DISCRETE PATTERN MATCHING OVER SEQUENCES AND INTERVAL SETS

(Ph.D. Dissertation)

James Robert Knight

TR 93-28

April 26, 1996

Department of Computer Science
The University of Arizona
Tucson, Arizona 85721

This research was supported in part by the National Institute of Health under Grant R01 LM04960, by the NSF under Grant CCR-9002351 and by the Aspen Center for Physics.

DISCRETE PATTERN MATCHING OVER SEQUENCES AND INTERVAL SETS

by

James Robert Knight

A Dissertation Submitted to the Faculty of the
DEPARTMENT OF COMPUTER SCIENCE
In Partial Fulfillment of the Requirements
For the Degree of
DOCTOR OF PHILOSOPHY
In the Graduate College
THE UNIVERSITY OF ARIZONA

1 9 9 3

ACKNOWLEDGMENTS

At a time like this, who don't you want to thank. Well, there was this one kid, back when I was . . ., but never mind that now. Top honors should, of course, go to the people who had a direct hand in the development of the dissertation. Beginning with my family, who got themselves involved with my, and hence its, creation. They taught me much of what I know, and more importantly, much of the way I think. To my advisor, Gene Myers, who among other things had the sense of timing to return from sabbatical during my second year of grad school, just when I was looking for a subject and advisor. To the rest of my committee, both past and present, for their helpful suggestions and advice. And finally to the folks in the department here. I've heard a number of horror stories about the trials and tribulations of other departments and companies around the country, and it is definitely a pity that I have to go out and discover whether they are true.

TABLE OF CONTENTS

LIST OF FIGURES

ABSTRACT

Finding matches, both exact and approximate, between a sequence of symbols A and a pattern P has long been an active area of research in algorithm design. Some of the more well-known byproducts from that research are the *diff* program and *grep* family of programs. These problems form a sub-domain of a larger area of problems called *discrete pattern matching* which has been developed recently to characterize the wide range of pattern matching problems. This dissertation presents new algorithms for discrete pattern matching over sequences and develops a new sub-domain of problems called discrete pattern matching over interval sets. The problems and algorithms presented here are characterized by three common features: (1) a "computable scoring function" which defines the quality of matches; (2) a graph based, dynamic programming framework which captures the structure of the algorithmic solutions; and (3) an interdisciplinary aspect to the research, particularly between computer science and molecular biology, not found in other topics in computer science.

The first half of the dissertation considers discrete pattern matching over sequences. It develops the alignment-graph/dynamic-programming framework for the algorithms in the sub-domain and then presents several new algorithms for regular expression and extended regular expression pattern matching. The second half of the dissertation develops the sub-domain of discrete pattern matching over interval sets, also called *super-pattern matching*. In this sub-domain, the input consists of sets of typed intervals, defined over a finite range, and a pattern expression of the interval types. A match between the interval sets and the pattern consists of a sequence of consecutive intervals, taken from the interval sets, such that their corresponding sequence of types matches the pattern. The name super-pattern matching comes from those problems where the interval sets corresponds to the sets of substrings reported by various pattern matching problems over a common input sequence. The pattern for the super-pattern matching problem, then, represents a "pattern of patterns," or super-pattern, and the sequences of intervals matching the super-pattern correspond to the substring of the original sequence which match that larger "pattern."

CHAPTER 1
INTRODUCTION

Discrete pattern matching is a domain of *exact* and *approximate pattern matching* problems which, in its most general form, compare an object such as a sequence of symbols and a pattern expressing a desired set of objects, such as the sequences described by a regular expression. This overall domain can be divided into sub-domains based on the object used in the comparison. Some of the sub-domains currently being researched are discrete pattern matching over (1) sequences, where the objects are sequences of symbols and patterns of symbols, (2) trees, where the objects are trees with value containing nodes, and (3) images, also called two-dimensional pattern matching, where the objects are two dimensional "images" made up of atomic symbols. While the objects in the various sub-domains may differ, similar sets of problems and approximate match criteria can be posed for those sub-domains.

Most of the research in this area has occurred in discrete pattern matching over sequences, however research in the other sub-domains has been on the increase as the core problems for discrete pattern matching over sequences have been efficiently solved. This dissertation presents several new algorithms solving problems in discrete pattern matching over sequences and develops a new sub-domain of discrete pattern matching over interval sets, also called *super-pattern matching*. In this new sub-domain, the input consists of sets of typed intervals and a pattern of interval types. Matches are formed from sequences of consecutive intervals whose types match a sequence in the pattern.

Discrete pattern matching has matured over the last twenty years into a distinct area of research, separate from the artificial intelligence work on pattern recognition and natural language processing and the parsing technology used in compiler design. For the two sub-domains of interest in this dissertation, discrete pattern matching over sequences and interval sets, the distinction rests on three characteristics common to the problems and algorithmic solutions:

- The notion of a "computable scoring function" used in approximate matching which serves both as a *metric* of similarity and as a guide for computing that metric. When comparing, say, sequences of symbols, the scores produced by the matching of pairs of sequences give a measure or metric identifying which sequences are similar and which are dissimilar. This is not unlike many other pattern recognition mechanisms. However, the scoring functions used in discrete pattern matching also identify the specific match between two sequences, out of all the possible ways those sequences could be matched, that produces the score representing the measure of their similarity. In doing so, the scoring function often presents the basic outline for computing similarity scores algorithmically.

- The use of state machines and a shortest-path-graph/dynamic-programming framework to derive the algorithms for exact and approximate matching problems. In the development of an algorithm solving an approximate matching problem, the matching problem is recast as a problem of finding either all paths or the shortest paths through a specially constructed graph. Then, efficient dynamic programming recurrences and algorithms solve the shortest path problem. The details of the structure of the graph and the terms in the recurrences depend on the language given in the pattern and the specified scoring function, but follow the guidelines of the framework. The solutions to the exact matching problems usually involve the development of a state machine, or some close relative, which is used to scan the input sequence for matches.

- The interdisciplinary nature of the research, particularly between computer science and molecular biology. Many of the algorithms and applications for discrete pattern matching have originated not in computer science, but in other fields such as molecular biology and speech processing.

While these characteristics may limit the scope of discrete pattern matching and exclude the more complex, context-sensitive issues in natural language processing and compiler design, the results produced in this research area have become part of the mainstream of computer science, as illustrated by the widespread use of utility programs such as *diff* and *grep* family. The area has also significantly altered research in molecular biology, making computer-based sequence analysis one of the daily activities of many biologists.

These characteristics have appeared throughout the history of discrete pattern matching. In fact, it could be said that discrete pattern matching became a separate problem domain when nine independent authors, by one count [SK83, Page 23-24], developed essentially the same algorithm, an algorithm with the characteristics mentioned above. Those results were motivated by two lines of research occurring in the mid 1960's: (1) the creation of metrics, such as the Hamming, evolutionary and Levenshtein distances, to describe the similarity of sequences and (2) the automatic correction of spelling mistakes and other typing errors [Dam64, Alb67, Mor70]. The fusion of the two produced a set of papers in computer science [NW70, WF74, Hir75], molecular biology [San72, RCW73] and speech processing [Vin68, VZ70, SC71, Hat74] which solve two related problems called the *edit distance* problem and the *longest common sequence* or LCS problem.

These two problems, and their relationship, are more easily described in terms of the key concept underlying the approximate matching problems over sequences, namely an *alignment* between two sequences. Informally, an alignment is a pairing of symbols of two sequences such that the lines of the induced *trace* don't cross, as in the following example:

```
a b c a d b b c c c
 \ \    / /  / / /
 a b b b a b c c
```

$$\begin{bmatrix}a\\a\end{bmatrix}\begin{bmatrix}b\\b\end{bmatrix}\begin{bmatrix}c\,a\,d\\\varepsilon\end{bmatrix}\begin{bmatrix}b\\b\end{bmatrix}\begin{bmatrix}b\\b\end{bmatrix}\begin{bmatrix}\varepsilon\\a\end{bmatrix}\begin{bmatrix}c\\b\end{bmatrix}\begin{bmatrix}c\\c\end{bmatrix}\begin{bmatrix}c\\c\end{bmatrix}$$

This figure illustrates two representations of an alignment, as a set of trace lines on the left and in a column-based arrangement on the right. The formal alignment definition is given in Chapter 2.

The edit distance problem involves finding the minimal number of edit operations which converts a sequence $A = a_1a_2\ldots a_M$ into a sequence $B = b_1b_2\ldots b_N$, where the three edit operations are (1) deleting a symbol, (2) inserting a symbol and (3) substituting one symbol for another. In terms of an alignment, the solution to a edit distance problem requires an alignment with the fewest unpaired symbols of A, unpaired symbols of B and mismatched pairs of symbols. This follows because each unpaired symbol of A corresponds to a required deletion operation in the edit, each unpaired symbol of B becomes an insertion and each mismatched pair becomes a substitution operation. Thus, the alignment with the fewest unpaired symbols and mismatches corresponds to the minimal set of editing operations.

The longest common subsequence of A and B asks for the longest sequence that forms a subsequence of both A and B. For example, "tusday" is the LCS of "tuesday" and "thursday." Under the alignment representation, the longest common subsequence corresponds to an alignment with the most paired symbols, where here no mismatched pairing are allowed. Since no crossing of an alignment's trace lines are allowed, each sequence of paired symbols forms a subsequence of both A and B, and the sequence with the most paired symbols must be the longest common subsequence of the two sequences. Note the relationship between the alignments for the two problems, as alignments with fewer unpaired symbols must contain more paired symbols. In fact, the LCS problem is the dual of the edit distance problem where no substitutions are allowed, or alternatively where substitutions count as two edit operations.

This relationship follows through to the algorithms solving the two problems. The key observation made by those nine authors is that every alignment between A and B must end in one of three ways: (1) by pairing a_M with b_N, (2) leaving a_M unpaired, and (3) leaving b_N unpaired. Thus, the "optimal alignment," i.e. the alignment corresponding to the edit distance or LCS solution, must be (1) the optimal alignment of $a_1 a_2 \dots a_{M-1}$ and $b_1 b_2 \dots b_{N-1}$ and the pairing of a_M and b_N, (2) the optimal alignment of $a_1 a_2 \dots a_{M-1}$ and $b_1 b_2 \dots b_N$ and an unpaired a_M, and (3) the optimal alignment of $a_1 a_2 \dots a_M$ and $b_1 b_2 \dots b_{N-1}$ and an unpaired b_N. This leads to the following *computational recurrence* describing the solution to the edit distance and longest common subsequence problems, respectively,

$$
\begin{array}{ll}
C_{0,0} = 0 & C_{0,0} = 0 \\
C_{i,j} = \min \{ \; C_{i-1,j-1} + \begin{cases} 0 & \text{if } a_i = b_j \\ 1 & \text{if } a_i \neq b_j \end{cases}, & C_{i,j} = \max \{ \; C_{i-1,j-1} + \begin{cases} 1 & \text{if } a_i = b_j \\ 0 & \text{if } a_i \neq b_j \end{cases}, \\
\qquad\qquad C_{i-1,j} + 1, & \qquad\qquad C_{i-1,j} + 0, \\
\qquad\qquad C_{i,j-1} + 1 & \qquad\qquad C_{i,j-1} + 0 \\
\qquad \} & \qquad \}
\end{array}
$$

where any terms with $i < 0$ or $j < 0$ are ignored. The value of $C_{M,N}$, then, equals either the minimal edit distance or the size of the LCS. Computing these recurrences involved the, then new, technique of dynamic programming and resulted in algorithms running in $O(MN)$ time.

From these initial two problems and their common solution, research into discrete pattern matching over sequences has expanded into a number of different problems and algorithms, as illustrated by Figure 1.1. Most of the research has been concerned with the matching of two sequences, or *sequence comparison*, and its variations. A growing number of results have been developed more recently for problems dealing with regular expressions and extended regular expressions. The first half of this dissertation surveys most of the work in this area (Chapter 2) and presents new algorithms solving (1) approximate regular expression pattern matching with concave gap penalties (Chapter 3) and (2) exact and approximate extended regular expression pattern matching (Chapter 4). All of these algorithms, both the previous work and the new results, are placed in an alignment-graph/dynamic-programming framework which simplifies much of the algorithms' complexity and links these results back to the simple edit distance and LCS solutions.

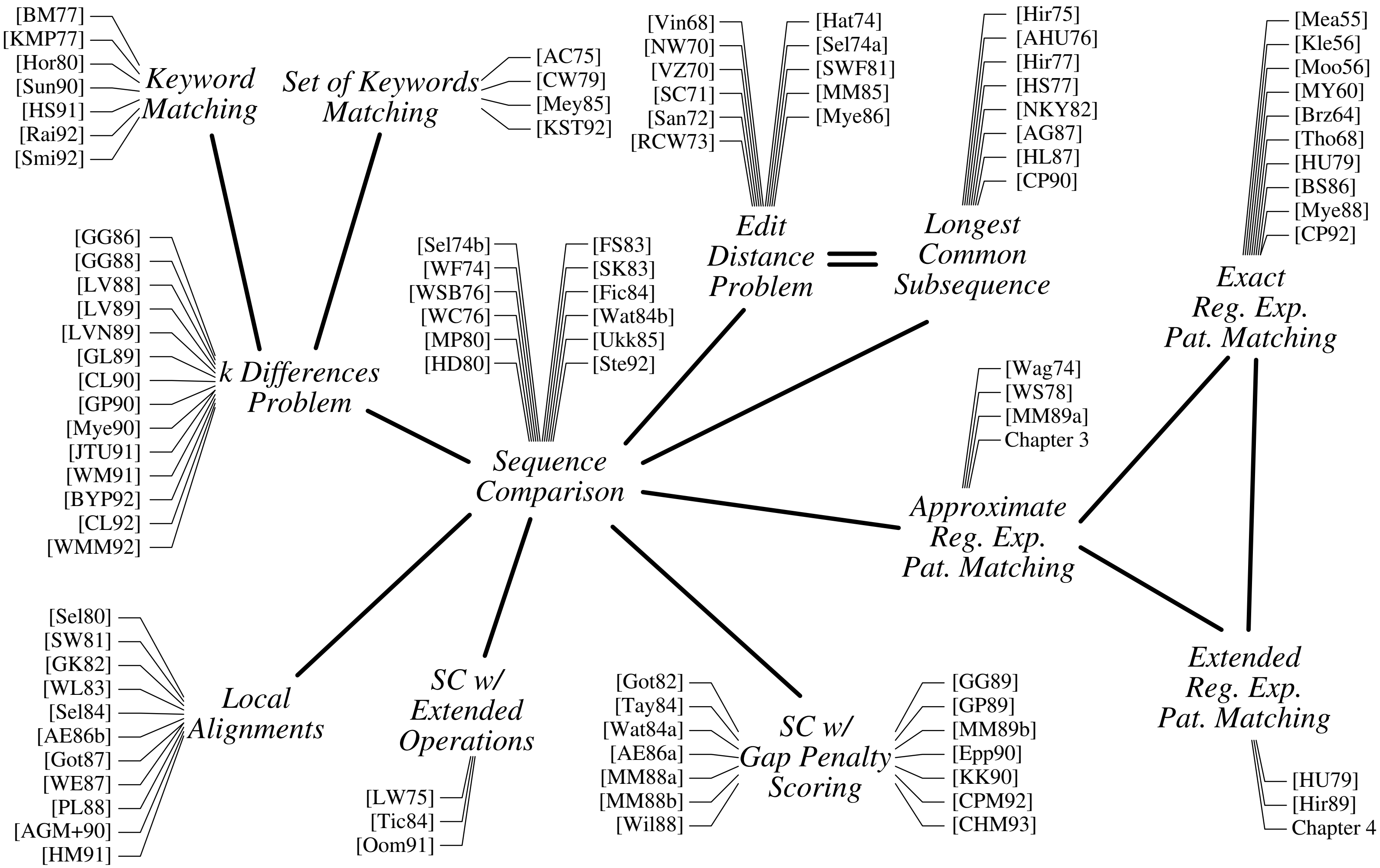


Figure 1.1: The various problems and solutions on discrete pattern matching of sequences.

The second half of the dissertation develops a new domain of problems called *super-pattern matching*, or discrete pattern matching over interval sets. It was developed for recognition problems that are either too complex or too ambiguous to be expressed using only pattern matching over sequences, i.e. the pattern cannot be expressed as a sequence or regular expression of the sequence symbols. In these cases, a richer environment is needed to describe the “patterns” and to perform the recognition of those “patterns.” Some researchers have turned to artificial intelligence techniques and multi-step matching approaches for the problems of gene recognition [FS90, GKDS92, Sea89], protein structure recognition [LWS87] and on-line character recognition [FCK+91]. Super-pattern matching defines a class of problems which involve finding matches to “patterns of patterns,” or super-patterns, given solutions to the lower-level patterns. The expressiveness of this problem class rivals that of traditional artificial intelligence characterizations, yet polynomial time algorithms are described for each problem in the class. Chapter 5 presents the domain of problems, containing several language classes, output requirements and error models, and then Chapter 6 presents algorithms solving those problems, using some of the shortest-path-graph/dynamic-programming techniques of discrete pattern matching over sequences.

CHAPTER 2

DISCRETE PATTERN MATCHING OVER SEQUENCES: PREVIOUS WORK

The sub-domain of discrete pattern matching over sequences covers a wide range of *exact* and *approximate pattern matching* problems, spanning a number of pattern language classes and scoring schemes. Figure 1.1 indicates the breadth of the field. Because of that breadth, most of this chapter concentrates only on the problems and algorithms needed by the rest of the dissertation. Specifically, this chapter presents the known solutions to the exact and up to five approximate pattern matching problems for sequence patterns (Section 2.1), regular expressions (2.2), and extended regular expressions (2.3). These problems make up the core of discrete pattern matching over sequences and contain all of the results needed by Chapters 3, 4 and 6.

The exact pattern matching problem asks simply whether the input sequence $A = a_1 a_2 \ldots a_M$ is in the language or set of sequences defined by pattern P, i.e. is $A \in L(P)$? Approximate pattern matching problems take an input sequence A, a pattern P and a scoring scheme S, and ask for the score of an *optimal alignment* between A and one of the sequences in $L(P)$, where the criterion of optimality is defined by S. 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 2.1. Scoring scheme S defines 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.

Formally, an alignment between sequences $A = a_1 a_2 \ldots a_M$ and $B = b_1 b_2 \ldots b_N$, over alphabet Σ, 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 2.1.

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 $i_0 = j_0 = 0$, $i_{t+1} = M$ and $j_{t+1} = N$. The score of the optimal alignment between A and a more complex pattern, such as a regular expression P, is RE(A, P, $\{pair, gap\}$) = $\min\{$ SEQ(A, B, $\{pair, gap\}$) $\mid B \in L(P)\}$.

The scoring schemes considered here consist of a *symbol-based* scheme and four *gap penalty* scoring schemes: affine, concave, convex and arbitrary. All of these 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, δ is extended to be defined over an additional symbol ε not in Σ, and the score of an unaligned symbol a is given by $\delta(a, \varepsilon)$. The score of a gap $a_{i_k+1} a_{i_k+2} \ldots a_{i_{k+1}-1}$

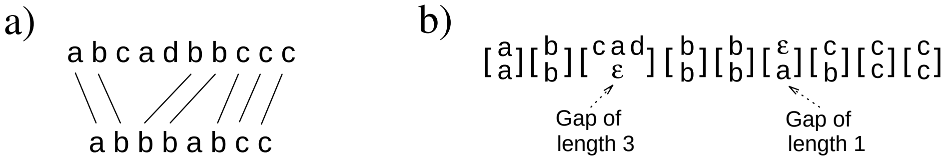


Figure 2.1: An alignment a) as a set of trace lines and b) in a column-oriented display.

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.

For the gap penalty models, 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. The four gap penalty scoring schemes specify four different classes of w functions. In an affine scoring scheme, w is a linear function of k, or $w(k) = e \cdot k + c$ for given "gap extension" and "gap creation" constants $e > 0$ and $c \geq 0$. For the concave schemes, 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 $\Delta w(k) \equiv w(k+1) - w(k)$. The convex scoring schemes require *convex* functions whose forward differences are non-decreasing, i.e. $\Delta w(1) \leq \Delta w(2) \leq \Delta w(3) \leq \ldots$. The arbitrary gap penalty scheme allows any function w.

The solutions to all of the problems described above, for any combination of pattern language class and scoring scheme, share a common *alignment-graph/dynamic-programming* framework. This framework uses four major steps in developing the algorithm from the specific problem definition. The first step constructs a state machine equivalent to the pattern, i.e. a machine which accepts the same language as the pattern expression. Second, the matching problem is recast as the problem 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. In the third major step, *dynamic programming recurrences* are derived from this graph which compute the shortest path costs from the source to each vertex. 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. Finally, algorithms solving these recurrences are given.

2.1 Sequence Comparison

Skipping the trivial solution to the exact matching problem, the algorithm solving sequence comparison under a symbol-based scoring scheme, SEQ(A, B, $\{\delta\}$), generalizes the edit distance and longest common subsequence algorithms presented in the introduction. This generalization was first given by Wagner and Fischer [WF74] and Sellers [Sel74a, Sel74b] and differs from the other algorithms only in the use of the more general δ function scores, rather than the simpler 0 and 1 scores. For this problem, the state machine for A is a row of states modeling the successive prefixes of A. The alignment graph construction takes the cross product of that state machine and a similar state machine for B. The resulting graph forms an $M+1$ by $N+1$ grid or matrix with

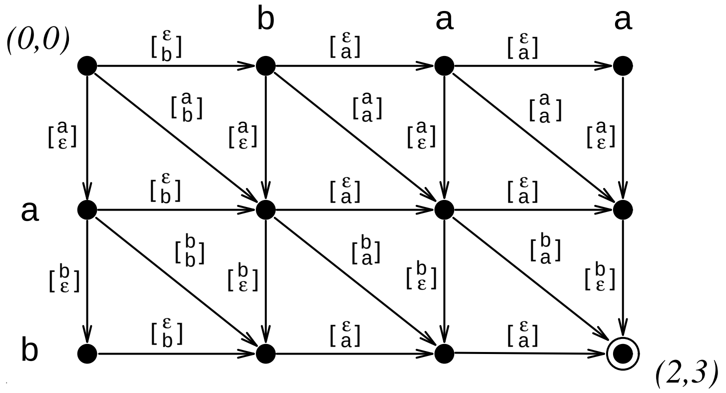


Figure 2.2: The sequence vs. sequence alignment graph for $A = ab$ and $B = baa$.

vertices denoted (i, j), for $i \in [0, M]$ and $j \in [0, N]$. One such graph is illustrated in Figure 2.2. For a 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 $j < N$). These terms for the edge types stem from the edit distance problem. They are useful here in that they distinguish the cases where symbols of A are left unaligned (deletions) from those where symbols of B are left unaligned (insertions).

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) 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 $\delta(a_i, b_j)$. 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:

$$C_{i,j} = \min\{\ C_{i-1,j-1} + \delta(a_i, b_j),\ \ C_{i-1,j} + \delta(a_i, \varepsilon),\ \ C_{i,j-1} + \delta(\varepsilon, b_j)\ \} \tag{2.1}$$

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.

Waterman, Smith and Beyer [WSB76] first proposed the gap penalty scoring model and solved SEQ(A, B, $\{\delta, w\}$) for an arbitrary function w. Converting their solution to the alignment-graph/dynamic programming framework, the constructed alignment graph augments the alignment graph for SEQ(A, B, $\{\delta\}$) with insertion and deletion edges that model multi-symbol gaps. These edges are needed because the cost of multi-symbol gaps 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

$(i+1,j)$, $(i+2,j)$, ... (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 $(i,j+1)$, $(i,j+2)$, ... (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). Thus, the recurrence becomes

$$C_{i,j} = \min\{\ C_{i-1,j-1} + \delta(a_i,b_j),\ \min_{0\leq k<i}\{C_{k,j} + w(i-k)\},\ \min_{0\leq k<j}\{C_{i,k} + w(j-k)\}\ \} \tag{2.2}$$

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.

The inefficiency of the naive dynamic programming algorithm 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 given an arbitrary gap penalty function w. However, under a more restricted choice of w, the sequence of deletion term minimums computed across a given column can be exploited to improve to the running time of the algorithm as a whole. Specifically for a given column j, one needs to deliver the sequence of deletion terms,

$$D_i = \min_{0\leq k<i}\{V_k + w(i-k)\} \tag{2.3}$$

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 in better than $O(M^2)$ time.

Gotoh [Got82] recognized that for affine gap penalty functions $w(k) = e\cdot k + c$, Equation 2.3 could be simplified to

$$\begin{aligned} D_i &= \min\{\ V_{i-1} + c + e,\ \min_{0\leq k<i-1}\{V_k + w(i-k)\}\ \} \\ &= \min\{\ V_{i-1} + c + e,\ \min_{0\leq k<i-1}\{V_k + w((i-1)-k) + e\}\ \} \\ &= \min\{\ V_{i-1} + c + e,\ D_{i-1} + e\ \}. \end{aligned}$$

Thus, the deletion and insertion terms can be computed in $O(M)$ and $O(N)$ time per row and column, resulting in an $O(MN)$ algorithm solving the overall sequence comparison problem.

For concave and convex gap penalties, two groups of papers have presented improvements over the naive $O(M^2)$ algorithm. The first group [MM88a, GG89, HL87] employed the concept of a *minimum envelope* and its corresponding *candidate list* to improve the running time to $O(M\log M)$ time. In their algorithms, the key to achieving a faster computation for Equation 2.3 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

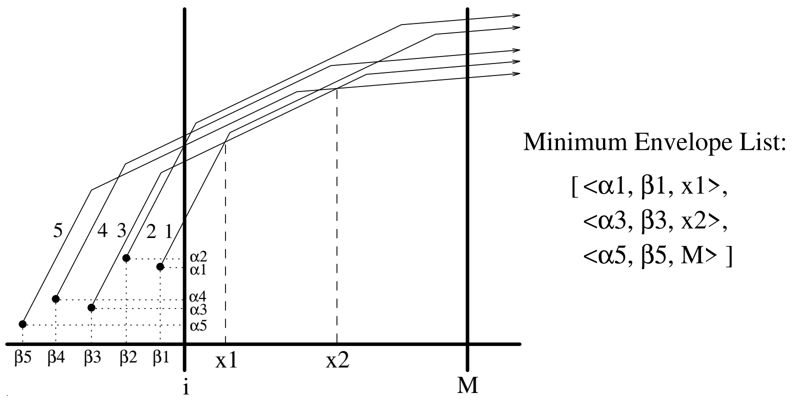


Figure 2.3: A minimum envelope E_i and its list representation.

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 α and β (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 α and β in the y- and x-axes, respectively. Because every candidate is a translation of the same concave or convex curve, it follows that any pair of such curves intersect at most once, although the intersection may occur over an interval of x values as opposed to just a single point. To see this in the concave case, 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), and then stays above it for all larger x. A similar argument shows this “single intersection” property for convex curves.

The minimum envelope at i, $E_i(x) = \min_{0 \leq k < i}\{C_k(x)\}$, is the minimum of a collection of variously translated copies of the same curve as illustrated in Figure 2.3. 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 these 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, β : 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:

$$E_i(x) = c_j.\alpha + w(c_j.\beta + x) \text{ for } x \in [c_{j-1}.x + 1, c_j.x] \tag{2.4}$$

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, for concave curves, it is also true that $c_{j-1}.\beta < c_j.\beta$ for all j because curves with small β's rise more quickly than those with larger β's. The inverse is true for convex curves.

The equations above for D_i and E_{i+1} in terms of E_i 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 2.3:

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

where [ ] denotes an empty candidate list.

Computationally, using 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. Considering operation *Add* for the concave case, 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 occurs because C_{i-1} has a smaller β value, $-(i-1)$, than any candidate in E and so rises faster than those candidates. C_{i-1}'s interval is empty if $V_{i-1} + w(1) \geq c_1.\alpha + w(c_1.\beta + i)$, 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 $1 \leq k \leq h$, $V_{i-1} + w(-(i-1) + c_k.x) \leq c_k.\alpha + w(c_k.\beta + c_k.x)$; 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. Again, the inverse is true for convex curves. Testing and removing from the tail of the list implements *Add* for convex curves, since a new candidate's interval of representation must span from the intersection point to M.

The time complexity of this algorithm is $O(M)$ times the computation needed to find each intersection point in operation *Add*, because each candidate is inserted into and deleted from the list at most once. 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 or convex gap-cost function w, the intersection point can be found using a $O(\log M)$ binary search over the range $[i, c_{h+1}.x]$ in the concave case and $[c_{h-1}.x, M]$ in the convex case, assuming $c_h, c_{h+1}, \ldots, c_{|E|}$ were removed in the convex *Add*. This gives an $O(M \log M)$ time bound for the algorithm.

The second group of results [Wil88, GP89, KK90, Epp90] improve the $O(M^2)$ running time of Equation 2.3 by employing a matrix searching technique originally presented in [AKM+87]. This technique computes 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

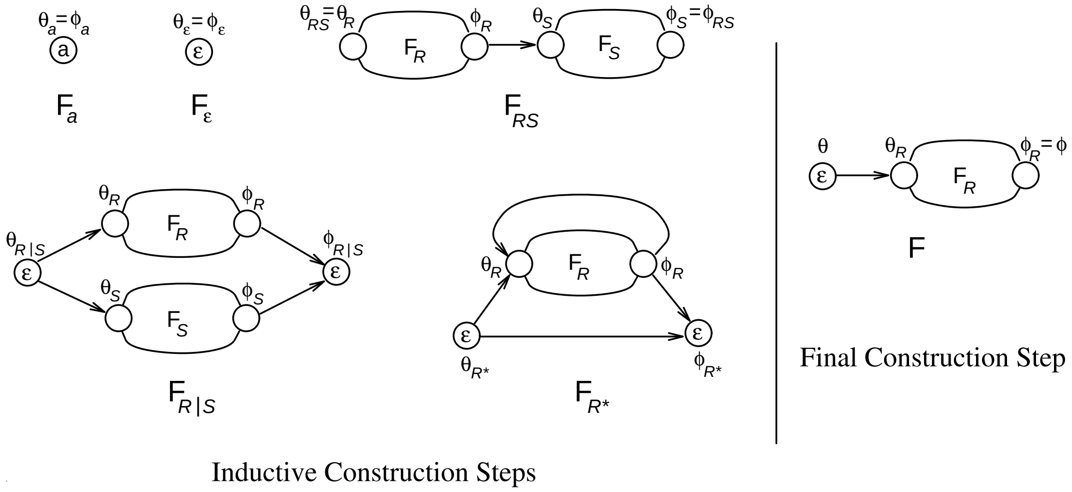


Figure 2.4: Constructing the NFA F for a regular expression R.

along each column h corresponds to the solution for D_h in Equation 2.3. The algorithms for convex [Wil88, GP89] and concave curves [KK90] use this matrix searching technique to solve the one-dimensional problem in $O(M)$ and $O(M\,\alpha(M))$ time respectively, where $\alpha(\ldots)$ is the inverse Ackermann function. Eppstein [Epp90] extends these results for piecewise convex/concave functions w with a resulting complexity of $O(MS\,\alpha(M/S))$, where S is the number of pieces. These results are only mentioned here because they are not applicable to the regular expression with concave gap penalties problem solved in Chapter 3.

2.2 Regular Expressions

We now turn our attention to problems that involve generalizing B to a regular expression P. Recall that a regular expression over alphabet Σ is any expression built from symbols in $\Sigma \cup \{\varepsilon\}$ using the operations of concatenation (juxtaposition), alternation (|), and Kleene closure (*). The symbol ε matches the empty string. For example, $a\,(b\,|\,\varepsilon)\,|\,c\,b*$ denotes the set $\{ab, a, c, cb, cbb, \ldots\}$.

There have been many results solving the exact matching problem. Most of these involve the construction of a *finite automaton* used to perform the matching [Mea55, Moo56, MY60, Brz64, Tho68, HU79, BS86, CP92]. The automaton construction algorithm presented below is used by Myers and Miller [MM89a] for their approximate matching solutions and is the construction used for the algorithms in Chapters 3, 4 and 6. Formally, this non-deterministic finite automaton, or 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 λ assigning a "label", $\lambda_s \in \Sigma \cup \{\varepsilon\}$, to each state s; (4) a designated "source" state θ; and (5) a designated "sink" state ϕ. Intuitively, F is a vertex-labeled directed graph with distinguished source and sink vertices. A directed path through F *spells* the sequence obtained by concatenating the non-ε state labels along the path. $L_F(s)$, the *language accepted at* $s \in V$, is the set of sequences spelled on all paths from θ to s. The *language accepted by* F is $L_F(\phi)$.

Any regular expression P of size N can be converted into an equivalent finite automaton F with the inductive construction depicted in Figure 2.4. For example, the figure shows that F_{RS} is

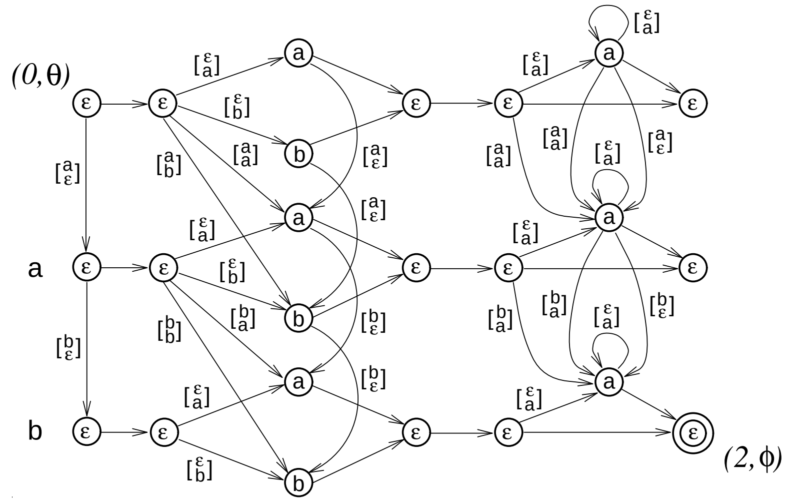


Figure 2.5: The regular expression alignment graph for A=ab and P=(a|b)a*.

obtained by constructing F_R and F_S, adding an edge from ϕ_R to θ_S, and designating θ_R and ϕ_S as its source and sink states. After inductively constructing F_R, an ε-labeled start state is added as shown in the figure to arrive at F. This last step guarantees that the 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 θ is 0; (2) the out-degree of ϕ is 0; (3) every state has an in-degree and an out-degree of 2 or less; and (4) $|V| \leq 2|N|$, i.e. the number of states in F is less than or equal to twice P's length. In addition, the structure of cycles in the graph $<V, E>$ of F has a special property. Term those edges introduced from ϕ_R to θ_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* [All70] or as having a *loop connectedness parameter* of 1 [HU75]. In summary, the key observations are that for any regular expression P there is an NFA whose graph is reducible and whose size, measured in either vertices or edges, is linear in the length of P.

For the symbol-based approximate matching problem RE(A, P, $\{\delta\}$) solved by Wagner and Seiferas [WS78] and Myers and Miller [MM89a], the alignment graph for A versus P consists of $M+1$ copies of F, as illustrated in Figure 2.5. 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 λ_s unaligned (in whatever word of P 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 λ_s. Note that by the construction of F, there are at most two insertion and two substitution edges out of each vertex, and $O(MN)$ 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 ε

unaligned and substitution edges to s model aligning a_i with ε. 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 P reduces to finding a least cost path between source vertex $(0, \theta)$ and sink vertex (M, ϕ). It is further shown in [MM89a] that all substitution and deletion edges entering ε-labeled vertices except θ 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 2.5 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:

$$C_{i,s} = \min\{ \min_{t \to s}\{C_{i-1,t} + \delta(a_i, \lambda_s)\},\ C_{i-1,s} + \delta(a_i, \varepsilon),\ \min_{t \to s}\{C_{i,t} + \delta(\varepsilon, \lambda_s)\} \} \tag{2.5}$$

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 δ 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 $O(N)$ 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 P 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 $O(MN)$ 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).

Introducing gap penalty scoring schemes has an effect on the alignment graphs of RE(A, P, $\{\delta\}$) 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, P, $\{\delta, w\}$) 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.

Myers and Miller [MM89a] present an $O(MN\,(M+N))$ algorithm for arbitrary, monotone increasing w, i.e. $w(k) \leq w(k+1)$, and an $O(MN)$ algorithm for affine gap costs. Beginning with the arbitrary, monotone increasing algorithm, the monotonicity of w implies that 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-ε symbols. Let $G_{t,s}$, hereafter called the *gap distance* between t and s, be the number of non-ε 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 $O(N^2)$ time.

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

$$C_{i,s} = \min\{\, \min_{t \to s}\{C_{i-1,t} + \delta(a_i, \lambda_s)\},\ \min_{0 \leq k < i}\{C_{k,s} + w(i-k)\},\ \min_{\forall t : t \xrightarrow{*} s}\{C_{i,t} + w(G_{t,s})\}\} \tag{2.6}$$

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. The direct computation of this recurrences using the two-sweep, node-listing algorithm described above can find $C_{M,\phi}$ in $O(MN(M+N))$ time, as the deletion and insertion terms require $O(M)$ and $O(N)$ time per graph node.

In [MM89a], Myers and Miller actually restrict the allowed functions w to those that are *sub-additive*, i.e. for all m and n, $w(m+n) \leq w(m) + w(n)$. This restriction is included to preclude sequences of insertion edges from consideration. When w is sub-additive, the minimal cost insertion gap from t to s must be a single insertion of a string spelled on a path from t to s rather than multiple insertions of smaller strings whose concatenation spells a path from t to s. However, 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 the sequence of insertion edges must form an acyclic path in F. Since the recurrence above plus the two-sweep algorithm consider all combinations of insertion edges which map to acyclic paths in F, two sweeps suffice even when w is not sub-additive.

In order to improve this algorithm for either the affine case considered here or the concave case presented in Chapter 3, the insertion terms $\min_{\forall t: t \xrightarrow{*} s}\{C_{i,t} + w(G_{t,s})\}$ of Equation 2.6 must be computed in less than $O(N^2)$ time. These insertion term computations along each row of the alignment graph are captured in the following one-dimensional problem:

$$I_s = \min_{\forall t: t \xrightarrow{*} s} \{V_t + w(G_{t,s})\} \tag{2.7}$$

where V_t is $C_{i,t}$. For the affine case, Myers and Miller present an algorithm which effectively simplifies this recurrence to

$$I_s = \begin{cases} \min_{t \to s}\{I_t,\ V_s + c\} & \text{if } \lambda_s = \varepsilon \\ \min_{t \to s}\{I_t + e,\ V_s + c + e\} & \text{if } \lambda_s \neq \varepsilon \end{cases}$$

Using this simplification, along with the Gotoh's simplification of the deletion gap recurrence, yields an $O(MN)$ complexity for the two-sweep algorithm as it takes $O(1)$ time to compute the recurrences at each vertex in the alignment graph.

2.3 Extended Regular Expressions

Extended regular expressions, as its name suggests, extends the set of regular expression operators with two additional operators, intersection ($\&$) and difference ($-$). The matches to expressions $R \,\&\, S$ and $R - S$ are defined as the set intersection and set difference, respectively, of the strings matching R and S. So for example, the expression $(a^*b^*cc \,\&\, ab^*c^*) - a^*bbbb^*c^*$ denotes the set of strings $\{acc, abcc, abbcc\}$.

A variation of this set of operators, allowing intersection and complementation instead of intersection and difference, formed the original operator set in the initial papers [Kle56, Brz64] defining regular expressions. As regular expressions were originally developed to describe nerve nets and logical circuits, the operators intersection and complementation naturally characterized the logical AND and NOT circuits. We use the alternative version with a difference operator, because it results in more useful expressions for the super-pattern matching in Chapters 5 and 6. The two are essentially equivalent as R' (complement of R) equals $\Sigma^* - R$ and $R - S$ equals $R \,\&\, S'$.

Despite its long history, few papers have tackled the matching of extended regular expressions. One paper [BS86] actually considered the exact matching problem and then remarked that it could not be solved using the algorithms described in the paper. Only two efficient algorithms have appeared for the exact matching problem, one by Hopcroft and Ullman [HU79, Exercise 3.23] and one by Hirst [Hir89]. The approximate matching problem has not been considered previously, although Section 4.2 may shed some light on this lack of results.

Hopcroft and Ullman solve the exact matching problem between sequence A and extended regular expression P using an $O(M^3N)$ dynamic programming algorithm. The algorithm computes the set of substrings of A matching each sub-expression of P in a bottom-up manner. A set of inductive recurrence rules, similar to the inductive NFA construction rules of Figure 2.4, sets the value of $C(i, j, R)$ to either 1 or 0 depending on whether $a_{i+1}a_{i+2}\ldots a_j$ matches the expression R. Specifically, the rules are:

1. $C(i, j, \varepsilon) = (i == j)$
2. $C(i, j, a) = (i + 1 == j) \wedge (a_j == a)$
3. $C(i, j, R \mid S) = C(i, j, R) \vee C(i, j, S)$
4. $C(i, j, R\, S) = \exists i \leq k \leq j : C(i, k, R) \wedge C(k, j, S)$
5. $C(i, j, R^*) = (i == j) \vee \exists i \leq k \leq j : C(i, k, R) \wedge C(k, j, R^*)$
6. $C(i, j, R \,\&\, S) = C(i, j, R) \wedge C(i, j, S)$
7. $C(i, j, R - S) = C(i, j, R) \wedge \overline{C(i, j, S)}$

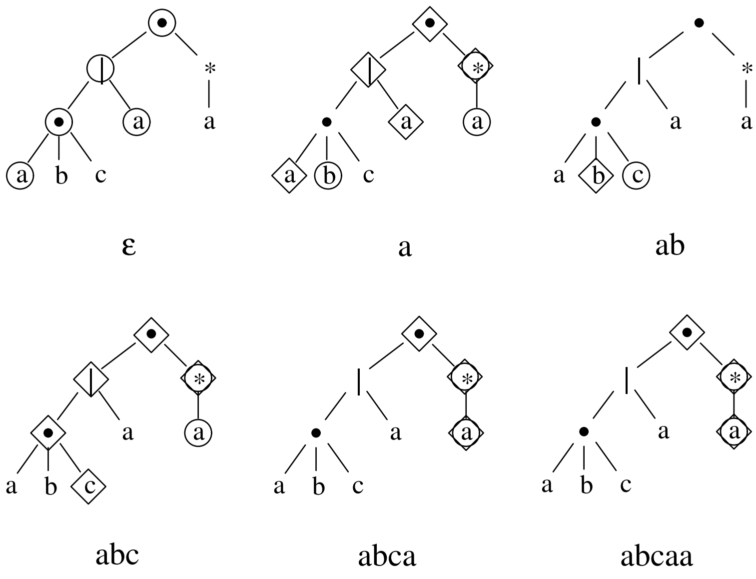
where "==" denotes a boolean equality with *true* as 1 and *false* as 0. Sequence A matches P iff $C(0, M, P)$ equals 1.

The complexity of this algorithm is $O(M^3 N)$, where M and N are the sizes of A and P, respectively. The algorithm computes the match between each sub-expression of P and all of the $O(M^2)$ substrings of A. In addition, the application of recurrences 4 and 5 take an average $O(M)$ time per substring of A. Curiously, with this algorithm, it is the concatenation and Kleene closure operations that directly account for the M^3 behavior, and not the new intersection and difference operations.

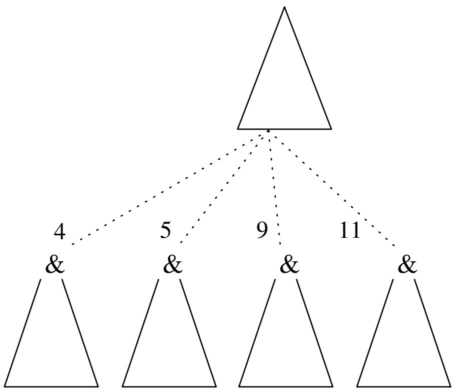
Hirst improved on this algorithm, although not its worst case complexity, by building a data structure based on the extended regular expression's parse tree and then using that to perform the matching. Although his specific algorithm is rather involved, the concepts and techniques used in that algorithm can be described by a tree marking algorithm. In this algorithm, the sequence A is scanned left to right and, for each successive symbol, the nodes of parse tree for P are annotated with two types of marks, one marking the sub-expressions to be matched against the substring of A not yet scanned and the other marking sub-expressions which have been completely matched against a suffix of the substring already scanned. Figure 2.6a gives the sequence of markings for an example sequence and simple regular expression, where the circles mark the sub-expressions to be matched and the diamond mark the sub-expressions which have been matched. Henceforth, the two types of marks will be denoted using the terms circle and diamond.

The algorithm begins by marking the root of the parse tree with a circle. It then follows a set of rules to generate the rest of the markings on the tree. The rules themselves are straightforward, given an understanding of the concatenation, alternation and Kleene closure operations (the intersection and difference operations are discussed momentarily). For instance, three rules govern a sub-tree whose root represents a concatenation operation: (1) when the root is marked with a circle, mark the leftmost sub-tree with a circle; (2) when a sub-tree of the root, except the rightmost sub-tree, is marked with a diamond, mark its right sibling with a circle; (3) when the rightmost sub-tree has a diamond mark, mark the root with a diamond. Once the markings on the tree are complete, i.e. no more rules apply, then the next symbol in the input is scanned and all nodes which are labeled with that symbol and marked with a circle are now marked with a diamond. All of the previous markings are then removed, and the rules are again applied to the new markings. This process ends when the last symbol of the sequence has been scanned and a complete set of markings has been generated. If the root of the tree is marked with a diamond, then the sequence A is accepted as a sequence in the language of the extended regular expression.

When the extended regular expression contains intersection and difference operators, the situation becomes more complex. The complexity results from the requirement that, with a sub-expression such as $R \,\&\, S$, the matches to R and S occur with the same substring of A. In an

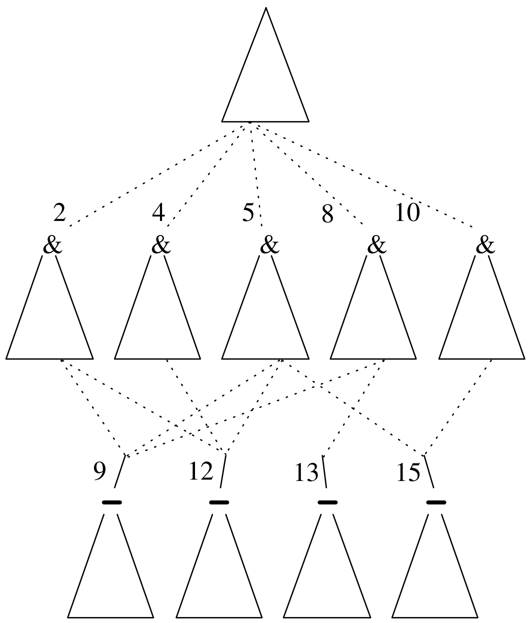


(a) Tree markings for pattern (abc | a) a* and sequence abcaa.
(Circles denote subtrees to be matched, diamonds denote matched subtrees)



(b) Duplicated subtrees for matching a single intersection sub-expression.

(Numbers denote the start positions of each substring match)



(c) Nested intersection and difference duplicated subtrees, with the extra links.

Figure 2.6: Examples of Hirst's tree marking algorithm.

algorithm which scans the text from left to right, this implies that extra information must be maintained for the currently active matches to sub-expressions of P. This extra information is used to guarantee that only valid matches to $R \,\&\, S$ can be computed from the matches to R and the matches to S, and similarly for the $R - S$ sub-expressions. For expressions containing no nested intersection or difference operators, Hirst creates this extra state information by duplicating the sub-tree of each intersection and difference node for each text position at which a potential match to that sub-expression could begin. Figure 2.6b illustrates this for an expression with one intersection operator. This duplication ensures that when the sub-tree for both R and S have been matched in one of the duplicates, then a valid match to $R \,\&\, S$ has occurred. Similarly, a valid match for $R - S$ occurs when a match to R but no match to S occurs in one of the duplicates.

When nested intersection and difference operators occur in the extended regular expression, the data structure used by Hirst's algorithm becomes that of Figure 2.6c. Rather than blindly creating duplicates within the duplicated sub-trees, only one duplicate of each intersection and difference sub-tree is created per text position. Extra links connect the nested duplicates to each of the higher level duplicated sub-trees for which a valid match to the nested sub-expression could extend the match to the higher level sub-expression. The algorithm performs the normal tree marking algorithm on all of the duplicated sub-trees, but reports matches to intersection and difference sub-expressions along all of its connecting extra links.

The basic tree marking algorithm used for the regular expression operators runs in $O(N)$ time per symbol of A. Hirst's actual algorithm also runs in $O(N)$ time per symbol, but improves the constant factor using a more efficient method for generating the marks on the tree. The complete algorithm's worst-case behavior results from the $O(M^3 N)$ time required to construct and maintain the duplicate sub-trees and extra links for the intersection and difference operators. For each "row" of duplicated sub-trees, a completed match to the intersection or difference sub-expression may have to be reported to $O(M)$ higher level duplicates modeling the various matches to the enclosing intersection or difference sub-expression. Thus, the processing of each "row" can take $O(M^2)$ time per text position. Multiplying by the number of possible rows and text positions gives the $O(M^3 N)$ time bound. Note, however, that this worst-case complexity does not reflect the behavior of the algorithm for many extended regular expressions, unlike the Hopcroft-Ullman algorithm. The actual time taken by the algorithm depends on the structure of the extended regular expression and the number of substrings matching each intersection and difference sub-expression. In many cases, that time will be less than $O(M^3 N)$.

CHAPTER 3

APPROXIMATE REGULAR EXPRESSION PATTERN MATCHING WITH CONCAVE GAP PENALTIES

The approximate regular expression pattern matching problem with concave gap penalties involves finding the score of the optimal alignment between a sequence $A = a_1 a_2 \ldots a_M$ and a regular expression P of size N. The various alignments are scored using the concave gap penalty scoring scheme $S = \{\delta, w\}$ described in Chapter 2. Section 2.1 presents the $O(MN(\log M + \log N))$ solution to SEQ(A, B, $\{\delta, w\}$) with concave w, i.e. the sequence comparison problem with concave gap penalties, and Section 2.2 presents the $O(MN(M + N))$ solution to RE(A, P, $\{\delta, w\}$) with monotone increasing but otherwise arbitrary w. Here, N denotes the size of either B or P. This chapter combines the techniques used in those two algorithms to solve RE(A, P, $\{\delta, w\}$) with concave w in $O(MN(\log M + \log^2 N))$ time. However, this merger is not straightforward, requiring an *applicative* implementation of the candidate lists used in the sequence comparison problem (see Section 2.1) and the use of a stack of those candidate lists by the new algorithm.

The $O(MN(M + N))$ regular expression algorithm from Section 2.2 can be improved to $O(MN(\log M + \log^2 N))$, when w is restricted to a concave function, by more efficiently computing Equation 2.6 for each node (i, s) in the alignment graph. Specifically, this improvement comes from the insertion and deletion terms of Equation 2.6 along each row and column of the alignment graph, isolated here in terms of the following one-dimensional problems:

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

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

The sub-cubic complexity of the algorithm solving RE(A,P,$\{\delta, w\}$) with concave w rests on computing (1) the D_i values in less than $O(M^2)$ time and (2) the I_s values in less than $O(N^2)$ time. Note that Equation 2.6 and these one-dimensional recurrences actually only hold for the concave gap penalty problem when w is also monotone increasing, but, as is discussed below, the concave functions which are not monotone increasing can be treated using a variation of the solution for the monotone increasing case.

One of the algorithms in Section 2.1 solves the deletion term recurrence above in $O(M \log M)$ time by using *minimum envelopes* and their *candidate list* implementation. This chapter presents an $O(N \log^2 N)$ algorithm for computing the I_s values by combining the minimum envelope techniques with the two-sweep, node-listing approach used in approximate regular expression matching. Section 3.1 presents the minimum envelope characterization of the insertion term recurrence above and reimplements the candidate list data structure in order to handle the greater complexities of that recurrence. Sections 3.2 and 3.3 then give the algorithms for each of the two sweeps used to compute the I_s values. The complete algorithm solving RE(A, P, $\{\delta, w\}$) is the two-sweep, node-listing algorithm of Section 2.2, which computes the shortest paths to each vertex in the alignment graph row by row. For each graph row i, the algorithm uses $O(M)$ candidate lists to deliver the deletion and substitution terms of Equation 2.6 to the computations given in Section 3.2 and 3.3. They then deliver the final $C_{i,s}$ values, for each vertex (i, s) in the row.

In the treatment that follows, we focus only on the case where w, in addition to being concave, is monotone increasing. This is because the problems involving more general concave w can be solved using the monotone increasing solution, as follows. 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 P contains at least one Kleene closure operator, the Kleene closure admits arbitrarily long gaps scoring $-\infty$, and consequently such matching problems are ill-posed. However, when P 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 2.6. $L_{t,s}$ is the largest number of non-ε symbols spelled on a path from t to s. The first sweep algorithm in Section 3.2 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 second sweep algorithm is not needed, since there are no cycles in the alignment graph. The complete algorithm for problems with concave but not monotone increasing w concurrently executes two versions of the algorithms in Section 3.2, 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.

3.1 Generalizing the Minimum Envelopes

For this problem, an envelope at s or E_s captures the future contribution of s's predecessors and is used to expedite the computation of I-values at the successors to s. What is desired at each s is

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

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 F, 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,

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

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

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 this chapter, 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.

Unfortunately, the more complex structure of the NFA F gives rise to a number of complications, requiring a more general data structure modeling the minimum envelopes. 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, θ, is x. However, this fails for regular expressions because there are automata for which two states at equal gap distance from θ 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 Sections 3.2 and 3.3 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 non-zero. However, candidates with a β-value of 0 cannot be treated under the minimum envelope formulation of Section 2.1, 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 zero β-valued candidates separately.

The most significant complication arises because of the multiple paths introduced by alternation and Kleene closure sub-automata. Any incremental algorithm building each E_s from s's immediate predecessors must use the envelope at each $\theta_{R\,|\,S}$ and θ_{R^*} at two different successors. The construction at those two successor states, which make different alterations to the original envelope, cannot be allowed to affect each other. The implementation of candidate lists modeling an envelope must be made *persistent* or *applicable*, so that the construction occurring along each path through F does not affect the construction on concurrent paths.

At each $\phi_{R\,|\,S}$ and ϕ_{R^*} state, the candidate lists for that state's two predecessors must be merged, in some fashion, so that $E_{\phi_{R\,|\,S}}$ or $E_{\phi_{R^*}}$ contains the union of the candidates at the predecessors. 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 $O(N)$, such a merge operation at each $\phi_{R\,|\,S}$ and ϕ_{R^*} yields an $O(N^2)$ time bound. Barring a more efficient merge operation, it does not appear than any algorithm using either a single or constant number of candidate lists can compute the I_s values in less than $O(N^2)$ time. Too many candidates occur in both envelopes in the merges: either 1) candidates from the same state t occurring in the lists being merged at a single $\theta_{R\,|\,S}$ or θ_{R^*}, or, more subtle duplication, 2) candidates from a state t needed in the series of final states occurring when F contains 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, Δ): returns a candidate list E' for which $E'(x) = E(x + \Delta)$ when $\Delta \geq 0$.
(3) *Add* (E, α, β): 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 rest of this 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) ≤ Len(E2)
    then E ← E1; E' ← E2
    else E ← E2; E' ← E1
  for c ∈ E do
    E' ← Add (E', c.α, c.β)
  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 $E2$ models $\min\{E1(x), E2(x)\}$ 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 2.1 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 variation of the algorithm in Section 2.1 correctly computes the D_i values specified by Equation 2.3:

```
E ← [ ]
for i ← 1 to M do
{ E ← Add (Shift (E, 1), V_{i−1}, 1)
  D_i ← Value (E, 0)
}
```

3.1.1 Operations *Value*, *Shift* and *Add*

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

$rp(k) \;\equiv\; E_k.x$ # 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
$LE(k) \equiv\; m = |E|$ **or** $E_k(lp(k)) \leq \alpha + w(\beta + lp(k))$ # if E_k is minimal at
$rp(k)$ and $lp(k)$, resp.

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

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

Add (E, α, β)
{ **if** $|E| = 0$ **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* $(k : k > m \;\&\; RE(k))$
  $F \leftarrow$ *Append* $(E_{1..l},\ [<\alpha, \beta, \infty>],\ E_{h..|E|})$
  if $l > 0$ **then**
    $F_l.x \leftarrow$ *Intersect* (F_l, F_{l+1})
  if $h \leq |E|$ **then**
    $F_{l+1}.x \leftarrow$ *Intersect* (F_{l+1}, F_{l+2})
  return F
}

Figure 3.1: The procedures *Value*, *Shift*, and *Add*.

following operations, in addition to the typical list operations, are assumed. Their implementation is discussed in the next section.

(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 ∞ 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 s.t. $e(x) < f(x)$.
(5) *Offset* (E, Δ): returns E' s.t. 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 $[1, |E|]$. 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 ∞. We will use ∞ in such contexts to denote a number that is sufficiently large. In all cases such a number is easy to arrive at, e.g. $|E| + 1$ in the case that the predicate of $Findmin$ is defined over list E.

Figure 3.1 presents the procedures implementing *Value*, *Shift*, and *Add*. The realization of operation *Value* follows directly from Equation 2.4 in Section 2.1. 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 ∞ occurring in *Value*, an appropriate choice when solving RE(A, R, $\{\delta, w\}$) is $w(1) \cdot (M + P) + 1$.

Operation *Shift* must add Δ to each record's β-field and subtract Δ from each record's x-field, since in the desired envelope $E'_k.\beta = E_k.\beta + \Delta$ and $E'_k.x = E_k.x - \Delta$. 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 Δ. Because record in the list appear in increasing order of this field, the call to ***Findmin*** finds the leftmost candidate which remains active. The sublist to the right of that candidate, inclusively, is extracted from E. Then, operation *Offset* updates the x- and β-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 3.2. Recall that the candidates in a list occur in increasing order of their β-fields. Thus, the new candidate's interval of representation, if not empty, occurs between the candidates in E with lesser β's and those with greater β'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 β's (moving leftward) and rises more quickly than the candidates with larger β'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 $m < k \leq |E|$. Let rp = $rp(m)$ when $m > 0$ and 0 otherwise, and let lp = $lp(m+1)$ when $m < |E|$ and ∞ 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 "$m = |E|$" 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.

3.1.2 Implementing the Candidate Lists

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

[1] N suffices here as longer gap distances will not be encountered. Similarly M and N suffice for the one-dimensional column and row problems posed in SEQ(A, B, $\{\delta, w\}$).

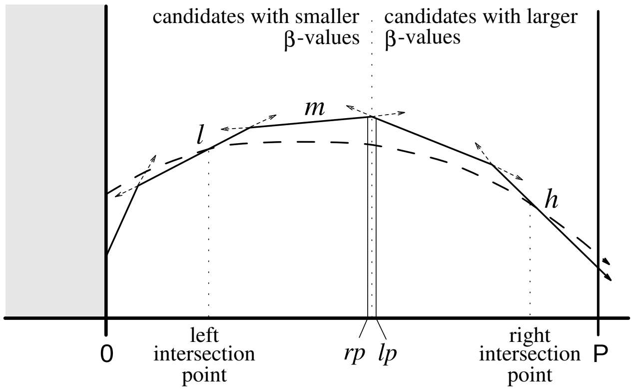


Figure 3.2: Adding a curve to an minimum envelope.

given by an inorder traversal of the tree. This well-known representation for a linear list [Knu73, pages 463-468] permits all of the typical list operations, plus the binary search used by ***Findmax*** and ***Findmin***, in either constant or logarithmic time of the length of the list. In addition, Myers [Mye84] 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 $O(\log N)$ operation takes $O(\log N)$ space as well.

Primitive operation (4), *Intersect*(e, f), involves the monotone predicate $e(x) < f(x)$ whose range is restricted to $x \in [0, N]$, because of the single intersection property of concave curves and because N is the largest possible gap distance between any two states. Thus an $O(\log N)$ binary search over this range implements *Intersect*. Operation *Offset*(E, Δ), primitive operation (5), can be realized in $O(1)$ time over a height-balanced tree as noted for link-cut trees [ST85]. The "trick" is to store the β and x values for a candidate record as *offsets relative* to the β and x values at the parent of the candidate in the tree. With such a scheme, $E_i.\beta$ is obtained by summing the β-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, Δ) simply involves adding and subtracting Δ to the β and x values at the root.

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 [ST85].

The final consideration is the extension of candidate lists to include candidates whose β-value is zero. These extended lists consist of two parts: $E.\mathit{list}$, a "standard" envelope for the candidates whose β-value is greater than zero; and $E.\alpha 0$, the α value of the best candidate with a β 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)                Shift⁺(E, Δ)                          Add⁺(E, α, β)
{ a ← E.α0 + w(x)           { if Δ = 0 then return E              { if β = 0 then
  b ← Value(E.list, x)        E.list ← Shift(E.list, Δ)               E.α0 ← min{E.α0, α}
  return min{a, b}            if E.α0 ≠ ∞ then                      else
}                             { E.list ← Add(E.list, E.α0, Δ)         E.list ← Add(E.list, α, β)
                                E.α0 ← ∞                            return E
                              }                                   }
                              return E
                            }
```

3.2 The First Sweep Algorithm

In the computation of envelope $E1_s(x) = \min_{\forall t : t \xrightarrow{*} s} \{V_t + w(G_{t,s} + x)\}$ at each state s in F, the first sweep algorithm need only consider the subgraph of F restricted to DAG edges. Over this acyclic subgraph, the set of *predecessors* of state s are those states t where $t \xrightarrow{*} s$, 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 P. This notion of up and down can be better illustrated by hypothetically extending the two-dimensional NFA's illustrations 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 $F_{R\,|\,S}$ 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.

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 3.3, 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 P and a root node corresponding to P 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 partition the states of 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 $t \xrightarrow{*} s$ and $N_s \xrightarrow{*} N_t$, 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

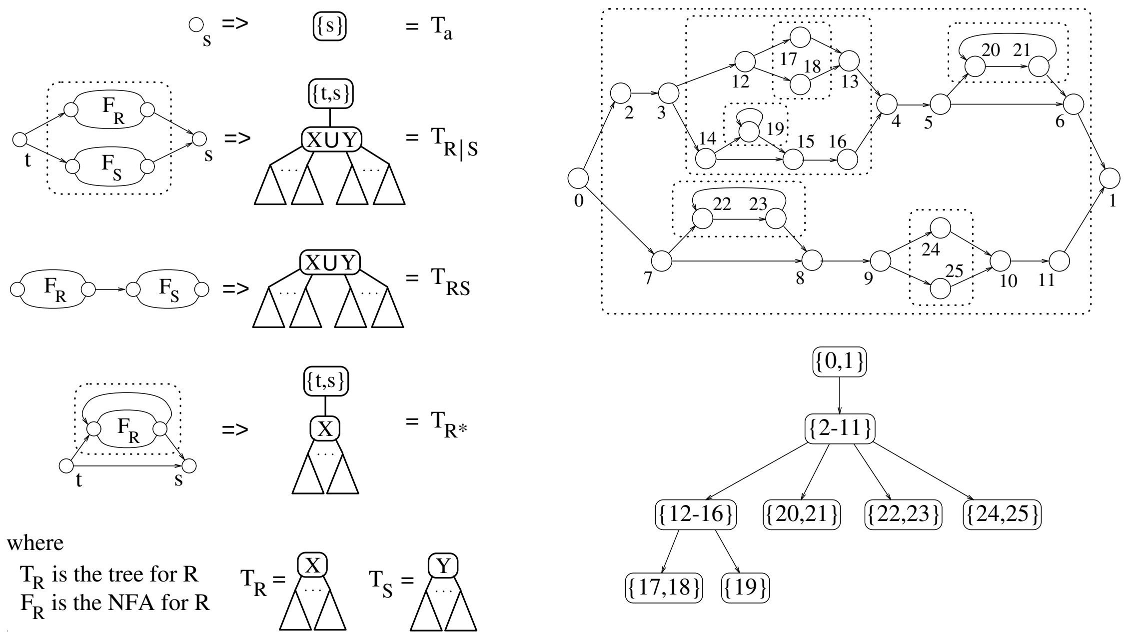


Figure 3.3: The nesting tree construction and an example nesting tree.

then decompose the computation of $E1_s$ into the computation of:

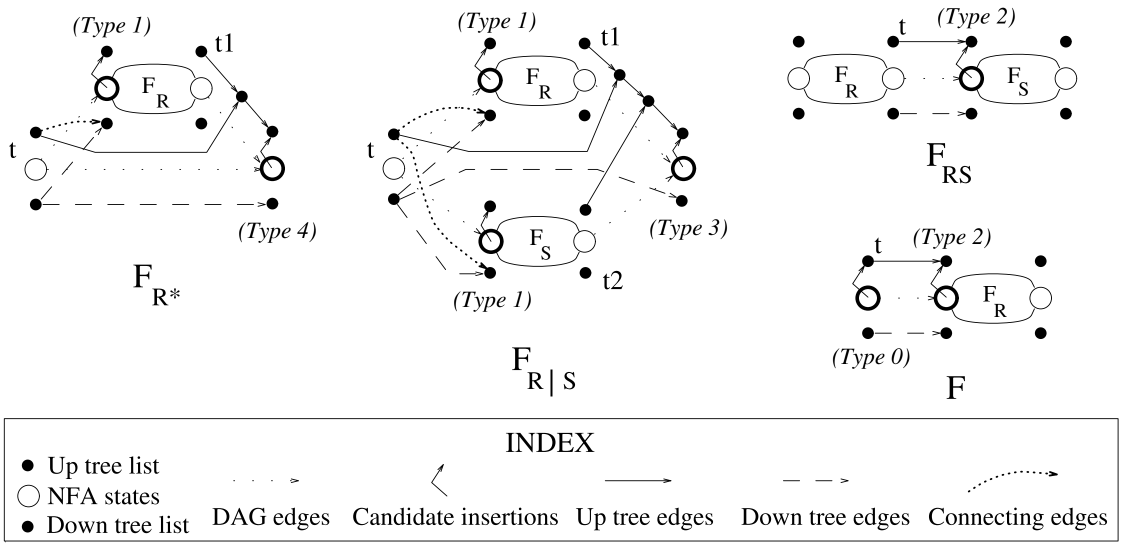
$$EU1_s(x) = \min_{\forall t: t \xrightarrow{*} s} \{V_t + w(G_{t,s} + x) \mid N_s \xrightarrow{*} N_t\}$$

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

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 the movement of candidate curves through the data structures constructed at each state. Figure 3.4 gives the inductive construction of the flow graphs. Figures 3.5 and 3.6 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 3.4. Figure 3.4 also gives the predecessor state notation



Type 0: The start state, θ (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 $F_{R\,|\,S}$ (predecessors $t1$ and $t2$, $t = \theta_{R\,|\,S}$).
Type 4: The final state of F_{R^*} (predecessor $t1$, $t = \theta_{R^*}$).

Figure 3.4: The inductive construction of the up and down trees at each state s.

which is used throughout this section. The single type 0 state is the start state θ, 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 3.4).

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

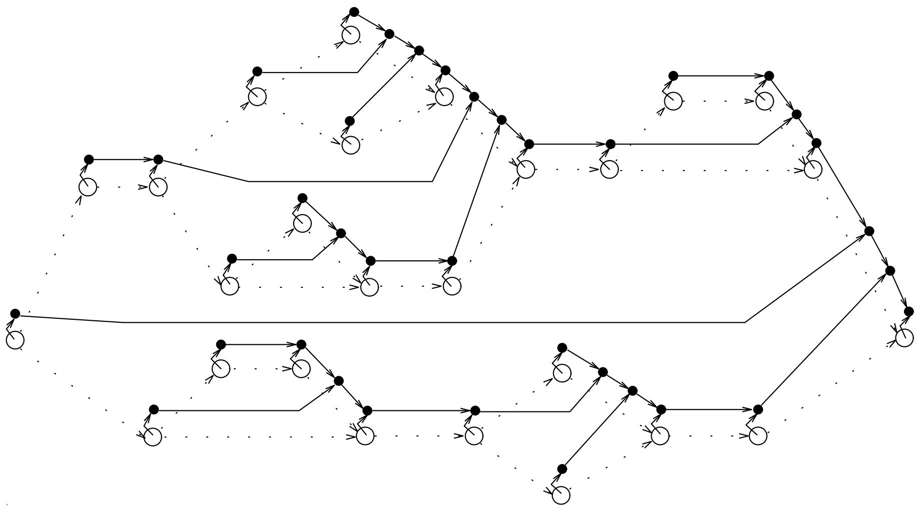


Figure 3.5: The up tree for the example NFA in Figure 3.3.

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 up lists U_s constructed by the algorithm model $EU1_s$ for every 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:

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

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. □

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

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

LEMMA 2. For any NFA F with N states, the *Merge* operations in the up list construction require at most $N \lfloor \log N \rfloor$ *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 3.5, which forms a tree, laying on its side, with N 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 N, s's candidate can be copied at most $\lfloor \log N \rfloor$ times. This argument holds for each state in F, so at most $N \lfloor \log N \rfloor$ *Add* operations are used by *Merge2* over the course of the construction. Therefore, no more than $N \lfloor \log N \rfloor$ *Adds* can be used under operation *Merge*. $\square$

3.2.2 The Down List Construction

The down list data structure uses up to $\lfloor \log N \rfloor + 1$ 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 $O(N^2)$ time however, because too many of the up lists can contain $O(N)$ active candidates. Instead, our algorithm uses up to $\lfloor \log N \rfloor + 1$ 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

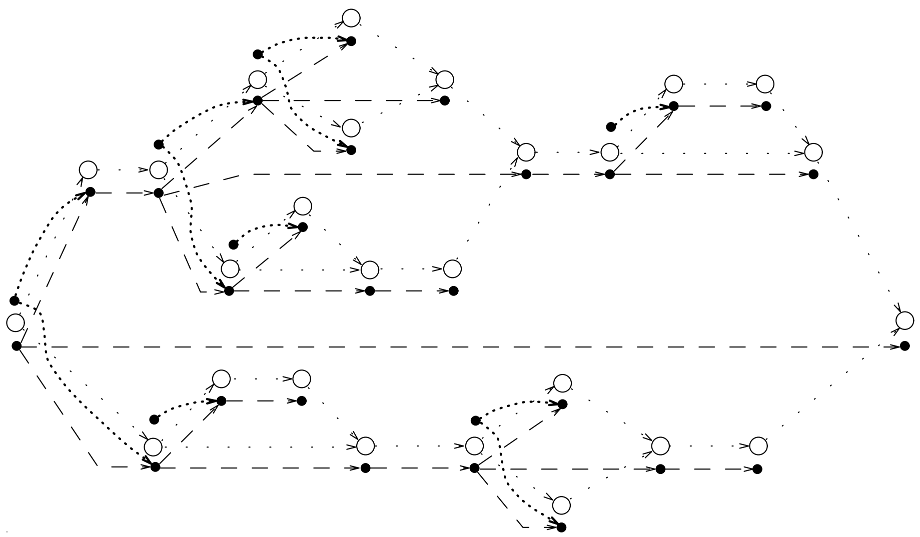


Figure 3.6: The example NFA's down tree.

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

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

Type 1: θ_R and θ_S 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: θ_S in F_{RS} and θ_R in 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 θ, $EH1_\theta$ contains no candidates (N_θ 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 all of the states in the sub-tree rooted at N_s appear 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 $EH1_s(0) = \min\{H_{0,s}(0), H_{1,s}(0), \ldots, H_{k_s,s}(0)\}$. Components 2 and 3 take $O(\log N)$ time multiplied by the height of the stack at each state. The time spent for component 1 is $O(N \log N)$ plus the cost of the *Merge* operations. The actual time bounds for these components depend 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 $\lfloor \log N \rfloor$ lists at any state and (2) the *Merge* operations used to construct H_0 require no more than $O(N \log N)$ *Add* operations, then the algorithm's complexity is $O(N \log^2 N)$. Thus, the $O(N \log^2 N)$ 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 P, so each edge $b \to c$ corresponds to the copy decision made at the start state of c's submachine. Denote this start state θ_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 $b \to c$ in the tree is labeled with an *edge set* $X_c = \{s \mid b \overset{*}{\to} N_s \;\&\; c \overset{*}{\not\to} N_s\}$, i.e. the states in the sub-tree rooted at b minus the states in the sub-tree rooted at c. Figure 3.7 gives the labeled tree for the example NFA being used in this section. The label removal 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.

Observe from the figure 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 $s \overset{*}{\to} \theta_c$. 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 a correct procedure for the label removal problem on T can be applied to the copy decision problem for the down list construction algorithm for F.

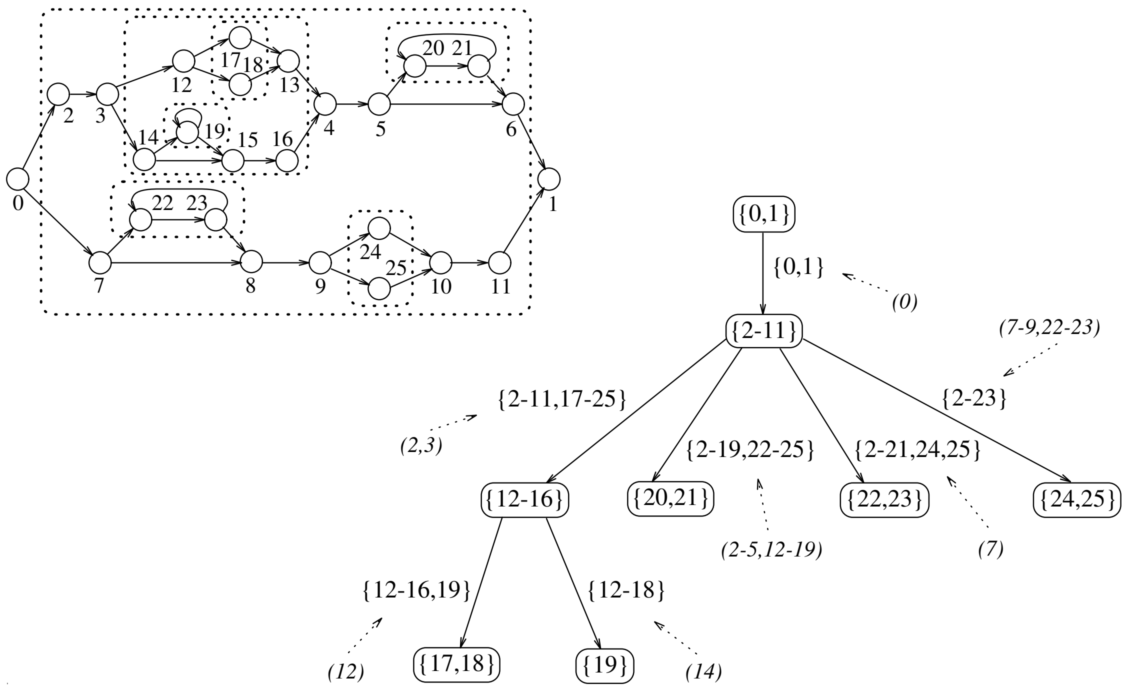


Figure 3.7: The nesting tree from Figure 3.3 labeled with edge sets X_c (in brackets) and the actual $EU1_{\theta_c}$ sets (in parentheses).

Proof. Let the fate of edge set X_c, labeling a tree edge $b \rightarrow c$, represent the copy decision needed at state θ_c as follows. The removal of X_c represents setting COPY1(θ_c) = **true** and merging U_{θ_c} into H_0. Retaining X_c represents setting COPY1(θ_c) = **false** and pushing U_{θ_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 $\lfloor \log |T| \rfloor \leq \lfloor \log N \rfloor$ up lists onto the stack. (2) If no state occurs in more than $\lfloor \log |T| \rfloor + 1$ removed edge sets, then at most $\lfloor \log N \rfloor + 1$ *Add* operations can be used to merge a particular state into H_0, resulting in an overall bound of $O(N \log N)$ *Add* operations. $\square$

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(θ_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(θ_{c_i}) = **true**] and retain the other edge sets [COPY1(θ_{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 $L_{root} + 1$ 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 $b \overset{*}{\to} N_s$, 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 $O(\log N)$, and Lemmas 6 and 5 show that $R_s(root)$ is $O(\log N)$ for any state s. Therefore, the height of the stack at any state is $O(\log N)$, no more than $O(N \log N)$ *Add* operations are used in the construction of H_0, and the overall time bound for the down list construction is $O(N \log^2 N)$.

3.3 The Second Sweep Algorithm

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

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

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. This can be seen from the graphical rules of Figure 2.4. 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-ε 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 computation of $E2_s$. Other versions coming from non-innermost surrounding back edges to s can be safely ignored by the data structures since, by the presence of the candidate from the innermost surrounding back edge, those versions can never contribute to a value to $E2$.

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 is used to propagate each up list to the states inside the corresponding back edge, but outside all nested back edges. At those nested back edges, a copy decision is made and that 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 3.8. This second sweep tree differs from that of the first sweep in that it only contains nodes for each of P's Kleene closure sub-expressions. Figure 3.8 presents the formal nesting tree 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 3.8. So, envelopes $EI2$ and $EH2$ model the innermost and down predecessors:

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

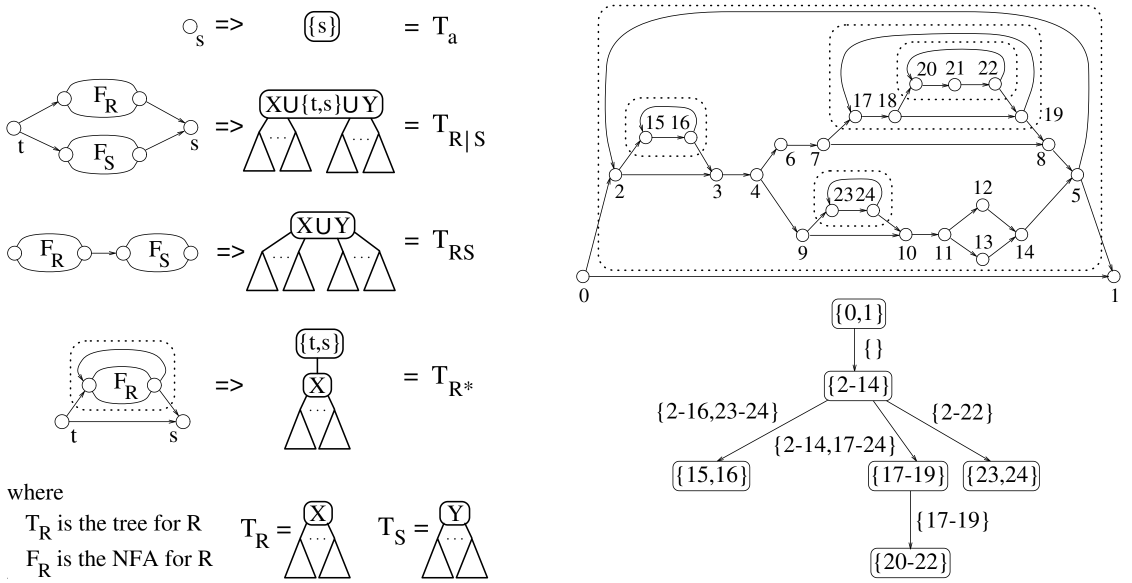
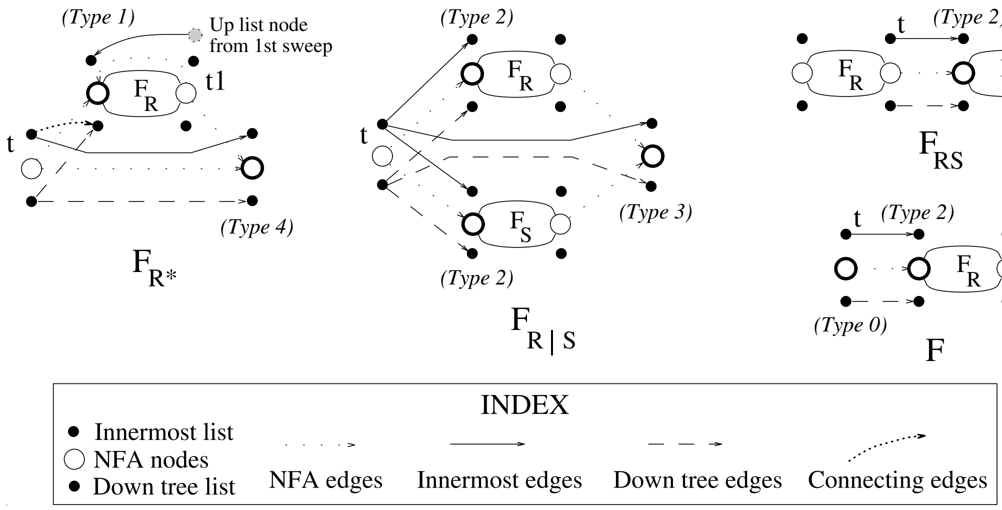


Figure 3.8: The second sweep nesting tree construction and an edge labeled example.

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

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

Figure 3.9 defines the construction of flow graphs for the innermost and down lists, along with the second sweep state types. Figures 3.10 and 3.11 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 $\lfloor \log N \rfloor + 1$ candidate lists, $H_{0,s}$, $H_{1,s}$, …, $H_{k_s,s}$. The construction steps for the two lists are:



Type 0: The start state, θ (no predecessors).
Type 1: Inner start states in F_{R^*} (DAG edge pred. t, back edge pred. $t1$).
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^*} (corres. start state t, other predecessor $t1$).

Figure 3.9: The second sweep flow graph construction.

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

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 init. 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 3.9 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.

The construction of the innermost list at each state and the construction of the down list at

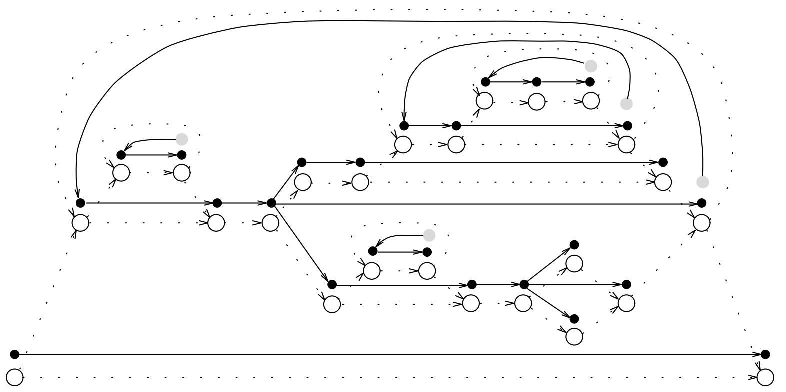


Figure 3.10: The flow graph for the innermost list.

all but the type 1 states are straightforward, and so are not considered further. 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 θ_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 exactly modeling $EJ2$ cannot be constructed at every type 1 state in $O(N \log^2 N)$ time. 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 $O(N \log^2 N)$ 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 $EH2_v$.

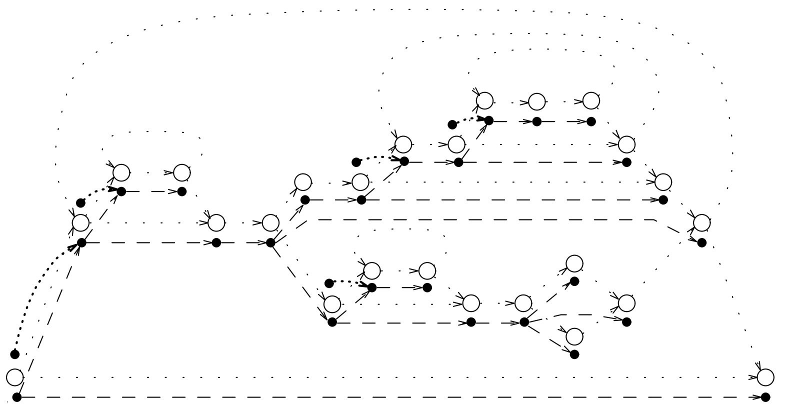


Figure 3.11: The flow graph for the second sweep down list.

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 $O(N \log^2 N)$ 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 $u \to v$, 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* $(J1_t, G_{t,s}))$ # $t = \theta_{R|S}, t1 = \phi_R$,
$J1_s \leftarrow$ *Add* (*Merge* $(J1_s, J1_{t2}), V_s, 0)$ # 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 $O(N \log N)$, since a single candidate list is incrementally constructed from predecessor states. $J1$ can be constructed in $O(N \log^2 N)$ 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 3.2.2 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 $O(\log N)$ and no more than $O(N \log N)$ candidates can be added into H_0, giving an overall time complexity of $O(N \log^2 N)$ for the down list construction.

CHAPTER 4
EXTENDED REGULAR EXPRESSION PATTERN MATCHING

The operations which set extended regular expressions apart from simple regular expressions have been around for quite some time, however few solutions have appeared that solve the range of exact and approximate pattern matching problems. In particular, none of the previous algorithms fit into the state machine, alignment-graph/dynamic-programming framework which characterizes the solutions to pattern matching of sequences and regular expressions. Although it appears that the $O(M^3N)$ worst-case complexity of exact matching cannot be improved upon, except possibly by a theoretically better but impractical $O(M^{2.81})$ matrix multiplication reduction, Section 4.1 presents the construction of an *extended NFA* or ENFA which accepts extended regular expressions and builds on the NFA state machine construction and simulation for regular expressions. Section 4.2 then goes on to consider the approximate matching of extended regular expressions. As is discussed in that section, the formal notion of an approximate match must be redefined, as the traditional definition of an optimal alignment does not yield an efficient algorithm. Under this new *recursive* match definition, where matches between any A and P are based on the matches to substrings of A and sub-expressions of P, an alignment-graph/dynamic-programming style algorithm using ENFA's can solve ERE(A,P,$\{\delta\}$) in $O(M^3N)$ time.

4.1 Exact ERE Matching

For any extended regular expression P, an extended NFA or ENFA F accepting $L(P)$ can be constructed using an extension of the NFA construction. The construction algorithm employs the same inductive rules as the NFA construction (Figure 2.4), but with additional rules in Figure 4.1 for the intersection and difference operators. The formal definition of F consists of an nine-tuple $<V, V_{re}, V_\theta, V_\&, V-, E, \lambda, \theta, \phi>$ where V, E, λ, θ and ϕ are as defined for NFA's. The new state sets V_{re}, V_θ, $V_\&$ and V_- form a partition of the complete state set V. V_{re} contains all of the states introduced using the regular expression construction rules, V_θ contains the set of $\theta_{R\&S}$ and θ_{R-S} states, $V_\&$ contains the set of $\phi_{R\&S}$ states, and V_- contains the ϕ_{R-S} states. The states of the last three sets must be distinguished from the states in V_{re}, because different computations will be required at those states.

As with regular expressions and NFA's, the language accepted at $s \in V$, $L_F(s)$ is the set of sequences spelled on all "paths" from θ to s, and $L_F(\phi)$ defines the language accepted by F. However, a new definition of a "path" is required, more than simply any sequence of edges through F, to maintain the equivalence between $L_F(\phi)$ and $L(P)$. This new definition is recursive, using the following cases (where the quoted "path" refers to the new definition):

1. Any sequence of edges in F which does not pass through both the start and final state of an $F_{R\&S}$ or F_{R-S} sub-automaton is considered a "path". Thus, when no such sub-automata

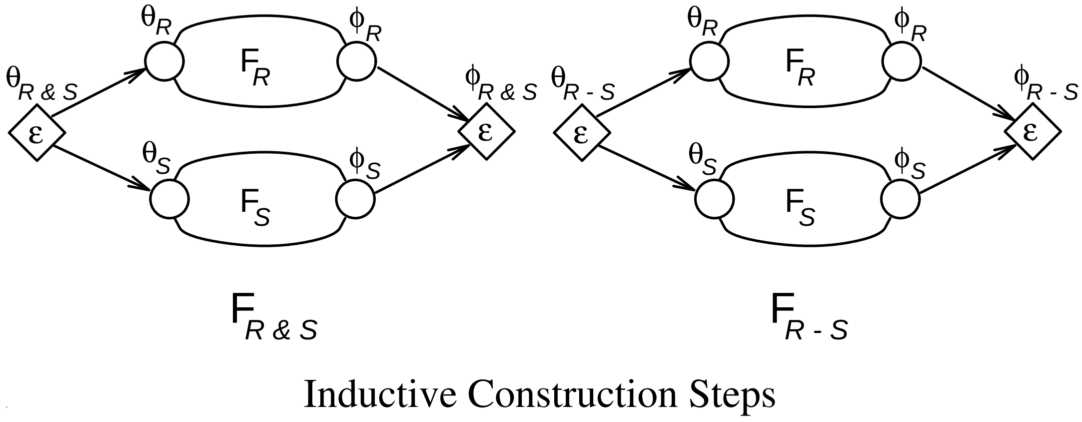


Figure 4.1: Constructing the ENFA F for an extended regular expression R.

occur in F, the new "paths" through F are simply the old paths through F as defined for NFA's. Note that this case includes sequence of edges which pass through $F_{R\&S}$ and F_{R-S} start states but do not pass through the corresponding final states.

2. For a sub-graph $F_{R\&S}$, a "path" from $\theta_{R\&S}$ to $\phi_{R\&S}$ consists of a pair of "paths," $\theta_{R\&S} \to \theta_R \xrightarrow{*} \phi_R \to \phi_{R\&S}$ and $\theta_{R\&S} \to \theta_S \xrightarrow{*} \phi_S \to \phi_{R\&S}$, which spell the same sequence. $L_{F_{R\&S}}(\phi_{R\&S})$ is simply the set of sequences for which these "path" pairs exist. This is equivalent to the language restriction that a sequence in $L(R \,\&\, S)$ must occur in both $L(R)$ and $L(S)$.

3. For a sub-graph F_{R-S}, the "paths" through F_{R-S} are the "paths" $\theta_{R-S} \to \theta_R \xrightarrow{*} \phi_R \to \phi_{R-S}$ for which no "path" spelling the same sequence exists through F_S. This satisfies the language restriction that a sequence in $L(R - S)$ must be in $L(R)$ but not in $L(S)$.

4. Finally, in general, the "paths" from θ to a state s consist of the sequence of edges outside any nested $F_{R\&S}$ or F_{R-S} sub-automaton combined with the "paths" through those nested $F_{R\&S}$ and F_{R-S} sub-automata.

$L_F(s)$, then, is the set of sequences spelled on "paths" from θ to s, and $L_F(\phi)$ is the language accepted by F. From this point on, we drop the quotes from the term *path*, and so path now refers to this new recursive definition. Also, let the phrase *sub-machines in F* denote the set of $F_{R\&S}$ and F_{R-S} sub-automata occurring in F.

Additional computation, beyond the NFA state simulation, is required to support this new definition of a path. Given an input string $A = a_1 a_2 \ldots a_M$, the recurrences below define the state simulation computation at position i and state s. They satisfy the new path definition by maintaining *partial path* information for the sub-machines in F. A partial path for a sub-machine is a path which passes through the sub-machine's start state and ends at a state "inside" the sub-machine. This information takes the form of the first positions, in A, of the substrings of A being spelled on the partial paths from each $\theta_{R\&S}$ and θ_{R-S} to s. Thus, as the simulation progresses and partial paths are extended to include a $\phi_{R\&S}$ [ϕ_{R-S}] state, only those pairs of paths which spell the same sequence from $\theta_{R\&S}$ [θ_{R-S}] through F_R and [not] through F_S are extended. Because the state simulation is solving the language acceptance problem, the sequence being spelled on all paths in the simulation is A. Thus, it suffices to extend those pairs of paths whose first position values for the corresponding $\theta_{R\&S}$ [θ_{R-S}] state are equal.

Different recurrences are defined for different states, based on the five-part partition of V. The recurrences for the states in V_{re} compute $S_{i,s}$ sets for each position i in $[0..N]$ and state $s \in V_{re}$, as follows:

(1) $s = \theta$:

$$S_{i,s} = \begin{cases} \{0\} & \text{if } i = 0 \\ \emptyset & \text{otherwise} \end{cases}$$

(2) $s \in V_{re} - \{\theta\}$:

$$S_{i,s} = \begin{cases} \bigcup\{S_{i,t} \mid t \to s\} & \text{if } \lambda_s = \varepsilon \\ \bigcup\{S_{i-1,t} \mid t \to s\} & \text{if } \lambda_s \neq \varepsilon,\ i > 0 \text{ and } \lambda_s = a_i \\ \emptyset & \text{if } \lambda_s \neq \varepsilon \text{ and either } i = 0 \text{ or } \lambda_s \neq a_i \end{cases}$$

These sets contain the beginning positions k, for $0 \leq k < i$, of matches between $a_{k+1}a_{k+2}\ldots a_i$ and the partial path through the *innermost enclosing* sub-machine of s. The innermost enclosing sub-machine of a state s is the most deeply nested $F_{R\&S}$ or F_{R-S} sub-automaton for which $s \in V_{R\&S}$ (or $s \in V_{R-S}$).

The partial path information for the other sub-machines enclosing a state s are kept in a series of *mapping tables* associated with each state in V_θ. These tables are used to perform the mapping between the valid paths in $\theta_{R\&S}$'s or θ_{R-S}'s sub-machine and the valid paths of their innermost enclosing sub-machine.

(3) $s \in V_\theta$:

$$T_{i,s} = \begin{cases} \bigcup\{S_{i,t} \mid t \to s\} & \text{if } \lambda_s = \varepsilon \\ \bigcup\{S_{i-1,t} \mid t \to s\} & \text{if } \lambda_s \neq \varepsilon,\ i > 0 \text{ and } \lambda_s = a_i \\ \emptyset & \text{if } \lambda_s \neq \varepsilon \text{ and either } i = 0 \text{ or } \lambda_s \neq a_i \end{cases}$$

$$S_{i,s} = \begin{cases} \{i\} & \text{if } T_{i,s} \neq \emptyset \\ \emptyset & \text{if } T_{i,s} = \emptyset \end{cases}$$

The $T_{i,s}$ values which make up the mapping table for s collect and store the first positions k of the matches between $a_{k+1}a_{k+2}\ldots a_i$ and the partial path from the innermost enclosing sub-machine's start state to s. These tables are retained throughout the computation and are used at the corresponding final states to extend the valid paths through the sub-machine. The value of $S_{i,s}$ here injects a new first position into the matching of the start state's sub-machine. Its value is either empty or contains the single position i, depending on whether a match between a prefix of $a_{i+1}a_{i+2}\ldots a_N$ and the sub-machine could result in an overall match between A and F.

The recurrences for $V_\&$ and V_-, the sub-machine final states, first compute the valid paths through the $F_{R\&S}$ or F_{R-S} sub-machine and then perform the mapping back to the valid paths through their innermost enclosing sub-machine. At a state $s \in V_\&$ [or V_-], the recurrence determines those first positions k for which paths exist through F_R and F_S [or through F_R but not through F_S] spelling $a_{k+1}a_{k+2}\ldots a_i$. It then unions the $T_{k,t}$ sets, where t is the start state $\theta_{R\&S}$ [θ_{R-S}] corresponding to s. This extends the partial path information for the enclosing sub-machine. Thus, the partial path information for $\theta_{R\&S}$ $[\theta_{R-S}]$ is extended to $\phi_{R\&S}$ $[\phi_{R-S}]$ using the valid paths through $F_{R\&S}$ $[F_{R-S}]$.

(4) $s \in V_{\&}$:
$$S_{i,s} = \bigcup \{T_{k,t} \mid k \in S_{i,t1} \; \& \; k \in S_{i,t2}\}$$
for $t1 \to s$, $t2 \to s$ and $t = \theta_{R\&S}$ corresponding to $s = \phi_{R\&S}$.

(5) $s \in V_{-}$:
$$S_{i,s} = \bigcup \{T_{k,t} \mid k \in S_{i,t1} \; \& \; k \notin S_{i,t2}\}$$
for $t1 \to s$, $t2 \to s$ and $t = \theta_{R-S}$ corresponding to $s = \phi_{R-S}$.

Finally, the simulation accepts if $S_{N,\phi} = \{0\}$ and rejects if $S_{N,\phi} = \emptyset$.

The actual algorithm performing the state simulation uses the same two-sweep technique described in Section 2.2 for regular expressions, taking the union of the sets computed during the two sweeps. Since F is reducible, the same arguments given for the regular expression matching hold here. For an input string A of size M and an extended regular expression P of size N, the time complexity for this algorithm is $O(M^3N)$ in the worst case. Each of the $S_{i,s}$ and $T_{i,s}$ sets can be of size $O(M)$, and the computation at the $V_{\&}$ and V_{-} states merges $|S_{i,s}|$ of the $T_{i,s}$ sets. So, $O(M^2N)$ time may be required for each text position i, and $O(M^3N)$ time overall.

However, this worst-case complexity does not hold for many patterns, and a tighter, match-sensitive complexity can be derived. First, isolate each intersection and difference sub-expression of P, and let I denote the size of the largest set of substrings of A which match one of those sub-expressions. Let L denote the length of the longest substring of A which matches any prefix of a string in $L(P)$. The running time of the algorithm above can be bounded by $O((M+I)NL)$, because the size of the $S_{i,s}$ and $T_{i,s}$ sets is bounded by L, and $O(IL)$ bounds the time needed to union the $T_{i,\theta_{R\&S}}$ or $T_{i,\theta_{R-S}}$ sets at each $\phi_{R\&S}$ or ϕ_{R-S} state. Note that for an extended regular expression containing no Kleene closure operations, the values of I and L are limited to $O(MN)$ and $O(N)$ respectively, giving a complexity bound of $O(MN^3)$.

4.2 Approximate ERE Matching

As with symbol-based approximate matching of sequences and regular expressions, ERE(A, P, $\{\delta\}$) involves finding the best "match" between sequence $A = a_1a_2 \ldots a_M$ and pattern P, using function δ to score symbol pairs and unaligned symbols. However, the *set-theoretic* definition of an approximate match used for sequences and regular expressions does not lend itself to an efficient algorithm for extended regular expressions. Under the set-theoretic definition, an approximate match between A and P is an optimal alignment between A and a specific sequence $B \in L(P)$. When P contains intersection and difference sub-expressions, the matches to those sub-expressions are required to use a common sequence from $L(R) \cap L(S)$ or $L(R) \cap \overline{L(S)}$. In a worst-case example, the expression $aba \; \& \; acb$ could never match a sequence since language it represents is empty, despite the fact that the two strings "aba" and "acb" are each only one insertion from a common string "acba."

This commonality restriction does not fit well with the "bottom-up" style of computations presented in Section 2.3 and the previous section. In those algorithms, the matches to the R and S sub-expressions are computed separately, then those matches are used to derive the matches for the intersection or difference sub-expression. The restriction that a common sequence from $L(R)$ and $L(S)$ must be used for the intersection and difference matches means that the bottom-up algorithms cannot be extended to handle the approximate matching case, as was done for sequence and regular expression patterns. In fact, the only solutions to the approximate matching problem under the set-theoretic definition appear to include either a cross-product technique, i.e. taking the cross-product of the state sets as in [Brz64], or a generative technique where the matches to the sub-expressions are generated in order of score. Both of these techniques contain an exponential time factor.

Under a slightly less restrictive, *recursive* definition of an approximate match, where the substrings of A match the sub-expressions of P without requiring a common alignment to a single sequence, a polynomial algorithm exists and is presented below. This recursive definition is embodied formally in the following function, *score (w, R)*, which returns the score of the best match between sequence $w = w_1 w_2 \dots w_k$ and extended regular expression R. It uses the following rules:

1. If $R \equiv \varepsilon$, then *score* $(w, \varepsilon) = \sum_{i=1}^{k} \delta(w_i, \varepsilon)$
2. If $R \equiv a$, then *score* $(w, a) = \begin{cases} \delta(w, a) & \text{if } |w| \leq 1 \\ \infty & \text{otherwise} \end{cases}$
3. If $P \equiv R\,S$, then *score* $(w, R\,S) =$
   $\min_{0 \leq i \leq k} \{\textit{score}\,(w_1 w_2 \dots w_i, R) + \textit{score}\,(w_{i+1} w_{i+2} \dots w_k, S)\}$.
4. If $P \equiv R \mid S$, then *score* $(w, R \mid S) = \min\{\textit{score}\,(w, R),\ \textit{score}\,(w, S)\}$.
5. If $P \equiv R^*$, then *score* $(w, R^*) =$
   $\begin{cases} 0 & \text{if } |w| = 0 \\ \min_{0 < i \leq k} \{\textit{score}\,(w_1 w_2 \dots w_i, R) + \textit{score}\,(w_{i+1} w_{i+2} \dots w_k, R^*)\} & \text{if } i \neq j \end{cases}$
6. If $P \equiv R \,\&\, S$, then *score* $(w, R \,\&\, S) = \mathcal{F}_{R\&S}(\textit{score}\,(w, R),\ \textit{score}\,(w, S))$
   where $\mathcal{F}_{R\&S}$ can be a general function (see below).
7. If $P \equiv R - S$, then *score* $(w, R - S) = \mathcal{F}_{R-S}(\textit{score}\,(w, R),\ \textit{score}\,(w, S))$
   where $\mathcal{F}_{R-S}$ can be a general function (see below).

For sequence and regular expressions, this definition coincides with the set-theoretic definition, since both compute the minimum sum of the paired symbols and unaligned symbols. However, the rules for the intersection and difference sub-expressions relax the set-theoretic requirement that the matches to R and S be made with a common sequence B in $L(R) \cap L(S)$. Here, the matches to sub-expressions R and S can occur separately and to possibly distinct sequences (or, given nested operators, to whole sets of sequences formed by the multiple matches to the nested sub-expressions).

In order to more closely capture the spirit of "intersection" and "difference" under this recursive definition, the rules also allow general functions $\mathcal{F}_{R\&S}$ and $\mathcal{F}_{R-S}$ to determine the best match of an intersection or difference sub-expression. More complex scoring methods are needed because using the "minimal sum of scores" criterion of optimality, where $F_{R\&S}$ and F_{R-S} are defined as the $\min$ function, reduces the intersection and difference operations to that of alternation. For the intersection operation, scoring schemes such as taking the maximal, average, or sum of the scores can give a more realistic score of the match to $R \,\&\, S$, under the assumption that finite or thresholded scores for both R and S exist. For an expression $R - S$, the scoring scheme which most preserves the essence of the difference operation is the following function:

$$\mathcal{F}_{R-S}(x,y) = \begin{cases} x & \text{if } y > t \\ \infty & \text{if } y \le t \end{cases}$$

This function returns the score of the match to R if the score of S's match is above a defined threshold t, and otherwise returns infinity. The general functions allowed here permit a wide range of scoring schemes, and in particular include all of the examples cited here as well as the traditional "minimal sum of scores."

The algorithm solving ERE(A, P, $\{\delta\}$), under the recursive match definition, combines the alignment-graph/dynamic programming framework with the ENFA's described in the previous section. The alignment graph is formed from $M+1$ copies of the ENFA F constructed from P. Substitution, insertion and deletion edges are added just as in the approximate regular expression problem with substitution edges from $(i-1,t)$ to (i,s), for each $t \to s$, deletion edges from $(i-1,s)$ to (i,s) and insertion edges from (i,t) to (i,s) for $t \to s$. The values computed by the recurrences consist of a pair $<c,k>$ combining the best score c with the ENFA first position information k. When such a pair occurs in the data computed at vertex (i,s), it specifies that the best match between $a_{k+1}a_{k+2}\ldots a_i$ and a partial path from s's innermost enclosing sub-machine to s has a score of c. The recurrences themselves are very similar to the exact matching case. Beginning with the states in V_{re},

1) $s = \theta$

$$C_{0,\theta} = \{\langle 0,0\rangle\}$$
$$C_{i,\theta} = \{\langle c+\delta(a_i,\varepsilon),0\rangle \mid \langle c,0\rangle \in C_{i-1,\theta}\}$$

2) $s \in V_{re} - \{\theta\}$

$$\begin{aligned} C_{i,s} = \min\{\ & \{\langle c+\delta(a_i,\lambda_s),k\rangle \mid \exists\, t \to s \text{ s.t. } \langle c,k\rangle \in C_{i-1,t}\} \\ & \{\langle c+\delta(a_i,\varepsilon),k\rangle \mid \langle c,k\rangle \in C_{i-1,s}\} \\ & \{\langle c+\delta(\varepsilon,\lambda_s),k\rangle \mid \exists\, t \to s \text{ s.t. } \langle c,k\rangle \in C_{i,t}\}\ \} \end{aligned}$$

Operation "min" here and below is a element-wise, set minimum operation which picks the minimum scoring pair for each position k appearing in the set of pairs, i.e. $\{\langle c,k\rangle \mid \nexists \langle c',k'\rangle : k' = k \;\&\; c' < c\}$.

For the three state sets introduced by the intersection and difference operators, V_θ, $V_\&$ and V_-, the recurrences are as follows:

3) $s \in V_\theta$

$$\begin{aligned} T_{i,s} = \min\{\ & \{\langle c+\delta(a_i,\lambda_s),k\rangle \mid \exists\, t \to s \text{ s.t. } \langle c,k\rangle \in C_{i-1,t}\} \\ & \{\langle c+\delta(a_i,\varepsilon),k\rangle \mid \langle c,k\rangle \in T_{i-1,s}\} \\ & \{\langle c+\delta(\varepsilon,\lambda_s),k\rangle \mid \exists\, t \to s \text{ s.t. } \langle c,k\rangle \in C_{i,t}\}\ \} \end{aligned}$$

$$C_{i,s} = \begin{cases} \langle 0,i\rangle & \text{if } T_{i,s} \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

4) $s \in V_\& \cup V_-$

$$\begin{aligned} PRED_{t1} = \min\{\ & \{\langle c+\delta(a_i,\lambda_s),\ k\rangle \mid \langle c,k\rangle \in C_{i-1,t1}\} \\ & \{\langle c+\delta(\varepsilon,\lambda_s),\ k\rangle \mid \langle c,k\rangle \in C_{i,t1}\}\ \} \\ PRED_{t2} = \min\{\ & \{\langle c+\delta(a_i,\lambda_s),\ k\rangle \mid \langle c,k\rangle \in C_{i-1,t2}\} \end{aligned}$$

$$\{\langle c + \delta(\varepsilon, \lambda_s),\ k\rangle \mid \langle c, k\rangle \in C_{i,t2}\}\ \}$$

$$\begin{aligned} C_{i,s} = \min\{\ &\{\langle c, k\rangle \mid \exists k', c' : \langle c1, k'\rangle \in PRED_{t1}\ \&\ \langle c2, k'\rangle \in PRED_{t2}\ \& \\ &\qquad\qquad \langle c', k\rangle \in T_{k',t}\ \&\ c = c' + \mathcal{F}\,(c1, c2)\}, \\ &\{\langle c + \delta(a_i, \varepsilon), k\rangle \mid \langle c, k\rangle \in C_{i-1,s}\}\ \} \end{aligned}$$

where $\mathcal{F}$ here generically refers to the $\mathcal{F}_{R\&S}$ or $\mathcal{F}_{R-S}$ function defined for each intersection and difference sub-expression. These recurrence assume that all possible substrings/sub-expressions can be aligned with some score, so that the same recurrence can be used for both the intersection and difference sub-automata. In practice however, threshold values typically limit the alignments between substrings and sub-expressions, so the recurrence above must be changed for the states in V_- to handle the case of non-existent, infinite scoring pairs.

As in the regular expression case, the alignment graph is reducible along each column, so the two-sweep, node-listing algorithm of Myers and Miller can be used to correctly compute the recurrences. The complexity of the algorithm is $O(M^3N)$, since each $C_{i,s}$ and $T_{i,s}$ set can contain $O(M)$ pairs and the $V_\&$ and V_- recurrences merge $|C_{i,s}|$ of the $T_{i,s}$ sets. Note that the worst-case complexity is also the expected complexity, unless a threshold cutoff is used to eliminate some of the substring/sub-expression matches.

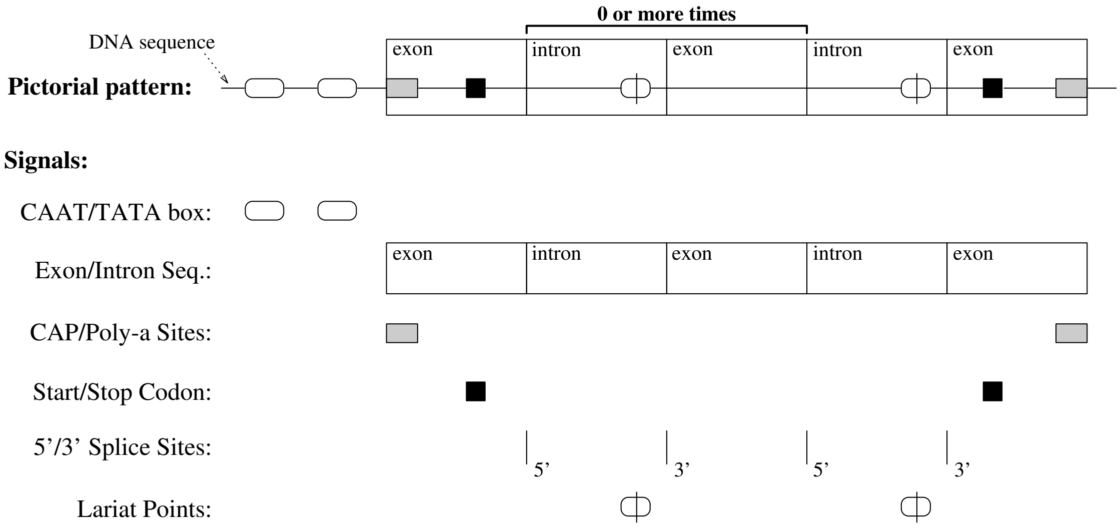
CHAPTER 5
SUPER-PATTERN MATCHING: INTRODUCTION

Super-pattern matching forms a domain of discrete pattern matching, akin to that of approximate pattern matching over sequences, where the input consists not of a sequence and a pattern of symbols, but of (1) a finite number of types of *features*, (2) for each feature, a *set of intervals* identifying the substrings of an underlying sequence having the feature, and (3) a *super-pattern* that is a pattern of features types. The objective is to find a sequence of adjacent feature intervals over the underlying sequence such that the corresponding sequence of feature types matches the super-pattern. The string spanned by the sequence of feature intervals is then identified as a match to the super-pattern. Such meta-pattern problems, i.e. a pattern of patterns, have traditionally been categorized in the realm of artificial intelligence and been solved using AI techniques such as backtracking and branch-and-bound search. Super-pattern matching's characterization is such that the dynamic programming techniques of discrete pattern matching can be used to derive practically efficient and worst-case polynomial time algorithms.

The concepts behind super-pattern matching were originally motivated by the gene recognition problem, now of great importance to molecular biologists because of the advent of rapid DNA sequencing methods. The problem is to find regions of newly sequenced DNA that code for protein or RNA products, and is basically a pattern recognition problem over the four letter alphabet $\{a, c, g, t\}$. Molecular biologists [Ld90] have developed an initial picture of the gene encoding structure, illustrated in Figure 5.1. Such a region consists of a collection of major features or "signals," constrained to be in certain positional relationships with each other. An important aspect is that the features are not linearly ordered, but frequently coincide or overlap each other. Referring to the figure, a sequence of *exons* and *introns* form the main components of a gene encoding. It is the sequence appearing in the exons, between the start and stop codons, that actually encodes the relevant protein or RNA structure. The introns, whose function is not currently known, are interspersed regions which do not contribute directly to the gene product. Overlapping these major components are smaller signals which (1) distinguish exon/intron boundaries (3' and 5' splice sites), (2) determine endpoints of the actual gene sequence (the start and stop codons) or the encoding structure (the CAAT and TATA boxes and POLY-A site), and (3) play significant roles in gene transcription (the lariat points). This view is by no means a complete description, and is still developing as biologists learn more.

At the current time, much work has been done on building recognizers for individual features using, for example, regular expressions [AWM$^+$84], consensus matrices [Sto88], and neural nets [LBB$^+$89]. Libraries of these component recognizers are currently being used to recognize either pieces of gene encodings or complete encodings. One gene recognition system, GM [FS90], uses eighteen "modules" in its gene recognition procedure.

Less work has been done on integrating these subrecognizers into an overall gene recognizer. The current methods involve hand coded search procedures [FS90], backtracking tree-search algorithms [GKDS92], and context-sensitive, definite clause grammars [Sea89]. These techniques



Pattern Expression:

CAAT <0,300> TATA <0,30> (Exon & (CAP <0,*> Start <0,*> 5'ss))
[(Intron & (5'ss <0,*> Lariat <12,40> 3'ss)) (Exon & (3'ss <0,*> 5'ss))]*
(Intron & (5'ss <0,*> Lariat <12,40> 3'ss)) (Exon & (3'ss <0,*> Stop <0,*> Poly-A))

Figure 5.1: Basic gene encoding structure.

either lack sufficient expressiveness or contain potentially exponential computations. Super-pattern matching attempts to provide the expressiveness needed to search for these patterns while keeping within polynomial time bounds in the worst case and performing efficiently in practice.

This multi-step approach to pattern matching has also appeared for such problems as protein structure prediction [LWS87] and on-line character recognition [FCK$^+$91]. In general terms, the matching procedure forms a *recognition hierarchy*, as depicted in Figure 5.2, where successively larger "patterns" are matched at higher and higher levels in the hierarchy. Super-pattern matching characterizes an isolated recognition problem in such a general recognition hierarchy and is defined in such a way as to facilitate the construction of multi-problem, super-pattern matching, recognition hierarchies.

This chapter introduces the formal concept of super-pattern matching and defines the domain of super-pattern matching problems. Section 5.1 presents the basic super-pattern matching problem, with several variations using different super-pattern expressions and output requirements. Section 5.2 then expands this basic problem into a problem class through a series of extensions, ranging from allowing flexible matches using spacing specifications to introducing scoring mechanisms and the notion of an approximate match. The next chapter then develops a matching-graph/dynamic-programming framework, similar to the alignment-graph/dynamic-programming framework of Chapters 2, 3 and 4, which characterizes all of the algorithmic solutions.

5.1 Basic Problem

The input to a basic super-pattern matching problem consists of the following:

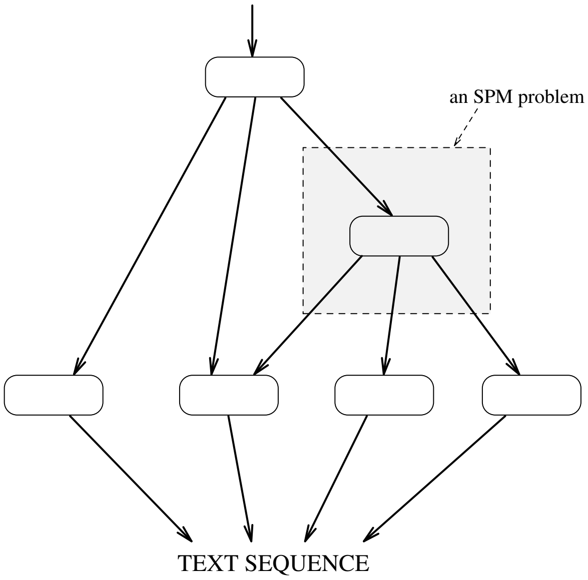


Figure 5.2: Pictorial description of a recognition hierarchy.

- A one-dimensional space, $[0..N]$.
- An alphabet $\Sigma = \{a, b, c, \ldots\}$ of *interval types*.
- An *interval set* I_a for each interval type $a \in \Sigma$. I_a is some subset of $\{[i, j] \mid 0 \leq i \leq j \leq N\}$.
- *Super-pattern* P. A sequence, regular expression or extended regular expression defined over Σ.

For a substring search solving a gene recognition problem, the one-dimensional space represents the underlying DNA sequence $A = a_1 a_2 \ldots a_N$, and the interval types in Σ are names identifying the recognizers providing input to the super-pattern search. Each of the recognizers constructs an interval set consisting of the intervals $[i, j]$ that correspond to the recognized substrings $a_{i+1} a_{i+2} \ldots a_j$. Finally, the super-pattern describes the gene encoding structure using the interval types identifying the recognizers.

The actual matching occurs between the sub-intervals of the one-dimensional space and the sub-expressions of the super-pattern. A set of *recursive* matching rules, similar to those in Section 4.2 on approximate extended regular expression pattern matching, defines the intervals matching an expression P in terms of the matches to P's sub-expressions. Formally, an interval $[i, j]$ *matches* P if and only if

1. If $P \equiv a$ where $a \in \Sigma$, then $[i, j] \in I_a$.
2. If $P \equiv \varepsilon$, then $i = j$.
3. If $P \equiv R\,S$, then $\exists\, i \leq k \leq j : [i, k]$ matches R and $[k, j]$ matches S.
4. If $P \equiv R \mid S$, then $[i, j]$ matches R or $[i, j]$ matches S.

5. If $P \equiv R^*$, then $i = j$ or $\exists\, i < k \leq j : [i, k]$ matches R and $[k, j]$ matches R^*.

6. If $P \equiv R \,\&\, S$, then $[i, j]$ matches R and $[i, j]$ matches S.

7. If $P \equiv R - S$, then $[i, j]$ matches R, but does not match S.

The intervals matching an expression P are called the *matching intervals* of P. The set of input intervals used in the match between interval $[i, j]$ and P is called the *interval sequence* matching $[i, j]$ and P.

One reason for defining the matches recursively is that the intersection and difference operators again cause computational problems for the set-theoretic definition of a match, where an interval sequence $<[i, i_1], [i_1, i_2], \ldots, [i_k, j]>$ aligns with a sequence $B = b_1 b_2 \ldots b_{k+1}$ in $L(P)$ using rule 1 above. It's an open question under the set-theoretic definition whether an algorithm exists which is polynomial in the size of the super-pattern, whereas the next chapter presents such a polynomial solution under the recursive definition.

However, a more practical reason exists with super-pattern matching for using the recursive definition. Under that match definition, the intersection and difference operators provide a natural method for specifying overlapping signals, one not permitted under the set-theoretic definition. For example, given a super-pattern $ABA \,\&\, AC$ and intervals $[0, 10], [40, 50] \in I_A$, $[10, 40] \in I_B$ and $[10, 50] \in I_C$, the interval $[0, 50]$ matches both ABA and AC and so can be reported as a match to $ABA \,\&\, AC$ under the recursive definition. The set-theoretic definition of a match does not permit this, since the language described by $ABA \,\&\, AC$ contains no common sequences. Figure 5.1 presents a more potent example of using this recursively-defined intersection operator to describe overlapping patterns of signals. Thus, this recursive definition results in a much more expressive and useful form of intersection and difference, while retaining an efficient solution.

The default type of output for the basic problem reports the matching intervals to the super-pattern. With this output, hierarchical recognition problems can be constructed by connecting the inputs and outputs of isolated super-pattern matching problems. Oftentimes however, different types of output are desired for truly isolated problems, particularly when the output can affect the complexity of the algorithms. We consider four levels of output, characterized by the following four problems. The output to the *decision* problem consists of a yes or no answer as to whether any interval in $[0..N]$ matches the super-pattern. In the *optimization* problem, the output reports the matching interval which best fits some criteria, such as longest, shortest or best scoring interval. The *scanning* problem requires the optimal matching intervals ending at each position j, for $0 \leq j \leq N$. Finally, the *instantiation* problem asks for the complete set of matching intervals.

5.2 Problem Domain

The domain of super-pattern matching problems extends from the basic problem in a number of directions, two of which have already been discussed (varying the super-pattern and required output). The other extensions introduce a positional flexibility in the interval matching and account for errors occurring in the input. Specifically, the five extensions are 1) *explicit spacing* in the super-pattern to model context free substrings occuring between recognized signals, 2) *implicit spacing* associated with input intervals which corrects for errors in the reported endpoints, 3) *interval scores* to represent significance levels of the lower-level recognition, 4) *repair intervals* used to construct interval sequences in the presence of errors allowing intervals to be missed and 5) *affine scoring schemes* to more realistically model endpoint reporting errors and missing input intervals. The rest of this section details the effect of each extension on the basic problem.

5.2.1 Explicit Spacing

Explicit spacing introduces *spacer pattern elements*, or simply spacers, into the super-pattern to model unrecognizable substrings of a certain size occurring between recognized signals. The only interesting property of these substrings is their size, and often their sole purpose is to separate the surrounding signals. One interesting example of this is the “space” of size 12 to 40 occurring in the gene encoding structure of Figure 5.1 between the lariat point and the 3’ splice site of each intron. After a copy of the DNA containing the gene has been made, each intron is then edited out of that copy. This editing process involves RNA molecules which attach at the 5’ splice site, 3’ splice site and lariat point of the copy. They then splice the intron out and connect the ends of the surrounding exons. In order to perform the splicing, the RNA molecules attached to the lariat point and 3’ splice site must also attach to each other. Since these molecules are of a certain size, the distance between attachment points on the DNA (and thus on its copy) must also be of a certain size. The specific DNA sequence appearing between the points, however, is not particularly relevant. Spacers provide a simple method for specifying these types of signals.

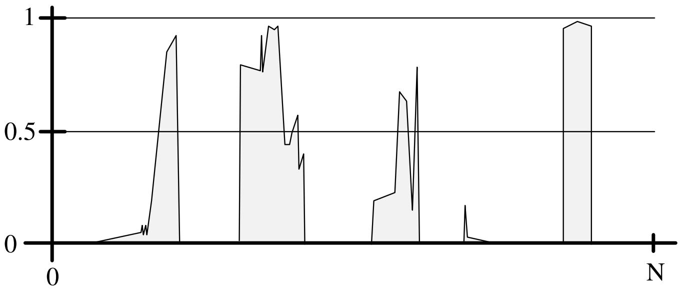
The super-pattern in Figure 5.1 illustrates the two forms of the spacer considered here, *bounded* ($<12, 40>$) and *unbounded* ($<0, *>$). Each spacer specifies a size range of matching intervals. In terms of the recursive definition, the following additional rules capture this property:

8. If $P \equiv <l, h>$, then $l \le j - i \le h$
9. If $P \equiv <l, *>$, then $l \le j - i$

We only consider spacers whose lower and upper bounds are non-negative, i.e. $l \ge 0$. Allowing the use of negative spacers such as $<-20, -5>$ involves the redefinition of an interval to include intervals of negative length, such as $[100, 97]$. The algorithms in Chapter 6 for regular expression and extended regular expression super-patterns depend heavily on the property that all intervals have a non-negative length. The introduction of negative length intervals invalidates those algorithms (which are based on the dynamic programming of Section 2.2 and Chapter 4) and requires more general path-finding algorithms to be used.

5.2.2 Implicit Spacing

Implicit spacing defines *neighborhoods* around the reported endpoints of each input interval which can be used in matches to the super-pattern. Some recognition algorithms can identify the presence or absence of a feature, but have difficulty pinpointing the exact endpoints of the feature. An example of this occurs in gene recognition. Exonic DNA regions are identified by sliding a fixed-width window along the DNA and feeding each window’s sequence to a trained neural net ([LBB$^+$89]). The raw output of this recognizer is a sequence of values, each between 0 and 1, giving a likelihood measure that the sequence in each window is an exon:



This output can be transformed into a set of intervals by thresholding the raw values and treating contiguous regions above the threshold as recognized intervals. In doing so, the general areas of exons are accurately predicted, but the endpoints of those intervals typically do not match the true ends of the exons. The use of implicit spacing, in combination with an accurate exon boundary recognizer, transforms this from an exonic region recognizer to an exon recognizer while still limiting the number of reported intervals.

We consider three types of implicit spacing, a *fixed* or *proportional* space specified for an interval type a and applied to the intervals in I_a, or a *per-interval* space reported for each input interval. Each type defines the *neighborhoods* of allowed matches around each input intervals' left and right endpoints, $<i + lmin, i + lmax>$ and $<j + rmin, j + rmax>$ for interval $[i, j]$. The fixed and per-interval spacing specify absolute $lmin$, $lmax$, $rmin$ and $rmax$ values for an interval type a or a particular input interval $[i, j]$, respectively. The proportional spacing defines two factors, $lprop_a$ and $rprop_a$ for interval type a, which are multiplied with the length of each interval in I_a to get the desired ranges.

In terms of the recursive matching rules, rule 1 (for $P \equiv a$) now becomes the following for (1) fixed, (2) proportional or (3) per-interval spacing:

1'. If $P \equiv a$, then $\exists\ [i', j'] \in I_a$ such that
  (1) $i' + lmin_a \leq i \leq i' + lmax_a$ & $j' + rmin_a \leq j \leq j' + rmax_a$
  (2) $i' - ldist \leq i \leq i' + ldist$ & $j' - rdist \leq j \leq j' + rdist$,
    where $ldist = (j' - i') * lprop_a$ and $rdist = (j' - i') * rprop_a$
  (3) $i' + lmin_{[i',j']} \leq i \leq i' + lmax_{[i',j']}$ & $j' + rmin_{[i',j']} \leq j \leq j' + rmax_{[i',j']}$

Negative values for $lmin, lmax, rmin$ and $rmax$ are permitted here with the restriction that the two neighborhoods of any input interval cannot overlap, i.e. for all $[i', j'] \in I_a$, $i' + lmax \leq j' + rmin$. The reasons for this are the same as given for negative-length explicit spacers.

5.2.3 Interval Scoring

Associating scores with input intervals provides a method for modeling errors and uncertainty at the lower-level recognizers. The scores can give a significance or likelihood measure about the validity of an interval, such as the mean neural net value occurring in each interval reported by the neural net exonic recognizer. The use of these scores changes the matching problem from one of finding matching intervals of a super-pattern to that of finding the *best scoring* matching intervals. The algorithms presented in this paper assume that all scores are non-negative and that the best scoring matching interval is the one with minimal score, except as described below for intersections and differences. They could be altered to allow negative scores and to solve maximization problems.

The recursive rules defining a match between $[i, j]$ and P now become rules in a function $score([i, j], P)$ which computes the best score of a match between $[i, j]$ and P. Specifically, $score([i, j], P)$ is

1. If $P \equiv a$, then $score\,([i, j], a) = \begin{cases} \sigma & \text{if } [i, j] \in I_a \text{ with score } \sigma \\ \infty & \text{otherwise} \end{cases}$
2. If $P \equiv \varepsilon$, then $score\,([i, j], \varepsilon) = \begin{cases} 0 & \text{if } i = j \\ \infty & \text{if } i \neq j \end{cases}$
3. If $P \equiv R\,S$, then $score\,([i, j], R\,S) = \min_{i \leq k \leq j} \{score\,([i, k], R) + score\,([k, j], S)\}$.
4. If $P \equiv R \mid S$, then $score\,([i, j], R \mid S) = \min\{score\,([i, j], R),\ score\,([i, j], S)\}$.
5. If $P \equiv R^*$, then $score\,([i, j], R^*) =$
$$\begin{cases} 0 & \text{if } i = j \\ \min_{i < k \leq j} \{score\,([i, k], R) + score\,([k, j], R^*)\} & \text{if } i \neq j \end{cases}$$
6. If $P \equiv R\,\&\,S$, $score\,([i, j], R\,\&\,S) = \mathcal{F}_{R\&S}(score\,([i, j], R),\ score\,([i, j], S))$
   where $\mathcal{F}_{R\&S}$ can be a general function (see below).
7. If $P \equiv R - S$, $score\,([i, j], R - S) = \mathcal{F}_{R-S}(score\,([i, j], R),\ score\,([i, j], S))$
   where $\mathcal{F}_{R-S}$ can be a general function (see below).

As with approximate extended regular expression pattern matching, this function computes the "minimal sum of scores" for the regular expression operations and allows general functions, such as the sum, minimum, maximum, . . ., to be defined for the intersection and difference sub-expressions.

The scoring of interval sequences changes the output requirements for the four problems described at the beginning of the chapter, as well as the specification of explicit and implicit spacing. The decision problem becomes that of reporting the best score of a matching interval, rather than the existence of a matching interval. For the other three problems, the scores of matching intervals are reported along with the intervals themselves, either the matching interval with the best score (the optimization problem) or the set of matching intervals and their best scores (the instantiation problem). The use of explicit and implicit spacing again require new rules 8 and 9 and the rewriting of rule 1, respectively. Some fixed cost $c \geq 0$ is now incorporated into those rules and either reported as the score of an explicit spacer's match or added to the score for each input interval when computing $score([i, j], a)$.

5.2.4 Repair Intervals

Repair intervals are a mechanism for inserting intervals, not appearing in the input, into the construction of interval sequences. Few recognition algorithms for complex features can correctly identify every "true" instance of that feature in a sequence. Even using interval scores to report possible matches, many recognizers are designed to achieve a balance between sensitivity and specificity. They allow a few true intervals to be missed (a sensitivity rate close to, but under, 100%) so that the number of false intervals reported does not explode (thus keeping the specificity rate high). These missed intervals, however, can disrupt the match to an entire interval sequence in the super-pattern matching problems described so far. Repair intervals are used to complete the construction of interval sequences in the face of small numbers of missing intervals.

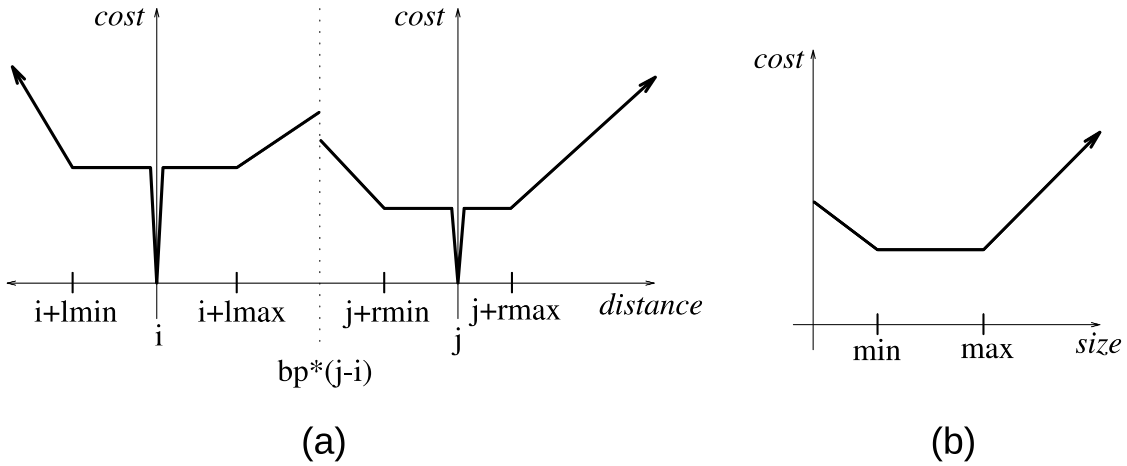


Figure 5.3: Affine scoring of a) implicit spacing for interval $[i, j]$ and b) bounded spacers/repair intervals.

A repair interval specification is given for an interval type. It consists of a non-negative size range, l to h, and a fixed cost c for using a repair interval in an interval sequence. Given such a specification for interval type a, each instance of a in the super-pattern can be matched with any interval $[i, j]$ where $l \leq j - i \leq h$. That match then contributes an additional cost c to the score of the resulting interval sequence. In terms of the recursive definition, this results in the following new rule for $P \equiv a$:

1. If $P \equiv a$, then $score([i, j], a) = \min \begin{cases} \sigma & \text{if } [i, j] \in I_a \text{ with score } \sigma \\ c & \text{if } l \leq j - i \leq h \\ \infty & \text{otherwise} \end{cases}$

assuming here that interval scores are being used and that no implicit spacing has been defined for the intervals in I_a.

5.2.5 Affine Scoring Schemes

The fixed range implicit spacing, explicit spacing and repair intervals often provide an unrealistic measure of the size distribution of endpoint errors, missing intervals and context free spaces. For some recognizers, a majority of the incorrectly reported endpoints may differ only slightly from the true endpoints, while a small but significant number are off by greater distances. At other times, no fixed bounds can be computed for either the endpoint errors or sizes of missing intervals. In these cases, a fixed cost, bounded range scoring scheme does not correctly model the distributions of sizes or error distances in the input. Affine scoring schemes for implicit spacing, explicit spacing and repair intervals are distance-based models where a match's score grows as the distance from a desired range grows, whether the distance is from a reported endpoint's position or an interval's size.

In its most complex form, the affine specification for an interval type's implicit spacing consists of the five-tuples $<lcl_a, lmin_a, lc_a, lmax_a, lcr_a>$ and $<rcl_a, rmin_a, rc_a, rmax_a, rcr_a>$ for the left and right endpoints of intervals in I_a, plus a boundary proportion bp_a used to separate the left and right endpoint neighborhoods. The graphical representation of the scoring scheme is shown in Figure 5.3a. For the left endpoint, the values of $lmin_a$ and $lmax_a$ specify a fixed size range in which the cost of implicit spacing within that neighborhood is lc_a. The values of lcl_a and lcr_a give the incremental cost for extending the implicit spacing to the left and right of the fixed space range. The right endpoint scoring is similar. The boundary between the two neighborhoods is given as a proportion on the length of each interval in I_a, and is necessary to avoid introducing negative-length intervals.

The score of a match between interval $[i, j]$ and the expression $P \equiv a$ is the minimum, over all intervals $[i', j'] \in I_a$, of the score associated with $[i', j']$ (assuming interval scores are being used), plus the cost of the implicit spacing at the two endpoints. In terms of the matching rules, this results in the following for $P \equiv a$:

$$score([i,j],a) = \min\{\textbf{\textit{left}}_{[i',j']} + \sigma + \textbf{\textit{right}}_{[i',j']} \mid [i',j'] \in I_a \text{ scores } \sigma\}$$

where

$$\textbf{\textit{left}}_{[i',j']} = \begin{cases} lc_a + lcl_a * ((i' + lmin_a) - i) & \text{if } i < i' + lmin_a \\ lc_a & \text{if } i' + lmin_a \le i < i' \\ 0 & \text{if } i = i' \\ lc_a & \text{if } i' < i \le i' + lmax_a \\ lc_a + lcr_a * (i - (i' + lmax_a)) & \text{if } i' + lmax_a < i \le i' + (j' - i') * bp_a \end{cases}$$

$$\textbf{\textit{right}}_{[i',j']} = \begin{cases} rc_a + rcl_a * ((j' + rmin_a) - j) & \text{if } i' + (j' - i') * bp_a \le j < j' + rmin_a \\ rc_a & \text{if } j' + rmin_a \le j < j' \\ 0 & \text{if } j = j' \\ rc_a & \text{if } j' < j \le j' + rmax_a \\ rc_a + rcr_a * (j - (j' + rmax_a)) & \text{if } j > j' + rmax_a \end{cases}$$

This assumes no repair interval specifications for a.

The affine specification for repair intervals and bounded spacers is a three part curve defined for non-negative size values and is shown in Figure 5.3b. Considering only the repair intervals, the numerical information consists of a similar five-tuple $<cl_a, min_a, c_a, max_a, cr_a>$ with a (now non-negative) size range min_a to max_a, fixed cost c_a for that size range, and incremental costs cl_a and cr_a for extending to the left and right of the size range. The cost of using a repair interval is computed from the size of that interval, as follows:

$$score([i,j],a) = \min \begin{cases} \sigma & \text{if } [i,j] \in I_a \text{ with score } \sigma \\ c_a & \text{if } min_a \le j - i \le max_a \\ c_a + cl_a * (min_a - (j - i)) & \text{if } j - i < min_a \\ c_a + cr_a * ((j - i) - max_a) & \text{if } j - i > max_a \end{cases}$$

where no implicit spacing is defined here. Including both affine implicit spacing and affine repair intervals simply requires combining the above two rules. The rule for computing an affine scored spacer uses the bottommost three terms of the repair interval rule.

CHAPTER 6
SUPER-PATTERN MATCHING: ALGORITHMS

The solution to each of the super-pattern matching problems defined in the previous chapter employs a matching-graph/dynamic-programming framework similar to that developed for the approximate pattern matching of Chapters 2, 3 and 4. The framework for super-pattern matching involves four major steps common for all of the algorithmic solutions. The first step constructs a state machine equivalent to the super-pattern, i.e. a machine which accepts the same language as the super-pattern expression. Second, the matching problem is recast as a graph traversal problem by constructing a *matching graph* from the state machine. The construction is such that the graph edges correspond to input intervals and paths through the graph correspond to interval sequences matching the super-pattern's sub-expressions. The third step derives dynamic programming recurrences which compute the paths (and hence the interval/sub-expression matches) to each vertex in the graph. Finally, algorithms solving these recurrences are developed.

Sections 6.1 and 6.2 present the four steps solving the scanning and instantiation problems, with interval scoring, for a sequence, regular expression and extended regular expression. The inclusion of interval scoring results in more interesting algorithms, since the graph traversal problem becomes the more complex problem of finding shortest paths through the graph, rather than just the existence of paths. The solutions to problems with no interval scoring or for the decision and optimization problems are simple variations of the algorithms presented below. Section 6.3 then presents the algorithmic extensions which solve for the explicit and implicit spacing, repair intervals and affine scoring schemes defined in Section 5.2.

6.1 Sequences and Regular Expressions

The solution to the super-pattern matching problem with a sequence or regular expression super-pattern P uses the same NFA construction as the approximate pattern matching solution described in Section 2.2. Reviewing the basic definitions, the 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 λ assigning a "label", $\lambda_s \in \Sigma \cup \{\varepsilon\}$, to each state s; (4) a designated "source" state θ; and (5) a designated "sink" state ϕ. The inductive construction is given in Figure 2.4. Intuitively, F is a vertex-labeled directed graph with distinguished source and sink vertices. A directed path through F *spells* the sequence obtained by concatenating the non-ε state labels along the path. $L_F(s)$, the *language accepted at* $s \in V$, is the set of sequences spelled on all paths from θ to s. The *language accepted by* F is $L_F(\phi)$. The key properties of this construction are that 1) for any sequence P of size M there is an acyclic NFA containing $M + 1$ states and 2) for any regular expression P of size M there is an NFA whose graph is reducible and whose size, measured in either vertices or edges, is $O(M)$.

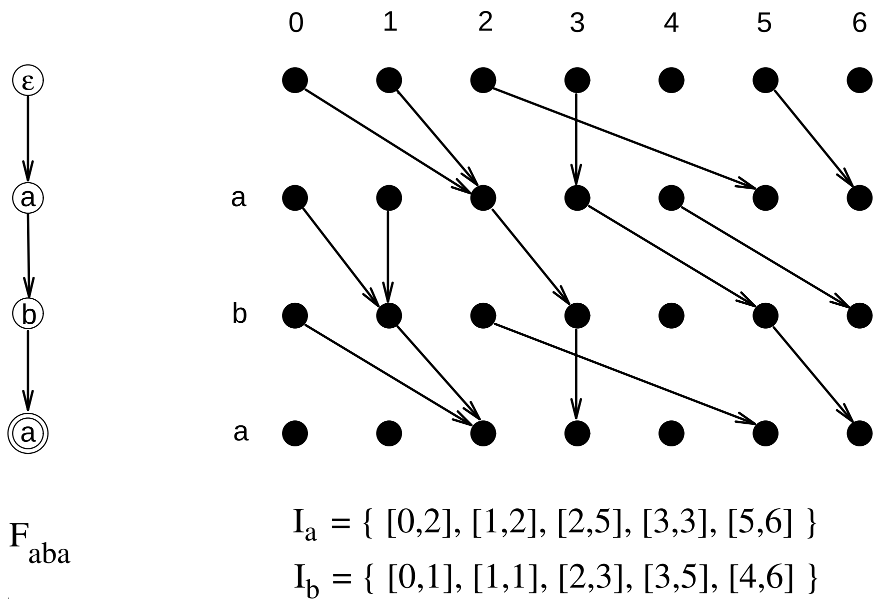


Figure 6.1: The NFA and matching graph for super-pattern $P = aba$ and $N = 6$.

The matching graphs for these super-patterns consist of $N + 1$ copies of the NFA for P, where N is the size of the one-dimensional space defined in the matching problem. Examples for a sequence and regular expression are shown in Figures 6.1 and 6.2. In this matrix-structured graph, the vertices are denoted using pairs (s, j) where $s \in V$ and $j \in [0..N]$. Weighted edges are added in a row-dependent manner, considering the vertices $(s, 0), (s, 1), \ldots, (s, N)$ as a "row." For a vertex (s, j), if the label of state s, λ_s, is some symbol $a \in \Sigma$, incoming edges are added from each vertex (t, i) where $t \rightarrow s$ is an edge in F and $[i, j] \in I_{\lambda_s}$. The weights on those edges equal the scores associated with the corresponding intervals in I_{λ_s}. This models the matches of symbol λ_s to the intervals in I_{λ_s}. When λ_s is ε, vertex (s, j) has incoming edges with weight 0 from each vertex (t, j) where $t \rightarrow s$. These edges model the match between the ε symbol and the zero-length interval $[j, j]$. A straightforward induction, using the recursive matching rules from Chapter 5.2.3, can show the correspondence between paths and matching intervals.

For the scanning problem with interval scoring, the dynamic programming recurrences compute the shortest paths from row θ to row ϕ in the graph, where the shortest path is the one whose sum of edge weights is minimal. The recurrence for sequences and regular expressions is

$$C_{\theta,j} = \langle 0, j \rangle$$
$$C_{s,j} = \begin{cases} \min\{\langle c + \sigma, k \rangle \mid t \rightarrow s \;\&\; [i, j] \in I_{\lambda_s} \text{ scores } \sigma \;\&\; \langle c, k \rangle \in C_{t,i}\} & \text{if } \lambda_s \in \Sigma \\ \min\{C_{t,j} \mid t \rightarrow s\} & \text{if } \lambda_s = \varepsilon \end{cases}$$

This recurrence finds the *position pairs* $\langle c, k \rangle$ for each vertex (s, j), such that c is the best score of a match between interval $[k, j]$ and the path in F from θ to s. The "min" operation returns the position pair with the minimal score c, breaking ties using an optimality criterion such as the smallest or largest k. Thus, the $C_{\phi,j}$ values give the score and left endpoint position for the best scoring matching interval whose right endpoint is j.

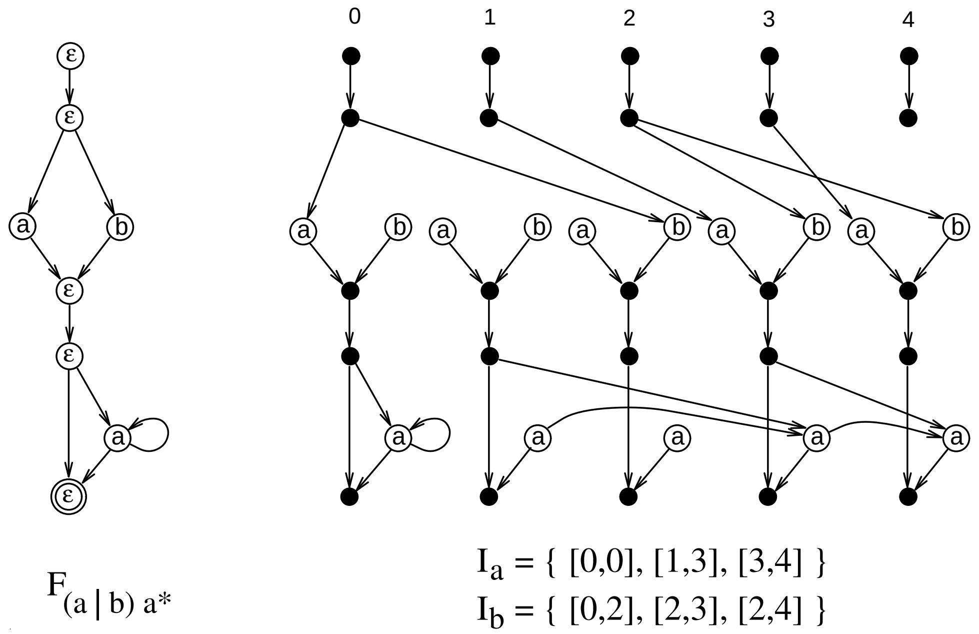


Figure 6.2: The state machine and matching graph for $P = (a \mid b)\, a^*$ and $N = 4$.

The recurrence for the instantiation problem is very similar, except that each $C_{s,j}$ value is a set of these position pairs, as follows:

$$C_{\theta,j} = \{\langle 0, j\rangle\}$$
$$C_{s,j} = \begin{cases} \bigcup_{\min}\{\langle c+\sigma, k\rangle \mid t \to s \;\&\; [i,j] \in I_{\lambda_s} \text{ scores } \sigma \;\&\; \langle c,k\rangle \in C_{t,i}\} & \text{if } \lambda_s \in \Sigma \\ \bigcup_{\min}\{C_{t,j} \mid t \to s\} & \text{if } \lambda_s = \varepsilon \end{cases}$$

The "$\bigcup_{\min}$" operation computes the best scoring match to each interval $[k, j]$ by taking the union of the minimum scoring position pairs for each position k, i.e. $\bigcup_{\min}(S) = \{\langle c, k\rangle \mid \langle c, k\rangle \in S \;\&\; \not\exists\, \langle c', k'\rangle \in S : k = k' \;\&\; c > c'\}$. The position pairs occurring in the $C_{\phi,j}$ sets identify the set of matching intervals of the super-pattern, or $\{[k, j] \text{ scoring } c \mid 0 \le j \le N \;\&\; \langle c, k\rangle \in C_{\phi,j}\}$.

A naive dynamic programming algorithm can solve the recurrences for sequence super-patterns, since the matching graph is acyclic. The decision, scanning and optimization solutions run in $O((N + I)M)$ time where I is the size of the largest I_a, because there are $O(NM)$ vertices and $O(IM)$ edges. The following solves the instantiation problem, using $0..M$ to reference states in F and [ ] to denote ordered lists:

for $j \leftarrow 0$ **to** N **do**
{ $C_{0,j} \leftarrow [\langle 0, j\rangle]$
  for $m \leftarrow 1$ **to** M **do**
  { $C_{m,j} \leftarrow [\,]$
    for $i, \sigma \leftarrow [i, j] \in I_{\lambda_m}$ scores σ **do**
      $C_{m,j} \leftarrow$ *Merge* $(C_{m,j},$ *Add* $(C_{m-1,i}, \sigma))$
  }
}

Operation *Merge* implements the $\bigcup_{\min}$ operation for two ordered lists by merging the lists according to left endpoint position k and by removing any non-optimal position pairs. *Add(L,v)* produces a new ordered list in which v is added to each position pair in L. The time complexity for this algorithm is $O((N+I)ML)$, where L is the length of the longest matching interval to any prefix of P. A more practical, and stricter, bound for L is the number of differently sized matching intervals to any prefix of P (which more closely reflects the $C_{m,j}$ sizes). But the longest matching interval bound gives a cleaner definition to the complexity measure. In the worst case where I is $O(N^2)$ and L is $O(N)$, this time complexity is $O(N^3M)$.

The regular expression algorithm is more complex as cyclic dependencies can occur in the dynamic programming recurrences, mirroring cycles in the matching graph. However, these cycles occur when the edges corresponding to zero-length input intervals and ε-labeled states link the vertices of Kleene closure sub-automata in a column's copy of F. Since no negative length intervals (or extensions resulting in negative length intervals) are permitted, these cycles can occur only down columns of the graph. Furthermore, the matching graph is acyclic except for the cycles along particular columns of the graph. Since each column of the graph has the same structure as F, the graph is reducible along each column and the two-sweep, node-listing technique from Section 2.2 can be used. The variation of that technique which solves for the instantiation problem is as follows:

for $j \leftarrow 0$ **to** N **do**
{ $C_{\theta,j} \leftarrow [\langle 0, j\rangle]$
  for $s \neq \theta$ **do**
    $C_{s,j} \leftarrow [\,]$
  for *sweep* $\leftarrow 1$ **to** 2 **do**
  { **for** s in topological order of DAG edges **do**
      if $\lambda_s \neq \varepsilon$ **then**
        for t, i, σ where $t \to s$ **and** $[i, j] \in I_{\lambda_s}$ scores σ **do**
          $C_{s,j} \leftarrow$ *Merge*$(C_{s,j},$ *Add*$(C_{t,i}, \sigma))$
      else
        for t where $t \to s$ **do**
          $C_{s,j} \leftarrow$ *Merge*$(C_{s,j}, C_{t,j})$
  }
}

Since F restricted to the DAG edges is acyclic, a topological order of the states exists. The complexity of the algorithm is $O((N + I)ML)$, since the recurrence is computed twice for each graph vertex and there are at most $2\,I\,M$ edges in the graph.

6.2 Extended Regular Expressions

For extended regular expression super-patterns, we use the extended NFA state machine, denoted as an ENFA, described in chapter 2.3 as the state machine accepting the language denoted by the super-pattern. As with the matching graphs for sequences and regular expressions, the matching graph for an extended regular expression super-pattern P consists of $N + 1$ copies of the ENFA

F constructed from P. The graph edges are added just as in the previous section for regular expressions, considering for the moment the intersection and difference sub-automata as alternation sub-automata.

The dynamic programming recurrences incorporate the details of the more complex ENFA simulation, defining different computations for each of the four subsets of states, V_{re}, V_θ, $V_\&$ and V_-. One simplifying factor in this incorporation is that the partial path information computed by the ENFA simulation is identical to the matching interval, left endpoint information computed by the instantiation problem. Thus, the recurrences use the same position pairs $\langle c, k\rangle$ to perform valid ENFA path extensions and to maintain the set of matching intervals. Because the position pairs are required by the ENFA simulation, the solutions to the decision, optimality and scanning super-pattern matching problems are essentially the same as for the instantiation problem, and so are not considered here.

Beginning with the states in V_{re}, i.e. the states introduced by regular expression operators:

$$
\begin{aligned}
C_{\theta,j} &= \{\langle 0, j\rangle\} \\
C_{s,j} &= \begin{cases} \bigcup_{\min}\{\langle c+\sigma, k\rangle \mid t \to s \;\&\; [i,j] \in I_{\lambda_s} \text{ scores } \sigma \;\&\; \langle c,k\rangle \in C_{t,i}\} & \text{if } \lambda_s \in \Sigma \\ \bigcup_{\min}\{C_{t,j} \mid t \to s\} & \text{if } \lambda_s = \varepsilon \end{cases}
\end{aligned}
$$

The computations at the intersection and difference sub-automata start states, $s \in V_\theta$, use mapping tables T_s to hold the partial path information for the enclosing sub-machine:

$$
\begin{aligned}
T_{s,j} &= \begin{cases} \bigcup_{\min}\{\langle c+\sigma, k\rangle \mid t \to s \;\&\; [i,j] \in I_{\lambda_s} \text{ scores } \sigma \;\&\; \langle c,k\rangle \in C_{t,i}\} & \text{if } \lambda_s \in \Sigma \\ \bigcup_{\min}\{C_{t,j} \mid t \to s\} & \text{if } \lambda_s = \varepsilon \end{cases} \\
C_{s,j} &= \begin{cases} \{j\} & \text{if } T_{s,j} \neq \emptyset \\ \emptyset & \text{if } T_{s,j} = \emptyset \end{cases}
\end{aligned}
$$

And the recurrences for states in $V_\&$ and V_- use those mapping tables to extend matches across each final state's sub-machine:

$$
\begin{aligned}
C_{s,j} = \bigcup_{\min}\{\langle c + \mathcal{F}_{R\&S}(c1, c2), k\rangle \mid \langle c,k\rangle \in T_{t,i} \;\&\; \langle c1, i\rangle \in C_{t1,j} \;\&\; \langle c2, i\rangle \in C_{t2,j}\} \\
\text{where } t1 \to s,\, t2 \to s \text{ and } t = \theta_{R\&S} \text{ corres. to } s = \phi_{R\&S} \\
C_{s,j} = \bigcup_{\min}\{\langle c + \mathcal{F}_{R-S}(c1, c2), k\rangle \mid \langle c,k\rangle \in T_{t,i} \;\&\; \langle c1, i\rangle \in C_{t1,j} \;\& \\
c2 = \begin{cases} c2' & \text{if } \langle c2', i\rangle \in C_{t2,j} \\ \infty & \text{if } \langle c2', i\rangle \notin C_{t2,j} \end{cases} \\
\} \\
\text{where } t1 \to s,\, t2 \to s \text{ and } t = \theta_{R-S} \text{ corres. to } s = \phi_{R-S}
\end{aligned}
$$

As with the instantiation problems for sequences and regular expressions, the values in $C_{\phi,j}$ for $0 \le j \le N$ give the left endpoints of the matching intervals of P.

The algorithm computing these recurrences is another column-based, two-sweeps per column algorithm, since the matching graph is again reducible. The time complexity is $O((N + I)ML)$, where L is defined as before but I is the size of either the largest input interval set or the largest number of different intervals matching an intersection or difference sub-expression of P.

6.3 Extension Algorithms

The four extensions described in Section 5.2, 1) explicit spacing, 2) implicit spacing, 3) repair intervals and 4) affine scoring, can be solved using extensions to the algorithms presented in the previous section. In addition, these algorithmic extensions require no major changes to the previous sections' algorithms and are independent of the super-pattern language, i.e. whether the super-pattern is a sequence, regular expression or extended regular expression. This occurs because the effects on the matching graphs from the extensions below can be thought of as "horizontal" changes along particular graph rows, whereas the algorithms of the previous section affect only the "vertical" structure along each column of the graph. Because of this fact, the extensions can be individually presented for a representative row of the matching graph. The overall algorithm for any combination of super-pattern and set of extensions is developed by starting with one of the base algorithms given in the previous two sections, and then applying the appropriate algorithmic extension to the relevant rows of the matching graph.

The descriptions that follow concentrate on the three additional algorithms used to solve these extensions. The first sub-section gives the application of a *sliding window* algorithm to bounded spacers, fixed range implicit spacing and fixed range repair intervals. The next sub-section describes a *range query tree/inverted skyline* solution for the proportional and per-interval implicit spacing. Finally, the third subsection presents a solution to the affine scoring scheme which employs *minimum envelopes* to efficiently model the contributions of the affine curves along a row. The unbounded spacer solution is not given as it can be solved using a running minimum as the overall algorithm progresses along row s where $\lambda_s = <l, *>$.

The description in each of the sub-sections isolates a particular row s of the matching graph, appropriately labeled, and solves the decision problem with interval scoring for that row. Also, it assumes that state s is labeled $\lambda_s = a$, unless otherwise noted, and has only one predecessor state t in F. The solutions to the other problems, and states with two predecessors, are straightforward variations of the algorithms below.

6.3.1 Sliding Windows

The bounded spacers, implicit spacing and repair intervals all involve the computation of values in fixed width windows, whether along the predecessor row of vertices or associated with the input intervals. Treating the bounded spacers $<l, h>$ first, the spacer is considered as an alphabet symbol in the construction of the state machine, resulting in one state $s \in V$ where $\lambda_s = <l, h>$. The edges from row t to row s in the graph connect each vertex (t, i), where $0 \leq i \leq N - l$, to the vertices $(s, i + l), (s, i + l + 1), \ldots, (s, \max\{N, i + h\})$. These edges model the match between intervals whose size is between l and h and the spacer. Looking at the incoming edges to a vertex (s, j) results in the following recurrence:

$$C_{s,j} = \min\{C_{t,i} \mid t \to s \;\&\; \max\{0, j - h\} \leq i \leq \max\{0, j - l\}\}$$

where "$\min$" here is the traditional minimum operation. From this point on, the "$\max\{0, \ldots\}$" boundary conditions are omitted and assumed in the equations and algorithms below.

Similar edges are added for fixed range repair intervals, but these edges are included in addition to the normal edges corresponding to input intervals. These new edges give the following recurrence, where the second term in the minimum reflects the repair intervals:

$$C_{s,j} = \min\{\ \min\{C_{t,i} + \sigma \mid [i,j] \in I_a \text{ scores } \sigma\},\ \min\{C_{t,k} + c_a \mid j - h_a \leq k \leq j - l_a\}\ \}$$

In each of the two cases above, the value of $C_{s,j}$ is the minimum over a window of $h - l$ $C_{t,i}$ values.

The fixed implicit spacing is more complex, because the fixed width windows are the neighborhoods occuring at both ends of each input interval. Along row s of the matching graph, the edges corresponding to each input interval $[i,j] \in I_a$ are replaced with (1) a new vertex $(s,[i,j])$ representing the interval, (2) edges connecting vertices $(t, i + lmin_a)$, $(t, i + lmin_a + 1)$, ..., $(t, i+lmax_a)$ to vertex $(s,[i,j])$ and (3) edges connecting vertex $(s,[i,j])$ to vertices $(s, j+rmin_a)$, $(s, j + rmin_a + 1)$, ..., $(s, j + rmax_a)$. The edges model the implicit spacing defined for each endpoint of $[i,j]$, and the additional vertex is needed to keep the number of edges proportional to the size of the implicit spacing. These changes result in the following two recurrences for the $C_{s,j}$ values:

$$C_{s,[i,j]} = \min\{C_{t,k} + \sigma \mid [i,j] \in I_a \text{ scores } \sigma \ \&\ i + lmin_a \leq k \leq i + lmax_a\}\}$$
$$C_{s,j} = \min\{C_{s,[i',j']} \mid [i',j'] \in I_a \ \&\ j' + rmin_a \leq j \leq j' + rmax_a\}$$

assuming no repair intervals have been specified for interval type a. These recurrences present two different algorithmic problems, the “front end” problem of computing each $C_{s,[i,j]}$ from the array of scores along row t, and the “back end” problem of computing the row of $C_{s,j}$ values as the minimum of the applicable $C_{s,[i',j']}$ values.

Naive dynamic programming algorithms computing these recurrences have a complexity of $O(NW)$ for explicit spacers or repair intervals, where $W = h-l$, and a complexity of $O((N+I)W)$ for implicit spacing, where $W = \max\{lmax_a - lmin_a + 1, rmax_a - rmin_a + 1\}$. More efficient algorithms use a “sliding window” technique for computing the sequence of minimums in $O(N)$ and $O(N + I)$ time. This technique computes a recurrence such as $D_j = \min_{j-w \leq i \leq j}\{E_i\}$ by incrementally constructing a list of indices $[i_1, i_2, \ldots, i_k]$ for each j. Index i_1 denotes the minimum value in the current window, index i_2 denotes the minimum value *to the right* of i_1, index i_3 gives the minimum to the right of i_2, and so on until i_k which always denotes the rightmost value in the window. The formal algorithm is as follows:

$L \leftarrow [\,]$
for $j \leftarrow 0$ **to** N **do**
{ **if** $L_1 < j - w$ **then**
    $L \leftarrow$ *DeleteHead* (L)
  while $|L| > 0$ **and** $E_{L_{|L|}} > E_j$ **do**
    $L \leftarrow$ *DeleteTail* (L)
  $L \leftarrow$ *Append* $(L, [j])$
  $D_j \leftarrow E_{L_1}$
}

using basic list operations *DeleteHead*, *DeleteTail* and *Append*. The list is updated as the window advances by 1) removing the head of the list if the window has slid past its value, 2) removing successive values from the tail of the list if the new value in the window is smaller and 3) inserting the new value at the tail of the list. The complexity of this is $O(N)$, since the value for each position j is inserted and deleted once from the list.

This algorithm directly applies to the explicit spacing and repair interval recurrences above, since the recurrence computing $C_{s,j}$ is simply a shifted version of the recurrence for D. The implicit spacing's front end problem can be solved by using the sliding window algorithm to precompute $\min\{C_{t,k} \mid i + lmin_a \leq k \leq i + lmax_a\}$ for each position $0 \leq i \leq N$. Then, $C_{s,[i,j]}$ equals the precomputed value at i plus the score associated with $[i, j]$. Note that the precomputed value needed by $C_{s,[i,j]}$ generally is not available when the overall algorithm is at position i, since the window for i cannot be computed until $C_{t,i+lmax_a}$ is available. But, since the implicit spacing ranges for the left and right endpoints cannot overlap, $C_{s,[i,j]}$ can be safely computed at any time between $i + lmax_a$ and $j + rmin_a$.

The application to the implicit spacing's back end problem is not as direct. In this case, there are possibly overlapping windows of size $rmax_a - rmin_a + 1$ where particular values hold, and the object is to find the minimum of the values holding at each position j. This can be solved using the data structure employed by the sliding window technique. As the overall algorithm progresses to each vertex (s, j), the values of each $C_{s,[i',j']}$ where $j' + rmin_a = j$ are inserted into the sliding window data structure. They are deleted either when dominated by another value or when $j' + rmax_a < j$. Since the neighborhoods of each $C_{s,[i',j']}$ are the same size, a dominated value can be safely removed from the list as it can never again contribute to a future $C_{s,j}$ value. With this algorithm, the value needed for each $C_{s,j}$ always appears at the head of the sliding window's list at j.

The use of this sliding window technique results in bounded spacer and repair interval computations taking $O(N)$ time per graph row and in implicit spacing computations taking $O(N + I)$ time per graph row.

6.3.2 Range Query Trees and Inverted Skylines

The sliding window algorithms cannot be applied to proportional and per-interval implicit spacing because neighborhood widths vary between the input intervals in I_a. The matching graph changes and recurrences are similar to that of fixed width spacing:

$$
\begin{aligned}
C_{s,[i,j]} &= \min\{C_{t,k} + \sigma \mid [i, j] \text{ scores } \sigma \ \&\ i + lmin \leq k \leq i + lmax\} \\
C_{s,j} &= \min\{C_{s,[i',j']} \mid [i', j'] \in I_a \ \&\ j' + rmin \leq j \leq j' + rmax\}
\end{aligned}
$$

where $lmin$, $lmax$, $rmin$ and $rmax$ henceforth generically denote the neighborhoods for the relevant input interval. Again, there are the "front end" and "back end" problems of computing $C_{s,[i,j]}$ from the values along row t and computing each $C_{s,j}$ as the minimum of the applicable $C_{s,[i',j']}$.

For the front end problem, the algorithm computing the $C_{s,[i,j]}$ values must be able to satisfy general *range queries* over the values along row t. These range queries ask for the minimum score over an arbitrary range x to y, or $\min\{C_{t,x}, C_{t,x+1}, \ldots, C_{t,y}\}$. The solution is to build a *range query tree* from the values along row t and use it to answer the queries. A range query tree is a binary tree with N leaves, corresponding to the $C_{t,i}$ values, and with additional pointers pointing up the tree. An example is depicted in Figure 6.3. Each node X in the tree contains seven values, denoted *X.l, X.h, X.value, X.left, X.right, X.lp* and *X.rp*. The first three values specify X's range and the minimum value over that range, i.e. *X.value* = $\min\{C_{t,X.l}, C_{t,X.l+1}, \ldots, C_{t,X.h}\}$. *X.left* and *X.right* point to the left and right children of X in the binary tree. *X.lp* and *X.rp* point to ancestors in the

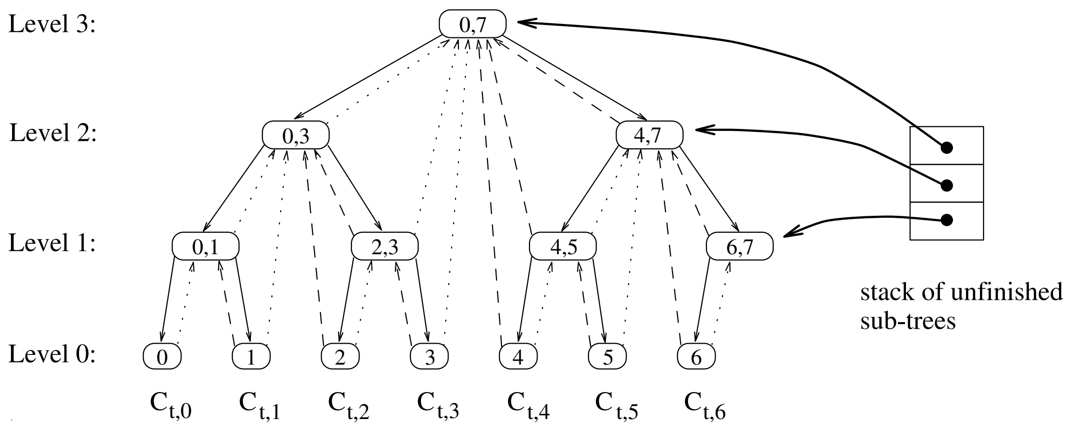


Figure 6.3: View of a partially constructed range query tree (dashed and dotted lines are *lp* and *rp* pointers).

tree. Specifically, *Y.rp* = *X* and *Y.lp* = *X.lp* for a left child Y of X, and *Z.lp* = *X* and *Z.rp* = *X.rp* for a right child Z of X.

The *lp* and *rp* pointers are used to answer the range queries x, y, as follows:

$X \leftarrow$ *Leaf*$_x$	$X \leftarrow$ *Leaf*$_y$
$v_l \leftarrow X$.*value*	$v_r \leftarrow X$.*value*
while *X.rp* $\neq$ `nil` **and** *X.rp.h* $< y$ **do**	**while** *X.lp* $\neq$ `nil` **and** *X.lp.l* $> x$ **do**
{ $X \leftarrow$ *X.rp*	{ $X \leftarrow$ *X.lp*
$v_l \leftarrow \min\{v_l,$ *X.right.value*$\}$	$v_r \leftarrow \min\{v_r,$ *X.left.value*$\}$
}	}

"$\min\{v_l, v_r\}$ is the minimal value of $C_{t,x}, C_{t,x+1}, \ldots, C_{t,y}$"

where *Leaf*$_x$ is the leaf of the tree containing $C_{t,x}$. The two traversals begin at *Leaf*$_x$ and *Leaf*$_y$ and move up the tree, using successive *rp* and *lp* pointers. The traversals end at the least common ancestor of the two leaves, which can be determined using the ranges stored at each node. The first traversal computes the minimum of the $C_{t,i}$'s from x to the midpoint of the LCA's range. The second traversal computes the minimum from the LCA's midpoint to y. This can be shown in a simple inductive proof, not given here, whose core argument uses the lemma below to show that each move up an *lp* or *rp* pointer extends the range of the minimum computation contiguously to the left or right of the current range, respectively.

The time taken by the query is $O(\log W)$, where $W = y - x$, since the range of *X.lp* and *X.rp* is at least twice as large as the range of each node X in the traversal and the range of the LCA is $\leq 2W$. Thus, arbitrary range queries can be satisfied in time logarithmic to the width of the range.

LEMMA 7. For a node X in a range query tree, 1) if *X.lp* $\neq$ `nil`, then *X.lp.left.h* = *X.l* -1 and 2) if *X.rp* $\neq$ `nil`, then *X.rp.right.l* = *X.h* $+1$.

Proof. We give only the proof for *X.lp*. There are two cases. First, if *X.lp.right* $=$ *X* (X is the right child of *X.lp*), then *X.lp.left* and X must be the two children of *X.lp*. Then, *X.lp.left.h* must equal $X.l - 1$, since the two children of a node divide that node's range in half. Second, if *X.lp.right* $\neq$ *X* (implying that *X.rp.left* $=$ *X*), then applying this proof inductively to *X.rp* yields that

X.rp.lp.left.h = *X.rp.l* -1. But *X.lp* = *X.rp.lp* by the range query tree definition. And *X.l* = *X.rp.l*, since X is the left child of *X.rp* and so the leftmost leaf in both their subtrees must be the same node. Thus, *X.lp.left.h* = *X.l* -1. $\square$

The construction of the range query tree occurs incrementally as the overall matching algorithm produces values of $C_{t,i}$. It uses a stack of ⟨*node,level*⟩ pairs to hold the roots of unfinished trees and their levels in the tree. Figure 6.3 shows the state of the construction for $i = 6$. The construction step for $i > 0$ is

```
Z ← New () ; Z.value ← C_{t,i}
⟨A, L⟩ ← Pop (Stack)
if L > 1 then        # The new leaf is a left child, so create and push its parent
{  X ← New () ; X.left ← Z ; X.lp ← Top (Stack).node
   Z.rp ← X ; Z.lp ← X.lp
   Push (Stack, ⟨A, L⟩) ; Push (Stack, ⟨X, 1⟩)
}
else       # L = 1 and the new leaf is a right child, so find the root of the largest
           # now finished sub-tree, create and push its parent, and then set the
           # rp pointers for the rightmost nodes of the finished sub-tree
{  A.right ← Z ; Z.lp ← A
   Z ← A            # In the loop, Z points to the finished sub-trees' roots
   while Size (Stack) > 0 and Top (Stack).level = L + 1 do
   {  ⟨A, L'⟩ ← Pop (Stack)
      A.value ← min{A.left.value, A.right.value}
      A.l ← A.left.l ; A.h ← A.right.h
      A.right ← Z ; Z.lp ← A
      Z ← A ; L ← L'
   }
   X ← New ()        # The new unfinished sub-tree root
   Z.rp ← X ; X.left ← Z
   if Size (Stack) > 0 then X.lp ← Top (Stack).node
   Push (Stack, ⟨X, L + 1⟩)
   for i ← L − 1 down to 1 do
   {  Z ← Z.right ; Z.rp ← Z.lp.rp }
}
```

Operations *Push*, *Pop*, *Top* and *Size* are the basic stack operations and *New* creates a new tree node. The construction at $i = 0$ is equivalent to the case above where the new leaf is a left child.

When the new leaf storing $C_{t,i}$ is a left child in the tree, it suffices to construct its parent and push the unfinished parent on the stack. When the new leaf is a right child, the construction is finished for the roots $R_1, R_2, \ldots, R_k$ of each sub-tree whose rightmost leaf is the new leaf. The completion involves first an upwards pass through these roots, setting the pointers and minimum values for each R_l. After the root of the new sub-tree whose left child is R_k has been created, an downward pass is made setting each of the *rp* pointers to that new root. This construction takes

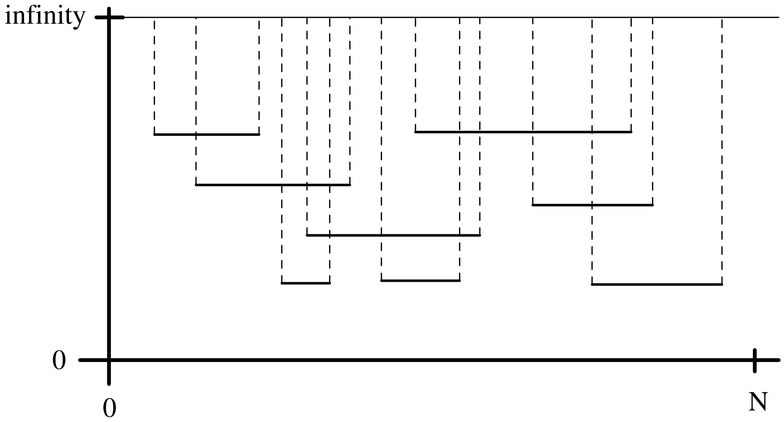


Figure 6.4: An inverted skyline.

$O(N)$ time for matching graph row t, since each value in a node is computed once and the size of the tree is $2\,N - 1$.

The back end problem for proportional and per-interval spacing takes the form of an *inverted skyline* problem and can be solved using a binary search tree. If the possible $C_{s,[i',j']}$ values which can contribute to various $C_{s,j}$ are plotted graphically, the picture takes the form of Figure 6.4. Each horizontal line represents the contribution of one $C_{s,[i',j']}$ to the $C_{s,j}$ values (the values of j form the x-axis of the figure). The actual values of the $C_{s,j}$ are those found on the lowest line at each position j in $0..N$. Thus, the various contributions of the $C_{s,[i',j']}$ form the "buildings" of the inverted skyline.

The solution to this problem is to keep a balanced binary search tree, ordered by score, that holds the $C_{s,[i',j']}$ values applicable at each position j. Thus at j, $C_{s,j}$ is the minimal value in the tree. The value of each $C_{s,[i',j']}$ is inserted and deleted from the tree at $j' + rmin$ and $j' + rmax + 1$, respectively. By applying one efficiency "trick," the time taken by this algorithm can be bounded by $O((N + I) \log W)$, where W is the width of the widest neighborhood. The trick is that when a value $C_{s,[j',j']}$ is being inserted into the tree, a query is made for any value in the tree which is to be removed at $j' + rmax + 1$. If no such value exists, the new value is inserted into the tree. If such a value exists, only the lower scoring value is kept in the tree, since the higher score cannot contribute to a future $C_{s,j}$. The use of this trick bounds the size of the tree at W nodes. Thus, all queries, insertions and deletions take $O(\log W)$ time.

The result of the algorithms described in this section is that proportional and per-interval implicit spacing can be computed $O((N+I) \log W)$ time, where W is the size of the widest input interval's neighborhood.

6.3.3 Minimum Envelopes and Affine Curves

In this section, we consider only the linear extension pieces to the affine-scored implicit spacing, bounded spacers and repair intervals. The fixed range sections of these affine scoring schemes can be handled separately by the algorithms of Sections 6.3.1 and 6.3.2. For explicit spacers and repair intervals, extra incoming edges must be added to vertex (s, j) from vertices $(t, 0)$, $(t, 1)$, $\ldots$, $(t, j - max - 1)$ and from $(t, j - min + 1)$, $(t, j - min + 2)$, $\ldots$, (t, j). The following two recurrences capture the new computations required for those edges.

$$L_{s,j} = \min\{C_{t,k} + cl * (k - (j - min)) \mid j - min < k \leq j\}$$
$$R_{s,j} = \min\{C_{t,k} + cr * ((j - max) - k) \mid 0 \leq k < j - max\}$$

With these recurrences for an explicit spacer, $C_{s,j} = \min\{L_{s,j} + c,\ R_{s,j} + c, \ldots$the fixed range recurrence$\ldots\}$ where c is the fixed range spacer cost. The repair interval case is similar, except the recurrences dealing with the input intervals must also be included.

The extra edges for affine scored implicit spacing correspond to the the four affine curves given in the specification and can be derived from the following four recurrences:

$$LL_{s,[i,j]} = \min\{C_{t,k} + lcl_a * ((i + lmin) - k) \mid 0 \leq k < i + lmin\}$$
$$LR_{s,[i,j]} = \min\{C_{t,k} + lcr_a * (k - (i + lmax)) \mid i + lmax < k \leq b\}$$
$$RL_{s,j} = \min\{C_{s,[i',j']} + rcl_a * ((j' + rmin) - j) \mid [i',j'] \in I_a \ \&\ b \leq j < j' + rmin\}$$
$$RR_{s,j} = \min\{C_{s,[i',j']} + rcr_a * (j - (j' + rmax)) \mid [i',j'] \in I_a \ \&\ j' + rmax < j \leq N\}$$

where $lmin$, $lmax$, $rmin$, $rmax$ and b generically denote the neighborhoods and boundary point for an interval. With these recurrences, the computations for implicit spacing become

$$C_{s,[i,j]} = \min\{LL_{s,[i,j]} + lc_a + \sigma,\ LR_{s,[i,j]} + lc_a + \sigma,\ \ldots\text{the fixed range comp.}\ldots\}$$
$$C_{s,j} = \min\{RL_{s,j} + rc_a,\ RR_{s,j} + rc_a,\ \ldots\text{the fixed range computation}\ldots\}$$

where lc_a and rc_a are the base implicit spacing costs and σ is the score associated for input interval $[i,j]$.

The rest of this section presents the algorithms for the six recurrences above by grouping them into three sets, 1) R, LL and RR, 2) L and LR and 3) RL, based on the algorithms used to compute the recurrences. For each group, abstract forms of the recurrences are constructed which simplifies the recurrences and better illustrates their commonality. Then, the solution for one representative abstract form (per group) is presented, along with the complexity for the resulting algorithm. The mapping back to the original recurrences is straightforward, and so not explicitly described.

The R, LL and RR recurrences can be abstracted as $D1_i = \min_{0 \leq k < i}\{E_k + c * (i - k)\}$ for R and LL and $D2_i = \min\{E_{[i',j']} + c * (i - k) \mid k = j' + rmax < i\}$ for RR. In this abstract form, each D_i is the minimum of the *candidates*, $f(m) = e_k + c * (m - k)$ from each position $k < i$, that are evaluated at i. The difference between the two forms is that multiple candidates can occur with the same k value in the second form. All of the candidates involved in the $D1_i$ (or $D2_i$) equations have the same slope c. Because lines with different origins and the same slope must intersect either zero or an infinite number of times, the minimum candidate at a position i must remain minimum over the candidates from $k < i$ at every $i' > i$. Therefore, only the current minimum at i is needed to compute future $D_{i'}$ values, and the recurrence for each D can be rewritten as $D1_i = \min\{D_{i-1} + c,\ E_i\}$ and $D2_i = \min\{D_{i-1} + c,\ \min\{E_{[i',j']} \mid j' + rmax = i\}\}$. These recurrences can be computed in $O(N)$ and $O(N + I)$ time for $0 \leq i \leq N$.

The L and LR recurrences take the abstract forms $D_i = \min_{l \leq k \leq i}\{E_k + c * (k - l)\}$ and $D_{[i,j]} = \min_{l \leq k \leq b}\{E_{[i,j]} + c * (k - l) \mid l = i + lmin \ \&\ b = (j - i) * bp_a\}$. The D_i form is a special case of $D_{[i,j]}$, where only one value is needed for any position i (rather than values for each $[i,j]$) and where all of the widths $i - l$ are of equal size (instead of the varying $b - l$). Only the solution to the more complicated $D_{[i,j]}$ is presented here. Each $D_{[i,j]}$ is the minimum, at position l, of candidates,

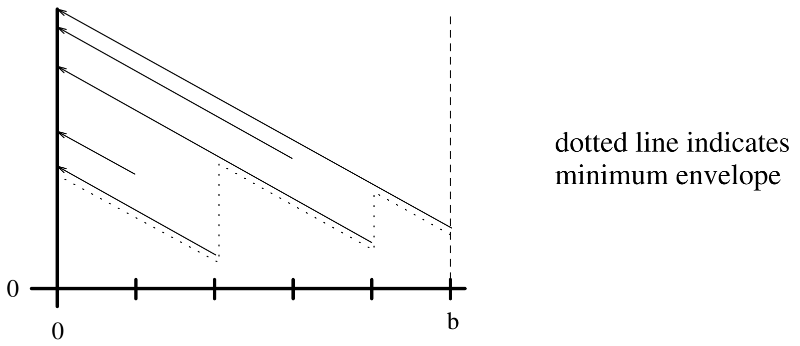


Figure 6.5: Five candidate lines and their minimum envelope.

$f(m) = y + c * (x - m)$, whose origin on the x-axis is somewhere between l and b. Considering the $D_{[i,j]}$ recurrence from the viewpoint of a particular position b, multiple $D_{[i,j]}$ values might be required at b, each with $(j - i) * bp = b$ and with differing l values. The solution is to construct a data structure at each position b which stores $\forall\, 0 \leq m \leq b : \min_{m \leq k \leq b}\{E_k + c * (k - m)\}$. Graphically, this is illustrated in Figure 6.5 as the *minimum envelope* of the candidate lines for $0 \leq m \leq b$. The value of $D_{[i,j]}$ is then computed by searching the data structure at $b = (j - i) * bp$ for the minimal value at $m = i + lmin$.

The data structure constructed at each position b is an ordered list of the candidates in the minimum envelope and the sub-ranges of $0..b$ in which each candidate is minimal. Since the candidates ordered by their minimal sub-ranges are also ordered by their origin positions k and since each candidate is minimal over a contiguous region of $0..b$ by the zero or infinite intersection property, constructing the list at $b + 1$ from the list at b involves 1) removing candidates at the tail of the list which are eliminated by the new candidate with origin position at $b + 1$ and 2) inserting the new candidate at the tail of the list. Implementing the list with a balanced search tree yields an $O(N \log N)$ construction algorithm and $O(\log N)$ searches for the I $D_{[i,j]}$ values.

The solution to the RL recurrence is essentially the inverse of the LR algorithm. The abstract RL recurrence takes the form of $D_j = \min\{E_{[i',j']} + c * (h - j) \mid [i', j'] \in I_a$ where $b \leq j < h = j' + rmin\}$. Graphically, the picture looks like that of Figure 6.5 again except that the range is $j..N$, not $0..b$, and the intervals are not evenly distributed at each position, but occur according to the individual $j' + rmin$ values. The algorithm is the inverse of the previous algorithm for the following two reasons. First, only the value at i must be retrieved from the data structure constructed at i, unlike the previous algorithm in which queries could vary over the range $0..b$. Second, new candidates at j, i.e. the candidates from each interval $[i', j']$ where $(j' - i') * bp = j$, can have origin positions, $j' + rmin$, anywhere from j to N. So, those candidates can be inserted anywhere into the minimum envelope of j. The construction of the list at $j + 1$ from the list at j in this case involves 1) removing the head of the list if that candidate's origin position $j' + rmin = j$, 2) inserting the new candidates, where $(j' - i') * bp = j + 1$, which will now appear in the minimum envelope at $j + 1$ and 3) removing the candidates from the list at j which are eliminated from the minimum envelope by the insertion of the new candidates at $j + 1$. Steps 2 and 3 are equivalent to the procedure described in the last paragraph for inserting new candidates into the LR data structure, except that the insertion uses only the sub-list of the current envelope which is minimal from $j + 1$ to $j' + rmin$, instead of the whole list, and the candidate currently minimal at $j' + rmin$

is not necessarily removed from the list, as it may still be minimal to the right of $j' + rmin$. Implementing this using a balanced binary search tree gives an $O((N + I) \log N)$ time complexity to the algorithm, since the three construction steps use a constant number of list operations.

Taken together, these four algorithms compute the linear extensions to the affine scored explicit spacers, implicit spacers and repair intervals in $O((N + I) \log N)$ time per matching graph row.

CHAPTER 7
CONCLUSIONS

This dissertation considers the problems and algorithms of discrete pattern matching over sequences and interval sets, presenting a coherent framework and new algorithms for discrete pattern matching over sequences and developing the sub-domain of discrete pattern matching over interval sets. Much of the work in discrete pattern matching over sequences has focused on specific problems and the techniques required to solve those specific problems. Despite 20 years of research, the range of algorithms for pattern matching over sequences have never really been characterized in a framework similar to that presented in this dissertation, although the use of such a framework has been implicit in many papers. Even the survey papers either attempt no such framework, or concentrate only on smaller regions of the domain (i.e. the keyword, set of keyword and k differences triangle or the edit distance, LCS, sequence comparison triangle). Chapter 2 presents the elements of a alignment-graph/dynamic-programming framework for this sub-domain of discrete pattern matching which are required by the later chapters of the dissertation. The structure of the framework appears to be general enough, however, to capture the rest of the sub-domain's algorithms.

Using this framework, Chapter 3 presents the first sub-cubic algorithms for approximate regular expression pattern matching with concave gap penalties. The use of the framework greatly simplifies the algorithm's description, since the computations of the insertion and deletion gaps, which are the cause of the cubic-time behavior for the naive algorithm, are characterized easily in terms of two "one-dimensional" problems. However, the final solution requires mechanisms and computations above and beyond the straightforward dynamic programming algorithms in order to achieve the improvement in the complexity bound. The most significant of those are the applicative candidate lists used to model minimum envelopes and the use of a "balancing act" between the cost of maintaining multiple candidate lists throughout the algorithm and the cost of merging separate candidate lists.

Chapter 4 develops the alignment-graph/dynamic-programming framework for extended regular expressions and presents new algorithms for their exact and approximate matching. The framework and algorithms for these patterns are not obvious extensions to those of regular expressions, but require new assumptions about the notion of a path through the alignment graph and the definition of an approximate match between a sequence and an extended regular expression.

And finally, Chapters 5 and 6 develop the sub-domain of discrete pattern matching over interval sets, or super-pattern matching. This problem formulation is useful for those applications where a simple sequence or regular expression of symbols cannot characterize the desired "pattern" and a more general recognition hierarchy is required. The two chapters cover a wide range of exact and approximate matching problems for this sub-domain and develop a matching-graph/dynamic-programming framework to describe the algorithms for that range of problems. And, as with the regular expression problem of Chapter 3, the more complicated matching problems are solved using techniques beyond naive dynamic programming, such as the sliding window, range tree and inverted skyline algorithms.

There are still a number open problems in discrete pattern matching over sequences. However, most of these appear to be of a more theoretical vein. A number of questions exist involving extending sequence comparison solutions to approximate pattern matching of regular expressions, extended regular expressions and even for context-free grammars. The most notable open problems for regular expressions are (1) using the Four Russians approach to produce a sub-quadratic approximate matching algorithm, (2) an $O(ND)$ approximate matching algorithm under the edit distance scoring model, where D is the optimal alignment score, (3) extending the work on approximate matching from fragments and (4) approximate regular expression pattern matching with non-monotone increasing convex gap penalties. One very interesting open problem for extended regular expressions and context-free grammars is approximate matching with concave or convex gap penalties. For CFG's, the best algorithm appears to be an $O(N^5)$ or $O(N^6)$ algorithm, whereas for ERE's it is not clear whether an optimal match can be defined (as the recursive match definition and the gap penalty scoring model are not easily merged).

For super-pattern matching, one major question left unaddressed is its efficiency in practice. While the theoretical complexity bounds do give pattern and input dependent limits to the running time for specific problems, the actual behavior of the algorithms for common super-pattern matching problems has not been explored. How fast does it really run on realistic problems? Second to that, and related to it, is the question of incorporating context-sensitive information into the matching. The information in many of the problems requiring recognition hierarchies and AI techniques cannot easily be modeled using only intervals and interval types. One example of this is on-line handwriting recognition, where the one-dimensional space can model the passing of time and the interval sets can represent the drawn lines and arcs, but there is no easy way to represent the geographic relationship between those lines and arcs using only intervals. A mechanism for incorporating this context-sensitive information greatly simplifies the description of the problem. However, including that information into the problem description invalidates the polynomial complexity bounds for the algorithms. Despite the worst-case exponential behavior resulting from allowing this information, can the super-pattern matching algorithms do "well enough" for realistic problems, in terms of either the end user's perception of the speed of the algorithm or by comparison with other artificial intelligence approaches. Apart from those two questions, the other open problems for super-pattern matching are the completion of the range of corresponding problems and sub-cases forming the domain of approximate pattern matching over sequences, such as extending the affine scoring scheme to allow concave or convex functions. This dissertation developed the core of the domain, and covered the problems with practical applications. However, the more theoretical problems from discrete pattern matching over sequences were left unanswered for super-pattern matching.

REFERENCES

[AC75] Alfred V. Aho and Margaret J. Corasick. "Efficient String Matching: An Aid to Bibliographic Search." *C. ACM* 18,6 (June 1975), 333–340.

[AE86a] S. Altschul and B. W. Erickson. "Optimal Sequence Alignments Using Affine Gap Costs." *Bull. Math. Biol.* 48 (1986), 606–616.

[AE86b] Stephen F. Altschul and Bruce W. Erickson. "Locally Optimal Subalignments Using Nonlinear Similarity Functions." *Bull. Math. Biol.* 48,5/6 (1986), 633–660.

[AG87] Alberto Apostolico and C. Guerra. "The Longest Common Subsequence Problem Revisited." *Algorithmica* 2 (1987), 315–336.

[AGM+90] S. Altschul, W. Gish, Webb Miller, Eugene W. Myers, and D. Lipman. "A Basic Local Alignment Search Tool." *J. Mole. Biol.* 215 (1990), 403–410.

[AHU76] Alfred V. Aho, Daniel S. Hirschberg, and Jeffery D. Ullman. "Bounds on the Complexity of the Longest Common Subsequence Problem." *J. ACM* 23,1 (January 1976), 1–12.

[AKM+87] Alok Aggarwal, Maria M. Klawe, Shlomo Moran, Peter Shor, and Robert Wilber. "Geometric Applications of a Matrix-Searching Algorithm." *Algorithmica* 2 (1987), 195–208.

[Alb67] C. N. Alberga. "String Similarity and Misspellings." *C. ACM* 10,5 (May 1967), 302–313.

[All70] F. E. Allen. "Control Flow Analysis." *SIGPLAN Notices* 5 (1970), 1–19.

[AP72] Alfred V. Aho and Thomas G. Peterson. "A Minimum Distance Error-Correcting Parser for Context-Free Languages." *SIAM J. Computing* 1,4 (December 1972), 305–312.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, Reading, Mass. (1986).

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation and Compiling, Vol. 1, Parsing*. Prentice-Hall, Englewood Cliffs, NJ. (1972).

[AWM+84] R. M. Arbarbanel, P. R. Wieneke, E. Mansfield, D. A. Jaffe, and D. L. Brutlag. "Rapid Searches for Complex Patterns in Biological Molecules." *Nucl. Acids Res.* 12,1 (1984), 263–280.

[BM77] Robert S. Boyer and J. Strother Moore. "A Fast String Searching Algorithm." *C. ACM* 20,10 (October 1977), 762–772.

[Brz64] J. A. Brzozowski. "Derivatives of Regular Expressions." *J. ACM* 11 (1964), 481–494.

[BS86] Gerard Berry and Ravi Sethi. "From Regular Expressions to Deterministic Automata." *Theo. Comp. Sci.* 48,1 (1986), 117–126.

[BYP92] Ricardo A. Baeza-Yates and Chris H. Perleberg. "Fast and Practical Approximate String Matching." In *LNCS 664: Proc. 3rd Symp. Combinatorial Pattern Matching* (1992), 185–192.

[CHM93] Kun-Mao Chao, Ross C. Hardison, and Webb Miller. "Constrained Sequence Alignment." *Bull. Math. Biol.* 55,3 (1993), 503–524.

[CL90] William I. Chang and Eugene L. Lawler. "Sublinear Expected Time Approximate String Matching and Biological Applications." In *Proc. 31st FOCS* (October 1990), 116–124.

[CL92] William I. Chang and Jordan Lampe. "Theoretical and Empirical Comparisons of Approximate String Matching Algorithms." In *LNCS 664: Proc. 3rd Symp. Combinatorial Pattern Matching* (1992), 175–184.

[CP90] Francis Chin and C. K. Poon. "A Fast Algorithm for Computing Longest Common Subsequences of Small Alphabet Size." Tech. Report TR-90-08. Univ. of Hong Kong, Dept. of Computer Science (July 1990).

[CP92] Chia-Hsiang Chang and Robert Paige. "From Regular Expressions to DFA's Using Compressed NFA's." In *LNCS 664: Proc. 3rd Symp. Combinatorial Pattern Matching* (1992), 90–110.

[CPM92] Kun-Mao Chao, William R. Pearson, and Webb Miller. "Aligning Two Sequences within a Specified Diagonal Band." *CABIOS* 8,5 (1992), 481–487.

[CW79] B. Commentz-Walter. "A String Matching Algorithm Fast on the Average." In *Proc. 6th Int. Colloq. Automata, Lang., and Prog.* (July 1979), 118–132.

[Dam64] F. J. Damerau. "A Technique for Computer Detection and Correction of Spelling Errors." *C. ACM* 7,3 (March 1964), 171–176.

[Ear70] Jay Earley. "An Efficient Context-Free Parsing Algorithm." *C. ACM* 13,2 (February 1970), 94–102.

[Epp90] David Eppstein. "Sequence Comparison with mixed Convex and Concave Costs." *J. Algorithms* 11 (1990), 85–101.

[FCK+91] T. Fujisaki, T. E. Chefalas, J. Kim, C. C. Tappert, and C. G. Wolf. "Online Run-On Character Recognition: Design and Performance." *Inter. J. Pat. Rec. and Art. Intell.* 5 (1991), 123–137.

[Fic84] James W. Fickett. "Fast Optimal Alignment." *Nucl. Acids Res.* 12,1 (1984), 175–179.

[FS83] W. M. Fitch and Temple F. Smith. "Optimal Sequence Alignments." *Proc. Nat. Acad. Sci. U. S. A.* 80 (1983), 1382–1386.

[FS90] C. A. Fields and C. A. Soderlund. "gm: A Practical Tool for Automating DNA Sequence Analysis." *CABIOS* 6,3 (1990), 263–270.

[GG86] Zvi Galil and Raffaele Giancarlo. "Improved String Matching with k Mismatches." *SIGACT News* 17,4 (1986), 52–54.

[GG88] Zvi Galil and Raffaele Giancarlo. "Data Structures and Algorithms for Approximate String Matching." *J. Complexity* 4 (1988), 33–72.

[GG89] Zvi Galil and Raffaele Giancarlo. "Speending Up Dynamic Programming with Applications to Molecular Biology." *Theo. Comp. Sci.* 64 (1989), 107–118.

[GHR80] S. Graham, M. Harrison, and W. Ruzzo. "An Improved Context-Free Recognizer." *ACM TOPLAS* 2,3 (July 1980), 415–462.

[GK82] Walter B. Goad and Minoru I. Kanehisa. "Pattern Recognition in Nucleic Acid Sequences. I. A General Method for Finding Local Homologies and Symmetries." *Nucl. Acids Res.* 10,1 (1982), 247–263.

[GKDS92] Roderic Guigó, Steen Knudsen, Neil Drake, and Temple F. Smith. "Prediction of Gene Structure." *J. Mole. Biol.* 226 (1992), 141–157.

[GL89] R. Grossi and F. Luccio. "Simple and Efficient String Matching with k Mismatches." *Info. Proc. Let.* 33 (1989), 113–120.

[Got82] Osamu Gotoh. "An Improved Algorithm For Matching Biological Sequences." *J. Mole. Biol.* 162 (1982), 705–708.

[Got87] Osamu Gotoh. "Pattern Matching of Biological Sequences with Limited Storage." *CABIOS* 3,1 (1987), 17–20.

[GP89] Zvi Galil and Kunsoo Park. "A Linear-Time Algorithm for Concave One-Dimensional Dynamic Programming." *Info. Proc. Let.* 33 (1989), 309–311.

[GP90] Zvi Galil and Kunsoo Park. "An Improved Algorithm for Approximate String Matching." *SIAM J. Computing* 19,6 (December 1990), 989–999.

[Hat74] J. P. Haton. "Practical Application of a Real-Time Isolated-Word Recognition System using Syntactic Constraints." *IEEE Trans. Acoustics, Speech and Signal Processing* 22,6 (1974), 416–419.

[HD80] Patrick A. V. Hall and Geoff R. Dowling. "Approximate String Matching." *Comp. Surveys* 12,4 (December 1980), 381–402.

[Hir75] Daniel S. Hirschberg. "A Linear Space Algorithm for Computing Maximal Common Subsequences." *C. ACM* 18,6 (June 1975), 341–343.

[Hir77] Daniel S. Hirschberg. "Algorithms for the Longest Common Subsequence Problem." *J. ACM* 24,4 (October 1977), 664–675.

[Hir89] Stephen Hirst. "A New Algorithm Solving Membership of Extended Regular Expressions." Univ. of Syndey, Dept. of Computer Science. Draft Copy (1989).

[HL87] Daniel S. Hirschberg and Lawrence L. Larmore. "The Least Weight Subsequence Problem." *SIAM J. Computing* 16,4 (August 1987), 628–638.

[HM91] Xiaoqiu Huang and Webb Miller. "A Time-Efficient, Linear-Space Local Similarity Algorithm." *Adv. Appl. Math.* 12 (1991), 337–357.

[Hor80] R. N. Horspool. "Practical Fast Searching in Strings." *Software - Pract. Exper.* 10 (1980), 501–506.

[HS77] James W. Hunt and Thomas G. Szymanski. "A Fast Algorithm for Computing Longest Common Subsequences." *C. ACM* 20,5 (May 1977), 350–353.

[HS91] Andrew Hume and Daniel M. Sunday. "Fast String Searching." *Software - Pract. Exper.* 21,11 (November 1991), 1221–1248.

[HU75] Matthew S. Hecht and Jeffrey D. Ullman. "A Simple Algorithm for Global Data Flow Analysis." *SIAM J. Computing* 4,4 (December 1975), 519–532.

[HU79] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, Mass. (1979).

[JTU91] Petteri Jokinen, Jorma Tarhio, and Esko Ukkonen. "A Comparison of Approximate String Matching Algorithms." Tech. Report A-1991-7. Univ. of Helsinki, Dept. of Computer Science (1991).

[Kas65] T. Kasami. "An Efficient Recognition and Syntax-Analysis Algorithm for Context-Free Languages." Tech. Report AFCRL-65-758. Air Force Cambridge Research Laboratory (1965).

[Kil85] J. Kilbury. "A Modification of the Earley-Shieber Algorithm for Direct Parsing with ID/LP-Grammars." In *Informatik-Fachberichte 103: Proc. 8th German Workshop on AI*, J. Laubsch (ed.). Springer-Verlag, Berlin. (1985), 39–48.

[KK90] Maria K. Klawe and Daniel J. Kleitman. "An Almost Linear Time Algorithm for Generalized Matrix Searching." *SIAM J. Disc. Math.* 3,1 (February 1990), 81–97.

[Kle56] S. C. Kleene. "Representation of Events in Nerve Nets and Finite Automata." In *Automata Studies*, C. E. Shannon and J. McCarthy (eds.). Princeton University Press. (1956), 3–41.

[KMP77] Donald E. Knuth, James H. Morris, and Vaughan R. Pratt. "Fast Pattern Matching in Strings." *SIAM J. Computing* 6,2 (June 1977), 323–350.

[Knu73] Donald E. Knuth. *Sorting and Searching: The Art of Computer Programming, Vol. 3*. Addison-Wesley, Reading, Mass. (1973).

[KST92] Jong Yong Kim and John Shawe-Taylor. "Fast Multiple Keyword Searching." In *LNCS 664: Proc. 3rd Symp. Combinatorial Pattern Matching* (1992), 41–51.

[LBB+89] Alan S. Lapedes, Christopher Barnes, Christian Burks, Robert M. Farber, and Karl M. Sirotkin. "Application of Neural Networks and Other Machine Learning Algorithms to DNA Sequence Analysis." In *Computers and DNA, SFI Studies in the Sciences of Complexity, Vol. VII*, George Bell and T. Marr (eds.). Addison-Wesley, Redwood City, CA. (1989).

[Ld90] Lectures and discussions. *Workshop on Recognizing Genes*. Aspen Center for Physics, (May 1990).

[LV88] Gad M. Landau and Uzi Vishkin. "Fast String Matching with k Difference." *J. Comp. Sys. Sci.* 37,1 (August 1988), 63–78.

[LV89] Gad M. Landau and Uzi Vishkin. "Fast Parallel and Serial Approximate String Matching." *J. Algorithms* 10 (1989), 157–169.

[LVN89] Gad M. Landau, Uzi Vishkin, and Ruth Nussinov. "Fast Alignment of DNA and Protein Sequences." Tech. Report CS-TR-2199 or UMIACS-TR-89-20. Univ. of Maryland, Inst. for Adv. Computer Studies, Dept. of Computer Science (1989).

[LW75] Roy Lowrance and Robert A. Wagner. "An Extension of the String-To-String Correction Problem." *J. ACM* 23,1 (April 1975), 177–183.

[LWS87] Richard H. Lathrop, Teresa A. Webster, and Temple F. Smith. "Ariadne: Pattern-Directed Inference and Hierarchical Abstraction in Protein Structure Recognition." *C. ACM* 30,11 (November 1987), 909–921.

[Mea55] G. H. Mealy. "A Method for Synthesizing Sequential Circuits." *Bell System Technical Journal* 34 (September 1955), 1045–1079.

[Mey85] Bertrand Meyer. "Incremental String Matching." *Info. Proc. Let.* 21 (1985), 219–227.

[MM85] Webb Miller and Eugene W. Myers. "A File Comparison Program." *Software - Pract. Exper.* 15 (1985), 1025–1040.

[MM88a] Webb Miller and Eugene W. Myers. "Sequence Comparison with Concave Weighting Functions." *Bull. Math. Biol.* 50,2 (1988), 97–120.

[MM88b] Eugene W. Myers and Webb Miller. "Optimal Alignments in Linear Space." *CABIOS* 4,1 (1988), 11–17.

[MM89a] Eugene W. Myers and Webb Miller. "Approximate Matching of Regular Expressions." *Bull. Math. Biol.* 51,1 (1989), 5–37.

[MM89b] Eugene W. Myers and Webb Miller. "Row Replacement Algorithms for Screen Editors." *ACM TOPLAS* 11,1 (January 1989), 33–56.

[Moo56] E. F. Moore. "Gedanken Experiments on Sequential Machines." In *Automata Studies*, C. E. Shannon and J. McCarthy (eds.). Princeton University Press. (1956), 129–153.

[Mor70] Howard L. Morgan. "Spelling Correction in System Programs." *C. ACM* 13,2 (February 1970), 90–94.

[MP80] William J. Masek and Michael S. Paterson. "A Faster Algorithm for Computing String Edit Distances." *J. Comp. Sys. Sci.* 20,1 (1980), 18–31.

[MY60] R. McNaughton and H. Yamada. "Regular Expressions and State Graphs for Automata." *IRE Trans. on Elect. Computers* 9,1 (March 1960), 39–47.

[Mye84] Eugene W. Myers. "Efficient Applicative Data Types." In *Proc. 11th Symp. POPL* (1984), 66–75.

[Mye86] Eugene W. Myers. "An O(ND) Difference Algorithm and Its Variations." *Algorithmica* 1 (1986), 251–266.

[Mye88] Eugene W. Myers. "A Four-Russians Algorithm for Regular Expression Pattern Matching." Tech. Report TR-88-34. Univ. of Arizona, Dept. of Computer Science (1988).

[Mye90] Eugene W. Myers. "A Sublinear Algorithm for Approximate Keyword Searching." Tech. Report TR-90-25. Univ. of Arizona, Dept. of Computer Science (1990).

[NKY82] Narao Nakatsu, Yahiko Kambayashi, and Shuzo Yajima. "A Longest Common Subsequence Algorithm Suitable for Similar Text Strings." *Acta Info.* 18 (1982), 171–179.

[NS83] Andrew S. Noetzel and Stanley M. Selkow. "An Analysis of the General Tree-Editing Problem." In *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, David Sankoff and Joseph B. Kruskal (eds.). Addison-Wesley, Reading, Mass. (1983), Chapter 8, 237–263.

[NW70] Saul B. Needleman and Christian D. Wunsch. "A General Method Applicable to the Search for Similarities in the Amino Acid Sequence of Two Proteins." *J. Mole. Biol.* 48,3 (March 1970), 443–453.

[Oom91] B. John Oommen. "String Editing with Substitution, Insertion, Deletion, Squashing and Expansion Operations." Tech. Report SCS-TR-194. Carleton Univ., School of Computer Science (September 1991).

[PL88] William R. Pearson and David J. Lipman. "Improved Tools for Biological Sequence Comparison." *Proc. Nat. Acad. Sci. U. S. A.* 85 (April 1988), 2444–2448.

[Rai92] Timo Raita. "Tuning the Boyer-Moore-Horspool String Search Algorithm." *Software - Pract. Exper.* 22,10 (October 1992), 879–884.

[RCW73] T. A. Reichert, D. N. Cohen, and A. K. C. Wong. "An Application of Information Theory to Genetic Mutations and the Matching of Polypeptide Sequences." *J. Theo. Biol.* 42 (1973), 245–261.

[San72] David Sankoff. "Matching Sequences Under Deletion/Insertion Constraints." *Proc. Nat. Acad. Sci. U. S. A.* 69,1 (January 1972), 4–6.

[SC71] H. Sakoe and S. Chiba. "A Dynamic-Programming Approach to Continuous Speech Recognition." In *1971 Proc. Inter. Cong. Acoustics* (1971), Paper 20 C 13.

[Sea89] David B. Searls. "Investigating the Linguistics of DNA with Definite Clause Grammars." In *Proc. North Amer. Conf. Logic Prog., Vol. 1* (1989), 189–208.

[Sel74a] Peter H. Sellers. "An Algorithm for the Distance Between Two Finite Sequences." *J. Comb. Theo.* Series A,16 (1974), 253–258.

[Sel74b] Peter H. Sellers. "On the Theory and Computation of Evolutionary Distances." *SIAM J. Appl. Math.* 26,4 (June 1974), 787–793.

[Sel80] Peter H. Sellers. "The Theory and Computation of Evolutionary Distances: Pattern Recognition." *J. Algorithms* 1,1 (March 1980), 359–373.

[Sel84] Peter H. Sellers. "Pattern Recognition in Genetic Sequences by Mismatch Density." *Bull. Math. Biol.* 46,4 (1984), 501–514.

[SK83] David Sankoff and Joseph B. Kruskal (eds.). *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*. Addison-Wesley, Reading, Mass. (1983).

[Smi91] P. D. Smith. "Experiments with a Very Fast Substring Search Algorithm." *Software - Pract. Exper.* 21,10 (October 1991), 1065–1074.

[ST85] Daniel Sleator and Robert Endre Tarjan. "Self-Adjusting Binary Search Trees." *J. ACM* 32,3 (July 1985), 652–686.

[Ste92] Graham A. Stephen. "String Search." Tech. Report TR-92-gas-01. Univ. College of North Wales, School of Elect. Eng. Sci. (October 1992).

[Sto88] Gary D. Stormo. "Computer Methods for Analyzing Sequence Recognition of Nucleic Acids." *Rev. Biophys. Chem.* 17 (1988), 241–263.

[Sun90] Daniel M. Sunday. "A Very Fast Substring Search Algorithm." *C. ACM* 33 (1990), 132–142.

[SW81] Temple F. Smith and Michael S. Waterman. "Identification of Common Molecular Subsequences." *J. Mole. Biol.* 147 (1981), 195–197.

[SWF81] Temple F. Smith, Michael S. Waterman, and W. M. Fitch. "Comparative Biosequence Metrics." *J. Mole. Evol.* 18 (1981), 38–46.

[Tay84] Philip Taylor. "A Fast Homology Program for Aligning Biological Sequences." *Nucl. Acids Res.* 12,1 (1984), 447–455.

[Tho68] Ken Thompson. "Regular Expression Search Algorithm." *C. ACM* 11,6 (June 1968), 419–422.

[Tic84] Walter F. Tichy. "The String-to-String Correction Problem with Block Moves." *ACM Trans. on Computer Systems* 2,4 (November 1984), 309–321.

[Tom86] Masaru Tomita. *Efficient Parsing for Natural Languages*. Kluwer Academic Publishers. (1986).

[Ukk85] Esko Ukkonen. "Algorithms for Approximate String Matching." *Info. and Control* 64 (1985), 100–118.

[Vin68] T. K. Vintsyuk. "Speech Discrimination by Dynamic Programming." *Cybernetics* 4,1 (1968), 52–57.

[VZ70] V. M. Velichko and N. G. Zagoruyko. "Automatic Recognition of 200 Words." *Inter. J. Man-Machine Studies* 2 (1970), 223–234.

[Wag74] Robert A. Wagner. "Order-n Correction of Regular Languages." *C. ACM* 17,5 (May 1974), 265–268.

[Wat84a] Michael S. Waterman. "Efficient Sequence Alignment Algorithms." *J. Theo. Biol.* 108 (1984), 333–337.

[Wat84b] Michael S. Waterman. "General Methods for Sequence Comparison." *Bull. Math. Biol.* 46,4 (1984), 473–500.

[WC76] C. K. Wong and Ashok K. Chandra. "Bounds for the String Editing Problem." *J. ACM* 23,1 (January 1976), 13–16.

[WE87] Michael S. Waterman and M. Eggert. "Letters to the Editor: A New Algorithm for Best Subsequence Alignments with Applications to tRNA-rRNA Comparisons." *J. Mole. Biol.* 197 (1987), 723–728.

[WF74] Robert A. Wagner and Michael J. Fischer. "The String-To-String Correction Problem." *J. ACM* 21,1 (January 1974), 168–173.

[Wil88] Robert Wilber. "The Concave Least-Weight Subsequence Problem Revisited." *J. Algorithms* 9 (1988), 418–425.

[WL83] W. J. Wilbur and David J. Lipman. "Rapid Similarity Searches of Nucleic Acid and Protein Data Banks." *Proc. Nat. Acad. Sci. U. S. A.* 80 (February 1983), 726–730.

[WM91] Sun Wu and Udi Manber. "Fast Text Searching With Errors." Tech. Report TR-91-11. University of Arizona, Department of Computer Science (1991).

[WMM92] Sun Wu, Udi Manber, and Eugene W. Myers. "A Sub-quadratic Algorithm for Approximate Limited Expression Matching." Tech. Report TR-92-36. Univ. of Arizona, Dept. of Computer Science (1992).

[WS78] Robert A. Wagner and Joel L. Seiferas. "Correcting Counter-Automaton-Recognizable Languages." *SIAM J. Computing* 7,3 (August 1978), 357–375.

[WSB76] Michael S. Waterman, Temple F. Smith, and W. A. Beyer. "Some Biological Sequence Metrics." *Adv. Math.* 20 (1976), 367–387.

[You67] Daniel H. Younger. "Recognition and Parsing of Context-Free Languages in Time N^3." *Info. and Control* 10,2 (1967), 189–208.