Algorithmica (1994) 12: 327–344

Algorithmica

Sublinear Approximate String Matching and Biological Applications[1]

W. I. Chang[2] and E. L. Lawler[3]

Abstract. Given a text string of length n and a pattern string of length m over a b-letter alphabet, the *k differences approximate string matching problem* asks for all locations in the text where the pattern occurs with at most k differences (substitutions, insertions, deletions). We treat k not as a constant but as a fraction of m (not necessarily constant-fraction). Previous algorithms require at least $O(kn)$ time (or exponential space). We give an algorithm that is sublinear time $O((n/m)k \log_b m)$ when the text is random and k is bounded by the threshold $m/(\log_b m + O(1))$. In particular, when $k = o(m/\log_b m)$ the expected running time is $o(n)$. In the worst case our algorithm is $O(kn)$, but is still an improvement in that it is practical and uses $O(m)$ space compared with $O(n)$ or $O(m^2)$. We define three problems motivated by molecular biology and describe efficient algorithms based on our techniques: (1) approximate substring matching, (2) approximate-overlap detection, and (3) approximate codon matching. Respectively, applications to biology are local similarity search, sequence assembly, and DNA-protein matching.

Key Words. Pattern matching, Edit distance, Suffix tree, Lowest common ancestor, Chernoff bound, Computational biology.

1. Introduction

1.1. Motivation. Pattern matching is one of the classical problems of computer science, and for exact matching many fast algorithms are known. However, in many applications a nonexact or *approximate* match is still meaningful. In the field of molecular biology, for example, a genomic database is likely to contain DNA or protein sequences of many individuals; therefore, matching a piece of DNA against the database must allow a small but significant error due to polymorphism (differences in DNA among individuals of the same species). Furthermore, current DNA sequencing techniques are not perfect, and experimental error can sometimes contribute as much as 5–10% inaccuracies. This problem persists even when two copies of the *same* DNA are compared. The degree to which two sequences match, and the plausibility of a particular alignment

[1] This work was supported in part by NSF Grants CCR-87-04184 and FD-89-02813; by the Human Genome Center, Lawrence Berkeley Laboratory, supported by the Director, Office of Health and Environmental Research, of the U.S. Department of Energy under Contract DE-AC03-76SF00098; and by Department of Energy Grants DE-FG03-90ER60999 and DE-FG02-91ER61190. Earlier versions of this paper appeared as [8] and part of [5].

[2] Cold Spring Harbor Laboratory, P.O. Box 100, Cold Spring Harbor, NY 11724, USA.

[3] Computer Science Division, University of California, Berkeley, CA 94720, USA.

Received September 5, 1991; revised September 2, 1992. Communicated by Alberto Apostolico.

between them, have recently received some statistical analysis of significance (see [24] and [45]). The interpretation of the DNA distance metric as evolutionary distance between species has also been proposed. This is relevant to the construction of evolutionary trees and models [23]. The proposed *sequence-tagged sites* (STS) genomic map database of the Human Genome Project [34], as well as several gene mapping strategies, require the approximate matching of DNA sequences.

We have focussed on the following model of approximate matching: Given a pattern of length m and a text of length n, consisting of letters from an alphabet of size b, and an integer parameter k, find all occurrences of the pattern in the text, allowing in a (partial) match at most k *differences* (substitutions, insertions, or deletions). We treat k not as a constant but as some fraction of m (not necessarily constant-fraction). The text is assumed to be given *on-line* and to be scanned sequentially; the space requirement should be linear in the length of the *pattern.* We note that this simple model has a rich history and interesting combinatorics, and is a natural starting point before more complex, parametric cost functions are to be considered (e.g., [18] and [46]).

1.2. Exact Matching. The problem of locating the first occurrence or all occurrences of a pattern word P in a long text string T has its roots in the implementation of text editors and information retrieval systems in the 1960s to early 1970s. Particularly noteworthy are efficient algorithms of Knuth, Morris, and Pratt (*KMP*) [28], Boyer and Moore (*BM*) (also see [28]), and Karp and Rabin [26]. Each of these algorithms preprocesses the pattern and then scans the text string on-line. Slightly earlier, Weiner [47] solved the complementary problem where the long text string T is fixed and keywords which are given on-line must be located within it, in time proportional to the length of each keyword. Specifically, he constructed a linear-size finite state automaton to report the first occurrences of all substrings of the given text string. An auxiliary data structure used by Weiner, called the *position tree*, soon became widely used and a particular variant of it, the *suffix tree* of McCreight [32], is most widely known for its role in fast implementations of the Lempel–Ziv data compression algorithm (see, for example, [13] and [38]). The suffix tree of a string is a compressed *digital trie* of its suffixes, whose edge labels are substrings (denoted by indices). Recent applications of suffix trees include methods for finding biologically significant motif patterns in DNA [17] and elegant linear-time algorithms for computing:

(1) The longest substrings common to k out of m strings, for all k [21].
(2) The pairwise maximum exact overlaps of m strings [19].

It is interesting to note, however, that no one seemed to know how to use suffix trees to solve the basic string-matching problem in a simple way, using $O(n)$ time and $O(m)$ space, where n, m are respectively the lengths of text T and pattern P (the same time and space constraints as in *KMP*, assuming a fixed, finite alphabet). Such results have been reported for *directed acyclic word graphs* (*DAWG*) [10] and for *suffix automata* [11] which differ from suffix trees in an essential way:

their edge labels are single letters, not strings. We have observed that a linear-time algorithm can be derived from the incremental suffix tree construction given in [32]. (Ukkonen [43] made this observation independently.) This is pedagogically satisfying and unifies several previous results:

(1) One-pass multiple-keyword search [1], using a suffix tree of $P_1\$_1P_2\$_2\cdots$, where P_i are keywords and $\$_i$ are distinct delimiters. Insertions and deletions of keywords are easily accommodated if no keyword contains another.
(2) Sublinear exact matching similar to *BM* but superior when the alphabet is small or m is large.

This special case ($k = 0$) of our sublinear approximate matching algorithm examines on average only $O((n/m)\log_b m)$ letters of the text (b is alphabet size). Such a result was first stated by Knuth *et al.* [28], but their version has to run the basic *KMP* in parallel in order to ensure linearity. Our method is inherently linear (in the worst case) and is easily generalized to handle multiple keywords. Recently Park [35] has found yet another way to achieve the same expected running time (for one keyword), using a suffix tree of the *reverse* of the pattern and facts from string combinatorics.

Our technique uses in a crucial way a set of pointers, called *suffix links*, originally used by McCreight [32] in suffix tree construction. (Subsequent papers paid no attention to these pointers.) By simultaneously scanning the text and walking in the suffix tree of the pattern, our algorithm computes the longest substring of the pattern to appear at each position of the text. We call this information *matching statistics*. In Section 2.1 we give an abstract defintion of the suffix tree, as opposed to definition by construction [32]. This greatly simplifies the description and analysis of the matching statistics algorithm (Section 2.2), which is then used to simplify and to improve upon the current best approximate matching algorithm that is based on dynamic programming (Section 2.3).

1.3. Approximate Matching. Beginning in the 1980s, genetics and DNA sequence analysis research provided the impetus for advances in nonexact string matching. The *k differences approximate string-matching problem* specifies, in addition to T and P, the parameter k of differences (substitutions, insertions, deletions) allowed in a match. The problem is to find all locations in the text where a match *ends*. (So the output is of linear size. This is equivalent to finding where matches begin, by reversing the strings.) Substitutions and *indels* (insertions and deletions) are *edit operations*, and the minimum number needed to transform one string to another is called the *edit distance* or the Levenshtein distance [31].

The classical dynamic programming algorithm of Sellers [41] computes (column by column) an $m + 1$ by $n + 1$ table whose entry $D(i, j)$ is the minimum number of edit operations necessary to transform the length i *prefix* of the pattern into *some* text fragment ending at the jth letter. (Boundary conditions are $D(i, 0) = i$ and $D(0, j) = 0$. There is a match ending at text position j if and only if entry

$D(m, j)$ is at most k.) There is a simple recursive formula giving each entry in terms of the three adjacent entries above or to the left:

$$D(i, j) = \min\{1 + D(i-1, j),\ 1 + D(i, j-1),\ I_{ij} + D(i-1, j-1)\},$$

where $I_{ij} = 0$ if $P[i] = T[j]$; $I_{ij} = 1$ if $P[i] \neq T[j]$. The three expressions in the min correspond respectively to deleting $P[i]$ from the pattern, inserting $T[j]$ into the pattern, and substituting $T[j]$ for $P[i]$. A continuity argument implies (1) adjacent entries along rows and columns differ by at most one; that and the recurrence imply (2) forward diagonals ($\searrow$) are nondecreasing and adjacent entries differ by at most one. More recent methods by Ukkonen [42], Ukkonen and Wood [44], Landau and Vishkin [29], [30], Galil and Giancarlo [14] (survey and refinements), and Galil and Park [15] take advantage of these geometric properties in order to compute $O(kn)$ instead of mn entries.

The locations of the first $k + 1$ *transitions* (x to $x + 1$) along each forward diagonal are sufficient to characterize the solution, by the nondecreasing diagonal monotonicity property (2). Landau and Vishkin's algorithm (*LV*) [29], [30] computes each transition in constant time. Two bottlenecks kept this $O(kn)$ algorithm impractical:

(1) Computing the *lowest common ancestor* (*LCA*) of two nodes of the suffix tree of P required a complicated algorithm. However Schieber and Vishkin [40] found a new constant-time *LCA* algorithm, which we implemented efficiently [4].
(2) It was not known how to compute matching statistics using only the suffix tree of P; however, we now have a simple solution. We have reduced the space requirement of *LV* to $O(m)$ in addition to input/output; it now appears to be the best among $O(kn)$ worst-case algorithms.

When k is a constant-fraction of m, $O(kn)$ dynamic programming methods represent no theoretical improvement over the classical $O(mn)$ algorithm. In this paper we take a different approach. When the error tolerance k is not too large relative to m, $O(n)$ *expected time* (*LET*) or even faster algorithms are possible. The threshold on k for linear expected time is *logarithmic fraction*: $k < k^* = m/(\log_b m + c_1) - c_2$ (for suitable constants c_i). The choice $c_1 = 5.6$ used in the proof is pessimistic; experiments suggest $c_1 = 3$ suffices for $b = 2$; $c_1 = 2$ suffices for $b \geq 4$. In practice, for m in the hundreds the error thresholds k^* in terms of percentage of m are 35 ($b = 64$), 25 ($b = 16$), 15 ($b = 4$), and 7 ($b = 2$).

Our main tool is matching statistics. Computed in a single left-to-right scan of the text, this information is used to eliminate stretches of text where a match cannot possibly occur. Only rarely does the algorithm resort to a dynamic programming subroutine, so the average running time is linear in the length of the text sequence. (An algorithm similar to *LET*, discovered independently but without analysis of threshold, was recently implemented by Jokinen *et al.* [22], and was the fastest for large m and medium k among algorithms they tested.) For $k < k^*/2 - 3$, stretches of text can be skipped altogether (*SET*); this is sublinear

in the sense of Boyer–Moore. When $k = o(m/\log_b m)$ the expected running time is $o(n)$, truly sublinear. Indeed, approximate matches in random text are so infrequent that running time is dominated by the effort to verify mismatches. Our algorithms do so in linear or even sublinear expected time.

Algorithms *LET* and *SET* are described in Sections 3.1 and 3.2. In each case the total work of the dynamic programming subroutine is at most what it would cost to apply the subroutine to the entire text all at once. It therefore does not hurt to apply our algorithms, even for k greater than k^* or $k^*/2 - 3$. These algorithms can be viewed as *approximate string matching via data compression* (compare with [38]). Finally, we discuss recent developments in Section 3.3.

1.4. Biological Applications. In Sections 4.1–4.3 we apply our techniques to several problems motivated by biology. We feel the problem formulations we have chosen are faithful to the original tasks, and hope our algorithms will become useful to biologists.

(k, l) APPROXIMATE SUBSTRING MATCHING PROBLEM. Given T of length n, P of length m, and parameters l and k, find all i such that some substring of T ending at i matches a length l substring of P to at most k differences.

This is a general method for finding *local similarities*. We should mention that matching statistics is already a useful local similarity measure (see [2], [10], and [36]). It can be used to obtain a summary of *all* exact matches between fragments of the text and pattern. A natural extension would be to assign weights to matched words based on frequency. For this purpose, the suffix tree can be any dictionary with suitable statistics.

ρ APPROXIMATE-OVERLAP DETECTION PROBLEM. Given P of length m, Q of length n, and parameter ρ, find all l such that the length l prefix of P matches a suffix of Q to within ρl differences.

Sequence assembly is the task of matching pieces of DNA that are known to be overlapping fragments of a single long DNA sequence. In this application approximate-overlap is done for every pair of fragments and accounts for most of the computing time (98% according to Peltola *et al.* [37]). In this application, the experimental sequencing error contributes about 5% differences to overlapped regions. This is few enough errors ($\rho = 0.05$) for our algorithm to be sublinear on average. Recently Huang [20] has used a simplified (t-tuple) implementation of our method to speed up sequence assembly in a program called *CAP*.

APPROXIMATE CODON MATCHING PROBLEM. $T \in \Sigma^*$ and $P \in \Pi^*$ are from different alphabets. The translation map tr: $\Sigma^t \to \Pi$ via t-tuples is not necessarily one-to-one (i.e., different *codons* may translate to the same letter). A string σ over Σ *translates* to a string π over Π if $\text{tr}(\sigma_1\sigma_2\cdots\sigma_t) = \pi_1$, $\text{tr}(\sigma_{t+1}\cdots\sigma_{2t}) = \pi_2$, etc. Find all locations i such that some substring of T ending at i is within edit distance k of a string that translates to P.

The difficulty comes from having both multiple encodings and framing errors caused by insertions and deletions. In DNA-protein matching, each codon is a 3-tuple of nucleotides (A, C, G, or T), and is translated into one of 20 amino acids. We describe an efficient algorithm based on our methods.

2. Data Structures

2.1. Suffix Trees. Let $P[1, \ldots, m]\$$ be a string over a fixed finite alphabet $\Sigma \cup \{\$\}$ where \$ is a special end-marker that occurs nowhere else. The following is a precise characterization of the suffix tree of a string P, denoted $S(P\$)$.

Substrings of $P\$$ are called *words*; a word w is *branching* if there are different letters x, y such that wx and wy are words. Let w be a word; floor(w) denotes the longest prefix of w that is a branching word; ceiling(w) denotes the shortest extension of w that is either a branching word or a suffix of $P\$$. The following are evident: floor and ceiling are well defined; if w is branching, then floor(w) = ceiling(w) = w; and the empty string, denoted Λ, is a branching word.

If string u is a prefix of string v, $u^{-1}v$ denotes the suffix of v which if appended to u gives v. If string v is not an empty string, v_1 denotes the first letter of v.

Let us define $S(P\$)$. There is a one-to-one correspondence between nodes of $S(P\$)$ and a set of words:

$$\text{root} \leftrightarrow \Lambda,$$

$$\{\text{internal nodes}\} \leftrightarrow \{\text{branching words}\},$$

$$\{\text{leaves}\} \leftrightarrow \{\text{suffixes}\}.$$

The relation *is prefix of* defines a natural partial order on the above set of words that is a tree; this in turn defines the edges of $S(P\$)$. That is, node u is the parent of node v iff (branching word) u is a proper prefix of (branching word or suffix) v, and there is no branching word w that is a proper extension of u and a proper prefix of v. This directed edge from u to v is labeled with a triple (x, l, r) chosen such that $P\$[l] = x$ and $u^{-1}v = P\$[l, \ldots, r]$ (the choice of l, r may not be unique). We write $\text{son}(u, x) = v$, $\text{first}(u, x) = l$, and $\text{len}(u, x) = r - l + 1$, as well as refer to nodes by their corresponding words.

In addition define the function *shift*: {internal nodes other than the root} → {internal nodes} given by $\text{shift}(w) = w_1^{-1}w$ ($w \neq \Lambda$). McCreight [32] constructs the suffix tree $S(P\$)$ and the shift function ("suffix links") in $O(m)$ time, using an algorithm very similar to the matching-statistics algorithm given below.

2.2. Matching Statistics. Define the *matching statistics* of text $T[1, \ldots, n]$ with respect to pattern $P[1, \ldots, m]$ to be an integer vector $M(T, P)$ together with a vector $M'(T, P)$ of pointers to the nodes of suffix tree $S(P\$)$, where $M(T, P)_i = l$ if l is the length of the longest word (substring of $P\$$) matching exactly a prefix of $T[i, \ldots, n]$; and $M'(T, P)_i$ points to the node ceiling($T[i, \ldots, i + l - 1]$). The goal

is to compute M and M' in $O(n + m)$ time and as little additional storage as possible. In most applications M and M' are not stored all at once. Landau and Vishkin [30] reduced this problem to constructing the suffix tree of $P\$_1 T\$_2$, but that requires either $O(n + m)$ additional space or computing everything twice (by building overlapping $O(m)$-size suffix trees). Galil and Giancarlo [14] gave an automata-based algorithm which scans the text one way and then in reverse; for it to run in $O(m)$ space the text must be scanned four times (again by working on overlapping $2m$-size blocks of text). Although satisfactory automata-based solutions exist [10], [11], since the approximate matching algorithm given in [14] already uses the suffix tree $S(P\$)$ in a different part of the algorithm, it makes sense to try to compute matching statistics directly, using only $S(P\$)$. The following is a very simple linear-time algorithm that computes $M(T, P)_i$ and $M'(T, P)_i$ in order of increasing i, in a single left-to-right scan of the text. Very briefly, the longest match starting at the first position is found by walking down the tree, matching a letter at a time. Subsequent longest matches are found by following suffix links and carefully going down the tree.

MATCHING STATISTICS ALGORITHM (Comments). Variables i, j, k are indices into the text string; the ith iteration of step 3 computes $M(T, P)_i$ and $M'(T, P)_i$; position k of the text had just been scanned, and j is some position between i and k. At all times the following invariant is maintained:

(i) $T[i, \ldots, k-1]$ is a word; $T[i, \ldots, j-1]$ is a branching word.

After step 3.1 the following becomes true:

(ii) $T[i, \ldots, j-1] = \text{floor}(T[i, \ldots, k-1])$ and corresponds to the node v.

After step 3.2 the following becomes true as well:

(iii) $T[i, \ldots, k]$ is not a word.

The key observation is that if $j < k$ after step 3.1, then $T[i, \ldots, k-1]$ is not a branching word, so neither is $T[i-1, \ldots, k-1]$. Indeed, as substrings of P they must have the *same* unique single-letter word extension. We know from iteration $i-1$ that $T[i-1, \ldots, k-1]$ is a word but $T[i-1, \ldots, k]$ is not, so $T[k]$ cannot be this letter. Hence the match cannot be extended.

Together invariants (i) and (iii) imply $M(T, P)_i = k - i$. Step 3.4 uses the suffix link to go from v to shift(v) if v is not the root. Note that (i) is maintained.

1. construct suffix tree $S(P\$)$ by McCreight's algorithm [32] (see below)
2. let $v \leftarrow$ root; $j \leftarrow 1$; $k \leftarrow 1$
3. **for** $i \leftarrow 1$ to n **do**
3.1. **while** $(j < k)$ and $(j + \text{len}(v, T[j]) \leq k)$ **do**
$v, j \leftarrow \text{son}(v, T[j]), j + \text{len}(v, T[j])$
end while

```
3.2.  if (j = k) then
        while son(v, T[j]) exists and
            (T[k] = P$[first(v, T[j]) + k − j]) do
          k ← k + 1
          if (j + len(v, T[j]) = k) then
            v ← son(v, T[j]); j ← k end if
        end while
      end if
3.3.  M(T, P)_i ← k − i
      if (j = k) M'(T, P)_i ← v
      otherwise M'(T, P)_i ← son(v, T[j])
3.4.  if v is root and (j = k) then
        j ← j + 1; k ← k + 1 end if
      if v is root and (j < k) then
        j ← j + 1 end if
      if v is not root then v ← shift(v) end if
```

The positive integers i, j, and k never decrease and are bounded by n. For every constant amount of work in step 3, at least one of i, j, k is increased in value. The running time is therefore $O(m)$ for steps 1 and 2 and $O(n)$ for step 3.

As a corollary we get an *on-line* exact matching algorithm, based on suffix trees, that uses $O(m)$ space and preprocessing. That is, an exact match ending at position $k - 1$ is found before $T[k]$ is read (as soon as variable k takes on the value $i + m$ in the middle of step 3.2). Furthermore, since m iterations of step 3 take $O(m)$ time, this computation can be made *real-time* (i.e., constant throughput) by buffering m letters of the input text.

For completeness we now sketch very briefly how to construct $S(P\$)$. McCreight's algorithm builds the tree incrementally as it scans the string left to right; it has the same structure as our matching-statistics algorithm. Each iteration of "step 3" adds a leaf (suffix), and possibly a node u above it (splitting an edge below some node v); if so, the next iteration will patch the suffix link shift(u). Transition to the next iteration is accomplished by going from v to shift(v). Then walk down the tree in the same manner as matching statistics. If a node corresponding to the suffix of u already exists, set shift(u) to point to it and continue walking; otherwise, such a node is inserted and the new leaf is added below. Correctness and linear-time arguments are similar to those for matching statistics (for details see [32]). Although a node may have as many edges as there are letters in the alphabet, the tree has altogether a linear number of edges. In practice, suffix trees are *hash coded* in order to be compact; look-up operations son(u, x) are constant time modulo hashing.

2.3. The Landau and Vishkin Algorithm. Vectors M and M' are the main ingredients needed to speed up approximate string matching. Define "jump" $J(i, j)$ as the length of the longest common prefix between $P[i, \ldots, m]$ and $T[j, \ldots, n]$. Jumps can be computed in constant time using the identity $J(i, j) = \min(M_j$, length of word corresponding to $LCA(M'_j$, leaf $P[i, \ldots, m]\$))$. (The min is needed because

M'_j is a "ceiling" and can be the leaf's ancestor.) The following paragraph describes succinctly the dynamic programming component of *LV*. Recall that the locations of the first $k+1$ transitions (x to $x+1$) along each forward diagonal of table D suffice in characterizing the solution.

Call cell $D(i, j)$ an entry of diagonal $j-i$. However, instead of D, compute column by column a $k+1$ by $n+1$ table L where $L(x, y)$ is the row number of the last x along diagonal $y-x$. Let us first look at D, the original table. We can define $L(x, -1) = -\infty$ because every cell of diagonal $-1-x$ is at least $D(x+1, 0) = x+1$. Likewise we can define $L(x, -2) = -\infty$. It is easy to see $L(0, y) = J(1, 1+y)$. Entry $L(x, y)$ can be computed using jumps and the three cells above and to the left: $\alpha = L(x-1, y-2)$, $\beta = L(x-1, y-1)$, and $\gamma = L(x-1, y)$, which are respectively the row numbers of the last $x-1$ in diagonals $y-x-1$, $y-x$, and $y-x+1$. It can be inferred that along diagonal $y-x$, three cells at rows α, $\beta+1$, and $\gamma+1$ are at most x (respectively, insert $T[\alpha+y-x]$ after $P[\alpha]$, substitute $T[\beta+1+y-x]$ for $P[\beta+1]$, delete $P[\gamma+1]$). Let $i = \max(\alpha, \beta+1, \gamma+1)$; then $L(x, y) = i + J(i+1, i+1+y-x)$.

In order to compute jumps in constant time Landau and Vishkin used $O(n+m)$ space (to build the suffix tree of P concatenated with T). Galil and Giancarlo [14] only needed the suffix tree of P and matching statistics, but their algorithm for the latter (called "Best-Fit" in their paper) required $O(n)$ space. In recent papers Galil and Park [15] and Ukkonen and Wood [44] gave $O(m^2)$-space "practical" algorithms. With a little care Landau and Vishkin's algorithm can be implemented very efficiently in $O(m)$ working storage, using the matching-statistics subroutine given above (keeping at any time only $O(m)$ most recent entries) and Schieber and Vishkin's very fast *LCA* computation [40]. In our implementation [4] only simple machine instructions are used (such as decrement and complement, but not bit-shift). Logarithm in [40] is replaced by bit magic, using a table of *reversals* of binary representation of numbers. Fewer than 60 machine instructions suffice to compute an *LCA*.

For a fixed, finite alphabet this appears to be the first practical algorithm to run in $O(m)$ space and $O(kn)$ time in the worst case. For general alphabet the worst-case running time of $O(kn)$ is modulo hashing in $O(m)$ space, or deterministic in $O(bm)$ space.

3. Algorithms for Logarithmic Fraction Error

3.1. Linear Expected Time. We now describe an algorithm for approximate matching that runs in $O(n)$ expected time when:

(1) T is a uniformly random string of length n over a finite alphabet of size b.
(2) The number k of differences allowed in a match is less than the threshold $k^* = m/(\log_b m + c_1) - c_2$ (c_1 and c_2 are constants to be specified later; m denotes pattern length).

The pattern P does not have to be random.

Recall that $M(T, P)_i$ denotes the length of the longest substring of P to be found beginning at position i of text T, and that it is easily computed using the suffix tree $S(P\$)$.

LINEAR EXPECTED TIME ALGORITHM (*LET*). Set $S_1 = 1$ and for $j \geq 1$ compute $S_{j+1} = S_j + M(T, P)_{S_j} + 1$. (The $(j+1)$st start position is obtained by taking the "maximum jump" at the jth start position *plus one more letter*.) For $j = 1, 2, \ldots$, if $S_{j+k+2} - S_j \geq m - k$ apply the Landau–Vishkin algorithm (*LV*) to $T[S_j, \ldots, S_{j+k+2} - 1]$ (call this "work at start position S_j"). All that is required to keep the worst-case running time bounded by $O(kn)$ is for *LV* to remember its state (i.e., the column of table L it last computed) so it can resume from that point.

CLAIM. *This correctly solves k differences approximate matching.*

PROOF. If $T[p, \ldots, p + d - 1]$ matches P and $S_j < p \leq S_{j+1}$, then this string can be written in the form $w_1 x_1 w_2 x_2 \cdots w_{k+1} x_{k+1}$ where each x_l is a letter or empty, and each w_l is a substring of P. It can then be shown by induction that, for every $0 \leq l \leq k+1$, we have $S_{j+l+1} \geq p + \text{length}(w_1 x_1 \cdots w_l x_l)$. In particular, $S_{j+k+2} \geq p + d$ which implies $S_{j+k+2} - S_j > d \geq m - k$. Therefore the above algorithm will perform work at start position S_j and thereby detect there is a match ending at position $p + d - 1$. □

Next we show that the probability of having to perform work at S_1 is small. Since the event $S_{k+3} - S_1 \geq m - k$ implies $S_{k^*+3} - S_1 \geq m - k^*$, it suffices to prove the following lemma.

MAIN LEMMA. *For suitably chosen constants c_1 and c_2, and*

$$k^* = m/(\log_b m + c_1) - c_2,$$

$$\Pr[S_{k^*+3} - S_1 \geq m - k^*] < 1/m^3.$$

PROOF. For ease of presentation let us assume (1) $b = 2$ ($b > 2$ gives slightly smaller c_i's) and (2) k^* and $\lg m$ are integers ($\lg$ denotes log to the base 2).

Let X_j be the random variable $S_{j+1} - S_j$. First note that the X_j's are independent and identically distributed since each position S_{j+1} is beyond the last letter looked at in order to compute the maximum jump at S_j. Also note $S_{k^*+3} - S_1 = X_1 + X_2 + \cdots + X_{k^*+2}$.

Since $X_1 = M(T, P)_1 + 1$ and there are at most m different words of length $\lg m + d$ in P out of $m2^d$ different strings of that length, we have

$$(*) \qquad \Pr[X_1 = \lg m + d + 1] < 2^{-d} \qquad \text{for all integer} \quad d \geq 0.$$

In particular $E[X_1] < \lg m + 3$. Let us consider the probability that $X_1 + X_2 + \cdots + X_{k^*+2} \geq m - k^*$. Let $Y_i = X_i - (m - k^*)/(k^* + 2)$. Choose $c_2 > 2$ so

$$m/(k^* + 2) > m/(k^* + c_2) = \lg m + c_1.$$

Then

$$\begin{aligned} Y_i &< X_i - m/(k^* + 2) + 1 \\ &< X_i - (\lg m + c_1) + 1 \\ &< X_i - \lg m - 3 \end{aligned}$$

provided we choose $c_1 > 4$. This implies $E[Y_i] < 0$, so we can apply the Chernoff bound technique (see [25]) to get

$$\begin{aligned} \Pr[X_1 + \cdots + X_{k^*+2} \geq m - k^*] &= \Pr[Y_1 + \cdots + Y_{k^*+2} \geq 0] \\ &\leq E[e^{tY_1}]^{k^*+2} \end{aligned}$$

for any $t > 0$. Inequality $(*)$ is equivalent to

$$\Pr[Y_1 = \lg m + d + 1 - (m - k^*)/(k^* + 2)] < 2^{-d} \qquad \text{for all integer} \quad d \geq 0.$$

Therefore, for any $t > 0$,

$$E[e^{tY_1}] < \sum_{d=0}^{\infty} e^{t \lg m + td + t - t(m-k^*)/(k^*+2)} \cdot 2^{-d},$$

where the first term of the summation ($d = 0$) bounds

$$E[e^{tY_1} \mid Y_1 \leq \lg m + 1 - (m - k^*)/(k^* + 2)]$$

and the remaining terms bound

$$E[e^{tY_1} \mid Y_1 > \lg m + 1 - (m - k^*)/(k^* + 2)] \cdot \Pr[Y_1 > \lg m + 1 - (m - k^*)/(k^* + 2)].$$

Choose $t = (\log_e 2)/2$. Then

$$E[e^{tY_1}] < \sqrt{2}^{\lg m + 1 - (m-k^*)/(k^*+2)} \cdot \sum_{d=0}^{\infty} \sqrt{2}^{-d},$$

which gives $E[e^{tY_1}] < \sqrt{2}^{\lg m + 1 - (m-k^*)/(k^*+2) + 3.6}$; so raising to the power $(k^* + 2)$ yields

$$E[e^{tY_1}]^{k^*+2} < \sqrt{2}^{(k^*+2)\lg m - (m-k^*) + 4.6(k^*+2)}$$
$$< \sqrt{2}^{(5.6-c_1)(k^*+2) - (c_2-2)(c_1 + \lg m)}$$

after some algebra. This is less than $1/m^3$ if $c_1 = 5.6$ and $c_2 = 8$. □

From this we deduce that the expected work at start position S_1 is $O(1)$. This is true for any start position S_j because the event $S_{j+k+2} - S_j \geq m - k$ implies the event $S_{j+k^*+2} - S_j \geq m - k^*$, which occurs with probability less than $1/m^3$. There is no conditioning because the theorem of total expectation implies

$$E[E[\text{work at start position } S_j \text{ given any conditioning}]]$$
$$= E[\text{work at start position } S_j].$$

Hence the expected total work is $O(n)$.

This type of analysis can be applied to more general settings, such as a biased alphabet, if certain assumptions are made about the pattern as well as about the text. In fact, the following is clear from the above proof:

THEOREM 1. *Suppose, for all i, the random variable*

$$Z_i = (\textit{length of longest exact match at position } i \textit{ of text } T \textit{ with some substring of pattern } P[1, \ldots, m])$$

is independent from $T[1, \ldots, i-1]$ *and has the same distribution as* Z_1, *and suppose* $\Pr[Z_1 > E[Z_1] + d \cdot \text{StdDev}(Z_1)]$ *decreases exponentially in d. Then constants* c_1 *and* c_2 *exist such that, for* $k \leq k^* = m/(E[Z_1] + c_1 \cdot \text{StdDev}(Z_1)) - c_2$, *k differences approximate matching takes linear expected time.*

3.2. Sublinear Expected Time. We now derive from the above an algorithm *SET* that is *sublinear* when $k < k^*/2 - 3$, where $k^* = m/(\log_b m + c_1) - c_2$ (same constants as before). Partition the text into regions of length $(m - k)/2$. Any substring of T that matches P must contain the whole of at least one region. Starting from the left end of each region R, compute $k + 1$ "maximum jumps," say ending at some position p. If position p is within R, there can be no match containing the whole of R. If p is beyond R, apply the Landau–Vishkin algorithm (*LV*) to a stretch of text beginning $(m + 3k)/2$ letters to the left of region R and ending at position p.

It can be shown by a trivial variation of the lemma that $\Pr[p \text{ is beyond } R] < 1/m^3$ (essentially, divide expressions $(m - k^*)$ and $(k^* + 2)$ by 2 everywhere

in the proof), so LV is seldom applied. Therefore, the expected running time of this algorithm is sublinear. On the average only $L = (k + 1)(\log_b m + O(1))$ letters are examined from each region, for a total of $2nL/(m - k)$. (A variation of this algorithm examines only $nL/(m - k - L)$ letters of text, on the average.) A combination of LET (for $k \geq k^*/2 - 3$) and SET (for $k < k^*/2 - 3$) runs in $O((n/m)k \log_b m)$ expected time, for any $k < k^*$. Since edit distance *lower bounds* Hamming distance (no indels) and longest common subsequence metric (no substitutions), our result applies to these models as well.

THEOREM 2. *The average-case complexity of k error (edit distance, Hamming distance, or longest common subsequence metric) approximate string matching is* $O((n/m)k \log_b m)$, *when k is bounded by the threshold* $m/(\log_b m + O(1))$. *In particular, when* $k = o(m/\log_b m)$ *the complexity is* $o(n)$.

3.3. Recent Developments. The choice $c_1 = 5.6$ used to prove the Main Lemma is pessimistic; experiments suggest $c_1 = 3$ suffices for $b = 2$; $c_1 = 2$ suffices for $b \geq 4$. In practice, for m in the hundreds the error thresholds k^* in terms of percentage of m are 35 ($b = 64$), 25 ($b = 16$), 15 ($b = 4$), and 7 ($b = 2$). For the purpose of comparison, the smallest k in terms of percentage of m for which there is a match (using $n = 100m$) are about 85 ($b = 64$), 72 ($b = 16$), 45 ($b = 4$), and 25 ($b = 2$).

The gaps and the mystery of these percentages were motivations for the work described in [7], where the average of row m of table D is shown to be $\Theta(m)$, and conjectured o be $(1 - 1/\sqrt{b} + o(1/\sqrt{b})) \cdot m$. Using the combinatorial lemma from [7], generalized matching statistics from [8], and a *bootstrapping* technique from [5], Chang has recently shown that constants $\rho_b > 0$ exist such that, for all $k < \rho_b m$, k-error approximate matching (edit distance, Hamming distance, or longest common subsequence metric) has average-case complexity $\Theta((n/m)(k + \log_b m))$ [6]. Very briefly, this $m^{O(1)}$ space and preprocessing algorithm extends the notion of matching statistics to nonexact matches; treats the pattern as text, and $\Theta(\log_b m)$-tuples of the text as patterns. It is very different from algorithms based on the *Four-Russians* technique, such as Ukkonen [42] (linear time but exponential space) and more recently Wu *et al.* [48]. The lower bound comes from the obvious $\Omega(kn/m)$ and Yao's classical $\Omega((n/m) \log_b m)$ result for exact matching [49]. Finally, we mention that Myers [33] has independently arrived at a sublinear algorithm based on two important techniques: Divide the pattern into length $\log_b n$ blocks, then:

(1) For each block, generate its "neighborhood" at the given error rate, and look up each element in a linear-size index of the text.
(2) Apply a "doubling trick" in order to extend a block match at the given error rate.

The success of [6] and [33] suggests the combinatorial complexity of string neighborhood is not as high as current bounds would indicate.

4. Biological Applications. Motivation for the following problems are given in the Introduction.

4.1. Substring Matching (Local Similarity)

(k, l) APPROXIMATE SUBSTRING MATCHING PROBLEM. Given T of length n, P of length m, and parameters l and k, find all i such that some substring of T ending at i matches a length l substring of P to at most k differences.

For $k < k^* = l/(\log_b m + c_1) - c_2$ (same constants as in Section 3), the linear and sublinear expected-time algorithms carry over nearly verbatim. The only changes are: $l - k$ appears instead of $m - k$, and the dynamic programming checking phase is more expensive. The Main Lemma and its proof remain correct when $(m - k^*)$ is replaced by $(l - k^*)$, which implies Pr[length of $k + 2$ maximum jumps $\geq l - k] < 1/m^3$. (For the sublinear version, Pr[length of $k + 1$ maximum jumps $\geq (l - k)/2] < 1/m^3$.) When dynamic programming is invoked, let Q denote the block of text to be matched against all length l substrings of P, and let q be the length of Q. The naive method requires $m - l + 1$ applications of $O(kq)$ dynamic programming. In practice, the following is likely to be an improvement.

Instead of matching Q against each substring of P, consider the reciprocal problem of matching P (as text) against Q. If we *bootstrap* and compute the matching statistics of P against Q, we can further reduce the problem size. Finally, dynamic programming is applied, matching Q against those substrings of P that are not yet eliminated.

4.2. Approximate-Overlap (Sequence Assembly)

ρ APPROXIMATE-OVERLAP DETECTION PROBLEM. Given P of length m, Q of length n, and parameter ρ, find all l such that the length l prefix of P matches a suffix of Q to within ρl differences.

Previous algorithms [37] are $O(\rho mn)$. (The dynamic programming algorithm for approximate string matching can be used to solve this problem: match Q against P; look for entries $D(l, n)$ in the last column that are less than or equal to ρl. Note that $D(l, n)$ is the minimum edit distance between $P[1, \ldots, l]$ and suffixes of Q. See [7] for a proof that this algorithm is $O(\rho mn)$ for random strings.) Our method builds a suffix tree of Q and computes matching statistics of P against Q. For each $i = 1, 2, \ldots, \rho m + 1$, if the length of i maximum jumps starting at $P[1]$ is at least $(i - 1)/\rho$, then apply the dynamic programming (DP) method to the prefix of P covered by those jumps. There is a constant c such that $\rho < 1/(\log_b m + c)$ implies Pr[DP is invoked for $i] < 1/m^3$. The expected running time is $O(\rho m \log_b m)$.

4.3. *Codon Matching (DNA-Protein Matching)*

APPROXIMATE CODON MATCHING PROBLEM. $T \in \Sigma^*$ and $P \in \Pi^*$ are from different alphabets. The translation map $\text{tr}: \Sigma^t \to \Pi$ via t-tuples is not necessarily one-to-one (i.e., different *codons* may translate to the same letter). A string σ over Σ *translates* to a string π over Π if $\text{tr}(\sigma_1\sigma_2\cdots\sigma_t) = \pi_1$, $\text{tr}(\sigma_{t+1}\cdots\sigma_{2t}) = \pi_2$, etc. Find all locations i such that some substring of T ending at i is within edit distance k of a string that translates to P.

If there were no *indels* (insertions or deletions) of single letters from T, matching could be performed after translation, i.e., $\text{tr}(T[1,\ldots,t])\text{tr}(T[t+1,\ldots,2t])\cdots$ against P. If the translation is one-to-one, we can *reverse translate*, i.e., match T against $\text{tr}^{-1}(P[1])\text{tr}^{-1}(P[2])\cdots$. The difficulty is in having both types, multiple encodings and framing errors caused by indels.

The scanning phase of our algorithm computes a modified matching statistics $M_i = \max\ l$ such that $T[i,\ldots,i+l-1]$ *translates* to a substring of P. The suffix tree used is that of P over alphabet Π; t separate passes over the text string T combine to produce the matching statistics (for every possible framing). The "jump" that starts at S_j (as in *LET*) is the *farthest* jump considering all possible framings; the next start position S_{j+1} is one letter beyond: $S_{j+1} = \max\ S_j + i + M_{S_j+i} + 1$ $(0 \le i \le t-1)$. Note that this gives a conservative estimate of the error. If $k+2$ jumps span at least $tm-k$ letters of the text (i.e., $S_{j+k+2} - S_j \ge tm - k$), resort to the checking phase; otherwise there can be no match. It follows from a simple variation of the Main Lemma that if $k < m/(\log_b m + O(1))$, then $\Pr[S_{j+k+2} - S_j \ge tm - k] < 1/m^3$.

The checking phase requires a dynamic programming algorithm for approximate codon matching. Let $\text{ed}'(u, v)$ be the natural generalization of edit distance to this domain where $u \in \Sigma^*$ and $v \in \Pi^*$: find the minimum x such that u is within x differences of a string that translates to v. Let

$$D'(i,j) = \min_{j'} \text{ed}'(P[1,\ldots,i],\ T[j',\ldots,j]),$$

analogous to the dynamic programming formulation of approximate string matching (table D). It is clear that $D'(i,0) = ti$ and $D'(0,j) = 0$. The recurrence for $D'(i,j)$ at first appears complicated because a codon for $P[i]$ may be a noncontiguous subsequence of some block $T[j-l+1,\ldots,j]$ against which $P[i]$ is matched. However, we make the following observation: If $P[i]$ has only one codon w, let D_w be the $t+1$ by $n+1$ dynamic programming table matching T against pattern w, whose top row (boundary condition) is given by row $i-1$ of D'. Then row i of D' is just the bottom row of D_w. If, on the other hand, $P[i]$ has a set of codons, then row i of D' is found by minimizing over that set. The work to compute row i of table D' is $O(tn \cdot$ number of codons for $P[i])$. The total work to compute D' is therefore within a factor ($t \cdot$ average multiplicity) of classical $O(mn)$ dynamic programming.

Conclusions. We have given a systematic treatment of exact and approximate string matching based on suffix trees and matching statistics, simplifying or improving upon several previous results. The main analytical result is an application of Chernoff bounds to string matching, from which we derive a sublinear expected-time algorithm when the error rate is less than logarithmic fraction. The edit distance model can be replaced by Hamming distance or longest common subsequence metric. Variations of our techniques can be used to give simple and practical solutions to a wide range of problems.

Finally, we would like to point out a difference between our approach and several heuristics used in biology which also consist of a scanning phase and a checking phase. In the absence of a clear definition of "match," these programs are tuned using positive and negative controls, typically data sets with known homology (or lack of it). Since comparison with exhaustive search is generally not available as a control, false negatives cannot be detected except by eye. We hope our techniques can be incorporated into these methods to make them more rigorous.

Acknowledgments. We would like to thank Dan Gusfield, Sampath Kannan, Peter Li, Dalit Naor, Frank Olken, and Tandy Warnow who are our colleagues in computational biology. Members of the Lawrence Berkeley Laboratory Human Genome Center, particularly Cassandra Smith, Sylvia Spengler, and Frank Olken suggested applications of our work. David Aldous, Maxime Crochemore, David Haussler, Dick Karp, Gad Landau, Gene Myers, Kunsoo Park, Esko Ukkonen, and Mike Waterman provided helpful comments and encouragement. Finally, we would like to thank the referees whose comments we found extremely helpful.

References

[1] A. V. Aho and M. J. Corasick, Efficient String Matching: An Aid to Bibliographic Search, *Comm. ACM* **18** (1975), 333–340.

[2] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, A Basic Local Alignment Search Tool, *J. Molecular Biology* **215** (1990), 403–410.

[3] A. Apostolico, The Myriad Virtues of Subword Trees, in A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words*, NATO ASI Series F, Vol. 12, Springer-Verlag, New York, 1985, pp. 85–96.

[4] W. I. Chang, Fast Implementation of the Schieber–Vishkin Lowest Common Ancestor Algorithm, Computer program, 1990.

[5] W. I. Chang, Approximate Pattern Matching and Biological Applications, Ph.D. thesis, University of California, Berkeley, August 1991. Also available as Computer Science Division Reports UCB/CSD 91/653–654.

[6] W. I. Chang, Approximate String Matching and Local Similarity, *Proc. Fifth Annual Symposium on Combinatorial Pattern Matching*, Asilomar, CA, June 5–8, 1994, Lecture Notes in Computer Science, Springer-Verlag, Berlin, in press.

[7] W. I. Chang and J. Lampe, Theoretical and Empirical Comparisons of Approximate String Matching Algorithms, *Proc. Third Annual Symposium on Combinatorial Pattern Matching*, Tucson, AZ, April 29–May 1, 1992, Lecture Notes in Computer Science, Vol. 644, Springer-Verlag, Berlin, 1992, pp. 175–184.

[8] W. I. Chang and E. L. Lawler, Approximate String Matching in Sublinear Expected Time, *Proc. 31st Annual IEEE Symposium on Foundations of Computer Science*, St. Louis, MO, Oct. 22–24, 1990, pp. 116–124.

[9] W. I. Chang and E. L. Lawler, Approximate String Matching and Biological Sequence Analysis (poster), *Human Genome II Official Program and Abstracts*, San Diego, CA, Oct. 22–24, 1990, p. 24.

[10] B. Clift, D. Haussler, R. McConnell, T. D. Schneider, and G. D. Stormo, Sequence Landscapes, *Nucleic Acids Res.* **14**(1) (1986), 141–158.

[11] M. Crochemore, Longest Common Factor of Two Words, *Proc. TAPSOFT '87*, Lecture Notes in Computer Science, Vol. 249, Springer-Verlag, Berlin, 1988, pp. 26–36.

[12] R. F. Doolittle, ed. *Molecular Evolution: Computer Analysis of Protein and Nucleic Acid Sequences, Methods in Enzymology*, Volume 183, Academic Press, New York, 1990.

[13] E. R. Fiala and D. H. Greene, Data Compression with Finite Windows, *Comm. ACM* **32**(4) (1989), 490–505.

[14] Z. Galil and R. Giancarlo, Data Structures and Algorithms for Approximate String Matching, *J. Complexity* **4** (1988), 33–72.

[15] Z. Galil and K. Park, An Improved Algorithm for Approximate String Matching, *SIAM J. Comput.* **19**(6) (1990), 989–999.

[16] G. H. Gonnet and R. Baeza-Yates, *Handbook of Algorithms and Data Structures: in Pascal and C*, 2nd edn., Addison-Wesely, Reading, MA, 1991.

[17] D. Gusfield, *Efficient Algorithms for String Manipulation and Pattern Matching*, Lecture Notes, University of California, Davis, 1989.

[18] D. Gusfield, K. Balasubramanian, and D. Naor, Parametric Optimization of Sequence Alignment, *Proc. Third Annual ACM–SIAM Symposium on Discrete Algorithms*, Jan. 1992, pp. 432–439.

[19] D. Gusfield, G. M. Landau, and B. Schieber, An Efficient Algorithm for the All Pairs Suffix–Prefix Problem, *Proc. Sequences 91*, Italy, July 1991.

[20] X. Huang, A Contig Assembly Program Based on Sensitive Detection of Fragment Overlaps, *Genomics* **14**(1) (1992), 18–25.

[21] L. C. Hui, Color Set Size Problem with Applications to String Matching, *Proc. Third Annual Symposium on Combinatorial Pattern Matching*, Tucson, AZ, April 29–May 1, 1992, Lecture Notes in Computer Science, Vol. 644, Springer-Verlag, Berlin, pp. 230–243.

[22] P. Jokinen, J. Tarhio, and E. Ukkonen, A Comparison of Approximate String Matching Algorithms, Manuscript, 1990.

[23] S. Kannan and T. Warnow, Inferring Evolutionary History from DNA Sequences, *Proc. 31st Annual IEEE Symposium on Foundations of Computer Science*, St. Louis, MO, October 1990, pp. 362–371.

[24] S. Karlin, F. Ost, and B. E. Blaisdell, Patterns in DNA and Amino Acid Sequences and Their Statistical Significance, in M. S. Waterman, ed., *Mathematical Methods for DNA Sequences*, CRC Press, Boca Raton, FL, 1989, pp. 133–157.

[25] R. M. Karp, *Probabilistic Analysis of Algorithms*, Lecture notes, University of California, Berkeley, Spring 1988; Fall 1989.

[26] R. M. Karp and M. O. Rabin, Efficient Randomized Pattern-Matching Algorithms, *IBM J. Res. Develop* **31** (1987), 249–260.

[27] J. D. Kececioglu, Exact and Approximate Algorithms for DNA Sequence Reconstruction, Ph.D. thesis, University of Arizona, Tucson, 1991. Also available as Technical Report TR91-26, Computer Science Department, University of Arizona, Tucson.

[28] D. E. Knuth, J. H. Morris, and V. R. Pratt, Fast Pattern Matching in Strings, *SIAM J. Comput.* **6**(2) (1977), 323–350.

[29] G. M. Landau and U. Vishkin, Fast String Matching with k Differences, *J. Comp. System Sci.* **37** (1988), 63–78.

[30] G. M. Landau and U. Vishkin, Fast Parallel and Serial Approximate String Matching, *J. Algorithms* **10** (1989), 157–169.

[31] V. Levenshtein, Binary Codes Capable of Correcting Deletions, Insertions and Reversals, *Soviet Phys. Dokl.* **6** (1966), 126–136.

[32] E. M. McCreight, A Space-Economical Suffix Tree Construction Algorithm, *J. Assoc. Comput. Mach.* **23**(2) (1976), 262–272.

[33] E. W. Myers, A Sublinear Algorithm for Approximate Keyword Matching, Technical Report TR90-25, Computer Science Department, University of Arizona, Tucson, September 1991.

[34] National Center for Human Genome Research, *Understanding Our Genetic Inheritance* (The U.S. Human Genome Project: The First Five Years FY 1991–1995), NIH Publication No. 90-1580, April 1990.

[35] K. Park, Fast String Matching On the Average, Manuscript, 1990.

[36] W. R. Pearson and D. J. Lipman, Improved tools for biological sequence comparison, *Proc. Nat. Acad. Sci. USA* **85** (1988), 2444–2448.

[37] H. Peltola, H. Söderlund, and E. Ukkonen, SEQAID: A DNA Sequence Assembling Program Based on a Mathematical Model, *Nucleic Acids Res.* **12**(1) (1984), 307–321.

[38] M. Rodeh, V. R. Pratt, and S. Even, Linear Algorithms for Data Compression via String Matching, *J. Assoc. Comput. Mach.* **28**(1) (1981), 16–24.

[39] D. Sankoff and J. B. Kruskal, eds., *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, MA, 1983.

[40] B. Schieber and U. Vishkin, On Finding Lowest Common Ancestors: Simplification and Parallelization, *SIAM J. Comput.* **17**(6) (1988), 1253–1262.

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

[42] E. Ukkonen, Finding Approximate Patterns in Strings, *J. Algorithms* **6** (1985), 132–137.

[43] E. Ukkonen, Personal communications.

[44] E. Ukkonen and D. Wood, Approximate String Matching with Suffix Automata, Report A-1990-4, Department of Computer Science, University of Helsinki, April 1990.

[45] M. S. Waterman, Sequence Alignments, in M. S. Waterman, ed., *Mathematical Methods for DNA Sequences*, CRC Press, Boca Raton, FL, 1989, pp. 53–92.

[46] M. S. Waterman, M. Eggert, and E. Lander, Parametric Sequence Comparisons, *Proc. Nat. Acad. Sci. USA* **89** (1992), 6090–6093.

[47] P. Weiner, Linear Pattern Matching Algorithms, *Proc. IEEE Symposium on Switching and Automata Theory*, 1973, pp. 1–11.

[48] S. Wu, U. Manber, and E. Myers, Improving the Running Times for Some String Matching Problems, Technical Report TR91-20, Computer Science Department, University of Arizona, Tucson, August 1991.

[49] A. C. Yao, The Complexity of Pattern Matching for a Random String, *SIAM J. Comput.* **8** (1979), 368–387.