Theoretical and Empirical Comparisons of Approximate String Matching Algorithms [1]

William I. Chang [2] Jordan Lampe [3]

Abstract

We study in depth a model of non-exact pattern matching based on *edit distance*, which is the minimum number of substitutions, insertions, and deletions needed to transform one string of symbols to another. More precisely, 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 have carefully implemented and analyzed various $O(kn)$ algorithms based on dynamic programming (DP), paying particular attention to dependence on b the alphabet size. An empirical observation on the average values of the DP tabulation makes apparent each algorithm's dependence on b. A new algorithm is presented that computes much fewer entries of the DP table. In practice, its speedup over the previous fastest algorithm is 2.5X for binary alphabet; 4X for four-letter alphabet; 10X for twenty-letter alphabet. We give a probabilistic analysis of the DP table in order to prove that the expected running time of our algorithm (as well as an earlier "cut-off" algorithm due to Ukkonen) is $O(kn)$ for random text. Furthermore, we give a heuristic argument that our algorithm is $O(kn/(\sqrt{b}-1))$ on the average, when alphabet size is taken into consideration.

keywords: approximate string matching, edit distance
abbreviated title: Theoretical and Empirical Comparisons

[1]This research was conducted at the University of California, Berkeley, and was supported in part by Department of Energy grant DE-FG03-90ER60999. An earlier version of this paper appeared as part of [3].

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

[3]Dept. of Computer Science and Engineering FR-35, University of Washington, Seattle, WA 98195.

1 Introduction and Summary of Results

Beginning in the 1980s, 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 text string T and pattern string 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 problem formulation is due to Sellers [31], and is equivalent to finding where matches begin, by reversing the strings.) In this paper, the text is assumed to be given on-line and to be scanned sequentially; the space requirement should (preferably) be linear in the length of the pattern. While this is a simple model of non-exact matching, we note that it has a rich history ([31, 34, 20, 21, 9, 10, 4, 17, 3] chronologically) and interesting combinatorics, and is a natural starting point before more complex, parametric cost functions are to be considered (e.g. Gusfield, *et al.* [12, 13]).

Notation. Text $T[1,\ldots,n]$ and pattern $P[1,\ldots,m]$ over fixed, finite alphabet of size b. The *edit distance* (also called Levenshtein distance [23]) $\mathbf{ed}(u,v)$ of two strings u,v is the minimum number of substitutions, insertions, deletions needed to transform one string into the other.

Seller's dynamic programming (DP) algorithm [31] (**mn.dp**) 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 $P[j]$.

Remark. The classical dynamic programming algorithm for computing the edit distance of two strings u,v differs from the above only in the boundary condition $D(0,i) = i$. Speedups to $O(\mathrm{ed}(u,v) \cdot$ length of shorter string) are due to Ukkonen [33] and Myers [26]. A different model of approximate matching based on *longest common subsequence* (each $I_{ji} = 2$; equivalent to not allowing substitutions) has received a great deal of attention from Myers [27] and Manber, Wu [25].

It can be seen from the recurrence that $(*)$ adjacent entries along rows and columns differ by at most one; and $(**)$ forward diagonals $(\searrow)$ are non-decreasing and adjacent entries differ by at most one. More recent methods by Ukkonen, *et al.* [34, 36, 17]; Landau, Vishkin [20, 21] (survey and refinements by [9]); and Galil, Park [10] take advantage of these geometric properties in order to compute $O(kn)$ instead of mn entries.

The simplest of these, Ukkonen's "cut-off" algorithm (**kn.uk**) [34] never computes the bottom portion of a column if those entries can be inferred to be greater than k. Despite statement in [34] that "It should be quite obvious," no rigorous analysis was done that shows kn.uk has $O(kn)$ expected running time [35]. We give the first proof of this fact in section **2**. 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**) [20, 21] computes each transition in constant time. Several practical improvements (see [6]) have made kn.lv the best among $O(kn)$ worst case algorithms. It turns out, however, that by replacing the $O(1)$ time diagonal transition subroutine with a brute-force method, the resulting algorithm (**kn.dt**) has $O(kn)$ *expected* running time and is *faster* in practice [27]. See section **3** for a succinct description.

We have done careful theoretical and empirical comparisons of these methods; apart from questions of overhead, they do not have the same dependence on b the alphabet size. We discovered a speedup of the dynamic programming method whose running time depends on the *row averages* of table D (the higher the averages, the *faster* our algorithm). Our method works by *partitioning* each column into *runs of consecutive integers* (e.g. 012 23 234), and is many times faster than previous algorithms based on dynamic programming. Its expected running time is $O(kn)$ because it is always faster than kn.uk. In addition, it has given us special insights into the *statistics* of sequence matching. With these insights we are able to formulate empirical running times of various algorithms as functions of n, m, k, and b (see **Table 1**). Variations of column partitioning are given in section **4**.

Recall that the minimum number of substitutions, insertions, and deletions needed to transform one string into another is called *edit distance*. It is a surprising fact that relatively little is known about the average case behavior of edit distance. Early qualitative results of Chvátal, Sankoff [7] and Deken [8] on *longest common subsequence* (*LCS*) can be carried over to edit distance: the expected edit distance between two uniformly random strings of size m (as $m \to \infty$) is $C_b m$ for some constant C_b that depends only on alphabet size. But exact bounds are not easily converted, and there has been no "formula" given in the literature for C_b. (Sankoff, Mainville [29] conjectured $\lim_{b\to\infty} C_b'\sqrt{b} = 2$ where C_b' are the corresponding constants for length of LCS.) Since any match where the text differs from the pattern by much fewer than C_b-fraction differences can be considered "significant," a basic understanding of these constants is of paramount importance. The primary difficulties are (1) the proof of convergence by the Subadditive Ergodic Theorem is non-constructive; and (2) the algorithmic formulation of edit distance (like approximate matching) is highly recursive, and leads to exponentially many states in the natural Markov model.

We have made the following empirical observation: columns of the dynamic programming table D for approximate matching consist of runs of consecutive integers, of average length very close to $\sqrt{b}$. This observation leads to **Conjecture 1**, row m of table D has average value $(1-1/\sqrt{b}+o(1/\sqrt{b}))\cdot m$ as $m \to \infty$; and **Conjecture 2**, $C_b = 1 - 1/\sqrt{b} + o(1/\sqrt{b})$. A probabilistic analysis of partitions of columns, subject to several simplifying assumptions, has yielded heuristic arguments (but no proof) in favor of our conjectures (see section **5**). In addition, we state **Conjecture 3**, the expected *minimum value* of row m (i.e. best match) is $(1 - 1/\sqrt{b} + o(1/\sqrt{b}))\cdot(m - \Theta(\log_b n))$ as $m, n \to \infty, n < b^m$. This conjecture implies *linear expected time, polynomial space* (in length of pattern), *constant-fraction differences* approximate string matching (see companion paper Chang, Lawler [6]).

We have carefully optimized the simpler algorithms based on dynamic programming: mn.dp (Sellers [31]); kn.uk (Ukkonen [34]); kn.dt (see Galil, Park [10]); mn.clp and kn.clp (described below). Generally, our code for these is faster than others we have seen. Several more complicated algorithms have also been implemented but are unoptimized: kn.gp (verbatim from Galil, Park [10]); kn.lv (Landau, Vishkin [21]); let.cl and set.cl (Chang, Lawler [6]). Speedup by a small factor can be expected from careful optimization. As presently implemented the *linear expected time* let.cl and *sublinear expected* *time* set.cl [6] are not competitive with kn.clp, primarily because the hash coded suffix tree [24] is not the fastest implementation possible. An algorithm similar to let.cl (discovered independently, but without analysis of threshold) was implemented by Jokinen, Tarhio, and Ukkonen [17], and was the fastest for small k among algorithms they tested. In addition, [17] includes extensive tables of running times for mn.dp; kn.uk; kn.gp; and a new algorithm [32].

Our programs are allowed only $O(m)$ space, except kn.gp which is $O(m^2)$. Text is read *on-line*. While text buffering on-the-fly is slightly slower compared to reading the entire input into real or virtual memory, we feel it is more realistic not to make a copy of the input, for the types of applications we envision: at least n in the millions; m in the hundreds; and k in the tens. Our results are summarized in **Table 1**.

2 Cut-Off Algorithm and Its Average Case Analysis

Recall that dynamic programming table D has two important geometric properties:
(*) Adjacent entries along horizontal and vertical directions differ by 0 or 1
(**) Forward diagonals are non-decreasing and adjacent entries differ by 0 or 1.

Ukkonen Cut-Off Algorithm (**kn.uk**). Let $l_i = \max j$ s.t. $D(j,i) \leq k$ ($l_0 = k$). Given l_{i-1}, compute $D(j,i)$ for j up to $l_{i-1}+1$, and set l_i to largest $j \leq l_{i-1}+1$ such that $D(j,i) \leq k$. Correctness follows immediately from property (**).

Despite statement in [34] that "It should be quite obvious," it was not previously proved that kn.uk has $O(kn)$ expected running time. We will prove this fact. Assume the text is uniformly random over a size b alphabet. Given two strings u, v, let $\text{lcs}(u,v)$ denote the length of the longest common subsequence between u, v.

Proposition 1. $D(j,i) \geq (1/2) \cdot \text{ed}(P[1,\ldots,j], T[i-j+1,\ldots,i]$.

Proof. Consider the edit distance y between the length j strings $P[1,\ldots,j]$ and $T[i-j+1,\ldots,i]$. The value $D(j,i) = x$ must come from matching $P[1,\ldots,j]$ against a text fragment ending at i whose length differs from j

by at least $y - x$ (edit distance satisfies the triangle inequality). Then $y - x$ is a lower bound on x just by consideration of length. So $x \geq y/2$. *Q.E.D.*

Proposition 2. There exist constants $c < 1$, $\beta < 1$ and α s.t. Pr[two random, length j strings have a common subsequence of length $cj] < (1/j) \cdot \alpha\beta^j$.

Proof. For convenience assume cj is an integer. By Stirling's formula $j!/(cj)!(j-cj)! = (1+o(1))\cdot(\sqrt{2\pi c(1-c)j}\cdot c^{cj}(1-c)^{(1-c)j})^{-1}$. Let $p =$Pr[length cj common subsequence]. Then $p \leq \sum b^{-cj}$ where the summation is over all size cj bipartite matchings of positions. Hence

$$p \leq \binom{j}{cj}^2 \cdot b^{-cj} = \frac{1+o(1)}{2\pi c(1-c)} \cdot \frac{1}{j} \cdot (c^c(1-c)^{1-c})^{-2j} \cdot b^{-cj}.$$

This last expression decreases exponentially in j if $\beta = (c^c(1-c)^{1-c})^{-2} \cdot b^{-c} < 1$. This condition is satisfied for all $b \geq 2$ by the choice $c = 7/8$. As $b \to \infty$, it suffices to choose $c > e/\sqrt{b}$. Choose $\alpha > (2\pi c(1-c))^{-1}$, sufficiently large to overcome the error term in Stirling's formula. *Q.E.D.*

Theorem. The expected running time of algorithm kn.uk is $O(kn)$.

Proof. It suffices to prove $\mathrm{E}[l_i] = O(k)$ since l_i bounds the work in column $i+1$. Let $l = 2k/(1-c)$, so $l - 2k = cl$ and $j - 2k \geq cj$ for all $j \geq l$. We have $\mathrm{E}[l_i] < l - 1 + \sum_{j \geq l} j \cdot \Pr[D(j,i) \leq k]$. By Proposition 1, $D(j,i) \leq k$ implies $\mathrm{ed}(u,v) \leq 2k$ where $u = P[1,\ldots,j]$ and $v = T[i-j+1,\ldots,i]$ are length j strings. Since $j-\mathrm{lcs}(u,v)$ is clearly a lower bound on $\mathrm{ed}(u,v)$, this implies $\mathrm{lcs}(u,v) \geq j - 2k \geq cj (j \geq l)$. By Proposition 2, for $j \geq l$, $\Pr[D(j,i) \leq k] \leq \Pr[\mathrm{lcs}(u,v) \geq cj] < (1/j) \cdot \alpha\beta^j$ for some constant α and constant $\beta < 1$. Hence $\mathrm{E}[l_i] < l - 1 + \sum_{j \geq l} j \cdot (1/j) \cdot \alpha\beta^j = l - 1 + O(1) = O(k)$. *Q.E.D.*

3 Diagonal Transition Algorithms

Diagonal monotonicity (**) implies the locations of the first $k+1$ transitions *along each diagonal* are sufficient to characterize D for the solution to the k differences problem [21]. A key ingredient is the "jump" $J(j,i) =$ length of the longest exact match $P[j,\ldots] = T[i,\ldots]$. The following algorithms differ only in how jumps are computed.

Landau & Vishkin Algorithm (kn.lv). Call cell $D(j,i)$ an entry of diagonal $i - j$. But instead of D, compute column by column a $(k+1) \times (n+1)$

table L where $L(x,y) = \max j$ s.t. $D(j, j+y-x) \leq x$ ($0 \leq x \leq k$; $0 \leq y \leq n$). That is, $L(x,y)$ is the row number of the last x along diagonal $y-x$. Let us first look at D the original table. Since $D(j,0) = j$, every cell of diagonal $-j$ is at least j. We can define $L(x,-1) = -\infty$ because there is no j s.t. $D(j, j-1-x) \leq x$. Likewise it is convenient to 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: $\alpha = L(x-1, y-2), \beta = L(x-1, y-1), \gamma = L(x-1, y)$, which are respectively the row numbers of the last $x-1$'s in diagonals $y-x-1, y-x, y-x+1$. More precisely, it can be inferred that $D(\alpha, \alpha+y-x) \leq 1 + D(\alpha, \alpha+y-x-1) = x$ (by an insertion into P of $T[\alpha+y-x]$). Similarly, $D(\beta+1, \beta+1+y-x) \leq x$ (by substitution) and $D(\gamma+1, \gamma+1+y-x) \leq x$ (by deletion of $P[\gamma+1]$). So along diagonal $y-x$, three cells at rows $\alpha, \beta+1, \gamma+1$ are known to be at most x. Let $j = \max(\alpha, \beta+1, \gamma+1)$. Then it is easy to see that for $j' > j, D(j', j'+y-x) = x$ iff $J(j+1, j+1+y-x) \geq j'-j$. To summarize, $L(x,y) = j + J(j+1, j+1+y-x)$.

Jumps are computed according to Chang, Lawler [6]. Two key ingredients are *matching statistics* (a summary of all exact matches between the text and pattern) and *lowest common ancestor* (LCA). In our implementation [2] of the Schieber, Vishkin LCA algorithm [30], only simple machine instructions are used (such as add, decrement, and complement, but not *bit-shift*). Logarithm in [30] is replaced by bit magic, using a table of *reversals* of binary representation of numbers. Fewer than sixty machine instructions suffice to compute an LCA. The worst case running time of $O(kn)$ for kn.lv is modulo hashing in $O(m)$ space, or deterministic in $O(bm)$ space. The $O(m)$ space hash coded implementation [24] is slower in practice.

Diagonal Transition Algorithm (kn.dt). Compute jumps by brute force. This is algorithm *MN2* in [10]; it is a variation of an edit distance algorithm given in [33]. Expected running time can be shown to be $O(kn)$, first stated by Myers [27]. Briefly, the fact that a jump at (j,i) is needed or not is determined solely by P and $T[1,\ldots,i-1]$, so $\mathrm{E}[J(j,i)|\text{jump is needed}] = 1/(b-1)$; an extra comparison is needed to find the mismatch that ends a jump. Our optimized code for kn.dt is faster than kn.uk.

Galil & Park Algorithm (kn.gp). See [10]. Strictly $O(kn)$, but requires $O(m^2)$ space for a table of lengths of exact matches $P[j,\ldots] \doteq P[j',\ldots]$.

4 Column Partition Algorithms

Column Partition Algorithm (**mn.clp**). Each column of table D can be partitioned into runs of consecutive integers: entry $D(j,i)$ belongs to *run* δ of column i iff $j - D(j,i) = \delta$ (note $j - D(j,i)$ is non-decreasing in j). For $\delta \geq 0$, we say run δ of column i ends at j if j is smallest possible such that $D(j+1,i)$ belongs to run $\delta' > \delta$. A run may be of zero length (whenever $D(j+1,i) < D(j,i)$), but $(*)$ implies no two consecutive runs δ, $\delta+1$ may both be of zero length. The goal is to compute where each run ends in *constant time*; the algorithm would then perform $O(m - D(m,i))$ work on column i, for a total of $O((m-\mu)n)$ where μ is the average of row m of D.

Proposition 3. If run δ of column i ends at j and is of zero length, then run δ of column $i+1$ ends at $j+1$.

Proof. The condition means $D(j+1,i) < D(j,i)$ and $\delta = j - D(j,i) + 1$; $(**)$ implies $D(j+1,i+1) = D(j,i)$ but $D(j+2,i+1) \leq D(j,i)$. *Q.E.D.*

Proposition 4. If run δ of column i ends at j and is of length $l \geq 1$, and $j' \in [j-l+2, j+1]$ is smallest possible such that $P[j'] = T[i+1]$, then run δ of column $i+1$ ends at $j'-1$. If no such j' exists and run $\delta+1$ of column i is not of zero length, then run δ of column $i+1$ ends at $j+1$; otherwise it ends at j.

Proof. We know $D(j-l,i) \geq D(j-l+1,i)$, so $D(j-l+1,i+1) \geq D(j-l+1,i)$ by $(**)$. Also, $D(j+2,i+1) \leq D(j+1,i)+1 \leq D(j,i)+1$. Run δ of column $i+1$ must therefore end within the range $[j-l+1, j+1]$. The proposition then follows easily from the recurrence. *Q.E.D.*

Implementation. Pre-compute and tabulate the partial function $\mathbf{loc}(j,x) =$ **min** j' s.t. $P[j'] = x$ and $j' \geq j$ (this requires $O(bm)$ space). Keep track of only the column partitions, not the actual entries of D. An alternative, $O(m)$ space implementation, using linked lists $\mathbf{loc}_x$ consisting of those j s.t. $P[j] = x$, is $O(mn)$ worst case but has the same running time in practice.

Remark. This can be viewed as a sparse matrix computation, cf. [11].

k **Differences Column Partition Algorithm** (**kn.clp**). In a manner similar to kn.uk, mn.clp can be "cut off" at k. The expected running time is $O(kn)$ because it is always faster than kn.uk. Empirically, it is much faster than previous algorithms based on dynamic programming (2.5X for binary alphabet; 4X for four-letter alphabet; 10X for twenty-letter alphabet compared to kn.dt).

Sparser k Differences Column Partition Algorithm (kn$'$.clp). Using sophisticated data structures it is possible to reduce the work on column i to $O(\log\log m)$ for each j s.t. $P[j] = T[i]$ and $O(1)$ for each run of length zero. The locations of ends of runs are stored by their *diagonal* number modulo m in a data structure that allows $O(\log\log m)$ insertion, deletion, and *nearest neighbor* lookup. When $P[j] = T[i]$ the run that needs to be modified according to Proposition 4 (i.e. would have contained cell $D(j,i)$ had $P[j] \neq T[i]$) can be looked up in the table, as a nearest neighbor of $i-j$ mod m. Run ends that need to be modified because of runs of zero length can be handled separately, by keeping a sublist of runs of zero length. Finally, the remaining run ends stay on the same diagonal so are automatically taken care of. The expected running time of this algorithm is $O(b^{-1}kn\log\log m)$. Unfortunately the overhead appears to be very high.

Remark. A similar result, for the *longest common subsequence metric* (equivalent to not allowing substitutions), is described in Manber, Wu [25].

5 Heuristic Analysis of Column Partitions

We showed in section 2 that $\mathrm{E}[D(j,i)] = \Theta(j)$; the bound we obtained is not tight, and does not fully characterize the running times of "cut-off" algorithms kn.uk and kn.clp. In this section we give a sketch of a heuristic argument that $\mathrm{E}[D(j,i)] \approx (1 - 1/\sqrt{b}) \cdot j$, which agrees very well with simulation results.

The first simplification we make is to throw away the strings and consider instead an abstract dynamic programming model given by the same recurrence and boundary conditions as $D(j,i)$ but with random variables $I'_{ji} = 1$ w.p. $1 - 1/b$; 0 w.p. $1/b$:

$$D'(j,i) = \min\ \{\ 1 + D'(j-1,i),\ 1 + D'(j,i-1),\ I'_{ji} + D'(j-1,i-1)\ \}$$

and $D'(j,0) = j; D'(0,i) = 0$. Furthermore, we let $m,n \to \infty$.

Let us call a run of length l an l-run. Let ϕ = probability that a run is of zero length (all runs equally likely to be chosen). Assume $\phi = O(1/b)$.

Next, focus on a column. Let x denote a cell chosen uniformly at random. Let $S_l = \Pr[x$ belongs to an l-run$]$ ($l \geq 1$); so $\sum_{l \geq 1} S_l = 1$. A given, longer run is more likely to be hit than a given, shorter one: $S_l = l \cdot \#l$-runs/area. Then $S_l/l = \Pr[x$ is the end of an l-run$]$, and is also the *odds* that a run of positive-length is of length l. Furthermore, the average length of a positive-length run is given by $1/\sum_{l \geq 1} S_l/l$ (*call this* λ). (We also have: $\Pr[x$ is the end of a run$] = \sum_{l \geq 1} S_l/l$; $\Pr[x$ is the k-th cell of a run$] = \sum_{l \geq k} S_l/l$; E[length of run containing $x] = \sum_{l \geq 1} S_l \cdot l$.)

The assumption that run-lengths of positive-length runs are *geometrically distributed* by length (this fits simulation data) is equivalent to the assumption that for a random cell x the events (1) it is the k-th cell of a run; (2) it is the end of a run, are *independent.* If we make this simplifying assumption, then it follows by a calculation that $S_l = (1/\lambda^2) \cdot l \cdot (1 - 1/\lambda)^{l-1}$. Also E[length of run containing $x] = 2\lambda - 1$.

Next, calculate $E[x - y]$ where y is the cell adjacent and to the left of x. This expectation approaches 0 as $n \to \infty$. A case analysis of the column partitioning process yields the following after some calculation: (1) $\Pr[x-y = 1] = \Pr[y$ is first cell of a run, and there is no "match" for the entire run above $y] = (1 + o(1)) \cdot (\lambda^{-1} + b^{-1})$; (2) $\Pr[x - y = -1] = \Pr[y$ is *not* first cell of a run, and there is some "match" above y in the run$] = (1 + o(1)) \cdot (\lambda/b)$. Hence $\lambda \approx \sqrt{b}$ (highest order term in b), and also average run length $\approx \sqrt{b}$ (highest order term in b).

Open Problems. Show $\phi = O(1/b)$. Remove the independence assumption.

Conjecture 1. $E[D(j,i)] = (1 - 1/\sqrt{b} + o(1/\sqrt{b})) \cdot j$ as $j \to \infty$.

Conjecture 2. E[edit distance between two strings of length $l] = (1 - 1/\sqrt{b} + o(1/\sqrt{b})) \cdot l$ as $l \to \infty$.

Conjecture 3. $E[\min D(m,i), 1 \leq i \leq n] = (1 - 1/\sqrt{b} + o(1/\sqrt{b})) \cdot (m - \Theta(\log_b n))$ as $m, n \to \infty, n < b^m$.

Acknowledgment

We would like to thank Gene Lawler for guidance and support. David Aldous, K. Balasubramanian, Maxime Crochemore, Dan Gusfield, Dick Karp, Gad Landau, Dalit Naor, Frank Olken, Kunsoo Park, Esko Ukkonen, and Sun Wu provided helpful comments and encouragement.

Table 1. Summary of Theoretical and Empirical Running Times (based on runs with n = 100,000; m = 100; $k = 10, 20, 30, \ldots, C_b m$; $b = 2, 3, 4, 8, 16, 32, 64$ on a VAX 8600 using the Unix program gprof; empirical running times for random text are formulated as functions of n, m, k, b and given in *microseconds*)

Alg'm	Worst case	Empirical ($k < C_b m$)	Notes	Attribution
mn.dp	$O(mn)$	$3.5mn$		Sellers [31]
kn.uk	$O(mn)$	$4.1\sqrt{b}/(\sqrt{b}-1) \cdot kn$	a	Ukkonen [34]
kn.dt	$O(mn)$	$4.2b/(b-1) \cdot kn$	b	"diagonal transition" [10, 27]
kn.gp	$O(kn)$	$50kn$	c	Galil, Park [10]
kn.lv	$O(kn)$	$40kn$	d	Landau, Vishkin [21]
mn.clp	$O((m-\mu)n)$	$1.4/\sqrt{b} \cdot mn$	e	Chang, Lampe
kn.clp	$O((m-\mu)n)$	$1.4/(\sqrt{b}-1) \cdot kn$	e	Chang, Lampe
let.cl		$80n$	f	Chang, Lawler [6]
set.cl		$160(k \log_b m)(n/m)$	g	Chang, Lawler [6]

Notes.

a. We showed in section 2 that kn.uk is $O(kn)$ on the average.

b. Myers [27] was first to state it is $O(kn)$ on the average (proof is simple).

c. (Unoptimized.) Requires $O(m^2)$ space.

d. (Unoptimized.) This is Landau, Vishkin [21] using McCreight *suffix tree* [24]; Chang, Lawler *matching statistics* [6]; and Schieber, Vishkin *lowest common ancestor* [30] with logarithm replaced by bit magic [2]. The worst case running time of $O(kn)$ is modulo hashing in $O(m)$ space, or deterministic in $O(bm)$ space.

e. Running time depends on d.p. table row averages; μ = average of last row. To guarantee the worst case running time of $O((m-\mu)n)$, $O(bm)$ space is needed. An alternative, $O(m)$ space implementation has the same running time in practice. Expected running time of kn.clp is $O(kn)$ because it is always faster than kn.uk.

f. (Unoptimized.) Linear expected time when error tolerance k is less than the threshold $k^* = m/(\log_b m + c_1) - c_2$ (for suitable constants c_i). 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. Worst case performance is same as dynamic programming based subroutine.

g. (Unoptimized.) Sublinear expected time when $k < k^*/2 - 3$ (in the sense that not all letters of the text are examined). The expected running time is $o(n)$ when k, treated as some fraction of m and not as a constant, is $o(m/\log_b m)$.

References

[1] R. Arratia and M.S. Waterman, Critical Phenomena in Sequence Matching, *The Annals of Probability* 13:4(1985), pp. 1236–1249.

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

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

[4] 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.

[5] 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.

[6] W.I. Chang and W.L. Lawler, Sublinear Expected Time Approximate String Matching and Biological Applications, submitted.

[7] V. Chvátal and D. Sankoff, Longest Common Subsequences of Two Random Sequences, *Technical Report STAN-CS-75-477*, Stanford University, Computer Science Department, 1975.

[8] J. Deken, Some Limit Results for Longest Common Subsequences, *Discrete Mathematics* 26(1979), pp. 17–31. *J. Applied Prob.* 12(1975), pp. 306–315.

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

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

[11] Z. Galil and K. Park, Dynamic Programming with Convexity, Concavity, and Sparsity, manuscript, October 1990.

[12] 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.

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

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

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

[16] N.I. Johnson and S. Kotz, *Distributions in Statistics: Discrete Distributions*, Houghton Mifflin Company (1969).

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

[18] 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.

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

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

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

[22] 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.

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

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

[25] U. Manber and S. Wu, Approximate String Matching with Arbitrary Costs for Text and Hypertext, manuscript, February 1990.

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

[27] E.W. Myers, Incremental Alignment Algorithms and Their Applications, *SIAM J. Comput.*, accepted for publication.

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

[29] D. Sankoff and S. Mainville, Common Subsequences and Monotone Subsequences, in D. Sankoff and J.B. Kruskal, eds., *Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley (1983), pp. 363–365.

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

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

[32] J. Tarhio and E. Ukkonen, Approximate Boyer-Moore String Matching, Report A-1990-3, Dept. of Computer Science, University of Helsinki, March 1990.

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

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

[35] E. Ukkonen, personal communications.

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

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

[38] M.S. Waterman, L. Gordon, and R. Arratia, Phase transitions in sequence matches and nucleic acid structure, *Proc. Natl. Acad. Sci. USA* 84(1987), pp. 1239–1243.