Sublinear Expected Time Approximate String Matching and Biological Applications [1]

William I. Chang [2] Eugene L. Lawler [3]

Abstract

The *k differences approximate string matching problem* specifies a text string of length n, a pattern string of length m, the number k of differences (substitutions, insertions, deletions) allowed in a *match*, and asks for all locations in the text where a match occurs. 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 else exponential space). We are interested in much faster algorithms for restricted cases of the problem, such as when the text string is random and the allowable error rate is not too high (*log-fraction*). We have devised an algorithm that is *sublinear* time $O(n/m)k\log_b m)$ *on the average*, when 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 still an improvement in that it is practical and uses $O(m)$ space compared to $O(n)$ or $O(m^2)$. We define three problems inspired by molecular biology and describe efficient algorithms based on our techniques: (1) *approximate substring matching*; (2) *approximate-overlap detection*; and (3) *approximate codon transcription.* Respectively, applications to genetics are local similarity search; sequence assembly; and DNA-protein matching.

keywords: pattern matching, sequence analysis
abbreviated title: Sublinear Approximate Matching

---

[1]This research was conducted at the University of California, Berkeley, and 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 grant DE-FG03-90ER60999. Earlier versions of this paper appeared as [8] and part of [6].

[2]Cold Spring Harbor Laboratory, Hershey Bldg., P.O. Box 100, Cold Spring Harbor, NY 11724. *Electronic mail:* wchang@cshl.org

[3]Computer Science Division, University of California, Berkeley, CA 94720. *Electronic mail:* lawler@ucbarpa.berkeley.edu

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 non-exact, or *approximate* match is still meaningful. In the field of molecular biology, for example, a genomic data base 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* between them, have recently received some statistical analysis of significance (see [29, 54]). 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. The proposed *sequence tagged sites* (STS) genomic map database of the Human Genome Project [41], 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. Gusfield, *et al.* [20, 21]).

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, Pratt (*KMP*) [32]; Boyer, Moore (*BM*) (also see [32]); and Karp, Rabin [31]. Each of these algorithms preprocesses the pattern and then scans the text string on-line. Slightly earlier Weiner [55] solved the complementary problem where the long text string T is fixed and one must locate within it keywords which are given on-line, 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 [38], is most widely known for its role in fast implementations of the Lempel-Ziv data compression algorithm (see for example [14, 46]). Recent applications of suffix trees include methods for finding biologically significant motif patterns in DNA [18] and elegant linear time algorithms for computing (1) the longest substrings common to k out of m strings, for all k [26]; (2) the pairwise maximum exact overlaps of m strings [22].

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* [12]. We have observed that a linear time algorithm can be derived from the incremental suffix tree construction given in [38]. (This observation was made simultaneously and independently by Ukkonen [52].) This is pedagogically satisfying and unifies several previous results, for example one-pass multiple-keyword search [1]. Furthermore, it allows us to simplify the current best approximate matching methods (see below). Our technique uses in a crucial way a set of pointers, called *suffix links*, originally used by McCreight [38] in suffix tree construction. Subsequent papers paid no attention to these pointers. After building a suffix tree of the pattern, we can compute for each position i of the text the longest substring that begins at position i and appears somewhere in the

pattern. We call this information *matching statistics* and have used it to solve the following *longest common substring query problem*: Preprocess T and P in linear time and space, in order to answer queries of the type "Given indices i and j, output longest common substring(s) between $T[i,\ldots,j]$ and P" in time linear in the size of the output string.

We can compute matching statistics in a single linear scan of the text. In **2.1** (section 1 of chapter 2) we give an abstract definition of the suffix tree (as opposed to definition by construction [38]), which simplifies the description and analysis of the matching statistics algorithm (**2.2**). For completeness we include in **2.3** a *sublinear* exact matching algorithm similar to BM but superior when the alphabet is small or m is large. On average it examines only $O((n/m)\log_b m)$ letters of the text (where b is alphabet size). Such a result was first stated by Knuth, Morris, Pratt [32], but their version has the drawback that in the worst case it must resort to the basic KMP algorithm in order to maintain linearity. Our variant uses a suffix tree of the pattern plus matching statistics and is inherently linear. Recently Park [43] has found yet another way to achieve the same expected running time, using a suffix tree of the *reverse* of the pattern and facts from string combinatorics.

1.3 Approximate Matching

Beginning in the mid 1980's, genetics and DNA sequence analysis research provided the impetus for advances in non-exact string matching. The k *differences approximate string matching problem* specifies, in addition to T and P, the parameter k of *differences* (insertions, deletions, substitutions) 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.) The minimum number of *indels* (insertions and deletions) and substitutions needed to transform one string to another is called *edit distance* or Levenshtein distance [36].

The classical dynamic programming algorithm [49] computes (column by column) an $m+1$ by $n+1$ table whose entry $D(j,i)$ is the minimum number of edit operations (substitutions, insertions, deletions) necessary to transform the length j *prefix* of the pattern into *some* text fragment ending at the i-th letter. (Boundary conditions are $D(j,0) = j$ and $D(0,i) = 0$. There is a match ending at text position i if and only if entry $D(m,i)$ is at most k.) There is a simple recursive formula giving each entry in terms of the three

adjacent entries above and to the left:

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

where $I_{ji} = 0$ if $P[j] = T[i]$; $I_{ji} = 1$ if $P[j] \neq T[i]$. The three expressions in the **min** correspond respectively to deleting $P[j]$ from the pattern; inserting $T[i]$ into the pattern; and substituting $T[i]$ for $P[j]$. It can be seen from the recurrence that (1) adjacent entries along rows and columns differ by at most one; and (2) forward diagonals ($\searrow$) are non-decreasing and adjacent entries differ by at most one. More recent methods by Ukkonen, *et al.* [51, 53, 27]; Landau, Vishkin [33, 34] (survey and refinements by [15]); and Galil, Park [16] 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 (non-decreasing) diagonal monotonicity property. Landau, Vishkin (**kn.lv**) [33, 34] 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 in constant time required a complicated algorithm [24]. Schieber, Vishkin [48] found a new LCA algorithm, which we implemented efficiently [5]. (2) It was not known how to compute matching statistics using only the suffix tree of P; we now have a simple solution. The overall space requirement of kn.lv in addition to input/output is $O(m)$. It now appears to be the best among $O(kn)$ worst case algorithms.

When k is a constant-fraction of m, faster 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* or even faster algorithms are possible. The threshold on k for linear expected time is *log-fraction*: $k < k^* = m/(\log_b m + c_1) - c_2$ (for suitable constants c_i). The main tool is matching statistics, which is a a summary of *all* exact matches between fragments of the text and pattern. Computed in a single left-to-right scan of the text, this information is used to *infer* the non-existence of matches, and to eliminate most text locations from consideration. 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 (**3.1**; **let.cl**). An algorithm similar to let.cl (discovered independently, but without analysis of

threshold) was recently implemented by Jokinen, Tarhio, and Ukkonen [27], and was the fastest for small k among algorithms they tested.

For $k < k^*/2 - 3$, portions of the text can even be skipped (**3.2**; set.cl); 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, for random text approximate matches to k^* differences or fewer are so infrequent that running time is dominated by the effort needed to *verify* for almost all positions that there is no match. Our algorithms do so in linear expected time or even sublinear expected time. 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 ($b = 2$) percent. 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 ($b = 2$) percent.

The gaps and the mystery of these percentages were motivations for the work described in a companion paper Chang, Lampe [7], where it is conjectured that, the *minimum value* of row m of the dynamic programming table (i.e. best match) is $(1-1/\sqrt{b}+o(1/\sqrt{b}))\cdot(m-\Theta(\log_b n))$ as $m, n \to \infty, n < b^m$ (**Conjecture 3** of [7]). In **3.3** we describe an extension based on this conjecture to *constant-fraction* error, using space polynomial in m, length of pattern, and linear or sublinear expected time. Very briefly, this is done by extending the notion of matching statistics to *non-exact* matches, and by *bootstrapping*: treat the pattern as *text*, and short text fragments as *pattern*. In contrast, another algorithm by Ukkonen [51] runs in linear time but requires exponential space.

We would like to point out a difference between our approach and several algorithms used by biologists which also consist of a scanning phase and a checking phase. Typically in the biology literature a *program* is described—but the nature of the output is not clearly specified. These programs are certainly useful in practice, but are not *algorithms* in the strict sense of mathematics and computer science. These *heuristics* are not guaranteed to find every *match*—which often is left undefined. A *control* is sometimes missing: a data set with comprehensive matching information produced by an *exhaustive* method should be included for comparison, or as definition of *match*. We on the other hand are primarily interested in fast, practical algorithms that can be rigorously analyzed with respect to correctness and efficiency. While "tweaks" are necessary in practice, the basic algorithm should be sound.

1.4 Biological Applications

In **4.1–4.3** we apply our techniques to several problems inspired 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, compute the *union* of $m - l + 1$ applications of k differences approximate string matching, one for each length l substring of P. That is, find all i such that some substring of T ending at i matches a length l substring of P.

This is a general method for finding *local similarities*. (We should mention that matching statistics is already a useful local similarity measure (cf. [2, 37, 44, 56]); a natural extension would be to assign weights to matched words.)

ρ Approximate-Overlap Detection Problem. Given T of length n and P of length m, find all l such that the length l suffix of P matches a prefix of T 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 (Myers [40]; 98% according to Peltola, Söderlund, Ukkonen [45]). In this application, 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 the average.

Approximate t-Codon Transcription Problem. $T \in \Sigma^*$ and $P \in \Pi^*$ are from different alphabets. The transcription map tr: $\Sigma^t \to \Pi$ via t-tuples is not necessarily *one-to-one* (i.e. different *codons* may transcribe to the same letter). A string σ over Σ *transcribes* to a string π over Π if $\mathrm{tr}(\sigma_1\sigma_2\cdots\sigma_t) = \pi_1$; $\mathrm{tr}(\sigma_{t+1}\cdots\sigma_{2t}) = \pi_2$; etc. Find all locations i such that some $T[i', \ldots, i]$ is within edit distance k of a string that transcribes 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 transcribed into one of twenty 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 $\Sigma \cup \{\$\} = \Sigma'$ 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 suffix of $P\$$ or a branching word. 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 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:

$$\begin{aligned} \text{root} &\longleftrightarrow \Lambda \\ \{\text{ internal nodes }\} &\longleftrightarrow \{\text{ branching words }\} \\ \{\text{ leaves }\} &\longleftrightarrow \{\text{ suffixes }\} \end{aligned}$$

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 labelled with a triple (x, l, r) chosen such that $P[l] = x$ and $u^{-1}v = P[l,\ldots,r]$ (choice of l, r may not be unique). We will write son$(u, x) = v$, first$(u, x) = l$, and 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} $\longrightarrow$ {internal nodes} given by shift$(w) = w_1^{-1}w$ $(w \neq \Lambda)$. McCreight [38] 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 [35] reduced this problem to constructing the suffix tree of $P\$T$, but that required either $O(n+m)$ additional space or computing everything twice (by building overlapping $O(m)$-size suffix trees). The vector $M(T,P)$ is called "Best-Fit" in their paper. Galil and Giancarlo [15] 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 there exist satisfactory automata-based solutions [12, 10], since the approximate matching algorithm given in [15] 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 letter-at-a-time. Subsequent longest matches are found by following suffix links and cleverly going down the tree (see comments).

Matching Statistics Algorithm.

```
1   construct suffix tree S(P$) by McCreight's algorithm [38]
2   let v ← root; j ← 1; k ← 1
3   for i ← 1 to n do
3.1   while (j < k) and (j+len(v,T[j]) ≤ k) do
        v,j ← son(v,T[j]),j+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
```

$v \leftarrow \text{son}(v, T[j])$; $j \leftarrow k$ end if
end while
end if

3.3 $M(T,P)_i \leftarrow k - i$
if $(j = k)$ $M'(T,P)_i \leftarrow v$
otherwise $M'(T,P)_i \leftarrow \text{son}(v, T[j])$

3.4 if v is root and $(j = k)$ then
$j \leftarrow j + 1$; $k \leftarrow k + 1$ end if
if v is root and $(j < k)$ then
$j \leftarrow j + 1$ end if
if v is not root then $v \leftarrow \text{shift}(v)$ end if

Comments. Variables i, j, k are indices into the text string; the i-th iteration of step 3 computes $M(T,P)_i$ and $M'(T,P)_i$; position k of 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.

Together invariants (i), (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.

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

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. Since m iterations of step 3. take $O(m)$ time, this computation can be made *real-time* by buffering m letters of the input text.

As a corollary we get an *on-line* exact matching algorithm based on suffix trees, that uses $O(m)$ space and preprocessing: there is an exact match ending at position $k - 1$ iff k takes on the value $i + m$ during step 3.2.

Vectors M and M' are the main ingredients needed to speed up approximate string matching. ("Jumps" $J(j,i) =_{\text{def}}$ length of longest common prefix between $P[j,\ldots,m]$ and $T[i,\ldots,n]$ are computed using the identity $J(j,i) = \min(M_i$, length of word corresponding to LCA$(M'_i,$ suffix $P[j,\ldots,m]\$))$.) In their papers Landau and Vishkin stated $O(n+m)$ as the space required by their algorithm; Galil and Giancarlo [15] made the same remark concerning Landau and Vishkin's algorithm, and in recent papers Galil and Park [16], Ukkonen and Wood [53] gave $O(m^2)$ space "practical" algorithms. With a little care Landau and Vishkin's algorithm can now be implemented very efficiently in $O(m)$ space, using the matching statistics subroutine given above and Shieber and Vishkin's very fast LCA computation [48]. For a fixed, finite alphabet this appears to be the first practical algorithm to run in $O(m)$ space and $O(nk)$ time in the worst case.

2.3 Exact Matching in Linear Worst Case and Sublinear Expected Time

Let b denote alphabet size. Let $T[1,\ldots,n]$ and $P[1,\ldots,m]$ denote the text and pattern respectively. For $s = m, 2m, 3m, \ldots$, let R_s denote the list of indices j such that the string $T[s+1,\ldots,j]$ is a suffix of P (by convention, assume $s \in R_s$); let L_s denote the list of indices i such that $i+m \leq \max R_s$ and the string $T[i+1,\ldots,s]$ is a prefix of P. The following is evident:

Claim. There is a match ending at position j iff for some s, $j-m \in L_s$ and $j \in R_s$.

Algorithm. After building suffix tree $S(P\$)$, list R_s and then list L_s can be efficiently computed using a tree-walk similar to that used to compute matching statistics. (Each prefix of P corresponds to a position in $S(P\$)$ above the leaf $P[1,\ldots,m]\$$; each suffix of P corresponds to a node with an out-edge labelled \$.) Then lists L_s and R_s can be "merged" to produce all pairs i,j s.t. $i = j-m$.

For random text, Pr[$T[s+1,\ldots,s+2\log_b m]$ is not a substring of P] $> 1-m^{-1}$; if this is the situation, then $\max R_s < s+2\log_b m$. This implies L_s can usually be found by looking only at $T[s-m+1,\ldots,s-m+4\log_b m]$, because Pr[$T[s-m+2\log_b m+1,\ldots,s-m+4\log_b m]$ is not a substring of P] $> 1-m^{-1}$. Average case running time is $O((n/m)\log_b m)$, and worst case running time is $O(n)$.

3 Linear and Sublinear Algorithms

3.1 Log-Fraction Error in 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, and (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.cl). 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 j^{th} 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 (kn.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 kn.lv to remember the last column it computed, so it can resume computation from that point if S_j is to the left of that position.

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 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 $p + d - 1$. *Q.E.D.*

Next we show 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 i.i.d. 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 there are at most m different words of length $\lg m + d$ out of $m2^d$ different strings of that length, we have

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

In particular $\mathrm{E}[X_1] < \lg m + 2$. 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

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

provided we choose $c_1 > 3$. This implies $\mathrm{E}[Y_i] < 0$, so we can apply the Chernoff bound technique (see [30]) to get:

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

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

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

Therefore for any $t > 0$

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

where the first term of the summation ($d = 0$) bounds $\mathrm{E}[e^{tY_1} | Y_1 \leq \lg m - (m - k^*)/(k^* + 2)]$ and the remaining terms bound $\mathrm{E}[e^{tY_1} | Y_1 > \lg m - (m - k^*)/(k^* + 2)] \cdot \Pr[Y_1 > \lg m - (m - k^*)/(k^* + 2)]$. Choose $t = (\log_e 2)/2$. Then

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

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

$$\begin{aligned}\mathrm{E}[e^{tY_1}]^{k^*+2} &< \sqrt{2}^{(k^*+2)\lg m-(m-k^*)+3.6(k^*+2)} \\ &< \sqrt{2}^{(4.6-c_1)(k^*+2)-(c_2-2)(c_1+\lg m)}\end{aligned}$$

after some algebra. This is less than $1/m^3$ if $c_1 = 4.6$ and $c_2 = 8$. *Q.E.D.*

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[work at start position S_j given any conditioning]] = E[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. Suppose for all i the random variable Z_i = (length of longest exact match at position i of the text T with some substring of the pattern $P[1,\ldots,m]$) is independent from $T[1,\ldots,i-1]$ and has the same distribution as Z_1, and suppose $\Pr[Z_1 > \mathrm{E}[Z_1] + d\cdot \mathrm{StdDev}(Z_1)]$ decreases exponentially in d. Then there exist constants c_1 and c_2 such that for $k \leq k^* = m/(\mathrm{E}[Z_1]+c_1\cdot \mathrm{StdDev}(Z_1)) - c_2$, k differences approximate matching takes linear expected time.

Remark. This algorithm can be viewed as practically linear approximate string matching via data compression (cf. [46]).

3.2 Sublinear Expected Time

As before let $k^* = m/(\log_b m + c_1) - c_2$ ($c_1 = 4.6$, $c_2 = 8$). Approximate matching in *sublinear* expected time can be achieved when $k < k^*/2 - 3$ (**set.cl**). 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 (kn.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 is beyond R] $< 1/m^3$ (essentially, divide expressions $(m-k^*)$ and (k^*+2) by 2 everywhere in the proof), so kn.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 $2nL/(m-k)$.

Remark. A variation of this algorithm examines only $nL/(m-k-L)$ letters of text, on the average.

A combination of the linear (for $k \geq k^*/2-3$) and sublinear algorithms runs in $O((n/m)k \log m)$ expected time, for any $k < k^*$.

3.3 Extension to Constant-Fraction Error

In let.cl and set.cl the total work of the subroutine (kn.lv or any other DP algorithm) 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 slightly greater than k^* or $k^*/2-3$. Nevertheless, it is desirable to improve the error threshold k^*.

One approach to improving the error threshold k^* is to divide the pattern into 2^s fragments each of length $m' = m/2^s$, and match each fragment against the text, allowing $k'^* = m'/(\log_b m' + c_1) - c_2$ errors. Next, apply a "doubling trick" [40]. If a block matches the text, try to double the length of the match, allowing twice as many errors as before, by applying an edit distance computation to the left and right of the matched portions of the text and pattern. Repeat the doubling step for any remaining matches. One is guaranteed to find all matches to $k = k'^* m/m'$ differences because either the left or right half of P will have at most $k/2$ errors; then either the left or right half of that piece will have at most $k/4$ errors, and so on. This is a gain if $k'^* m/m' > k^*$. (Because of the subtractive c_2 in the formula for error threshold, this is not always the case.) For example, over a four-letter alphabet $k^* = 139$ for $m = 1000$. By dividing P into four pieces each of length $m' = 250$, we get $k'^* = 38$ so $4k'^* = 152$, a small improvement on k^*.

Alternatively, let us define h *differences matching statistics* $M^{(h)}(T,P)_i = (l, X)$ where X = the set of substrings of P that are at most h differences from

$T[i,\ldots,i+l-1]$, such that X is non-empty and l is maximized. Computing generalized matching statistics efficiently appears to be a hard problem, even for $h = 1$. In principle, one can enumerate all strings that are at most h differences from some family of substrings of P, and use a modified *digital trie* to compute "maximum jumps." By using $M^{(h)}(T,P)$ instead of $M(T,P)$, the error threshold of our algorithms can be improved substantially, but at a possibly prohibitive cost in space requirement. Recall from the introduction (1.3) the following conjecture of Chang, Lampe [7]: The *minimum value* of row m of the dynamic programming table D (i.e. best match) is $(1-1/\sqrt{b}+o(1/\sqrt{b}))\cdot(m-\Theta(\log_b n))$ as $m,n\to\infty, n<b^m$ (**Conjecture 3** of [7]). This conjecture gives an indication of what can be expected if h is such that the data structure for computing $M^{(h)}$ is of size polynomial in m.

Let $l = \alpha\log_b m$. For each string x of length l, match the *pattern* P against x, treating P as text and x as pattern. Conjecture 3 of [7] implies that for some constant β, the closest match is about $(1-1/\sqrt{b})\cdot(l-\beta\log_b m))$ differences from x; let h be this value. Then an algorithm based on h differences matching statistics will find h errors per block of length l, a rate of $(1-1/\sqrt{b})\cdot(1-\beta/\alpha)$. This is a constant fraction provided $\alpha>\beta$. The data structure for $M^{(h)}$ will be of size $O(m^\alpha)$. There is a *time-space-error threshold* tradeoff. If $k>(1-1/\sqrt{b})\cdot m$ then k differences approximate matching will likely find matches *nearly everywhere*. The linear expected time method we described will therefore come within a factor of $1-\beta/\alpha$ of this natural error threshold. The sublinear expected time method will allow half as many errors.

4 Biological Applications

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, compute the *union* of $m-l+1$ applications of k differences approximate string matching, one for each length l substring of P. That is, find all i such that some substring of T ending at i matches a length l substring of P.

For $k<k^*=l/(\log_b m+c_1)-c_2$ (same constants as in 3.1), the linear and sublinear expected time algorithms from 3.1 and 3.2 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 substrings 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.

This is a general method for finding *local similarities* (cf. [2, 37, 44, 56]).

4.2 Approximate-Overlap (Sequence Assembly)

ρ Approximate-Overlap Detection Problem. Given T of length n and P of length m, find all l such that the length l suffix of P matches a prefix of T to within ρl differences.

Previous algorithms [45] are $O(\rho mn)$. (The DP algorithm for approximate string matching can be used to solve this problem: match the reverse of T against the reverse of P; look for entries in the last column that are less than or equal to ρ times row number.) Our method builds a suffix tree of P and computes matching statistics of T against P. For each $j = 1, 2, \ldots, \rho m + 1$, if the length of j maximum jumps starting at $T[1]$ is at least $(j - 1)/\rho$, then apply the DP method to the prefix of T covered by those jumps. There is a constant c such that $\rho < 1/(\log_b m + c)$ implies Pr[DP is invoked for j] $< 1/m^3$. The expected running time is $O(\rho m \log_b m)$.

In applications such as *sequence assembly*, 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.

4.3 Codon Transcription (DNA-Protein Matching)

Approximate t-Codon Transcription Problem. $T \in \Sigma^*$ and $P \in \Pi^*$ are from different alphabets. The transcription map tr: $\Sigma^t \to \Pi$ via t-tuples is not necessarily *one-to-one* (i.e. different *codons* may transcribe to the same letter). A string σ over Σ *transcribes* to a string π over Π if $\mathrm{tr}(\sigma_1\sigma_2\cdots\sigma_t) = \pi_1$; $\mathrm{tr}(\sigma_{t+1}\cdots\sigma_{2t}) = \pi_2$; etc. Find all locations i such that some $T[i',\ldots,i]$ is within edit distance k of a string that transcribes to P.

If there were no *indels* (insertions or deletions) of single letters from T, matching could be performed after transcription, i.e. $\mathrm{tr}(T[1,\ldots,t])\mathrm{tr}(T[t+1,\ldots,2t])\cdots$ against P. If the transcription is one-to-one, one can *reverse transcribe*, i.e. match T against $\mathrm{tr}^{-1}(P[1])\mathrm{tr}^{-1}(P[2])\cdots$. 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 $M_i = \max\ l$ s.t. $T[i,\ldots,i+l-1]$ *transcribes* 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.cl in 3.1) 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 in 3.1 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 t-codon transcription. Let $\mathrm{ed}'(u,v)$ 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 transcribes to v. Let $C(j,i) = \min_{i'} \mathrm{ed}'(T[i',\ldots,i], P[1,\ldots,j])$, analogous to the DP formulation of approximate string matching (table D). Boundary conditions are $C(0,i) = 0$ and $C(j,0) = tj$. The recurrence is $C(j,i) = \min\{X,Y\}$ where

$$X = C(j,i-1) + 1 \quad \text{(deletion of } T[i]\text{)}$$

$$Y = \mathbf{min}_{l \ge 0}\ C(j-1,i-l) + Y_l \quad \text{(substitution)}$$

and $Y_l = \mathrm{ed}'(T[i-l+1,\ldots,i], P[j])$ is the cost of an optimal substitution of the length l block $T[i-l+1,\ldots,i]$ for the one letter $P[j]$. The term $Y_0 = t$ is

the cost of *inserting* a codon for $P[j]$. The substitution cost is complicated because a codon for $P[j]$ may be a (non-contiguous) subsequence of the block $T[i-l+1,\ldots,i]$ against which $P[j]$ is matched.

This may appear complicated, but the following observation makes it easy to compute C.

Observation. $Y = \min_w D_w(t,i)$ ($w \in \{\text{codons for } P[j]\}$) where D_w is a $(t+1) \times (n+1)$ DP table matching T against w but with a twist: same recurrence as D, but $D_w(s,0) = C(j-1,0) + s$ and $D_w(0,i) = C(j-1,i)$.

Hence the work to compute row j of table C is $O(tn \cdot$number of codons for $P[j])$. The total work of approximate t-codon transcription is therefore only a factor ($t \cdot$ average multiplicity) worse than classical $O(mn)$ approximate string matching. Furthermore, $C(j,i) \geq C(j-1,i-t)$ by the following argument. First note that row-wise adjacent entries of C differ by at most one. Next consider $D'(s,i) = \min_w D_w(s,i)$. Table D' (which is never computed) satisfies diagonal monotonicity; $D(0,i-t) = C(j-1,i-t)$ and $D'(t,i) = Y$ implies $Y \geq C(j-1,i-t)$. Finally, induction on i shows $C(j,i) = \min\{X,Y\} \geq C(j-1,i-t)$. Hence the "cut-off" method of 3.2 can be used to speed up computation, although we have not analyzed its performance.

An immediate application is that of matching a DNA sequence against a protein (amino acid) sequence.

Acknowledgment

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 Sun Wu provided helpful comments and encouragement. Jordan Lampe wrote very challenging dynamic programming code against which we compare our algorithms.

References

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

[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 (1985), pp. 85–96.

[4] A. Blumer, J. Blumer, D. Haussler, R. McConnell, and A. Ehrenfeucht, Complete Inverted Files for Efficient Text Retrieval and Analysis, *J. ACM* 34(1987), pp. 578–595.

[5] W.I. Chang, Fast Implementation of the Schieber-Vishkin Lowest Common Ancestor Algorithm, computer program, 1990.

[6] W.I. Chang, *Approximate Pattern Matching and Biological Applications*, Ph.D. thesis, U.C. Berkeley, August 1991.

[7] W.I. Chang and J. Lampe, Theoretical and Empirical Comparisons of Approximate String Matching Algorithms, submitted.

[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, October 1990, pp. 116–124.

[9] W.I. Chang and E.L. Lawler, Approximate String Matching and Biological Sequence Analysis (poster), abstract in *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 Research* 14:1(1986), pp. 141–158.

[11] M. Crochemore, Transducers and Repetitions, *Theoretical Comput. Sci.* 45(1986), pp. 63–86.

[12] M. Crochemore, Longest Common Factor of Two Words, *Proc. TAPSOFT '87*, Springer-Verlag LNCS 249, pp. 26–36.

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

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

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

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

[17] W.B. Goad, Computational Analysis of Genetic Sequences, *Ann. Rev. Biophys. Biophys. Chem.* 15(1986), pp, 79–95.

[18] D. Gusfield, *Efficient Algorithms for String Manipulation and Pattern Matching*, lecture notes, U.C. Davis (1989).

[19] D. Gusfield, Efficient Algorithms for Inferring Evolutionary Trees, Report CSE-88-4, Computer Science Division, U.C. Davis, May 1988.

[20] D. Gusfield, K. Balasubramanian, J. Bronder, D. Mayfield, D. Naor, PARAL: A Method and Computer Package for Optimal String Alignment using Variable Weights, in preparation.

[21] D. Gusfield, K. Balasubramanian and D. Naor, Parametric Optimization of Sequence Alignment, submitted.

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

[23] P.A.V. Hall and G.R. Dowling, Approximate String Matching, *Computing Surveys* 12:4(1980), pp. 381–402.

[24] D. Harel and R.E. Tarjan, Fast Algorithms for Finding Nearest Common Ancestors, *SIAM J. Comput.* 13(1984), pp. 338-355.

[25] A. Hartman and M. Rodeh, Optimal Parsing of Strings, in A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words*, NATO ASI Series F, Vol. 12, Springer-Verlag (1985), pp. 155–168.

[26] L.C. Hui, Color Set Size Problem with Applications to String Matching, Report CSE-91-5, Computer Science Division, U.C. Davis, January 1991.

[27] P. Jokinen, J. Tarhio, and E. Ukkonen, A Comparison of Approximate String Matching Algorithms, manuscript, October 1990.

[28] 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.

[29] 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 (1989), pp. 133–157.

[30] R.M. Karp, *Probabilistic Analysis of Algorithms*, lecture notes, U.C. Berkeley (Spring 1988; Fall 1989).

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

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

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

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

[35] G.M. Landau, U. Vishkin, and R. Nussinov, Locating alignments with k differences for nucleotide and amino acid sequences, *CABIOS* 4:1(1988), pp. 19–24.

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

[37] D.J. Lipman and W.R. Pearson, Rapid and Sensitive Protein Similarity Searches, *Science* 227(1985), pp. 1435–1441.

[38] E.M. McCreight, A Space-Economical Suffix Tree Construction Algorithm, *J. ACM* 23:2 (1976), pp. 262–272.

[39] E.W. Myers, An O(ND) Difference Algorithm and Its Variations, *Algorithmica* 1(1986), pp. 252–266.

[40] E.W. Myers, personal communications.

[41] 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.

[42] P.T. Nielsen, On the Expected Duration of a Search for a Fixed Pattern in Random Data, *IEEE Trans. on Information Theory* (1973), pp. 702–704.

[43] K. Park, Fast String Matching On the Average, manuscript.

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

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

[46] M. Rodeh, V.R. Pratt, and S. Even, Linear Algorithms for Data Compression via String Matching, *J. ACM* 28:1(1981), pp. 16–24.

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

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

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

[50] E. Ukkonen, Algorithms for Approximate String Matching, *Inf. Contr.* 64(1985), pp. 100–118.

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

[52] E. Ukkonen, personal communications.

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

[54] M.S. Waterman, Sequence Alignments, in M.S. Waterman, ed., *Mathematical Methods for DNA Sequences*, CRC Press (1989), pp. 53–92.

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

[56] W.J. Wilbur, and D.J. Lipman, Rapid Similarity Searches of Nucleic Acid and Protein Data Banks, *Proc. Natl. Acad. Sci. USA* 80(1983), pp. 726–730.

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