Approximately Matching Context-Free Languages[†]

Gene Myers

TR 94-22

ABSTRACT

Given a string w and a pattern p, approximate pattern matching merges traditional sequence comparison and pattern matching by asking for the minimum difference between w and a string exactly matched by p. We give an $O(PN^2(N+logP))$ algorithm for approximate matching between a string of length N and a context free language specified by a grammar of size P. The algorithm generalizes the Cocke-Younger-Kasami algorithm for determining membership in a context free language. We further sketch an $O(P^5N^88^P)$ algorithm for the problem where gap costs are concave, and pose two open problems for such general comparison cost models.

June 27, 1994

Department of Compter Science
The University of Arizona
Tucson, AZ 85721

[†] Partially supported by NLM Grant LM-04960 and the Aspen Center for Physics.

Approximately Matching Context-Free Languages

0. Introduction.

The problems of comparing two strings [NeW70, San72, WaF74, Ukk85, Mye86] and of finding all matches to a pattern in a text [Tho68, AhC75, McC76, BoM77, KaR87] have been extensively studied. Combining the notions of sequence comparison and pattern matching lead to a natural formulation of *approximate pattern matching*. Namely, given pattern p, text t, and threshold τ, find all substrings of the text that can be aligned to some string exactly matched by p with score not greater than τ. This problem has two major dimensions of variation: (1) the pattern class over which p is chosen, and (2) the notion of sequence comparison used to assess the difference between two strings. Some of the simplest variations of this problem have already been extensively studied. For example when p is a string and the measure of difference is the number of unaligned symbols and mismatched pairs, we have the approximate string matching problem [LaV89, CL90, GaP90, Mye90, Ukk92, WuM92, WMM94]. In applications such as molecular biology, one must more generally compare sequences using arbitrarily weighted mismatch costs [DBH83], and often one must use affine or concave costs for gaps of unaligned symbols [FiS83, Wat84, MiM88, GaG89].

Comparatively little work has been done for choices of p other than a simple string. Myers and Miller [MyM89, Mye92] considered the problem for p a regular expression, Sankoff and Kruskal [SaK83] considered the case of regular expressions without Kleene closure, and Akutsu [Aku94] has considered the case of keywords with "wildcard" symbols. Earlier work was done on what was thought of at the time as *error correction* problems: given a pattern p, say a context free language, and a string t, what is the minimal number of corrections that transform t into a string in the language accepted by p? A correction corresponds to a mismatch or unaligned symbol in our comparison model and the problem is phrased in terms of testing membership rather than performing a search. These differences are inconsequential, but it is the case that all work on "order correction" was done with unit costs, i.e. they were only interested in the number of corrections required. Wagner and Seiferas [WaS78] presented order correction algorithms for regular expressions and counter automata. Myers and Miller [MyM89] latter developed a much simpler $O(PN)$ algorithm for approximately matching regular expressions under arbitrarily weighted mismatch and unaligned symbol costs, and Knight and Myers [KnM92] have developed an $O(PN(log^2P+logN))$ algorithm that permits concave gap costs. Recall from the abstract that P is the length of the pattern p and N is the length of the text t.

This note concerns itself with the problem of approximately matching context free languages. We develop a generalization of the exact matching, dynamic programming algorithm of Cocke, Younger, & Kasami [Kas63, You67] that realizes an $O(PN^2(N+logP))$ algorithm for approximate matching under a weighted comparison model. Earlier relevant work is by Aho & Peterson [AhP72] and Lyons [Lyo74] on order correction for context free languages for which they devised a generalization of Early's algorithm [Ear70] to arrive at $O(N^3)$ algorithms. However, their results are specific and applicable only to the unit cost comparison model and they do not consider the complexity of their methods in terms of grammar size which as presented is $O(P^2)$. We further show that our algorithm extends to affine gap-cost comparison models but that when concave gap-costs are to be modeled an appeal to the pumping lemma [HoU79, pp. 125-127] seems necessary and leads to an $O(P^5N^88^P)$ algorithm that is exponential in grammar size (but not in N).

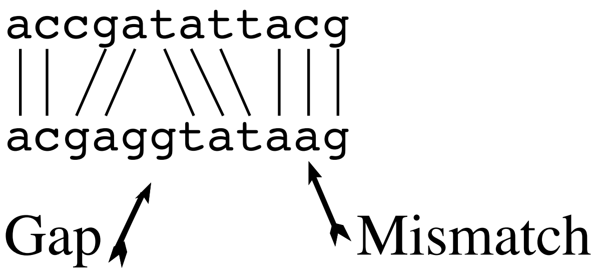

Figure 1: An alignment.

1. Preliminaries.

An *alignment* between two strings $w = a_1a_2\cdots a_N$ and $v = b_1b_2\cdots b_M$ is a set of pairs of aligned symbols such that if one were to draw lines between aligned symbols they would not cross as in Figure 1. Formally, alignment T $= \{ (i_1,j_1),(i_2,j_2),\cdots(i_L,j_L) \}$ aligns symbols a_{i_k} and b_{j_k} for all k, and satisfies the property that $i_k < i_{k+1}$ and $j_k < j_{k+1}$ for all $k < L$. Typically alignments are assigned scores according to the pairs of symbols they align/mismatch and the substrings of consecutive symbols, or *gaps*, they leave unaligned. We will assume as input to our problems a function δ that assigns a real-valued score, $\delta(a,b)$, to each aligned/mismatched pair and a score $\delta(u)$ to each gap $u \in \Sigma^+$. Under this scheme the score of alignment T is:

$$\sum_{k=1}^{L}\delta(a_{i_k},b_{i_k}) + \sum_{k=0}^{L}\delta(a_{i_k+1}a_{i_k+2}\cdots a_{i_{k+1}-1}) + \sum_{k=0}^{L}\delta(b_{j_k+1}b_{j_k+2}\cdots b_{j_{k+1}-1})$$

where for simplicity it is assumed that $i_0 = j_0 = 0$, $i_{L+1} = N+1$, $j_{L+1} = M+1$, and $\delta(\varepsilon) = 0$. Usually $\delta(a,a) = 0$ but this need not be the case. When comparing strings w and v, we seek an alignment of minimal score and call the minimal score, $\delta(w,v)$, the difference between w and v.

We have thus far left the manner in which δ scores gaps more general than it is usually treated in the literature. Typically gaps are attributed scores according to (1) their length, i.e. $\delta(c_1c_2\cdots c_h) = gap(h)$, or (2) as the sum over the symbols in the gap of scores attributed to leaving each symbol unaligned, i.e., $\delta(c_1 c_2 \cdots c_h) = \sum_{k=1}^{h} \sigma(c_k)$. This paper considers the later, symbol-dependent model in its central algorithm, and a number of versions of the length-dependent model in its final section. As a concrete example, the approximate string matching and order correction problems concern the scoring scheme:

$$\delta(a,b) = \begin{cases} 1 & \text{if } a \neq b \\ 0 & \text{if } a = b \end{cases} \quad \text{and} \quad \delta(c_1 c_2 \cdots c_h) = h. \tag{1.1}$$

Note that for this δ, gap scoring is an instance of both the length- and symbol-dependent models. While this simple scheme is useful in information retrieval and text processing contexts, in other applications such as comparing protein sequences, producing meaningful alignments requires the more general weighted models above.

Given a pattern p, a string w, and a scoring scheme δ, the difference $\delta(p,w)$ between p and w, is $\min\{\delta(v,w) : v \in L(p)\}$ where $L(p)$ is the set of strings exactly matched by p, also called the language accepted by p. In this note we focus on the basic approximate matching problem of computing the difference between a pattern and a string, analogous to the basic problem of determining membership in the language of a pattern. Algorithms for variations such as determining a string v in $L(p)$ realizing the difference and its alignment to w follow directly from the algorithm for the basic problem. It is also the case that our basic algorithm for context free languages readily solves the searching variation where one is searching a text t for all substrings whose difference from p is not greater than some threshold.

Let $G = \langle V, S, \Sigma, \Pi \rangle$ be a context free grammar where V is the set of *non-terminals*, $S \in V$ is the *sentence symbol*, Σ is the *terminal alphabet*, and $\Pi \subset V \times (V \cup \Sigma)^*$ is the *production set*. We write $\alpha \Rightarrow \beta$ if β can be obtained from α with the application of one production of Π, and $\alpha \Rightarrow^* \beta$ if β can be obtained via a chain of zero or more rewritings from Π,. In all three cases we say α *derives* β. The language accepted by G, $L(G)$ is the set of all $w \in \Sigma^*$ such that $S \Rightarrow^* w$, i.e., the set of all terminal strings derivable from S. We assume without loss of generality that G does not contain any useless productions [HoU79, pp. 88-90].

We consider the size, P, of a grammar to be, $\Sigma_{A \to \alpha}(|\alpha|+1)$, the sum of the lengths of all productions. For reasons of both simplicity and algorithmic complexity we must first transform this grammar into Chomsky Normal Form (CNF) [HoU79, pp. 92-94] save for the removal of unit productions. Recall that the first step of the transformation sequence replaces each occurrence of a terminal symbol a with a new non-terminal C_a in right-hand sides of length two or more, and adds a new production $C_a \to a$ to the grammar. In the second step one transforms each production $A \to B_1B_2\ldots B_k$ with $k > 2$ into the $k-1$ productions: $A \to B_1C_1$, $C_1 \to B_2C_2$, $\cdots$ $C_{k-2} \to B_{k-1}B_k$where $C_1, C_2, \cdots C_{k-2}$ are new, unique non-terminals added to the grammar. Finally, we determine all non-terminals that can derive the empty string (see [Hou79, pp. 90-92]), remove all ε-productions (i.e. productions of the form $A \to \varepsilon$), add $S \to \varepsilon$ if S can derive the empty string, and for every $A \to BC$ we add the production $A \to B$ ($A \to C$) if C (B) can derive the empty string. This leaves us with an equivalent transformed grammar $G' = < V', S, \Sigma, \Pi' >$ whose size is still *O(P)*, but for which every production is of the form $A \to BC$, $A \to B$, or $A \to a$ where $A,B,C \in V$ and $a \in \Sigma$, save for the one production $S \to \varepsilon$ iff $\varepsilon \in L(G)$. We call this form *pseudo-CNF* to emphasize that we have not removed unit productions (i.e. productions of the form $A \to B$) from the grammar. These are normally removed as a last step in the usual CNF transformation, but we do not do so as their removal can create a grammar of size *O(P*^{2}*)*.

2. The Central Algorithm.

Our central result is an algorithm for computing $\delta(G,w)$ where G is a context free grammar, $w = a_1a_2\cdots a_N$ is a string of length N, and δ is a weighted and symbol-dependent scoring scheme with non-negative gap costs. That is, $\delta(c_1c_2\cdots c_h) = \sum_{k=1}^{h}\sigma(c_k)$, and $\delta(a,b)$ and $\sigma(c) \ge 0$ are real numbers. Moreover, it is assumed without loss of generality that G is in pseudo-CNF form as described immediately above.

Our algorithm for approximately matching context free languages extends the dynamic programming algorithm of Cocke-Younger-Kasami [Kas63, You67] that computes the set of non-terminals that can derive every substring $w_{i\cdots j} = a_ia_{i+1}\cdots a_j$ of w. Their algorithm can be viewed as computing a predicate, $D(A,i,j) \equiv (A \Rightarrow^* w_{i\cdots j})$, for all $A \in V$ and $0 \le i-1 \le j \le N$, that is true if and only if A can derive $w_{i\cdots j}$. Note that in the boundary case where $j = i-1$ we are assuming that $w_{i\cdots i-1} = \varepsilon$. In direct analogy our algorithm computes a value $C(A,i,j)$for the *O(PN*^{2}*)* choices of A, i, and j that is the score of the best alignment between a non-empty string derivable from A and the substring $w_{i\cdots j}$. Formally:

$$C(A,i,j) = \min\{\, \delta(v, w_{i\cdots j}) : A \Rightarrow^* v \neq \varepsilon \,\}.$$

The recurrence we are about to develop for these C-values generalize those for the D-predicates of the Cocke-Younger-Kasami algorithm and are unusual in that they form a cyclic system of equations that preclude a straightforward application of dynamic programming.

Theorem 1:
$$
\begin{array}{rll}
C(A,i,j) \;=\; \min\{ & \min\limits_{A\to BC,k\in[i-1,j]} C(B,i,k)+C(C,k+1,j)\,, & (2.1)\\
& \min\limits_{A\to B} C(B,i,j)\,, & \\
& \min\limits_{A\to a,k\in[i,j]} \delta(a,a_k)+G(i,j)-\sigma(a_k)\,, & \\
& \min\limits_{A\to a} \sigma(a)+G(i,j) & \\
& \} &
\end{array}
$$

where $G(i,j) \;\equiv\; g(j)-g(i-1)$ and $g(i) = \begin{cases} 0 & \text{if } i=0 \\ g(i-1)+\sigma(a_i) & \text{if } i>0 \end{cases}$

Proof: First we show $C(A,i,j)$ is not greater than the right-hand side. Suppose for some $k\in[i-1,j]$ that $C(B,i,k) = \delta(t,w_{i\cdots k})$ where $B\Rightarrow^* t$, and $C(C,k+1,j) = \delta(u,w_{k+1\cdots j})$ where $C\Rightarrow^* u$. If $A\to BC\in\Pi$ then $A\Rightarrow BC\Rightarrow^* tu$, and so $C(A,i,j) \;\le\; \delta(tu,w_{i\cdots j}) \;\le\; \delta(t,w_{i\cdots k})+\delta(u,w_{k+1\cdots j}) \;=\; C(B,i,k)+C(C,k+1,j)$. Similarly, if $A\to B\in\Pi$ then one can conclude $C(A,i,j) \;\le\; C(B,i,j)$. Next, it is easy to see that $g(i) \;=\; \Sigma_{h\le i}\sigma(a_h)$ and thus that $G(i,j) \;=\; \Sigma_{i\le h\le j}\sigma(a_h) \;=\; \delta(\varepsilon,w_{i\cdots j})$, the cost of leaving $w_{i\cdots j}$ unaligned. So if $A\to a\in\Pi$ for $a\in\Sigma$ then one can conclude $C(A,i,j) \;\le\; \delta(a,w_{i\cdots j})$, but by the additivity of δ it follows that $\delta(a,w_{i\cdots j})$ is the minimum of either $\delta(a,\varepsilon)+\delta(\varepsilon,w_{i\cdots j}) \;=\; \sigma(a)+G(i,j)$ or $\min_k\, \delta(\varepsilon,a_i\cdots a_{k-1})+\delta(a,a_k)+\delta(\varepsilon,a_{k+1}\cdots a_j) \;=\; \min_k\, \delta(a,a_k)+G(i,j)-\sigma(a_k)$. We do not need to consider the possible production $S\to\varepsilon$ as A must derive a non-empty string. Thus $C(A,i,j)$ is not greater than any term in the minimum of the right-hand side.

For the converse, suppose that $A\Rightarrow^* v\ne\varepsilon$ and $C(A,i,j) = \delta(v,w_{i\cdots j})$. Since there must be some first step in the derivation of v from A, one of the following must be true: (1) $A\Rightarrow BC\Rightarrow^* v$, (2) $A\Rightarrow B\Rightarrow^* v$, or (3) $A\Rightarrow a = v$. If (1) is true then there are strings t and u such that $v = tu$, $B\Rightarrow^* t$, and $C\Rightarrow^* u$. Moreover, there must be some $k^*\in[i-1,j]$ such that $\delta(v,w_{i\cdots j}) \;=\; \delta(t,w_{i\cdots k^*})+\delta(u,w_{k^*+1\cdots j})$. It then follows that $C(A,i,j) \;=\; \delta(v,w_{i\cdots j}) \;=\; \delta(t,w_{i\cdots k^*})+\delta(u,w_{k^*+1\cdots j}) \;\ge\; C(B,i,k^*)+C(C,k^*+1,j)$. Similarly, if (2) or (3) are true then $C(A,i,j)\ge C(B,i,j)$ or $C(A,i,j)\ge\delta(a,w_{i\cdots j})$, respectively. Thus $C(A,i,j)$ must be greater or equal to at least one of the terms in the minimum of the right-hand side. ■

The recurrence of Theorem 1 expresses $C(A,i,j)$ in terms of other C-values for which $j-i$ is smaller except when i and j remain unchanged in the unit-production minimum of (2.1) and more subtly when $k = i-1$ or $k = j$ in the first minimum of (2.1). Thus $C(A,i,j)$ may potentially depend on itself since in a recursive grammar A can derive a string containing A. This is even more evident in the boundary case where $j = i-1$. First note that $C(A,i,i-1) \;=\; G(A)$ for all values of i, where $G(A) \;=$

$\min\{\,\delta(v,\varepsilon) : A \Rightarrow^{*} v \neq \varepsilon\,\}$. Thus a non-asymptotic efficiency gain is possible in the algorithm to follow by computing $G(A)$ once instead of $N+1$ times. Specializing (2.1) to $G(A)$ yields the recurrence:

$$G(A) = \min\{\min_{A \to BC} G(B) + G(C), \min_{A \to B} G(B), \min_{A \to a} \sigma(a)\} \tag{2.2}$$

whose cyclic dependencies are quite evident. This complication did not arise in the algorithm designed by Cocke, Younger, and Kasami as (1) they assumed the grammar *G* was in strict CNF form and hence did not have unit productions, and (2) one does not need to consider $k = i - 1$ or $k = j$ in the first minimum of (2.1) because *B* or *C* cannot *match* the empty string (whereas they can approximately match it in our case). While unit productions could be removed, we do not do so because it would result in an algorithm whose complexity in *P* would be $O(P^2)$ as noted in the preliminaries.

Using the dynamic programming paradigm, the algorithm of Figure 2 below computes a table, $C[A,i,j]$, of the *C*-values in increasing order of $len = j - i + 1$. Thus when one is about to compute *C*-values for a given *i* and *j* in lines 4-12, one can assume that $C(B,g,h)$ has been correctly computed and stored in $C[B,g,h]$ for all B, g, and h such that $h - g < j - i$. So it is only the terms of recurrence (2.1) *not* in:

$$known(A,i,j) = \min \left\{ \begin{array}{c} \min\limits_{A \to BC,\, k \in [i,j-1]} C[B,i,k] + C[C,k+1,j]\,, \\ \min\limits_{A \to \kappa} \sigma(\kappa) + G(i,j)\,, \\ \min\limits_{A \to a,\, k \in [i,j]} \delta(a,a_k) + G(i,j) - \sigma(a_k) \end{array} \right\} \tag{2.3}$$

that have not been computed. Namely, $C(A,i,j)$ depends on $C(B,i,j)$ iff $A \to BC$, $A \to CB$, or $A \to B$. Consider the following weighted graph model of these dependencies. There is a vertex for each non-terminal in *G* and a special source vertex ϕ. There is an edge from ϕ to every vertex *A* whose weight is $known(A,i,j)$. Also there is an edge of weight 0 from *B* to *A* iff $A \to B$ and an edge from *B* to *A* with weight *G(C)* iff $A \to BC$ or $A \to CB$. It follows directly from the construction that the value of $C(A,i,j)$ is the weight of a shortest path from ϕ to *A* in this graph. Moreover, since σ is positive, *G(A)* is positive for every non-terminal, and so every edge weight is positive. Thus Dijkstra's shortest path algorithm [Dij59] may be applied directly as detailed in lines 4-12 of Figure 2.

```
0.   Var C: array [V,1..N+1,0..N] of real
1.   For $len \leftarrow 0$ to $N$ do
2.     For $i \leftarrow 1$ to $N - len + 1$ do
3.       { $j \leftarrow i + len - 1$
4.         For $A \in V$ do
5.           $C[A,i,j] \leftarrow known(A,i,j)$
6.         $H \leftarrow$ heap of $V$ (ordered by $C[?,i,j]$)
7          While $H \neq \varnothing$ do
8.           { $A \leftarrow$ extract_min($H$)
9.               For $B \in H$ and ($B \to AC \in \Pi$ or $B \to CA \in \Pi$) do
10.                { $C[B,i,j] \leftarrow \min\{C[B,i,j], C[A,i,j] + G(C)\}$; reheap($H$,$B$) }
11.              For $B \in H$ and $B \to A \in \Pi$ do
12.                { $C[B,i,j] \leftarrow \min\{C[B,i,j], C[A,i,j]\}$; reheap($H$,$B$) }
13.          }
14.      }
15.  Print "$\delta(G,w)$ is" $\min\{C[S,1,N], G(0,N)$ if $S \to \varepsilon\}$
```

Figure 2: *$O(PN^2(N+logP))$* algorithm for computing $\delta(G,w)$.

Theorem 2: After taking *$O(PN^2(N+logP))$* time the algorithm of Figure 2 concludes with $C[A,i,j] = C(A,i,j)$ for all *A*, *i*, and *j*. Also $\delta(G,w) = \min\{C[S,1,N], G(0,N)$ if $S \to \varepsilon\}$.

Proof: With regard to correctness, the preceding paragraph presents the algorithm of Figure 2 as a dynamic programming algorithm centering on recurrence (2.1) with an application of Dijsktra's algorithm in the inner loop to properly handle the cyclicity induced by the grammar. Thus correctness follows from the correctness of (2.1) and that of Dijkstra's algorithm. There is, however, the subtle situation where $i = 1$ and $j = 0$, that is, when we are computing *G*-values for the first time in accordance with recurrence (2.2). Note that in this case the graph dependency model no longer holds as the weight $G(A) \equiv C[A,1,0]$ of some edges isn't known! In this case one should conceptually think of applying Dijkstra's algorithm to a hyper-graph where there is a hyper-edge of weight 0 from *B* and *C* to *A* whenever $A \to BC$. An exercise that revisits the details of the proof of Dijkstra's algorithm reveal the correctness of the algorithm in this boundary situation. Finally, either the empty string is closest to *w* (if $\varepsilon \in L(G)$) or some non-empty string is closest to *w*. By construction the former is $G(0,N)$ and the later is $C[S,1,N]$. Thus line 15 correctly reports $\delta(G,w)$

With regards to time, observe that the outer two **for**-loops of lines 1 and 2 are repeated *$O(N^2)$* times. Assuming that *O(N)* time is taken earlier to precompute a table of the values *g(i)*, a total of *O(PN)* time is spent each time the **for**-loop of lines 4&5 is executed.

Constructing the initial heap in lines 6 takes *O(P)* time. To account for the time spent in the **while**-loop of lines 7-13, observe that when a non-terminal A is extracted, the number of repetitions of the subsequent **for**-loops equals the number of productions involving A in their right hand side. Summed over all non-terminals this is P. Each **reheap** is an *O(logP)* operation, giving the dominant cost of *O(PlogP)* for the **while**-loop. In summary, *O(P(N+logP))* time is spent each time lines 3-14 are executed, and these lines are executed $O(N^2)$ time. ■

In keeping with the dynamic programming paradigm, the algorithm computes the best alignment between a non-empty string derivable from every non-terminal and every substring of w, in order to efficiently compute $C[S,1,N]$, the best alignment between a non-empty string v derivable from S and all of w. To obtain v, its derivation from S, and the alignment to w achieving this minimum, one need only start at the entry $C[S,1,N]$, determine how its minimal value was achieved, and then recursively *traceback* through the relevant terms in the table C. This can be done in *O(P(M+N))* time where M is the length of v.

Theorem 2, our primary and fastest result, requires that $G(A) \geq 0$ for all A, which is true whenever $\sigma(a) \geq 0$ for all $a \in \Sigma$. That is, in order to apply Dijkstra's algorithm, all that is required is that σ-values be non-negative, the alignment values $\delta(a,b)$ of scoring scheme δ may be arbitrary. In applications in molecular biology, such a restriction on gap costs is almost always reasonable. Nonetheless, for arbitrary σ an algorithm is still possible: rather than using Dijkstra's algorithm we can use an $O(P^2)$ algorithm (e.g. [Bel58]) for computing shortest paths over a graph with arbitrary edge weights. One subtlety: if such a subprocedure detects a negative cycle then the overall algorithm should halt and report a difference of $-\infty$. Thus we also have an $O(PN^2(N+P))$ algorithm for approximately matching a context free language under arbitrary scoring scheme δ.

3. Generalizing Gap Costs.

We now sketch extensions to our basic result for the case where gap costs are length dependent. For affine gap costs, i.e., $gap(h) = r + sh$ for constants r and s, we show that an easy extension gives algorithms with the same complexities as those of the previous section. Concave gap costs are characterized as having non-increasing forward differences, i.e., $\Delta gap(h) \geq \Delta gap(h+1)$ for all $h \geq 1$ where $\Delta gap(h)$ = $gap(h+1) - gap(h)$. In this case we present recurrences leading to an $O(P^6 N^8 8^P)$ algorithm and pose the open problem of designing an algorithm that is not exponential in P.

Finally, for arbitrary functions *gap(h)* we pose the problem of determining if the approximate match problem is recursive or not.

In the case of affine gap costs, the key is to develop recurrences for:

$$C^{\tau}_{\nu}(A,i,j) = \min\{\delta(x)+\delta(v,w_{i \cdots j})+\delta(y) : A \Rightarrow^{*} xvy \neq \varepsilon \text{ and } |x| \,\tau\, 0 \text{ and } |y| \,\nu\, 0\}$$

where $\tau, \nu \in \{=,\neq\}$. The idea is to compute four different C-values modulated by τ and ν that break alignments with strings derivable from A (hereafter called *A-strings*) into four cases depending on whether the alignment begins or ends with a gap of symbols of the A-string. For example, $C^{\neq}_{=}(A,i,j)$ is the score of the best alignment between an A-string t with $w_{i \cdots j}$ that leaves the first symbol of t unaligned and its last symbol aligned. Following the logic of the previous section leads directly to the following recurrences for these quantities.

Theorem 3: For all $A \in V$, $0 \leq i-1 \leq j \leq N$, and $\tau, \nu \in \{=,\neq\}$:

$$\begin{aligned} C^{\tau}_{\nu}(A,i,j) = \min\{ & \min_{A \to BC, k \in [i-1,j], \kappa, \sigma \in \{=,\neq\}} C^{\tau}_{\kappa}(B,i,k) + C^{\sigma}_{\nu}(C,k+1,j) - (r \text{ if } \kappa = \sigma = '\neq'), \\ & \min_{A \to B} \; C^{\tau}_{\nu}(B,i,j), \\ & (\min_{A \to a, k \in [i,j]} \delta(a,a_k) + (j-i)s + (r \text{ if } k>i) + (r \text{ if } k<j)) \;\; \textbf{if } \tau = \nu = '=', \\ & (\min_{A \to a} \; (j-i+2)s + 2r) \;\; \textbf{if } \tau \neq \nu \textbf{ and } j \geq i, \\ & (\min_{A \to a} \; r + s) \;\; \textbf{if } \tau = \nu = '\neq' \textbf{ and } j < i \\ \} & \end{aligned}$$

Proof: The proof is a relatively direct extension of that for Theorem 1, so we only make a couple of comments to assist the reader's intuition. In the minimum over productions $A \to BC$, r is subtracted when $\kappa = \sigma = '\neq'$ because two gaps are being merged into one. On the other hand, one need not be concerned with end-gaps of $w_{i \cdots j}$ as the minimum is over all possible k. Finally, the basis minimums over productions $A \to a$ of (2.1) have $G(i,j)$ replaced with $r + (j-i+1)s$, $\sigma(a)$ replaced with $r + s$, and are spread across the four choices of τ and ν, according to the underlying alignment. ■

Gap(h) ≥ 0 for all h if and only if $r + s \geq 0$ and $s \geq 0$. In this event we have a scoring scheme with non-negative gap costs and a direct adaptation of the algorithm of Figure 2 to the recurrence of Theorem 3 yields an $O(PN^2(N+logP))$ algorithm for approximate matching of context free languages with affine gap costs. When *gap(h)* is negative for some *h* then we can develop an *O(PN²(N+P))* algorithm by an appeal to a more powerful shortest paths algorithm as noted earlier.

In the more general case of concave gap costs we first have to be concerned with whether or not *gap(h)* is non-decreasing. If it isn't and *L(G)* is not finite (i.e. *G* is recursive) than the difference to *w* can be made arbitrarily negative by simply deriving arbitrarily large strings. So a first step is to screen this possibility and answer $-\infty$ if it arises. Hence forward, we assume *gap(h)* is non-decreasing. During the treatment under this assumption, it will become clear that a corollary is an $O(PN^4 8^P)$ algorithm for the case when *L(G)* is finite and *gap(h)* is decreasing.

Under the cost model of concave, non-decreasing gap costs the key is to develop recurrences for:

$$C(A,i,j,l,r) = \min\{\delta(v,w_{i\cdots j}) : A \Rightarrow^* xvy \neq \varepsilon \text{ and } |x|=l \text{ and } |y|=r\}$$

where $l,r \in [0,NP2^P]$. In this case we maintain additional parameters *l* and *r* that constrain the exact size of the end-gaps that occur in an *A*-string's alignment with $w_{i\cdots j}$, and the *C*-value does not include the cost of these end-gaps. The lack of additivity in scoring gaps requires that the gap extensions modeled by aligning an *A*-string with the empty string be explicitly represented in the recurrence using the auxiliary quantity:

$$L(t) = \{A : A \Rightarrow^* v \text{ and } |v| = t\}$$

that is the set of non-terminals that can derive strings of length $t \in [1,NP2^P]$. This has the modest compensation of removing the need for the boundary case $j = i-1$ since it is now explicitly represented in the recurrence of Theorem 4.

Theorem 4: For all $A \in V$, $0 \le i \le j \le N$, and $l,r \ge 0$:

$$\begin{aligned} C(A,i,j,l,r) = \min\{ & \min_{A\to BC,\, i\le k<t\le j,\, m,n\ge 0} C(B,i,k,l,m) + C(C,t,j,n,r) + gap(t-k) + gap(m+n), \\ & \min_{A\to BC,\, t\in[1,r],\, C\in L(t)} C(B,i,j,l,r-t), \\ & \min_{A\to BC,\, t\in[1,l],\, B\in L(t)} C(C,i,j,l-t,r), \\ & \min_{A\to B} C(B,i,j,l,r), \\ & (\min_{A\to a,\, k\in[i,j]} \delta(a,a_k) + gap(k-i) + gap(j-k)) \textbf{ if } l=r=0, \\ & \min_{A\to a} gap(j-i+1) \textbf{ if } l+r=1 \\ \} & \end{aligned}$$

where $L(t) = \cup_{k=1}^{t-1}\{A : A\to BC \in L(k)\times L(t-k)\} \cup \{A : A\to B \in L(t)\} \cup \{A : A\to a \text{ and } t=1\}$.

Proof: The correctness of the recurrence is established using a case-based argument on the strings derivable from A as in Theorem 1. ■

The one essential point that we will prove by an appeal to the pumping lemma is that it suffices to compute the recurrence only for end-gaps up to a maximum size of $NP2^P$, i.e. the variables l, r, m, and n of the recurrence can be restricted to range over the interval $[0,NP2^P]$ without loss of generality. To do so it suffices to show that $Best(A,i,j) = \min_{l,r \in [0,NP2^P]} C(A,i,j,l,r)$ where $Best(A,i,j) = \min\{\delta(v,w_{i \cdots j}) : A \Rightarrow^* xvy$ for some x and $y\}$ is the best alignment between a substring of an A-string and $w_{i \cdots j}$. Suppose that $Best(A,i,j) = \delta(u,w_{i \cdots j})$ and that the subsequence u^* is the sequence of symbols in u aligned with those of $w_{i \cdots j}$ in the optimal alignment. In the derivation tree of u from A, let a *junction* vertex be an interior vertex with two children (in CNF outdegree is one or two) both of which derive at least one symbol in u^*. Any subpath between two junction vertices with more than $|V|$ vertices must have a repeated non-terminal label. In such a case one could produce a derivation from A with shorter gaps between symbols in u^* by cutting out the section between the repeated labels, a contradiction to optimality. Thus in the derivation of u from A, all subpaths between consecutive junction vertices must have $|V|$-or-less vertices, and each *off-child* of a vertex on a subpath which is itself is not on the subpath must produce a shortest possible string derivable from its non-terminal label. But then from the facts that $\max_{A \in V} \min_{A \Rightarrow^* v} |v| \leq 2^{P-1}$ and $|u^*| \leq |w_{i \cdots j}| \leq N$, it follows that the largest gap in the alignment with u is not greater than the number of leftmost/rightmost subpaths between junction vertices times their length times the length of the minimum strings derived by an off-child, i.e. $(N+1)(P-1)2^{P-1} \leq NP2^P$.

Given that the recurrence of Theorem 4 need only be computed in the range of indices proved above it follows that one needs to compute $O(|V|P^2N^44^P)$ entries. The cost of handling the cyclic dependencies introduced by the unit production terms is dominated by the $O(P_AP^2N^44^P)$ time needed to computed the first minimum of the recurrence, where P_A is the number of productions with A as their left-hand side. Thus we can compute the difference between a context free language and a string under a concave, non-decreasing gap-penalty scoring scheme in worst-case time $O(P^5N^88^P)$. Note that when $L(G)$ is finite, then its longest string is of length not greater than 2^{P-1}. Thus in this case, the end-gap indices need only be varied over the range $[0,2^{P-1}]$ and an $O(PN^48^P)$ algorithm ensues regardless of whether $gap(h)$ is non-decreasing or not.

The immediate question is whether one can do better than above since the powers are very high and the dependence on grammar size is exponential. This appears very difficult.

Finally, if gap costs can be an arbitrary function of length we know of no result and ponder whether the problem is computable by any Turing machine.

References:

[AhC75] Aho, A.V., and M.J. Corasick, "Efficient string matching: an aid to bibliographic search," *Comm. of the ACM* **18** (1975), 333-340.

[AhP72] Aho, A.V. and T.G. Peterson, "A minimum distance error-correcting parser for context free languages," *SIAM J. on Computing* **1** (4) (1972), 305-312.

[Aku94] Akutsu, T., "Approximate string matching with don't care symbols," *Proc. 5th Combinatorial Pattern Matching Conference* (Asilomar, CA 1994). Published as *Lecture Notes in Computer Science*, #807 (Springer Verlag, New York, NY), 240-249.

[Bel58] Bellman, R.E., "On a routing problem," *Quant. Appl. Math* **16**, 1 (1958), 87-90.

[BoM77] Boyer, R.S., and J.S. Moore, "A fast string searching algorithm," *Comm. of the ACM* **20** (1977), 762-772.

[ChL90] Chang, W.I. and E.L. Lawler, "Approximate matching in sublinear expected time," *Proc. 31st IEEE Symp. on Foundations of Computer Science* (1990), 116-124.

[Dij59] Dijkstra, E.W., "A note on two problems in connexion with graphs," *Numerische Mathematik* **1** (1959), 269-271.

[DBH83] Dayhoff, M.O., W.C. Barker, and L.T. Hunt, "Establishing homologies in protein sequences," *Methods in Enzymology* **91** (1983), 524-545.

[Ear70] Early, J. "An efficient context-free parsing algorithm," *Comm. of the ACM* **13** (1970), 94-102.

[FiS83] Fitch, W., and T. Smith, "Optimal sequence alignments," *Proc. Natl Acad. Sci. USA* **80** (1983), 1382-1386.

[GaP90] Galil, Z. and K. Park, "An improved algorithm for approximate string matching," *SIAM J. on Computing* **19** (1990), 989-999.

[HoU79] Hopcroft, J.E. and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computations* (Addison-Wesley, 1979).

[KaR87] Karp, R.M., and M.O. Rabin, "Efficient randomized pattern matching algorithms," *IBM J. Res. Develop.* **31** (1987), 249-260.

[Kas63] Kasami, T., "An efficient recognition and syntax analysis algorithm for context-free languages," AFCRL-65-758, Air Force Cambridge Reserach Laboratory, Bedford, MA (1963).

[KnM92] Knight, J.R., and E.W. Myers, "Approximate regular expression pattern matching with concave gap costs," *Proc. 3rd Combinatorial Pattern Matching Conference* (Tucson, AZ 1992). Published as *Lecture Notes in Computer Science*, #644 (Springer Verlag, New York, NY), 67-76. To appear in *Algorithmica*.

[LaV89] Landau, G.M., and U. Vishkin, "Fast parallel and serial approximate string matching," *J. of Algorithms* **10** (1989), 157-169.

[Lyo74] Lyon, G, "Syntax-directed least-errors analysis for context-free languages: A practical aproach," *Comm. of the ACM* **17**, 1 (1974), 3-14.

[McC76] McCreight, E.M., "A space economical suffix tree construction algorithm," *J. of ACM* **23** (1976), 262-272.

[MiM88] Miller, W., and E.W. Myers, "Sequence comparison with concave weighting functions," *Bull. of Math. Biol.* **50** (1988), 97-120.

[Mye86] Myers, E.W., "An O(ND) difference algorithm and its variants," *Algorithmica* **1** (1986), 251-266.

[Mye90] Myers, E.W., "A sublinear algorithm for approximate keyword matching," Technical Report TR 90-15, Dept. of Computer Science, U. of Arizona, Tucson, AZ 85721 (1990). Also to appear in *Algorithmica* (1994).

[Mye92] Myers, E.W., "Approximate matching of network expressions with spacers," Technical Report TR 92-5, Dept. of Computer Science, U. of Arizona, Tucson, AZ, 85721 (1992). Submitted to *J. Computational Biology*.

[MyM89] Myers, E.W. and W. Miller, "Approximate matching of regular expressions," *Bull. of Math. Biol.* **51**, 1 (1989), 5-37.

[NeW70] 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. Mol. Biol.* **48** (1970), 443-453.

[San72] Sankoff, D., "Matching sequences with deletion/insertion constraints," *Proc. Nat. Acad. USA* **69** (1972), 4-6.

[SaK83] Sankoff, D. and J. Kruskal (Eds), *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison* (Addison-Wesley, Reading, MA, 1983), 265-310.

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

[Ukk85] Ukkonen, E., "Finding approximate patterns in strings," *J. of Algorithms* **6** (1985), 179-188.

[Ukk92] Ukkonen, E., "Approximate string matching with q-grams and maximal matches," *Theoretical Computer Science* (1992), 191.

[WaF74] Wagner, R.A. and M.J. Fisher, "The string-to-string correction problem," *J. of ACM* **21** (1974), 168-173.

[WaS78] Wagner, R.A., and J.I. Seiferas, "Correcting counter automaton recognizable languages," *SIAM J. on Computing* **7**, 3 (1978), 357-375.

[Wat84] Waterman, M.S., "General methods of sequence comparison," *Bull. Math. Biol.* **46** (1984), 473-501.

[WuM92] Wu, S. and U. Manber, "Fast text searching allowing errors," *Comm. of the ACM* **35** (1992), 83-91.

[WMM94] Wu, S., U. Manber, and E.W. Myers, "A sub-quadratic algorithm for approximate limited expression matching," to appear in *Algorithmica*.

[You67] Younger, D.H., "Recognition and parsing of context-free languages in time n^3," *Information and Control* **10**, 2 (1967), 189-208.