Super-Pattern Matching*

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

TR 92-29

Revised April 26, 1996

Abstract

Some recognition problems are either too complex or too ambiguous to be expressed as a simple pattern matching problem using a sequence or regular expression pattern. In these cases, a richer environment is needed to describe the "patterns" and recognition techniques used to perform the recognition. Some researchers have turned to artificial intelligence techniques and multi-step matching approaches for the problems of gene recognition [5, 7, 18], protein structure recognition [13] and on-line character recognition [6]. This paper presents 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 artifical intelligence characterizations, and yet polynomial time algorithms are described for each problem in the class.

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

Keywords Computational biology, gene finding, pattern recognition

*This work is 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.

Super-Pattern Matching

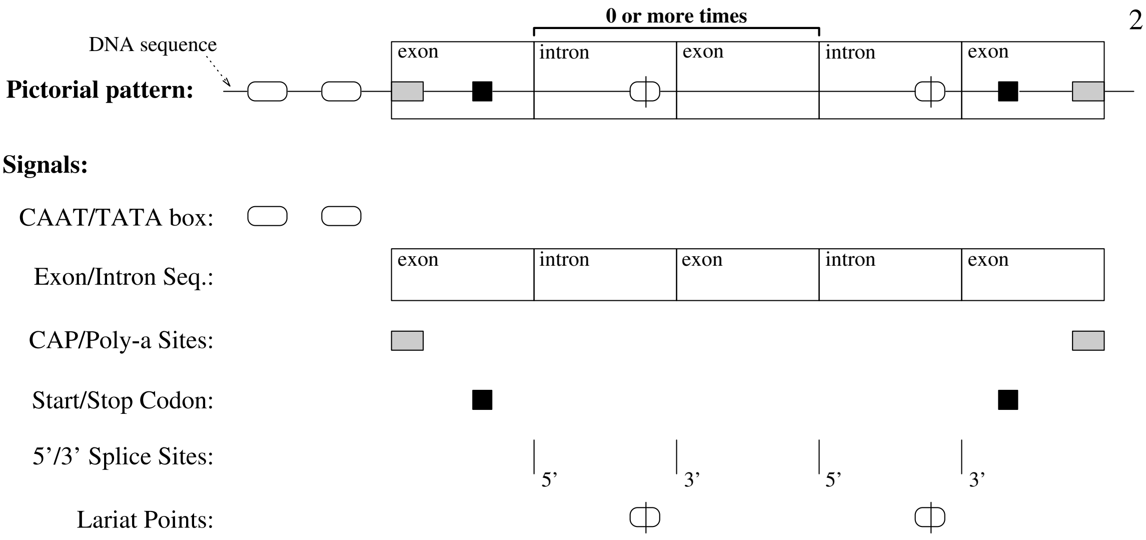
1 Introduction

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 [14] have developed a basic picture of a gene encoding structure, illustrated in Figure 1. Such a region consists of a collection of basic 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 don't 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 [2], consensus matrices [19], and neural nets [12]. Libraries of these component recognizers are currently being used to recognize either pieces of gene encodings or complete encodings. One gene recognition system, GM [5], 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 [5], backtracking tree-search algorithms [7], and context-sensitive, definite clause grammars [18]. These techniques 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 being efficient in practice.

This multi-step approach to pattern matching has also appeared for such problems as protein structure prediction [13] and on-line character recognition [6]. In general terms, the matching proce-



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 1 Basic gene encoding structure.

dure forms a *recognition hierarchy*, as depicted in Figure 2, where successively larger "patterns" are matched at higher and higher levels in the hierarchy. Super-pattern matching characterizes an isolated recognition problem in a general recognition hierarchy. The basic super-pattern matching problem definition (Section 2) presents an input and output interface facilitating such hierarchical recognition constructions and defines the meaning of a "match" under this interface. Section 2 also describes several problem variations with different super-pattern expressions and output requirements. Section 3 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.

Sections 4 and 5 describe solutions for each problem in the class. These solutions are set in a common matching-graph/dynamic-programming framework, similar to the edit-graph/dynamic-programming framework underlying the algorithms for approximate pattern matching over sequences. The worst-case time complexity of the algorithms considered here is $O(N^3M)$, given in terms of the sizes of the underlying sequence (N) and the super-pattern (M). However, tighter input-dependent bounds of $O((N+I)ML)$ to $O((N+I)ML\log N)$ are also derived, where I is the size of the largest interval set and L is the length of the longest match to a prefix of the super-pattern. For practical recognition problems with reasonably accurate recognizers and with small or rare super-patterns, the algorithms' performance is significantly better than the cubic-time worst-case behavior.

2 Basic Problem

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

- A one-dimensional space, $[0..N]$.

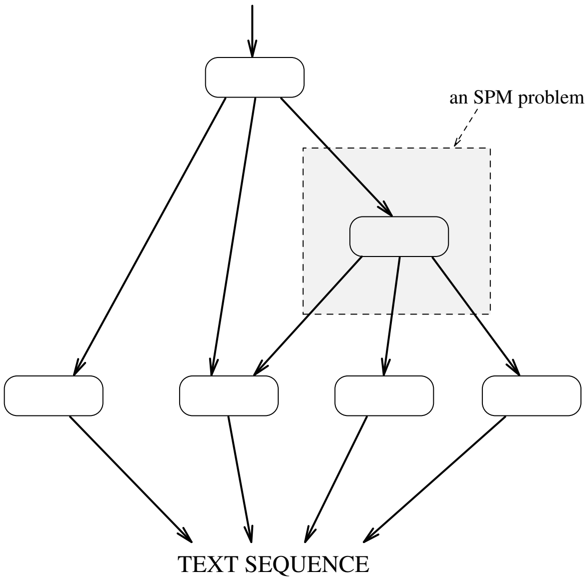


Figure 2 Pictorial description of a recognition hierarchy.

- 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 identify the recognizers of exons, introns, etc. 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 $[0, N]$ and sub-expressions of the super-pattern. A set of *recursive* matching rules (similar to those of [3] for pattern matching over sequences) defines the intervals matching an expression P in terms of 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 $P \equiv \varepsilon$ (the empty string), then $i = j$.
3. If $P \equiv R\, S$ (concatenation), then $\exists\, i \leq k \leq j : [i,k]$ matches R & $[k,j]$ matches S.
4. If $P \equiv R \mid S$ (alternation), then either $[i,j]$ matches R or $[i,j]$ matches S.
5. If $P \equiv R^*$ (Kleene closure), then either $i = j$ or $\exists\, i < k \leq j : [i,k]$ matches R & $[k,j]$ matches R^*.
6. If $P \equiv R \,\&\, S$ (intersection), then $[i,j]$ matches R and $[i,j]$ matches S.
7. If $P \equiv R - S$ (difference), then $[i,j]$ matches R, but does not match S.

The intervals matching an expression P are called the *matching intervals* of P. And 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.

Note that this recursive definition differs from a strict set-theoretic definition of a match, as is used in approximate pattern matching over sequences. There, a match of sequence A and pattern P consists of an alignment between A and a *specific sequence* $B \in L(P)$, where $L(P)$ the language defined by pattern P. The corresponding set-theoretic definition of a super-pattern match between $[i, j]$ and P pairs a sequence of intervals $<[i, i_1], [i_1, i_2], \ldots, [i_k, j]>$ with a sequence $B = b_1 b_2 \ldots b_{k+1}$ in $L(P)$ using rule 1 above. The recursive rules for the intersection and difference operators (rules 6 and 7) fail to maintain this definition as the matches of $[i, j]$ with R and S are not required to use a common B in $L(R)$ and $L(S)$. Thus, the "interval sequence" matching an interval and super-pattern may not necessarily be a sequence of intervals, but could contain overlapping "interval sequences" matching each intersection sub-expression of P.

We use the recursive definition of a matching interval for two reasons, one practical and one computational. First, extended regular expressions under the recursive definition provide a natural method for specifying overlapping signals, one not permitted under the set-theoretic definition. 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 1 presents a more potent use of this recursively-defined intersection operator. The second reason is that polynomial time algorithms exist, under the recursive definition, for the problems of approximate extended regular expression pattern matching over sequences and super-pattern matching with an extended regular expression. The recursive definition, super-pattern matching algorithm is presented in Section 4. The best known algorithms under the set-theoretic definition take time exponential in the size of the extended regular expression.

The default type of output for the basic problem is the super-pattern's set of matching intervals. With this type of 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.

3 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 occuring 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 incomplete input interval sets 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.

3.1 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 signals. One interesting example of this is the "space" of size 12 to 40 occuring 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 and which then splice the intron out, connecting the ends of the surrounding exons. One requirement of this process is that the RNA molecules attached to the lariat point and 3' splice site must also attach to each other. Since these molecule are of a certain size, the distance of the corresponding attachment points on the DNA (and thus on its copy) must also be of a certain size. Hence, the 12 to 40 spacing distance between those elements in the gene encoding structure.

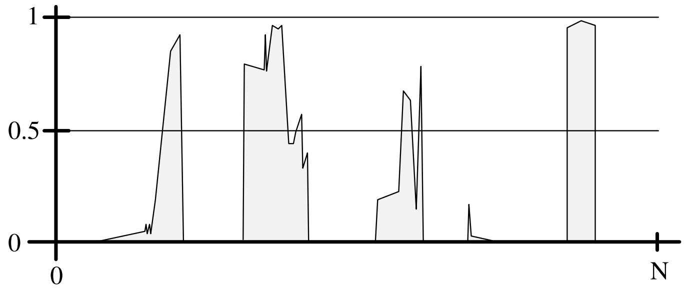
The super-pattern in Figure 1 illustrates the two forms of the spacer considered here, *bounded* ($<12, 40>$) and *unbounded* ($<0, *>$) spacers. Each spacer specifies a size range of intervals which match the spacer. In terms of the recursive definition from Section 2, the following additional rules capture this property:

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

This paper only considers spacers whose lower and upper bounds are non-negative, i.e. $l \geq 0$. 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 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 requires additional algorithmic support beyond the scope of this paper.

3.2 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 ([12]). 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.

3.3 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 $score([i, j], R\, S) = \min\limits_{i \leq k \leq j} \{score([i, k], R) + score([k, j], S)\}$.
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 $score([i, j], R^*) = \begin{cases} 0 & \text{if } i = j \\ \min\limits_{i < k \leq j} \{score([i, k], R) + score([k, j], R^*)\} & \text{if } i \neq j \end{cases}$

6. If $P \equiv R \,\&\, S$, then $score([i,j], R \,\&\, S) = \mathcal{F}_{R\&S}(score([i,j],R),\; score([i,j],S))$
   where $\mathcal{F}_{R\&S}$ can be a general function (see below).
7. If $P \equiv R - S$, then $score([i,j], R - S) = \mathcal{F}_{R-S}(score([i,j],R),\; score([i,j],S))$
   where $\mathcal{F}_{R-S}$ can be a general function (see below).

For sequence and regular expression super-patterns, this function simply sums the scores of the intervals in the best scoring interval sequence. The extended regular expression scoring rules allow any functions $\mathcal{F}_{R\&S}$ and $\mathcal{F}_{R-S}$ to determine the score of the extended regular expression. More complex scoring methods are needed, because the concept of "minimal is optimal" is not expressive enough to capture the meaning of intersection and difference in the realm of approximate matching. Scoring schemes such as taking the minimal, maximal or average score 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 uses a decision function returning the score of the match to R if the score of S's match is above a threshold. Otherwise, it returns infinity. Note that these examples do not always compute the minimal score resulting from evaluating $\mathcal{F}$ over the whole range of possible scores for R and S. The general functions allowed here permit a wide range of scoring schemes, and in particular include all of the example schemes cited above.

The scoring of interval sequences changes the output requirements for the four problems of Section 2 and 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 $score([i,j],a)$.

3.4 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.

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 $P \equiv a$:

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

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

3.5 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. 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 $bprop_a$ used to separate the left and right endpoint neighborhoods. The graphical representation of the scoring scheme is shown in Figure 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 $P \equiv a$:

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

where

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

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

This assumes no repair interval specifications for a.

The affine specification for bounded spacers and repair intervals is a three part curve defined for size values $s \geq 0$ and is shown in Figure 3b. 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

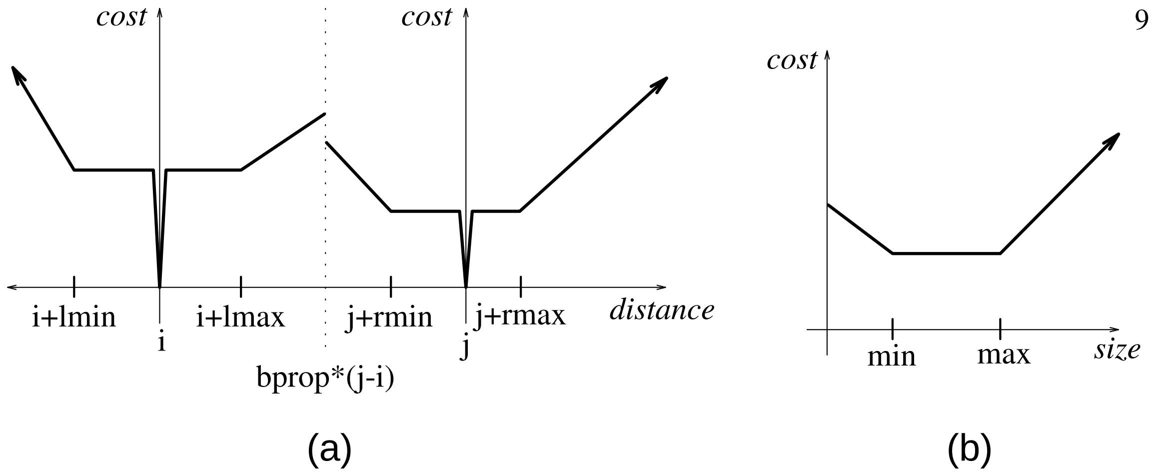


Figure 3 Affine scoring of a) implicit spacing for interval $[i, j]$ and b) bounded spacers/repair intervals.

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:

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

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.

4 Solving the Basic Problem

The solutions to each of the super-pattern matching problems employ a framework similar to that developed for sequence-based approximate pattern matching of sequences [16, 17, 20] and regular expressions [15, 21]. The framework for super-pattern matching involves four major steps common for all of the algorithmic solutions. The first step is to construct 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 is to derive 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 given.

The sub-sections that follow present the four steps describing the algorithms for the scanning and instantiation problems with interval scoring and with a 1) sequence, 2) regular expression and 3) extended regular expression super-pattern. The inclusion of interval scoring results in more interesting algorithms, since the the graph traversal problem changes from one of finding the existence of paths to one of finding the shortest paths through the matching graph. The solutions to problems with no interval scoring or for the decision and optimization problems are simple variations of the algorithms presented below.

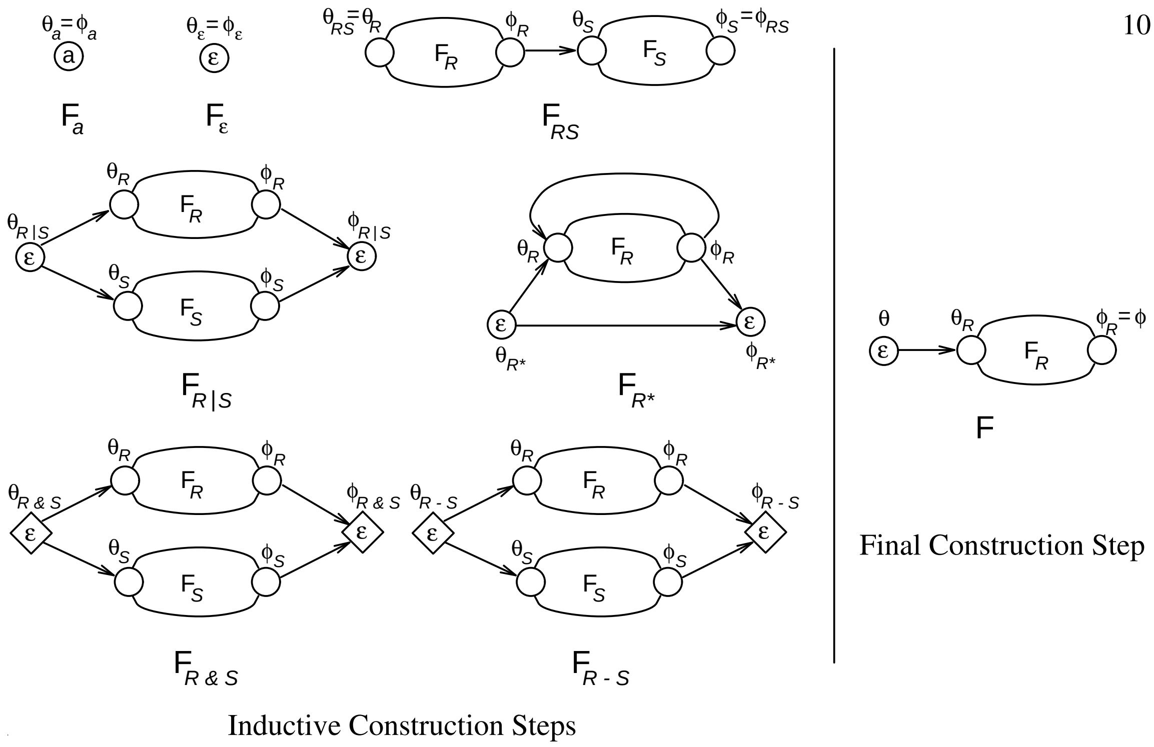


Figure 4 Inductive state machine construction rules for RE's and ERE's.

4.1 Sequences and Regular Expressions

The state machines constructed from sequence and regular expression super-patterns are deterministic or non-deterministic finite automata, hereafter referred to as NFA's. The finite automata used in this paper are the state-labeled automata used by Myers and Miller [15] for approximate regular expression pattern matching over sequences. Formally, an NFA $F = <V, E, \lambda, \theta, \phi>$ consists of: (1) a set V of vertices, called *states*; (2) a set E of directed edges between states; (3) a function 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 sequence or regular expression R can be converted into an equivalent NFA F using the inductive construction depicted in Figure 4, ignoring for the moment $F_{R\&S}$ and F_{R-S}. For example, the figure shows that F_{RS} is obtained by constructing F_R and F_S, adding an edge from ϕ_R to θ_S, and designating θ_R and ϕ_S as its source and sink states. After inductively constructing F_R, an ε-labeled start state is added as shown in the figure to arrive at F. This last step guarantees that the 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 matching graphs.

Note that for a sequence super-pattern $P = p_1p_2 \ldots p_M$, the construction of F uses only the construction rules F_a and F_{RS}, resulting in a deterministic NFA containing a row of $M + 1$ states. Successive states in F are labeled with successive symbols of P. This is illustrated in Figure 5 for $P = aba$. For the full regular expressions, a straightforward induction (given in [15]) shows that automata constructed 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)

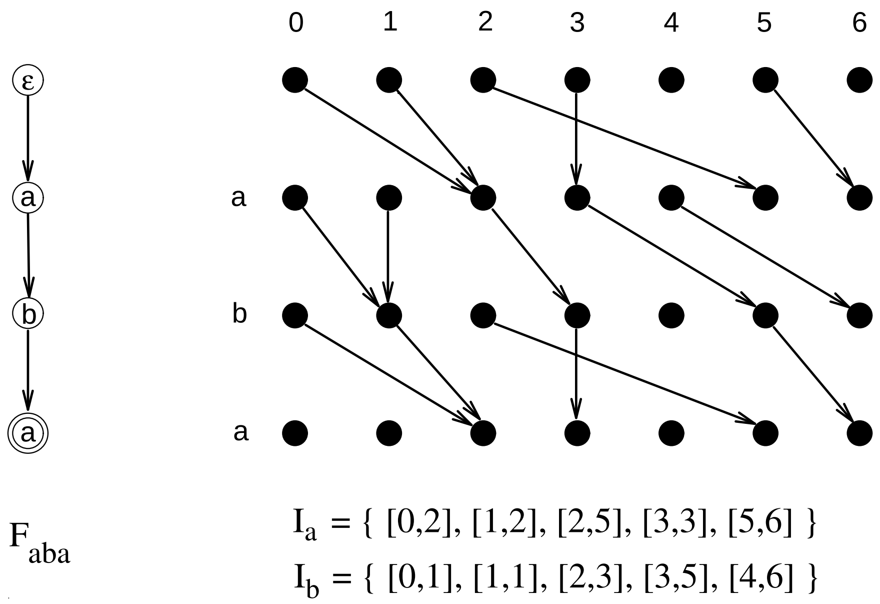


Figure 5 The NFA and matching graph for super-pattern $P = aba$ and 1-d space $[0, 6]$.

$|V| \leq 2\,|P|$, i.e. the number of states in F is less than 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* [1] or as having a *loop connectedness parameter* of 1 [8]. In summary, the key observations 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 $O(M)$.

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 5 and 6. 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 Section 2, shows 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

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

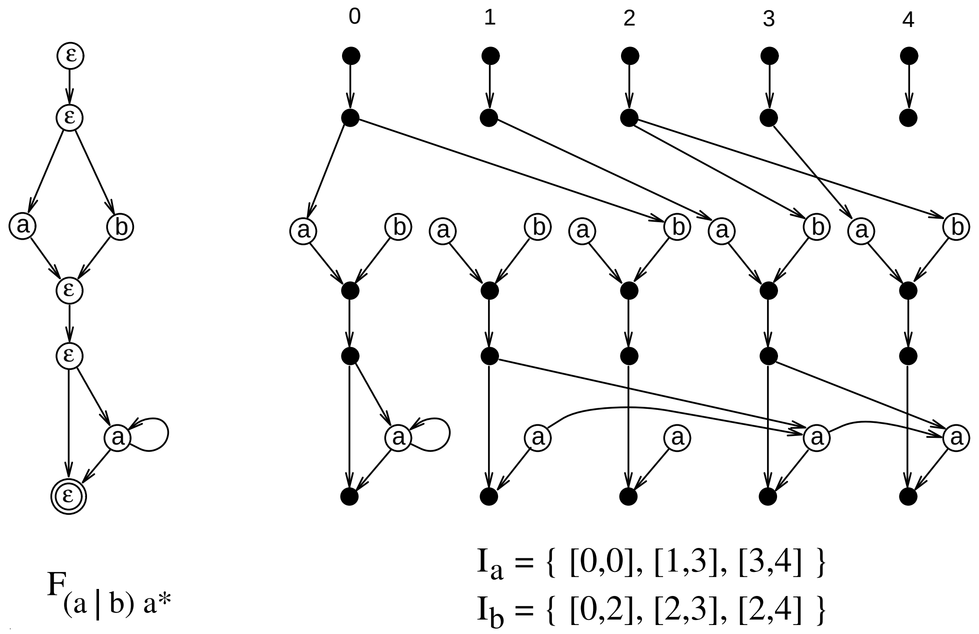


Figure 6 The state machine and matching graph for $P = (a \mid b)\, a^*$ and $N = 4$.

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 by taking either 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.

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 matches to each interval $[k, j]$ by unioning the minimum scoring position pairs for each position k from 0 to j, 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 set of matching intervals output for the instantiation problem is $\{[k, j] \text{ scoring } c \mid 0 \le j \le N \;\&\; \langle c, k\rangle \in C_{\phi,j}\}$.

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. Those cycles are reducible because of the structure of cycles in F (which forms the sub-graph along each column). This is useful because the recurrences above involve computing the shortest paths to a particular graph vertex, so only acyclic paths need to be considered. By the reducibility of F and as proved in [15] for a similar graph, it follows that any acyclic path through the matching graph contains at most one back edge from each column's copy of F.

These observations led Myers and Miller [15] to a column-based, two "sweeps" per column, dynamic programming algorithm for the approximate regular expression pattern matching problem over sequences. Their algorithm, applied here to the matching graph, sweeps the j^{th} column twice in topological order, computing the relevant terms of the recurrence in each sweep. That suffices to correctly compute the recurrence values for the j^{th} column, because any acyclic path to a vertex in the j^{th} column involves at most one back edge in column j. So, any path to a vertex (s,j) which enters column j at some other state, say t, consists of DAG edges to some vertex (v,j), a back edge to (w,j) and DAG edges to (s,j). The algorithm's first sweep correctly computes the value at (v,j), and the second sweep correctly computes the value at (w,j) and consequently at (s,j).

The algorithm below implements this approach for the instantiation problem:

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) M L), since the recurrence is computed twice for each graph vertex and there are at most $2\,I\,M$ edges in the graph.

4.2 Extended Regular Expressions

For extended regular expression super-patterns, we introduce a new machine, called an *extended NFA* or ENFA, which accepts languages denoted by extended regular expressions. There have been two previous solutions to this problem of recognizing ERE's, one by Hopcroft and Ullman [10] and one by Hirst [9]. The Hopcroft and Ullman algorithm is a naive dynamic programming solution taking $\Omega(N^3)$ time for any expression containing a Kleene closure operator. Hirst's algorithm, on the other hand, is essentially equivalent to the algorithm below in both complexity and algorithmic structure. However, the algorithm's details differ greatly from the framework of pattern matching over sequences, using the expression's parse tree instead of a state machine simulation, and it solves only the language acceptance problem, i.e. is a given string w in $L(P)$. The extended NFA provides a solution which can be explained as an extension to the NFA simulation and can be easily recast as a super-pattern matching algorithm. In fact, the ENFA state simulation is reminiscent of Earley's algorithm [4] for context-free grammar parsing, and it is a sparse, scanning version of the Hopcroft and Ullman's naive dynamic programming algorithm analogously to the way Earley's algorithm is a sparse, scanning version of the CYK algorithm [11, 22]. This section first presents the ENFA construction and its state simulation, and then develops the super-pattern matching algorithm from the state simulation.

For any extended regular expression R, an ENFA F formally consists of the same five-tuple $<V, E, \lambda, \theta, \phi>$ as an NFA, and it uses the inductive construction rules depicted in Figure 4 (now including $F_{R\&S}$ and F_{R-S}). In addition, 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 a sequence of edges through F, to maintain the equivalence between $L_F(\phi)$ and the language specified by R, $L(R)$. 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 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 $w = w_1 w_2 \ldots w_N$, 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 position, in w, of the substrings of w 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 w. 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 a five-part partition of V. The first subset of V contains only the start state of F, θ. Its computation is as follows:

(1) $s = \theta$:

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

The second subset, denoted V_{re}, contains all states except θ and the start and final states of the sub-machines in F. These are the states introduced into F by the regular expression construction rules from Figure 4.

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

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

The values contained in these $S_{i,s}$ sets consist of the beginning positions k in w of matches between $w_{k+1} w_{k+2} \ldots w_i$ 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 enclosing sub-machines are kept in a series of *mapping tables* associated with each $\theta_{R\&S}$ and θ_{R-S} state. 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 their innermost enclosing sub-machine. The third state set, denoted V_θ, contains the start states of each sub-machine, and its recurrences are the following:

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

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

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

The $T_{i,s}$ values, the mapping table for state s, collects and stores the first positions k of the matches between $w_{k+1} w_{k+2} \ldots w_i$ 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 sub-machine final states when matches between the nested sub-machine and a string $w_{i+1}w_{i+2}\ldots w_j$ is found. The value of $S_{i,s}$ here is either empty or contains the single position i, depending on whether a match between a prefix of $w_{i+1}w_{i+2}\ldots w_N$ and the nested sub-machine could result in an overall match between w and F.

The fourth and fifth subsets of V, $V_\&$ and V_-, contain the final states of the $F_{R\&S}$ and F_{R-S} sub-automata, respectively. At one of these final states, $s \in V_\&[V_-]$, the simulation determines those first positions k in the nested sub-machine, $F_{R\&S}$ $[F_{R-S}]$, for which paths exist through F_R and [not] through F_S spelling $w_{k+1}w_{k+2}\ldots w_i$. It then unions the $T_{k,t}$ sets, where $t = \theta_{R\&S}[\theta_{R-S}]$ corresponding to s, to extend 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) $s \in V_\&$:

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

(5) $s \in V_-$:

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

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

The actual algorithm performing the state simulation uses the two-sweep technique over the states of F. Since F is reducible, the same arguments given for the regular expression matching hold here. For an input string w of size N and an extended regular expression P of size M, the time complexity is $O(N^3M)$ in the worst case. However, this complexity depends on the structure of P, and a tighter bound of $O((N+I)ML)$ time can also be derived. In this case, I denotes the largest number of sub-strings of w which match an intersection or difference sub-expression of P. L, as in super-pattern matching, refers to the longest substring of w which matches a prefix of a string in the language denoted by P. This complexity bounds the running time, 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 $T_{i,\theta_{R\&S}}$ sets at each $\phi_{R\&S}$ state (or similarly at each 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 $O(NM)$ and $O(M)$ respectively, giving a complexity bound of $O(NM^3)$.

Returning to the super-pattern matching problem, the matching graphs consists of $N+1$ copies of ENFA F. The graph edges are added as described for regular expressions, as if the intersection and difference sub-automata were alternation sub-automata. The dynamic programming recurrences incorporate the more complex state machine simulation. The recurrences for the instantiation problem use the same five subsets of V, starting with the set containing just θ and the set V_{re}:

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

The computation at the sub-machine start states, $s \in V_\theta$, use similar mapping tables T to hold the partial path information for the enclosing sub-machine:

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

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

And finally, the sub-machine final states in $V_{\&}$ and V_{-} use those mapping tables to extend matches across the nested sub-machine:

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

As with the instantiation problems for sequences and regular expressions, the values in $C_{\phi,j}$ for $0 \leq j \leq N$ 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 I is defined here as 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.

5 Extension Algorithms

The four extensions described in Section 3, 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 section's 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 to 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 section, 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 schemes 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.

5.1 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:

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

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 $(t, i+lmin_a), (t, i+lmin_a+1), \ldots, (t, i+lmax_a)$ to vertex $(s,[i,j])$ and (3) edges connecting vertex $(s,[i,j])$ to vertices $(s, j+rmin_a), (s, j+rmin_a+1), \ldots, (s, j+rmax_a)$. 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 along a graph row s have a complexity of $O(NW)$ for explicit spacers or repair intervals, where $W = h-l$, and a complexity of $((N+I)W)$ 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 *Size*$(L) > 0$ **and** $E_{L_{Size(L)}} > E_j$ **do**
$L \leftarrow$ *DeleteTail*(L)
$L \leftarrow$ *Append*$(L, [j])$
$D_j \leftarrow E_{L_1}$
}

using basic list operations *DeleteHead*, *DeleteTail*, *Size* 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 anytime 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 $j' + rmax_a = j$. The neighborhood for each $C_{s,[i',j']}$ is the same size, so a dominated value can be safely removed from the list since 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.

5.2 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:

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

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*

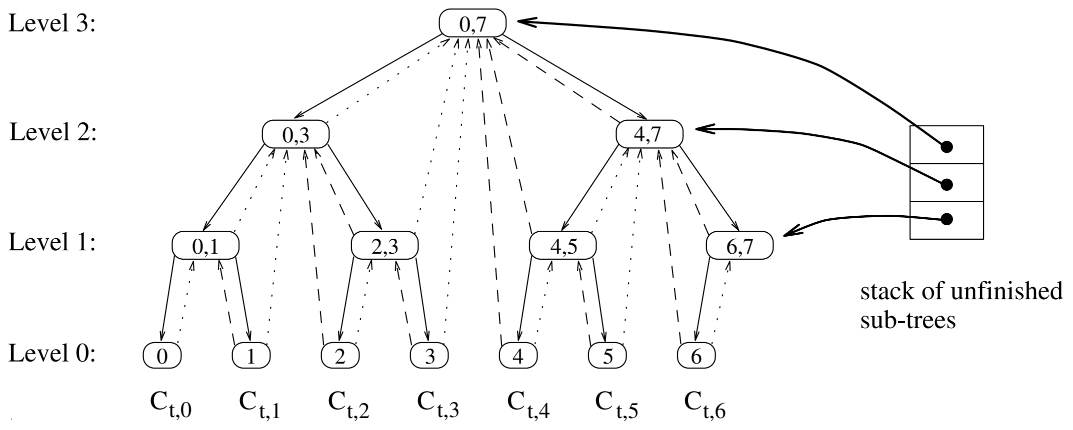


Figure 7 View of a partially constructed range query tree (dashed and dotted lines are *lp* and *rp* pointers).

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, illustrated in Figure 7. 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 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, to the node which is the least common ancestor of the two leaves. 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, resp., of the current range.

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 1. 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. □

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 $\langle$*node,level*$\rangle$ pairs to hold the roots of unfinished trees and their levels in the tree. Figure 7 shows the state of the construction for $i = 6$. The construction step for $i > 0$ is

$Z \leftarrow$ *New* () ; *Z.value* $\leftarrow C_{t,i}$
$\langle A, L\rangle \leftarrow$ *Pop* (*Stack*)
if $L > 1$ **then** # The new leaf is a left child, so create and push its parent
{ $X \leftarrow$ *New* () ; *X.left* $\leftarrow Z$; *X.lp* $\leftarrow$ *Top* (*Stack*)*.node*
  Z.rp $\leftarrow X$; *Z.lp* $\leftarrow$ *X.lp*
  Push (*Stack*, $\langle A, L\rangle$) ; *Push* (*Stack*, $\langle X, 1\rangle$)
}
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* $\leftarrow Z$; *Z.lp* $\leftarrow A$
  $Z \leftarrow A$ # In the loop, Z points to the finished sub-trees' roots
  while *Size* (*Stack*) > 0 **and** *Top* (*Stack*)*.level* $= L + 1$ **do**
  { $\langle A, L'\rangle \leftarrow$ *Pop* (*Stack*)
    A.value $\leftarrow \min\{$*A.left.value, A.right.value*$\}$
    A.l $\leftarrow$ *A.left.l* ; *A.h* $\leftarrow$ *A.right.h*
    A.right $\leftarrow Z$; *Z.lp* $\leftarrow A$
    $Z \leftarrow A$; $L \leftarrow L'$
  }
  $X \leftarrow$ *New* () # The new unfinished sub-tree root
  Z.rp $\leftarrow X$; *X.left* $\leftarrow Z$
  if *Size* (*Stack*) > 0 **then** *X.lp* $\leftarrow$ *Top* (*Stack*)*.node*
  Push (*Stack*, $\langle X, L + 1\rangle$)
  for $i \leftarrow L - 1$ **down to** 1 **do**
  { $Z \leftarrow$ *Z.right* ; *Z.rp* $\leftarrow$ *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 $O(N)$ time for matching graph row t, since 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 8. Each horizontal line represents the contribution of one $C_{s,[i',j']}$ to the $C_{s,j}$ values (the values of j form

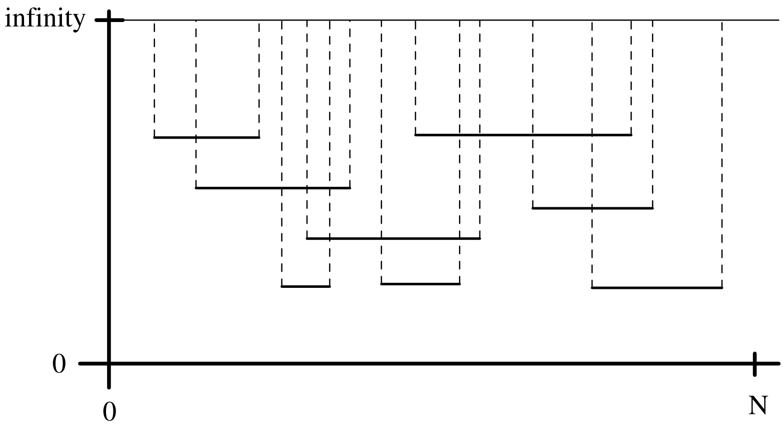


Figure 8 An inverted skyline.

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 $0..N$. The solution is to keep a balanced binary search tree, ordered by score, holding 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 $j'+rmax$, 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 $j'+rmax$. 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.

5.3 Minimum Envelopes and Affine Curves

In this section, we consider only the linear extension sections of the affine 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 5.1 and 5.2. For explicit spacers and repair intervals, extra incoming edges must be added to vertex (s,j) from vertices $(t,0)$, $(t,1)$, . . ., $(t,j-max-1)$ and from $(t,j-min+1)$, $(t,j-min+2)$, . . ., (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

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

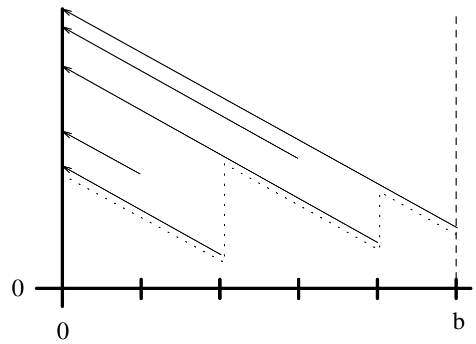
where lc_a and rc_a are the base implicit spacing costs and σ is the score associated for input interval $[i, j]$.

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 $D_{[i,j]} = \min_{l \leq k \leq b}\{E_{[i,j]} + c*(k-l) \mid l = i + lmin \;\&\; b = (j-i)*bprop_a\}$. 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, $f(m) = y + c * (x - m)$, 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 $(j - i) * bprop = b$ 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 9 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 $b = (j - i) * bprop$ 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.



dotted line indicates
minimum envelope

Figure 9 Five candidate lines and their minimum envelope.

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 9 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 $(j' - i') * bprop = j$, 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 $(j' - i') * bprop = j + 1$) 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.

6 Conclusions

The domain of discrete pattern matching over sequences has matured to the point where an outline for the problems in that domain has been developed and a unifying framework, using edit graphs and dynamic programming, for the solutions to the problem domain has appeared. This paper presents a problem class forming the core of a discrete pattern matching domain over something more than just sequences, namely intervals and interval sets. The characterization is such that 1) practical applications can be solved under this problem class, 2) a similar framework can be constructed for these problems, and 3) theoretical differences from the edit-graph/dynamic-programming framework and interesting algorithms appear at the edges of this domain.

Despite the range of problems presented in this paper, some limits were imposed on the problem class. The effects of introducing negative length intervals are not considered. Distance-based scoring schemes with concave or convex curves have been proposed as a realistic model for representing errors, yet this extension is not explored. Finally, this paper concentrates on the algorithms and complexity for isolated super-pattern matching problems, and does not consider the overall algorithms or overall complexities of recognition hierarchies.

References

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

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

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

[4] Earley, J. "An Efficient Context-Free Parsing Algorithm." *C. ACM* 13,2 (1970), 94-102.

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

[6] Fujisaki, T., T. E. Chefalas, J. Kim, C. C. Tappert and C. G. Wolf "Online Run-On Character Recognition: Design and Performance." *International Journal of Pattern Recognition and Artificial Intelligence* 5 (1991), 123-137.

[7] Guigó, R., S. Knudsen, N. Drake and T. Smith "Prediction of Gene Structure." *J. of Molecular Biology* 226 (1992), 141-157.

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

[9] Hirst, S. C. "A New Algorithm Solving Membership of Extended Regular Expressions." draft.

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

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

[12] Lapedes, A., C. Barnes, C. Burks, R. Farber and K. Sirotkin "Application of Neural Networks and Other Machine Learning Algorithms to DNA Sequence Analysis." *Computers and DNA, SFI Studies in the Sciences of Complexity, vol. VII (Eds. G. Bell and T. Marr).* Addison-Wesley, Redwood City, CA. (1989).

[13] Lathrop, R. H., T. A. Webster and T. F. Smith "Ariadne: Pattern-Directed Inference and Hierarchical Abstraction in Protein Structure Recognition." *C. ACM* 30,11 (1987), 909-921.

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

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

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

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

[18] Searls, D. "Investigating the Linguistics of DNA with Definite Clause Grammars." *Proc. of the N. American Conf. on Logic Programming, Vol. 1* (1989), 189-208.

[19] Stormo, G. "Computer Methods for Analyzing Sequence Recognition of Nucleic Acids." *Rev. Biophys. Chem.* 17 (1988), 241-263.

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

[21] Wagner, R. A. and J. I. Seiferas "Correcting Counter-Automaton-Recognizable Languages." *SIAM J. Computing* 7,3 (1978), 357-375.

[22] Younger, D. H. "Recognition and Parsing of Context-Free Languages in Time n^3." *Information and Control* 10,2 (1967), 189-208.