Reporting Exact and Approximate Regular Expression Matches

Eugene W. Myers [*] Paulo Oliva [†] Katia Guimarães [‡]

February 4, 1998

Abstract

While much work has been done on determining if a document or a line of a document contains an exact or approximate match to a regular expression, less effort has been expended in formulating and determining what to report as "the match" once such a "hit" is detected. For exact regular expression pattern matching, we give algorithms for finding a longest match, all symbols involved in some match, and finding optimal submatches to tagged parts of a pattern. For approximate regular expression matching, we develop notions of what constitutes a significant match, give algorithms for them, and also for finding a longest match and all symbols in a match.

1 Introduction

Much attention has been given to the problem of efficiently determining if a string contains a match to a pattern. But the problem of selecting which substring or substrings of the target to report as matching, is an important practical issue, that probably many have had to face, but for which there is a dearth of published work. This paper serves as a systematic exploration of a range of possibilities. We consider here the problem of reporting matches to a regular expression R of size P in a text A of length N, in both the cases of exact and approximate matching. The individual problems we encounter vary in difficulty. We treat them all in order to give a cohesive treatment.

We first make the observation that for the problems being considered here, reporting the desired or interesting thing is more important than doing so extremely efficiently. Generally, one can use a very fast algorithm such as that in `egrep` or `agrep`, to find the lines or regions of a text that contain a match. Thus the N in our context is typically on the order of the length of a line, and not that of a document. That is, we can afford to spend more time on each line containing a match, working on delivering a meaningful match rather than one that is an artifact of the scanning/filtration algorithm.

The widely accepted standard, e.g. `Perl` [WS91], `Tcl/Tk` [Ous94], and the IEEE Posix standard [IEE92], for exact regular expression pattern matching is to report the left-most longest match, i.e. the matching substring whose left end is leftmost, and if there are several with such a left end, then

*Dept. of Computer Science, University of Arizona Tucson, AZ 85721 (e-mail: gene@cs.arizona.edu). Partially supported by NLM grant LM-04960

†Dept. of Informatics, Federal University of Pernambuco, Recife, Brazil (e-mail: pbo@di.ufpe.br). Partially supported by CNPq-PIBIC grant 111371/97-0

‡Dept. of Informatics, Federal University of Pernambuco, Recife, Brazil (e-mail: katia@di.ufpe.br). Partially supported by CNPq grant 352775/96-3

the longest of those. The motivation for this definition appears to stem more from the limitations of searching with a finite-state automata then it does from any conceptual principle: a traditional implementation of a finite automaton easily admits the reporting of left-most and right-most, longest and shortest matches. In a recent paper, Clarke and Cormack, argue that shortest matches have superior search properties when looking at patterns that involve matching several regular expressions [CC97]. On the other hand, we know of no reported work on reporting approximate matches to regular expressions, save that there are connections to work on finding locally optimal alignments [SW81, Sel84].

Our goal is to report matches in a meaningful way. For example, suppose one is given the regular expression `babb*bab` and one is requested to find matches with 2 differences or less. One might like to see a report such as:

$$\textbf{aaaaaba}\boxed{\textbf{babbbbbab}}\textbf{aaaaaaaabaabaaaaa}$$

where the symbols in grey are those that are in some substring that approximately matches the pattern, and the substring in the heavily bordered box is, in a sense to be defined later, the "best" match. This motivates us to consider, for the case of exact matching, the problems of finding the longest match, all symbols of the text involved in a match, and on parsing a match so as to deliver a consistent set of matching substrings that optimize some criterion. For the case of approximate matching, we first examine the problem of defining and delivering an essential or significant match, an issue that does not arise in the exact matching case. We then conclude by solving the longest-match and all-matches problems for the case of approximate matches as well.

2 Reporting Exact Matches

2.1 Finding the Longest Exact Match

Using traditional scan-based regular expression matching methods based on finite automaton, one can easily find the left-most or right-most longest or shortest match to the pattern in a given string. But often this is not the most interesting or specific match. For example, given the pattern `bb*`, the left-most and right-most matches in `aabaabbbbbbbaabaa` are uninteresting compared to the long central match. This motivates the problem of finding the *longest* substring matching a regular expression, as it is the rarest match if symbols occur within the target with equal probability.

Interestingly, this problem can be solved in $O(PN)$ time using a specialization of the *approximate* regular expression matching algorithm of Miller and Myers [MM88] that accommodates any additive alignment scoring scheme δ. Quickly we review this result and then proceed to the specialization. First, recall that any regular expression R can be converted to a state-labeled ϵ-NFA F that has at most $O(P)$ states and transitions and a single final/accepting state. Figure 2 in Appendix A gives such a conversion especially suited to our purpose. Let θ and ϕ be the unique start and final states of F that by construction are labeled with ϵ. Further let $\lambda_s \in \Sigma \cup \{\epsilon\}$ denote the label of state s.

Miller and Myers first observation was that the cost of the best alignment between $A[1..i]$ and a word in $L_F(s)$, where $L_F(s)$ is the set of words accepted at state s, is the minimum-fixed point to the system of equations given by the recurrence:

$$C(i,s) = \max\{\max_{t\to s}\{C(i-1,t) + \delta(a_i,\lambda_s)\}, \max_{t\to s}\{C(i,t) + \delta(\epsilon,\lambda_s)\}, C(i-1,s) + \delta(a_i,\epsilon)\} \quad (1)$$

subject to the boundary condition $C(0,\theta) = 0$. Modifying the boundary condition to $C(i,\theta) = 0$ *for all* i, results in $C(i,s)$ being the score of the best match between *a suffix of* $A[1..i]$ and a word in $L_F(s)$. So in this instance, $C(i,\phi)$ is the score of the best match ending at i to a word recognized by R.

The second observation of Miller and Myers is that the graph of F is a reducible graph, so computing the desired fixed-point for a given i can be achieved by evaluating the relevant terms in two passes over the s parameter in topological order of the acyclic graph obtained by removing the *back-edges* from F (see Fig. 2). This gives the following algorithm outline:

```
$C(0,\theta) \leftarrow 0$
**for** $s \neq \theta$ in topological order of $F$ (less backedges) **do**
    $C(0,s) \leftarrow$ R.H.S. of (1) (exclude back-edge terms)
**for** $i \leftarrow 1, 2, \ldots n$ **do**
{  $C(i,\theta) \leftarrow 0$
    **for** $s \neq \theta$ in topological order of $F$ (less backedges) **do**
        $C(i,s) \leftarrow$ R.H.S. of (1) (exclude back-edge terms)
    **for** $s \neq \theta$ in topological order of $F$ **do**
        $C(i,s) \leftarrow \max\{C(i,s),$ R.H.S. of (1) (include back-edge terms)$\}$
}
```

Turning now to the longest exact match problem, consider the scoring function δ_{long} defined as follows: $\delta_{long}(a,b) = 1$ if $a = b \neq \epsilon$, $\delta_{long}(a,b) = 0$ if $a = b = \epsilon$, and $\delta_{long}(a,b) = -\infty$ otherwise. Effectively, every pair of aligned equal symbols scores 1, every insertion of an ϵ-state of F has score 0, and every other mis-alignment is not allowed. It then follows that $C(i,s) \neq -\infty$ *iff* there is an exact match between a suffix of $A[1..i]$ and a word in $L_F(s)$, and furthermore, that $C(i,s) = len \neq -\infty$ *iff* the longest suffix of $A[1..i]$ matching a word in $L_F(s)$ is of length len. Thus $C(i,\phi)$, under the cost function δ_{long} is the length of the longest match to R ending at i, or $-\infty$ if there is no such match.

In practice, one may model the $-\infty$ of δ_{long} with the value $-(N+1)$ provided that integers of value $[-P(N+1), N]$ can be modeled on the given machine. In that case $C(i,s) \geq 0$ is equivalent to the condition that $C(i,s) \neq -\infty$ above. Furthermore, the special form of δ_{long} allows one to specialize (1) for a non-asymptotic but practical gain in efficiency. First one must carefully construct F so that it is guaranteed to contain no cycles all of whose states are labeled with ϵ. Such an $O(P)$ time and space construction is novel and given in Figure 2 of Appendix A. Let F' be the subautomata consisting of the vertices of F and just those transitions directed into states labeled with ϵ. Because F contains no ϵ-cycles, it follows that F' is acyclic (*including* back edges). One then arrives at the following simple, one-pass algorithm:

```
Compute $C(0,s)$ for all $s$ as before
**for** $i \leftarrow 1, 2, \ldots n$ **do**
{  $C(i,\theta) \leftarrow 0$
    **for** $s \neq \theta$ in topological order of $F'$ **do**
        **if** $\lambda_s = \epsilon$ **then**
            $C(i,s) \leftarrow \max_{t \to s}\{C(i,t)\}$
        **else if** $\lambda_s = a_i$ **then**
            $C(i,s) \leftarrow \max_{t \to s}\{C(i-1,t)+1\}$
        **else**
            $C(i,s) \leftarrow -(N+1)$
}
```

The last consideration is how to actually report the longest matching substring. Running the algorithm above, one finds a left end of a longest possible match at a value of i, call it r, which maximizes $\max_i C(i,\phi)$. By then running the same algorithm on the reverse of R and $A[1..r]$ with the boundary condition that only $C(0,\theta)=0$, one finds the left end of the longest match whose right end is r. The overall time is $O(PN)$ and only $O(P)$ space is required.

2.2 Finding All Matching Positions

Next consider finding *all* the places where the pattern R matches the text, i.e., determine for all i, if symbol a_i is in a match to R or not. Solving this problem requires extending the well-known $O(PN)$ state-set simulation algorithm for matching regular expressions [Tho68, Sed83]. Recall that the algorithm computes in increasing sequence of text position i, what we call here the *forward state-set* $S_f(i) = \{s : \text{a suffix of } A[1..i] \text{ is in } L_F(s)\}$. That is, $S_f(i)$ is the set of all states that F could be in after scanning the first i symbols of A. Connecting this with our algorithm for longest matches above, it is easy to see that $S_f(i) = \{s : C(i,s) \neq \infty\}$. While this demonstrates that $S_f(i)$ can be computed from $S_f(i-1)$ in $O(P)$ worst-case time, the traditional reaching algorithm of [Tho68] does so in $O(|S_f(i-1)| + |S_f(i)|)$ time as one is free to discover the states in $S_f(i)$ in *any* order. This is superior in practice as it is frequently the case that the average size of the state sets is $O(1)$.

Let $s \xrightarrow{w} t$ be a predicate denoting that there is a path in F between states s and t whose state-label sequence spells w. With this notation, we may give the following, automata-centric definition: $S_f(i) = \{s : \exists j \leq i+1, \theta \xrightarrow{A[j..i]} s\}$. To solve our problem we will also need to be able compute what we call the *reverse* state-set at position i, $S_r(i) = \{s : \exists j \geq i-1, s \xrightarrow{A[i..j]} \phi\}$. These reverse state sets are easily computed by symmetry to the forward case: run the same algorithm, but on the reverse of both F and A. The following lemma follows directly from the definitions of S_f and S_r:

Lemma 1: For all i, a_i is in a match to R *iff* there exists $s \in S_f(i) \cap S_r(i)$ such that $\lambda_s \neq \epsilon$.

The one potentially interesting computational issue is how to deliver both $S_f(i)$ *and* $S_r(i)$ for all choices of i. If one computes and saves the state sets for all values of i then the resulting algorithm takes $O(PN)$ time *and space* in the worst case. Using less space is difficult because $S_f(i)$ is easily computable from $S_f(i-1)$ but not from $S_f(i+1)$. Similarly $S_r(i)$ is easily computable from $S_r(i+1)$ but not from $S_r(i-1)$. Thus the "grain" of the computations for S_f and S_r oppose each other. If space is a problem in a particular context, then one can employ the "going-against the grain" algorithm of Myers and Jain [MJ96], to compute $S_r(1), S_r(2), S_r(3), \ldots S_r(N)$ in the given order using $O(tPN)$ time and $O(PN^{1/t})$ space for any choice of $t \geq 1$. Choosing $t = \log N$ gives an $O(PN \log N)$ time and $O(P \log N)$ space worst-case guarantee.

2.3 Finding All Match Parses

In numerous contexts one desires not only a substring of the text that matches the pattern, but also the precise way that subexpressions of R are matched. For example, it is common to find a notation in line-based editing commands for tagging subexpressions of a pattern so that whatever matched the tagged part may be used in forming a string to replace the overall match. Consider then matching a regular expression R in which any number of subexpressions may be enclosed in curly braces. Each subexpression so enclosed is said to be *tagged* and when we match R we desire not only the match to R but also to each tagged subexpression. For example, consider `{a*{b*}}(a+b)*` and its match to all of `aabbabab`. The match to the subexpression `a*b*` could be either `aabb` or ϵ. If the former, then the match to the tagged subexpression `b*` must be `bb`, otherwise it must be ϵ. That is, the substrings returned for the tagged subexpressions must be *consistent* with a given parse. The two possible *tag-matches* for the example are the *ordered* pairs $(aabb, bb)$, and (ϵ, ϵ). Another subtlety is that in some cases a match to one or more of the subexpressions may not occur in the match to the entire pattern. For example, `(a{ba}b|bb)*` matches `bb` without matching the designated sub-expression. Basically this can happen whenever a tagged subexpression is part of an alternation ('`|`') construct, a Kleene closure ('`*`') construct, or an option ('`?`') construct. We will call expressions where tags do not occur *within* such constructs, *unambiguous*, and make the distinction because such cases can be handled with greater efficiency. Finally, note that by using curly braces to denote tags we are assuming that tagged subexpressions either nest or are disjoint.

Assume that the substring B of the text that R is to match has been selected, so that we can hereafter consider matches of R to the entirety of B. Rather than develop a particular method for selecting a parse to R, let's consider the problem of determining the graph $\mathcal{M}_{<B,R>}$ of all possible paths through R that match all of B. Let $S(i) = S_f(i) \cap S_r(i) \cap S_\Sigma$ and let $E(i) = S_f(i) \cap S_r(i+1) \cap S_\epsilon$ where $S_\Sigma = \{s : \lambda_s \in \Sigma\}$ and $S_\epsilon = \{s : \lambda_s = \epsilon\}$. Let the vertices in $\mathcal{M}_{<B,R>}$ be the set of ordered pairs $\{(0, s) : s \in E(0)\} \cup \{(i, s) : i \in [0, N] \text{ and } s \in E(i) \cup S(i)\}$. There is an edge $(i, s) \to (j, t)$ in the graph if and only if $s \to t$ is a transition in F and either (1) $j = i + 1$, or (2) $j = i$ and $t \in S_\epsilon$. It follows easily that there are at most $O(PN)$ vertices and edges in the graph and that it can be computed in time linear in its size. Note that $\mathcal{M}_{<B,R>}$ is acyclic and every path from vertex $(0, \theta)$ to vertex (N, ϕ) models a match between B and R. Also, given the graph and a selected path from $(0, \theta)$ to (N, ϕ), it is a simple matter to deliver the tagged submatches.

Now we develop criteria for selecting a path through the graph $\mathcal{M}_{<B,R>}$. Suppose that the k subexpressions $\alpha_1, \alpha_2, \cdots \alpha_k$ have been tagged. For a consistent match to these subexpressions, say $(a_1, a_2, \cdots a_k)$, let $\Sigma|a_i|$ be the *extent* of the match and let the k-tuple of integers, $(|a_1|, |a_2|, \cdots, |a_k|)$, be its *footprint*. Two possible selection criteria for a path through the graph are (1) to find the one whose submatches give the largest extent, or (2) to find the one that has the lexicographically largest footprint. Below we will solve for both of these problems by giving an algorithm that, more generally, works with respect to any ranking $\succeq$ of fingerprints satisfying the following monotonicity property: for all fingerprints F, G, H, $F \succeq G$ implies $F \oplus H \succeq G \oplus H$ where $\oplus$ is component-by-component (vector) addition.

In the case where k properly nested but otherwise arbitrary subexpressions are tagged, one must keep track of the highest-ranking fingerprint to each vertex v of $\mathcal{M}_{<B,R>}$ that involves a particular subset of tagged subexpressions, C, whose right-ends have been seen, and a particular subset of tagged subexpressions, I, disjoint from C, whose left-ends have been seen, but not yet their right ends. We say the tagged subexpressions in C are *complete*, and those in I are *in-progress*. Formally, we keep track of $Best_{C,I}(v)$ for every choice of C and I and every vertex v, computing these quantities in a topological order of $\mathcal{M}_{<B,R>}$. In the algorithm outline of Figure 1 below we do not worry about whether a particular (C, I) is legitimate for a given vertex v, but simply use $-\infty$ to fill illegitimate components of a fingerprint. That is, if $x \in C$ but there does not exist a path to v whose projection onto F passes through x's subautomaton, then the x component of the candidate will be $-\infty$. Similarly, if $x \in I$ but there does not exist a path to v whose projection

```
for $v \equiv (i,s)$ a vertex in $\mathcal{M}_{<B,R>}$ in topological order do
{ for $I \in 2^{[1,k]}$ and $C \in 2^{[1,k]-I}$ do
    { $Best_{(C,I)}(v) \leftarrow < -\infty, -\infty, \ldots, -\infty >$
      for $w \rightarrow v$ in $\mathcal{M}_{<B,R>}$ do
          if $Best_{(C,I)}(w) \succeq Best_{(C,I)}(v)$ then
              $(Best_{(C,I)}(v), Trace_{(C,I)}(v)) \leftarrow (Best_{(C,I)}(w), w)$
    }
    for $x$ a tagged subexpr. starting at $s$ do
      for $I \in 2^{[1,k]-x}$ and $C \in 2^{[1,k]-I}$ do
        { $f \leftarrow Best_{(C,I)}(v)$
          $f[x] \leftarrow 0$
          $(Best_{(C-x,I+x)}(v), Trace_{(C-x,I+x)}(v)) \leftarrow (f, Trace_{(C,I)}(v))$
        }
    if $\lambda_s \neq \epsilon$ then
      for $I \in 2^{[1,k]}$, $C \in 2^{[1,k]-I}$, and $x \in I$ do
          $Best_{(C,I)}(v)[x] \leftarrow Best_{(C,I)}(v)[x] + 1$
    for $x$ a tagged subexpr. ending at $s$ do
      for $I \in x + 2^{[1,k]}$ and $C \in 2^{[1,k]-I}$ do
        { if $Best_{(C,I)}(v) \succeq Best_{(C+x,I-x)}(v)$ then
              $(Best_{(C+x,I-x)}(v), Trace_{(C+x,I-x)}(v)) \leftarrow (Best_{(C,I)}(v), Trace_{(C,I)}(v))$
          $Best_{(C,I)}(v)[x] \leftarrow -\infty$
        }
}
```

Figure 1: Optimal R.E. Parsing Algorithm.

onto F enters x's subautomaton but does not leave it, then the x component of the candidate will be $-\infty$. For each *Best* value the algorithm retains a trace value *Trace* recording which predecessor vertex gave rise to the best value, so that one can trace back a desired path at the end of the computation. If the time to compare fingerprints under $\succeq$ is $O(c)$ then the algorithm of Figure 1 takes $O(c4^kPN)$ time and space in the worst case as all 2-partitions, (C, I) of all subsets of $[1, k]$ are considered. Note that c is $O(k)$ if we seek the lexicographically largest footprint, and c is $O(1)$ if we seek the footprint with the largest extent.

The algorithm above can be significantly improved by observing that for a given regular expression and a given choice of k tagged subexpressions within it, the set of (C, I) pairs that are actually legitimate at some vertex in $\mathcal{M}_{B,R}$ is usually much less than 4^k. For example, for the expression `x(x{1}x{2}x|x{3}x)*x{4}x`, of the 81 2-partitions of subsets of $\{1,2,3,4\}$, only the following 22 pairs are legitimate at some vertex: $(\emptyset,\emptyset)$, $(\emptyset,\{1\})$, $(\{1\},\emptyset)$, $(\{1\},\{2\})$, $(\{1,2\},\emptyset)$, $(\emptyset,\{3\})$, $(\{3\},\emptyset)$, $(\{2\},\{1\})$, $(\{3\},\{1\})$, $(\{1,3\},\emptyset)$, $(\{1,3\},\{2\})$, $(\{1,2,3\},\emptyset)$, $(\{1,2\},\{3\})$, $(\{2,3\},\{1\})$, $(\emptyset,\{4\})$, $(\{1,2\},\{4\})$, $(\{3\},\{4\})$, $(\{1,2,3\},\{4\})$, $(\{4\},\emptyset)$, $(\{1,2,4\},\emptyset)$, $(\{3,4\},\emptyset)$, and $(\{1,2,3,4\},\emptyset)$. Moreover the number of legitimate pairs at each vertex of the graph is even smaller, and is maximal at vertices whose state is final for the automaton F. In our current example, a maximum of 4 pairs need to be computed at each vertex as the legal pairs at the final vertex of the automaton is $(\{4\},\emptyset)$, $(\{1,2,4\},\emptyset)$, $(\{3,4\},\emptyset)$, and $(\{1,2,3,4\},\emptyset)$.

Lemma 2 gives recurrences bounding the number of subset pairs required for a particular regular expression and tags. Let T_R denote the number of legitimate (C, I) pairs required for expression R, excluding the initial pair $(\emptyset,\emptyset)$. Simultaneously, we will need to compute recurrences for M_R, the number of legitimate $(C,\emptyset)$ pairs found at the final state of R (including the pair $(\emptyset,\emptyset)$), and C_R, which is 1 or 0 depending on whether there is or is not, respectively, a path through R's automaton not involving a tagged subexpression.

Lemma 2:
$$\begin{array}{rcl}
C_a &=& (\textbf{if } a \text{ is tagged } \textbf{then } 0 \textbf{ else } 1) \\
M_a &=& 1 \\
T_a &=& (\textbf{if } a \text{ is tagged } \textbf{then } 2 \textbf{ else } 0) \\[1ex]
C_{RS} &=& (\textbf{if } RS \text{ is tagged } \textbf{then } 0 \textbf{ else } \min(C_R, C_S)) \\
M_{RS} &=& M_R M_S \\
T_{RS} &=& T_R + M_R T_S + (\textbf{if } RS \text{ is tagged } \textbf{then } (1 + M_{RS})) \\[1ex]
C_{R|S} &=& (\textbf{if } R|S \text{ is tagged } \textbf{then } 0 \textbf{ else } \max(C_R, C_S)) \\
M_{R|S} &=& M_R + M_S - \min(C_R, C_S) \\
T_{R|S} &=& T_R + T_S + (\textbf{if } R|S \text{ is tagged } \textbf{then } (1 + M_{R|S})) \\[1ex]
C_{R*} &=& (\textbf{if } R* \text{ is tagged } \textbf{then } 0 \textbf{ else } 1) \\
M_{R*} &=& 2^{M_R - C_R} \\
T_{R*} &\leq& 2^{M_R - C_R}(\frac{3}{4}T_R - (M_R - C_R) + 1) - 1 + (\textbf{if } R* \text{ is tagged } \textbf{then } (1 + M_{R*})) \\[1ex]
C_{R+} &=& (\textbf{if } R+ \text{ is tagged } \textbf{then } 0 \textbf{ else } C_R) \\
M_{R+} &=& 2^{M_R - C_R} - (1 - C_R) \\
T_{R+} &\leq& 2^{M_R - C_R}(\frac{3}{4}T_R - (M_R - C_R) + 1) - 1 + (\textbf{if } R+ \text{ is tagged } \textbf{then } (1 + M_{R+})) \\[1ex]
C_{R?} &=& (\textbf{if } R? \text{ is tagged } \textbf{then } 0 \textbf{ else } 1) \\
M_{R?} &=& M_R + (1 - C_R) \\
T_{R?} &=& T_R + (\textbf{if } R? \text{ is tagged } \textbf{then } (1 + M_{R?}))
\end{array}$$

While the lemma gives recurrences for bounding the size of legitimate sets, it is a simple step to extend them to recurrences for enumerating the legitimate sets at each vertex with a given state of F, in a prepass over F. The prepass takes $O(T_R)$ time and it can be shown that the maximum at any state is given by M_R. It is thus possible to modify the coarse algorithm above to only compute the legitimate pairs at each vertex, giving an $O(cM_R PN + T_R)$ time and space algorithm. A simple corollary is that if the tags are unambiguous, then there are at most $O(1)$ legitimate pairs at each vertex and T_R is $O(k)$. So the refined algorithm takes only $O(cPN)$ time in this case.

3 Reporting Approximate Matches

3.1 Finding the Most Significant Approximate Match

We now consider some problems in *approximate* regular expression pattern matching. A k-match to a regular expression R is a string whose minimal distance from a string exactly matching R is k, where distance is the standard unit-cost difference metric. The most significant issue in this context is what constitutes a match. For example, consider the expression `babb*bab` and suppose we are searching for all matches with 2-or-less differences to it, i.e., a *2-match*. When run against the text ...`aba|babbbab|aaa`... there is a 0-match to R shown between bars, but there are also 12 induced 2-matches in the vicinity that can be obtained with insertions and deletions at either end of the 0-match, i.e., `bababbbab`, `ababbba`, `ababbbab`, `ababbbaba`, `babbb`, `babbba`, `babbbaba`, `babbbabaa`, `abbba`, `abbbab`, `abbbaba`, and `bbbab`. Indeed, wherever there is a 0-match to the pattern there will *always* be another 12 2-matches *induced* by it. In addition, just by fortuitous circumstance, there can be additional overlapping matches, e.g., `abababbbab` in the example. Here we propose two schemes, first one for filtering out the induced matches, and then one for filtering the fortuitous matches.

It will be convenient in the ensuing treatment to think about alignments in terms of paths through an *edit graph* between A and the pattern R. Basically the edit graph $\mathcal{G}_{<A,R>}$ is just the dependency graph of the recurrence (1) with each edge weighted according to the δ-part of its recurrence term. Specifically, there is a vertex for each term (i, s), an insertion edge from (i, t) to (i, s) weighted $\delta(\epsilon, \lambda_s)$ for every edge $t \rightarrow s$ in F, a deletion edge from $(i-1, s)$ to (i, s) weighted $\delta(a_i, \epsilon)$, and a substitution edge from $(i-1, t)$ to (i, s) weighted $\delta(a_i, \lambda_s)$ for every edge $t \rightarrow s$ in F. By construction every path from (i, s) to (j, t) models an alignment between $a_{i+1}a_{i+2}\ldots a_j$ and a string spelled on the projection path from s to t in F excluding the first symbol on s. Moreover, the weight of the path is the score of the alignment it models. Thus in general the value $C(i, s)$ is the score of the least cost path to (i, s) from a θ-vertex of $\mathcal{G}_{<A,R>}$ (i.e. a vertex (j, θ) for some j).

3.1.1 Filtering Non-Essential Matches

In our first approach, we consider an alignment *essential* if (1) it begins and ends with aligned symbols (they need not be equal), and (2) the alignment has the lowest possible score of all alignments between the two strings involved. Note immediately, that none of the induced matches in the example above constitute essential matches. Also note that condition (2) is important: for example, there is a 2-alignment between `babb*bab` and `ababbbab` that begins and ends with aligned symbols, but there is also a 1-alignment that begins with an insertion. It is not difficult to prove that wherever there is a non-essential match, there is also at least one essential match. Thus every matching region will be reported when one restricts attention to just the essential matches. One must be careful to add a sentinel character at each end of the string A being searched in order that matches involving its ends be found.

Let a state, s, of F be termed θ-reachable if there is a path from θ to s all of whose states are labeled ε including s. Further let Θ be the set of θ-reachable states. With this definition we may then develop the recurrences below for $B(i, s)$ and $E(i, s)$ which are the best score of a path in $\mathcal{G}_{<A,R>}$ from a θ-vertex to (i, s) that (1) begins with aligned symbols, or (2) begins and ends with aligned symbols, respectively. Essentially the recurrences are exactly that for $C(i, s)$ save that certain edges in $\mathcal{G}_{<A,R>}$ are not permitted.

Lemma 3:

$$B(i,s) = \min \left\{ \begin{array}{c} \min\limits_{t \rightarrow s}\{B(i-1,t) + \delta(a_i, \lambda_s)\}, \\ \min\limits_{t \rightarrow s}\{B(i,t) + \delta(\epsilon, \lambda_s)\}, \\ B(i-1,s) + \delta(a_i, \epsilon), \\ \delta(a_i, \lambda_s) \ \ \textbf{if}\ t \rightarrow s \in \Theta \times \bar{\Theta} \\ \infty \end{array} \right\} \qquad E(i,s) = \min \left\{ \begin{array}{c} \min\limits_{t \rightarrow s : \lambda_s \neq \varepsilon}\{B(i-1,t) + \delta(a_i, \lambda_s)\}, \\ \min\limits_{t \rightarrow s : \lambda_s = \varepsilon} E(i,s) \\ \infty \end{array} \right\}$$

It follows that $E(i, \phi)$ is the score of a best alignment beginning and ending with aligned symbols between a suffix of $A[1..i]$ and a word in R. We can thus report as the left end of an essential k-match only those i for which $E(i, \phi) \leq k$ *and* $E(i, \phi) = C(i, \phi)$ is true.

3.1.2 Filtering Fortuitous Matches

While our first attempt at defining "true" matches removes the potential $O(k^2)$ induced matches, it does not distinguish or eliminate *fortuitous* matches which occur because by chance there is another way to complete the beginning or tail portion of a "match", e.g. `bababbbab` in our running example. For a given finite alphabet Σ, one can computationally approximate the limit:

$$r_\Sigma = \lim_{n\to\infty} E[\mathit{diff}(A,P)/|P| : A,\ P \text{ chosen uniformly from } \Sigma^n]$$

where $\mathit{diff}(A,P)$ is the score of the best alignment between A and P. Intuitively this is the number of differences per unit alignment length one expects to see in a "random" match. Any match with a lower *difference ratio* can be considered to be significant. Utilizing this, in our search for a good definition of an interesting match, let a *significant* alignment between two sequences be one for which the difference ratio of every prefix and suffix of the match is less than r_Σ. Intuitively, every "extension" of the match is significant. In the early 1980's Sellers [Sel84] explored algorithms for finding such matches in the context of molecular biology. This work appears to have been forgotten in the wake of the current popularity of the Smith-Waterman algorithm [SW81].

Sellers' basic idea is as follows. Suppose scoring is with respect to a general additive scoring scheme δ, and suppose one wants to detect only matches for which $\delta(A,P)/|P| \leq r$. Sellers observed that this is equivalent to finding matches for which $\delta_r(A,P) \geq 0$ where δ_r is a maximization scoring scheme derived from δ as follows: $\delta_r(a,b) = -\delta(a,b)$ if $b = \epsilon$, and $\delta_r(a,b) = r - \delta(a,b)$ if $b \neq \epsilon$. That this holds follows easily from the fact that δ_r has been constructed so that $\delta_r(A,P) = r|P| - \delta(A,P)$.

It now follows that what we seek are paths in the edit graph of A versus R that begin at a θ-vertex, end at a ϕ-vertex, and are both *prefix and suffix positive* under the weighting supplied by δ_r. A prefix positive path is one for which the score on every prefix of the path is positive. A suffix positive path is similarly defined. A basic exercise gives the following recurrence for $\mathit{PreP}(i,s)$ which is true if and only if there is a prefix positive path to (i,s).

Lemma 4: $$\mathit{PreP}(i,s) = \begin{cases} \mathit{or} \left\{ \begin{array}{l} \mathit{or}_{t\to s}\{\mathit{PreP}(i-1,t) \text{ and } (C(i-1,t) + \delta_r(a_i,\lambda_s) > 0)\}, \\ \mathit{or}_{t\to s}\{\mathit{PreP}(i,t) \text{ and } (C(i,t) + \delta_r(\epsilon,\lambda_s) > 0)\}, \\ \mathit{Pre}(i-1,s) \text{ and } (C(i-1,s) + \delta_r(a_i,\epsilon) > 0) \end{array} \right\} & \text{if } s \neq \theta \\ \mathit{true} & \text{if } s = \theta \end{cases}$$

Given the vertices on prefix positive paths, one can quickly infer the edges on such paths (i.e., $v \to w$ is on a prefix positive path if $\mathit{PreP}(v)$ is true and $C(v) + \delta_r(v \to w) > 0$). The suffix positive vertices and edges can similarly be found by developing recurrences for $\mathit{SufP}(i,s)$ over the reverse of R and A. By taking the set of vertices and edges that are on both prefix and suffix positive paths, one arrives at the subgraph $\mathcal{S}_{<A,R>}$ of $\mathcal{G}_{<R,A>}$ modeling all paths that are prefix and suffix positive, or equivalently all the significant matches, according to our definition of significant. One can compute $\mathcal{S}_{<A,R>}$ in $O(PN)$ time and space.

Now one may wish to report matches that are both essential and significant. While it is the case that a significant essential match occurs where ever a significant non-essential match is found, it is not true that an essential match is necessarily significant, or for that matter, that a k-match is significant. Consider then first computing the subgraph, $\mathcal{E}_{<R,A>}$ of all vertices and edges on essential matches. Intersecting this subgraph with $\mathcal{S}_{<R,A>}$ leads to a subgraph whose connected components may not include a θ- or ϕ-vertex, corresponding intuitively to a region where there is an essential match that is not significant. For such components, we suggest that one might either find the least cost extensions that reach a θ- and ϕ-vertex, or that one recompute $\mathcal{S}$ with increasing values of r until the intersection does admit an end-to-end path. This second approach yields an interesting subproblem in parametric dynamic programming that we leave open.

The computation of $\mathcal{E} \cap \mathcal{S}$ can be efficiently organized as we noted earlier in Section 2.2, where the grain or direction of two recurrences oppose each other, as do the recurrences for *PreP* and *SufP* here. With the method of [MJ96] we can compute $\mathit{SufP}_R(0)$, $\mathit{SufP}_R(1)$, ... $\mathit{SufP}_R(N)$ in the order given using $O(tPN)$ time and $O(PN^{1/t})$ space for any choice of $t \geq 1$, where $\mathit{SufP}_R(i)$ is the set of values $\{\mathit{SufP}(i,s) : s \in F\}$. Given this we may then simultaneously compute the intersection of C, B, E, and PreP with SufP in a single forward scan, in time and space dominated by the terms for delivering SufP against its grain. In particular, this gives us an $O(PN \log N)$ time, $O(P \log N)$ space algorithm for delivering the significant portion of an essential match of R to substrings of A.

3.2 Reporting Longest Matches and All Matches

We conclude, by sketching solutions for the longest-match and all-matches problems for the case of approximate matching. To compute the longest essential match, we need only augment the computation of $\mathcal{E}_{<R,A>}$ with a trace-back record of a minimum inducing predecessor that has the longest match achieving that minimum. Note that what we are doing is simply delivering the longest match achieving the minimum. If rather we want the longest match that is within the threshold k then this requires that we keep track of the longest solution with each of the scores in $[0,k]$. The additional complexity for doing so is $O(kPN)$ time and $O(kP)$ space. One can then further combine this with our result for finding the significant part of essential matches, resulting in an $O(PN(k + \log N))$ time and $O(P(k + \log N))$ space algorithm for finding the longest significant part.

Computing all match positions is straightforward. Simply compute $C(i,s)$ in the forward direction over A and also compute $C^r(i,s)$ in the reverse direction of A and R. Report all positions i for which there exists a state t for which $C(i,t) + C^r(i,t) \leq k$. Again this requires the simultaneous delivery of recurrences opposing each other, and can be solved with the by now, well understood, complexities.

References

[CC97] C.A. Clarke and G.V. Cormack. On the use of regular expressons for searching text. *ACM Trans. on Prog. Languages and Systems*, 19(3):413–426, 1997.

[IEE92] IEEE. *Portable Operating System Interface (POSIX).* IEEE Std 1003.2, Inst. of EE Engineers, New York, 1992.

[MJ96] E. Myers and M. Jain. Going against the grain. In Carleton University Press, editor, *Proc. 3rd South American Workshop on String Processing*, International Informatics Series #4, pages 203–213, 1996.

[MM88] E. Myers and W. Miller. Approximate matching of regular expressions. *Bulletin of Mathematical Biology*, 51(1):5–37, 1988.

[Ous94] J.K. Ousterhout. *Tcl and the TK Toolkit.* Addison-Wesley, Reading, Mass., 1994.

[Sed83] R. Sedgewick. *Algorithms.* Addison-Wesley, Reading, Mass., 1983.

[Sel84] P.H. Sellers. Pattern recognition in genetic sequences by mismatch density. *Bulletin of Mathematical Biology*, 46:501–514, 1984.

[SW81] T.F. Smith and M.S. Waterman. Identification of common molecular sequence. *J. of Molecular Biology*, 147:195–197, 1981.

[Tho68] K. Thompson. Regular expression search algorithm. *Comm. of ACM*, 11(6):419–422, 1968.

[WS91] L. Wall and R.L. Schwartz. *Programming Perl.* O'Reilly and Associates, Sebastopol, Calif., 1991.

Appendix A: ε-Cycle Free Automata.

We quickly show here the inductive construction of an ε-NFA for a regular expression that has $O(P)$ states and vertices and does not contain an ε-cycle. To this end, we need the predicate $Nil(r)$ that is true if and only if ε is a word in the language specified by regular expression r. The following recurrence for Nil follows easily by induction:

$$\begin{aligned}
Nil(a) &\equiv (a \notin \Sigma) \\
Nil(r^*) &\equiv \textbf{true} \\
Nil(r^+) &\equiv Nil(r) \\
Nil(r?) &\equiv \textbf{true} \\
Nil(rs) &\equiv Nil(r) \textbf{ and } Nil(s) \\
Nil(r|s) &\equiv Nil(r) \textbf{ or } Nil(s)
\end{aligned}$$

Given the Nil predicate for each subexpression of a regular expression r, we construct the ε-cycle free automata F for it as shown in Figure 2. The dashed edges labeled *"if Nil(?)"* are to be placed in the construction only if the predicate is true. The induction of the construction is that the machine built for expression r is one that accepts all words in r *except* for ε, if it happens to be matched by r. In the very last step of constructing F, we add a path accepting ϵ if r accepts ϵ.

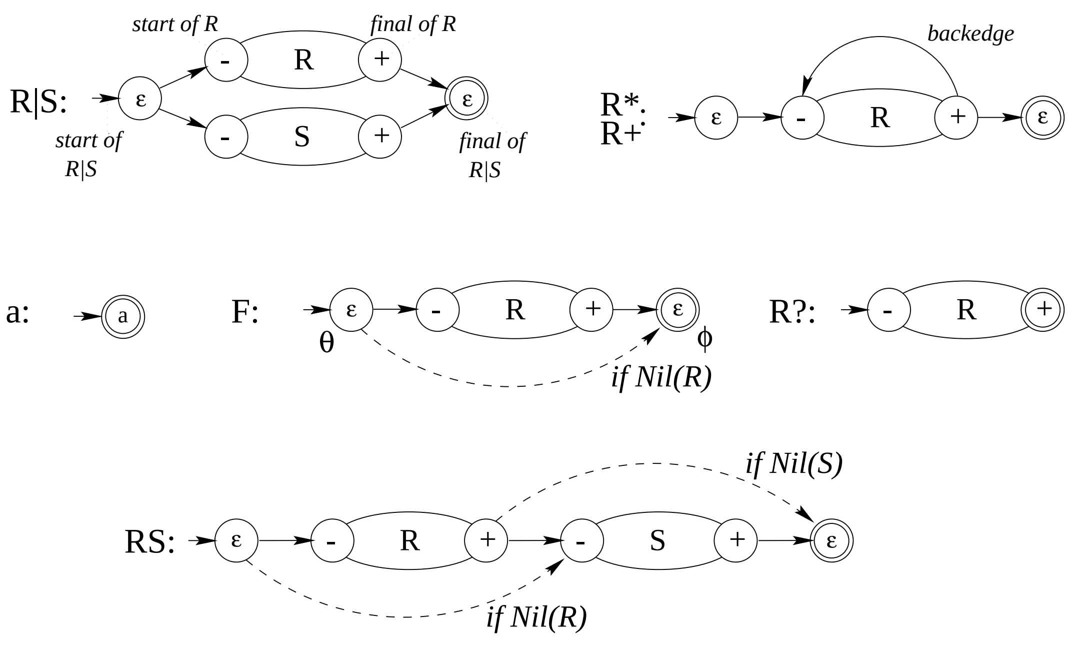

Figure 2: Inductive RE to ε-cycle free NFA Construction.