INCREMENTAL STRING COMPARISON

GAD M. LANDAU *, EUGENE W. MYERS †, AND JEANETTE P. SCHMIDT ‡

Abstract. The problem of comparing two sequences A and B to determine their LCS or the edit distance between them has been much studied. In this paper we consider the following incremental version of these problems: given an appropriate encoding of a comparison between A and B, can one incrementally compute the answer for A and bB, and the answer for A and Bb with equal efficiency, where b is an additional symbol? Our main result is a theorem exposing a surprising relationship between the dynamic programming solutions for two such "adjacent" problems. Given a threshold k on the number of differences to be permitted in an alignment, the theorem leads directly to an $O(k)$ algorithm for incrementally computing a new solution from an old one, as contrasts the $O(k^2)$ time required to compute a solution from scratch. We further show with a series of applications that this algorithm is indeed more powerful than its non-incremental counterpart by solving the applications with greater asymptotic efficiency than heretofore possible. For example, we obtain $O(nk)$ algorithms for the longest prefix approximate match problem, the approximate overlap problem, and cyclic string comparison.

Key words. String matching, edit-distance, dynamic programming.

AMS subject classifications. 68P99

* Dept. of Computer Science, Polytechnic University, 6 MetroTech, Brooklyn, NY 11201 (e-mail: landau@pucs2.poly.edu). Partially supported by NSF grant CCR-9305873 and the New York State Science and Technology Foundation Center for Advanced Technology.,

† Dept. of Computer Science, University of Arizona, Tucson, AZ 85721 (e-mail: gene@cs.arizona.edu). Partially supported by NLM grant LM-04960, NSF grant CCR-9002351, and DOE grant DE-FG05-91ER61132., and

‡ Dept. of Computer Science, Polytechnic University, 6 MetroTech, Brooklyn, NY 11201 (e-mail: jps@pucs4.poly.edu). Partially supported by NSF grant CCR-9305873, and the New York State Science and Technology Foundation Center for Advanced Technology.

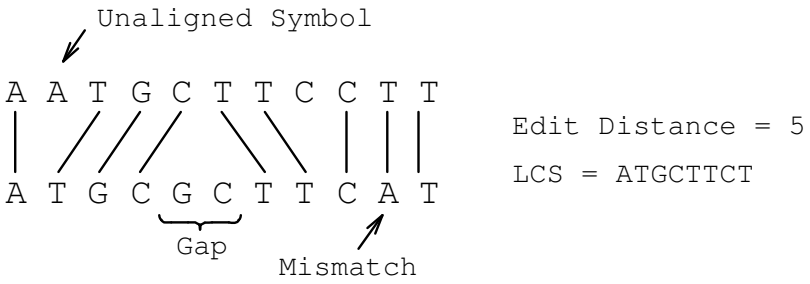

FIG. 1. *An Alignment between two strings.*

1. Introduction. Sequence comparison is an extensively studied topic. Applications are numerous and include file comparison [HS-77], spelling correction [HD-80], information retrieval [WM-92], and searching for similarities among biosequences [NW-70, Se-80, SW-81]. Given string $A = a_1a_2a_3 \ldots a_m$ and $B = b_1b_2b_3 \ldots b_n$, one seeks an *alignment* between the two strings that exposes their similarity. An alignment is any pairing of symbols subject to the restriction that if lines were drawn between paired symbols as in the Figure 1 below, the lines would not cross. Scores are assigned to alignments according to the concept of similarity or difference required by the context of the application, and one seeks alignments of optimal score [WF-74].

While for applications such as comparing protein sequences, the methods of scoring can involve arbitrary scores for symbol pairs and for gaps of unaligned symbols, in many other contexts simple unit cost schemes suffice. Two of these, the *longest common subsequence (LCS)* and the *edit distance* measures, have been studied extensively within computer science, and the unit cost nature of the scoring provides combinatorial leverage not found in the more general framework [Hi-77, HS-77, NKY-82, Uk-85a, My-86a, LV-89, GP-90]. In the edit distance problem, each mismatched aligned pair and unaligned symbol is called a *difference* and scores 1. All pairs of equal aligned characters score 0. One seeks an alignment that minimizes the score or number of differences and this minimal score, $ED(A, B)$, is called the *edit distance* between A and B. Conversely in the LCS problem, matched pairs score 1, mismatches and unaligned symbols score 0, and the goal is to find an alignment of maximum score, $LCS(A, B)$. In this case, the sequence of matched characters is a subsequence common to both sequences and is of maximum length, hence the name longest common subsequence. Figure 1 illustrates these measures.

Although certainly complementary, the LCS and edit distance problems are not formal duals. The LCS problem is equivalent to finding the minimum edit distance, *where mismatches are not allowed*, or equivalently, where a mismatch scores 2, so that it is no better than leaving the two characters unaligned. The reader may wish to also verify that the edit distance problem is equivalent to the following "LCS-like" problem: find an alignment of maximum score where matches score 1, unaligned characters score 0, and *mismatches score* $\frac{1}{2}$.

In this paper we consider the following incremental version of the sequence comparison problem: given a solution for the comparison of A and B, can one incrementally compute a solution for A versus bB, where b is an additional symbol prepended to B? By *solution* we mean some encoding of a relevant portion of the traditional dynamic programming matrix D computed in comparing A and B. D is

an $(m+1) \times (n+1)$ matrix, where entry $D[i,j]$ is the best score for the problem of comparing A^i with B^j, and A^i is the prefix, $a_1 a_2 \ldots a_i$, of A's first i symbols. As will be seen in detail later, the data-dependencies of the fundamental recurrence, used to compute an entry $D[i,j]$, is such that it is easy to extend D to a matrix D' for A versus Bb by computing an additional column. However, efficiently computing a solution for A versus bB given D is much more difficult, in essence requiring one to work against the "grain" of these data-dependencies. The further observation that the matrix for A versus B, and the matrix for A versus bB can differ in $O(mn)$ entries suggests that the relationship between such adjacent problems is non-trivial.

One might immediately suggest that by comparing the reverse of A and B, prepending symbols becomes equivalent to appending symbols, and so the problem, as stated, is trivial. But in this case, we would ask for the delivery of a solution for A versus Bb. The point is that our method allows one to append *and* prepend symbols to A and/or B in any order, efficiently solving one problem from the previous one. More formally, given an initial solution $S^{(0)}$ for $A^{(0)} = A$ versus $B^{(0)} = B$, and a sequence of operations $op^{(t)}$ that either prepend or append a single symbol to $A^{(t-1)}$ or $B^{(t-1)}$ to produce $A^{(t)}$ and $B^{(t)}$, we can deliver the corresponding sequence of solutions $S^{(t)}$ with linear efficiency. Moreover, we can do so on-line, i.e. the sequence of operations need not be known in advance. To keep matters simple, however, we will focus on the core problem of computing a solution for A versus bB, given a "forward" solution for A versus B. A "forward" solution of the problem contains an encoding of a comparison of all (relevant) prefixes of A with all (relevant) prefixes of B. It turns out that the ability to efficiently prepend a symbol to B when given all the information contained in a "forward" solution allows one to solve the applications given in Section 4 with greater asymptotic efficiency than heretofore possible.

As an example of such application, consider the problem of approximate string matching within k differences: find *all* substrings of a text B of length n whose edit distance from a query A is not greater than threshold k. With our $O(k)$ incremental version of the well-known $O(n+k^2)$ greedy algorithm for edit distance, the problem can be solved in $O(nk)$ time, by starting with an empty text and building it up from the right one character at a time. Proceeding formally, let B_l denote the *suffix* of B starting at its $l+1^{st}$ symbol, $b_{l+1} b_{l+2} \ldots b_n$. Begin the search by building the trivial k-thresholded solution for A versus B_n $(= \varepsilon)$. Then incrementally compute the solutions to A versus B_l for $l = n-1, n-2, n-3 \ldots 0$ in $O(k)$ time per step. While the overall time, $O(nk)$, is no better than previous results, [LV-89, GP-90], the algorithm is superior in that for each left index l it reports *all* right indexes r, (and for each right index r it hence reports *all left* indexes l), delimiting a substring $B_l^r = b_{l+1} b_{l+2} \ldots b_r$ that matches A within k differences. For each such match-pair (l,r), the algorithm delivers the number of differences, $ED(A, B_l^r)$, in the match. Previous algorithms either report for each right index r, the matches with the smallest number of differences ending at r, (the "forward" solution), or report the matches with the smallest number of differences starting at l, (the "backward" solution). In our solution at each potential left-index l, the entire k-thresholded solution for A versus B_l is available and can be examined in $O(k)$ time to find any corresponding right-indices and their match score. In addition if A does not match any prefix of B_l with at most k differences, we can report the longest prefix of A that matches a prefix of B_l with k differences. If desired, one can also build an $O(nk)$ table $T[0 \ldots n][-k \ldots k]$ during the search where $T[l][r-(l+m)]$ equals $ED(A, B_l^r)$, if (l,r) delimits a k-match, (i.e.

		A	G	G	A	T	A	T	T	A	
	0	1	2	3	4	5	6	7	8	9	0
A	1	0	1	2	3	4	5	6	7	8	1
T	2	1	1	2	3	3	4	5	6	7	2
G	3	2	1	1	2	3	4	5	6	7	3
G	4	3	2	1	2	3	4	5	6	7	4
T	5	4	3	2	2	2	3	4	5	6	5
A	6	5	4	3	2	3	2	3	4	5	6
T	7	6	5	4	3	2	3	2	3	4	7
A	8	7	6	5	4	3	2	3	3	3	8
	0	1	2	3	4	5	6	7	8	9	

FIG. 2. *A sample dynamic programming matrix.*

a match with at most k differences), and $k+1$ otherwise. The table T is a record of all k-matches found, as r must be in the interval $[l+m-k, l+m+k]$ if (l,r) delimits a k-match. It is impossible for previous string-matching algorithms to be augmented (1) to report, at each position of B, the longest prefix of A matching with k differences, (2) to build T, or, equivalently, (3) to report *all* k-matching substrings and their match distances in $O(nk)$ time. Such a capability is essential, for example, if one is searching for the best match under a scoring criterion that is a complex function of the length and number of differences in the match.

The algorithmic results of this paper hinge on what we find to be the rather surprising fact that there are exploitable relationships between the dynamic programming solutions of adjacent problems computed by several well-known comparison algorithms for LCS and edit distance. Throughout the paper we will focus on formulating incremental versions of the well-known $O(n+k^2)$ greedy-algorithms for finding the edit distance and the LCS between two sequences [My-86a, LV-88]. A similar and somewhat simpler incremental version [My-86b] also holds for the $O(r \log n)$ Hunt-Szymanski algorithm [HS-77]. After a presentation of preliminary concepts and a review of the $O(n+k^2)$ greedy algorithm in Section 2, we present the main theorem exposing the relationship between adjacent solutions and sketch an incremental algorithm based on it in Section 3. Section 4 then presents four applications of incremental algorithms in order to demonstrate the power of such algorithms.

2. Preliminaries.

2.1. The Dynamic Programming Algorithm. Consider the problem of computing the edit distance between strings $A = a_1 a_2 \ldots a_m$ and $B = b_1 b_2 \ldots b_n$ where without loss of generality we assume that $m \leq n$ hereafter. The well-known dynamic programming algorithm [NW-70, WF-74] computes an $(m+1) \times (n+1)$ *edit-distance matrix* $D[0 \ldots m][0 \ldots n]$, where entry $D[i,j]$ is the edit distance, $ED(A^i, B^j)$ between the prefixes A^i and B^j of A and B, where $A^i = A^i_0 = a_1 \ldots a_i$ and $B^j = B^j_0 = b_1 \ldots b_j$ as defined in the introduction. Figure 2 gives an example of the matrix $D[0 \ldots 8, 0 \ldots 9]$ for $A = ATGGTATA$ versus $B = AGGATATTA$. The edit distance between A and B is given in entry $D[8,9]$ which is 3. Conceptually we think of D as an $(m+1)\times(n+1)$ grid of *points* (i,j) to which we assign value $D[i,j]$.

A best alignment between $A^i = a_1 \ldots a_i$ and $B^j = b_1 \ldots b_j$ must either (1) leave

a_i unaligned and optimally align A^{i-1} and B^j, (2) leave b_j unaligned and optimally align A^i and B^{j-1}, or (3) align a_i and b_j and optimally align A^{i-1} and B^{j-1}. This observation leads immediately to the fundamental dynamic programming recurrence:

LEMMA 2.1. *For all* $i + j > 0$

$$D[i,j] = \min \left\{ \begin{array}{ll} D[i-1,j]+1 & \text{if } i > 0 \\ D[i,j-1]+1 & \text{if } j > 0 \\ D[i-1,j-1]+\delta_{a_i,b_j} & \text{if } i,j > 0 \end{array} \right\}$$

where $\delta_{a,b}$ *is 1 or 0 depending on whether* $a = b$ *or not, respectively.*

Coupled with the obvious boundary condition that $D[0,0] = 0$, this recurrence can be used to efficiently compute the $O(mn)$ entries of the matrix D in an order of i and j that observes the data-dependencies of the recurrence, i.e. an order that computes $D[i-1,j]$, $D[i,j-1]$ and $D[i-1,j-1]$ before $D[i,j]$. Traditionally, the lexicographical order of (i,j), (which clearly observes data-dependencies), is used to determine the value of the equation in Lemma 2.1 in $O(1)$ time. An algorithm based on this recurrence thus takes $O(mn)$ time to compute the value of every point of D. The edit distance $ED(A,B)$ is delivered in $D[m,n]$.

The matrix D has a number of useful monotonicity properties with respect to diagonals that are essential to our result.

DEFINITION 2.2. *Diagonal* d *is the list of all points* (i,j) *for which* $j = i + d$. Note that with this definition the lowest, leftmost diagonal is numbered $-m$ and the highest, rightmost diagonal is numbered n. The first essential property is that values are non-decreasing along diagonals and never increase by more than one:

LEMMA 2.3. *[Uk-85a] For all points (i,j):* $D[i,j] - D[i-1,j-1] \in \{0,1\}$. The second essential property is that adjacent values in adjacent diagonals never differ by more than one:

LEMMA 2.4. *[Uk-85b] For all points (i,j):* $D[i,j] - D[i-1,j], D[i,j] - D[i,j-1] \in \{-1,0,1\}$.

2.2. The Greedy Algorithm. In the mid-80s [NKY-82, Uk-85a] came upon the idea of computing the points of the matrix D in an order dictated by a greedy approach, (i.e. to compute the values in non-decreasing order), instead of the lexicographic order of (i,j). In this way entries whose values are 0 are computed first, then those whose values are 1, and then 2, and so on until either (i) some threshold k is reached, or (ii) the value $D[m,n]$ is determined.

By Lemma 2.1, $D[0,j] = j$ for $j = 0,\ldots,n$, and $D[i,0] = i$ for $i = 0,\ldots,m$. By Lemma 2.3, the values along a diagonal are non-decreasing, so that $D[0,d]$ (resp. $D[d,0]$) are the smallest values on diagonal d, (resp. $-d$). We conclude that all values not greater than k are on diagonals $-k, -k+1 \ldots, k$ in D, and there are therefore at most $(2k+1)m$ entries with such values. This idea gives rise to an $O(km)$ algorithm to determine the k-thresholded edit distance of A and B.

The algorithm can be realized by a modified version of Dijkstra's shortest path algorithm on a directed, weighted *edit graph* $G = (V,E)$. V consists of the set of all

points (i,j) of D. For each point (i,j) there is an edge into it from $(i-1,j)$ weighted 1, another from $(i,j-1)$ also weighted 1, and a third edge from $(i-1,j-1)$ weighted δ_{a_i,b_j}. For points along the left and upper boundaries of D, there is an edge from a predecessor of (i,j) only if the predecessor exists. Observe that the edge weights are chosen in exact correspondence with Lemma 2.1, and that $D[i,j]$ is the length of the shortest path from $(0,0)$ to (i,j) in this edit graph. The algorithm stops when either (i) the distance to (m,n) is determined, or (ii) the first node with distance greater than k from $(0,0)$ is reached. Since the lengths of all paths are integers in $[0 \ldots k]$, and the outdegree of each node in the graph is at most 3, a standard modification of Dijkstra's algorithm will run in $O(km)$ time and space.

Ukkonen [Uk-85a] noticed that it suffices to determine, for all h, the last h on each diagonal d of the matrix D, as this determines all other values in D by Lemma 2.3. More precisely, let $L^h(d)$ denote the largest row index of a point on diagonal d that has value h:

DEFINITION 2.5. $L^h(d) = \max\{\, i \;:\; D[i,i+d] = h \,\}$. Figure 3 illustrates this definition by labeling just these *furthest h-points* with their values. Note that the row number, $i = L^h(d)$, of the furthest h-point on diagonal d identifies the point itself as $(i, i+d)$. So henceforward, we will liberally use $L^h(d)$ to denote either the point $(i, i+d)$ or the row i, which interpretation to make being obvious from context.

As observed earlier, there are no h-points outside diagonals $[-h \ldots h]$. Thus it suffices to compute for each value h, the *h-wave*:

DEFINITION 2.6. $L^h = < L^h(-h), L^h(-h+1), \ldots, L^h(0), \ldots L^h(h-1), L^h(h) >$. That is, L^h is the orderered list consisting of the $2h+1$ furthest h-points in diagonals $-h$ up to h.

The algorithm then computes wave L^h, for $h = 0, 1, 2, \cdots$, until either (i) a wave e is computed for which $L^e(n-m) = m$, or (ii) wave L^k is computed in the case that the algorithm is thresholded by k. In the event termination is by condition (i) it follows that $ED(A,B) = D[m,n] = e$.

Note that the highest values on some of the diagonals may be less than h. In Figure 3 for example, the highest value in diagonal -2 is 2 (in $D[8,6]$), and $L^3(-2)$ therefore does not exist, as there is no 3-point in diagonal -2. We handle this overflow by adding a "dummy" point $L^h(d)$ with value ∞ to wave h, whenever no real point $L^h(d)$ exists, so that L^h always has $2h+1$ points. In addition, when the extreme point on diagonal d has value h, it is also convenient to set $L^h(d)$ to ∞ whenever (1) this extreme point is $(m, m+d)$, $D[m, m+d] = h$ and $D[m, m+d+1] = h-1$, or when (2) this extreme point is $(n-d, n)$, $D[n-d, n] = h$ and $D[n-d+1, n] = h-1$. Intuitively, this is justified because if we were to extend the matrix D with one more row and column then $D[m+1, m+d+1]$ (resp. $D[n-d+1, n+1]$) would always still have value h, regardless of how we interpret the "dummy" character associated with that row (resp. column). This is shown in Figure 3, where $L^3(-3)$ is shown outside the boundary of the matrix and hence assigned ∞, although the preceding point on that diagonal, $D[8,5]$, has also value 3.

Given wave L^{h-1}, wave L^h is computed from it by induction. Consider $i = L^{h-1}(d)$, the furthest $(h-1)$-point in diagonal d, and a corresponding optimal align-

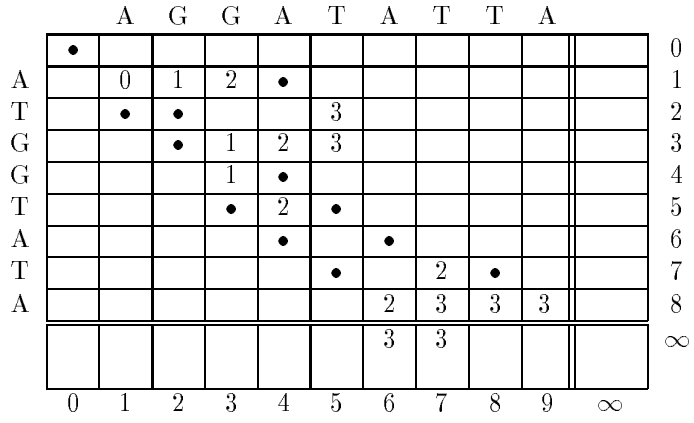

FIG. 3. *The h-waves of the matrix of Figure 2.*

ment between A^i and B^j where $j = i+d$. The alignment involves $h-1$ differences and $a_{i+1} \neq b_{j+1}$, as otherwise $(i+1, j+1)$ would also be an $(h-1)$-point (a contradiction). Observe that in this case this alignment can be maximally extended with one additional difference in the following three ways: (1) leave a_{i+1} unaligned and then align a_{i+1+q} with b_{j+q} for $q > 0$ as long as the symbols are equal, (2) leave b_{j+1} unaligned and then align a_{i+q} with b_{j+1+q} for $q > 0$ as long as the symbols are equal, and (3) mismatch a_{i+1} with b_{j+1} and then align a_{i+1+q} with b_{j+1+q} for $q > 0$ as long as the symbols are equal. In each case, we visualize the alignment of equal symbols as "sliding down" the relevant diagonal: $d-1$ in case (1), $d+1$ in case (2), and d in case 3.

We capture such a substring of equal characters, resulting in a slide down a diagonal, with the definition:

DEFINITION 2.7. $Slide_d(i) = \max\{\ q\ :\ A_i^q = B_{i+d}^{q+d}\ \}$. That is, $Slide_d(i)$ corresponds to a slide in diagonal d starting on row i. In order to correctly handle the cases where $L^h(d)$ is or becomes ∞, we define $Slide_d(i) = \infty$, when $i > m$ or $i + d > n$. Note that by definition $Slide_d(i) = i$ when $i = m$ or $i + d = n$.

The gist of earlier papers was a proof that the furthest point reached in diagonal d over the relevant extensions from the points $L^{h-1}(d-1)$, $L^{h-1}(d)$ and $L^{h-1}(d+1)$ on wave $h-1$, is the point $L^h(d)$ [1]. Combining this with the identity, $\max\{Slide_d(a), Slide_d(b)\} = Slide_d(\max\{a, b\})$, leads to the following recurrence for a furthest h-point in terms of furthest $(h-1)$-points.

LEMMA 2.8.

$$\textit{For all } h > 0,\ L^h(d) = Slide_d\left(\max\left\{\begin{array}{ll} L^{h-1}(d+1)+1 & \textit{if } d < h-1 \\ L^{h-1}(d)+1 & \textit{if } -h < d < h \\ L^{h-1}(d-1) & \textit{if } d > -h+1 \end{array}\right\}\right).$$

Using Lemma 2.8 one can immediately compute the $2h+1$ points of wave L^h, given

[1] This is not immediately obvious as indicated by the fact that this property is *not* true for more general, weighted-cost comparison models

the $2h-1$ points of wave L^{h-1}. The induction of the algorithm is started by observing that wave L^0 is the single point $L^0(0) = Slide_0(0)$.

In Figure 3 the furthest points of waves 0 through 3 are indicated by the placement of their value at their location, and the solid circles annotate points that the function *Slide* extends through, to reach a furthest point. Note that many of the points in the matrix D of Figure 2, that have value 3 or less, are not marked in Figure 3.

The time complexity of the greedy algorithm depends on the efficiency with which the function *Slide* is realized. When *Slide* is computed by a brute-force comparison of the relevant characters, computing $s = Slide_d(i)$ takes $O(s-i)$ time, resulting in $O(km)$ total time. However, Myers [My-86a] has shown that when one of the strings, say A, is a random string[2] then the algorithm takes $O(m + k^2)$ *expected time*. This result is true even if B is chosen so as to maximize the time spent by the Slide function.

2.3. An $O(n + k^2)$ algorithm. The worst case time of the previous algorithm is improved to $O(n + k^2)$ by computing *Slide* in constant time [My-86a, LV-88].

In a preprocessing step one computes a suffix tree [Wn-73, Mc-76] of the string $AxBy = a_1a_2 \ldots a_m x b_1 b_2 \ldots b_n y$ where $x \neq y$ are two symbols not in the alphabets of A and B. One further preprocesses this suffix tree using any of the algorithms [HT-84, SV-88, BV-93], to allow any LCA (least common ancestor) query over the tree to be answered in $O(1)$ time. This preprocessing takes $O(n)$ time[3].

For given indices i, j the *Slide* function must return the largest q for which $a_i \ldots a_{i+q} = b_j \ldots b_{j+q}$. The key observation is that this q can be retrieved from the suffix tree described above with an LCA query in $O(1)$ time. Specifically, it has been shown that $Slide_d(i) = depth(LCA(leaf(i), leaf(m + 1 + i + d))$ where $leaf(t)$ is the leaf in the suffix tree for suffix $(AxBy)_t$, $LCA(u, v)$ is the LCA of vertices u and v, and $depth(v)$ is the length of the string that labels the path from the root of the suffix tree to vertex v.

As noted earlier, wave L^h has $2h + 1$ points, each of which can now be computed in $O(1)$ time. Thus it takes $O(k^2)$ worst-case time to compute the $(k + 1)^2$ points in waves L^0 through L^k after the initial $O(n)$ preprocessing. This results in an $O(n+k^2)$ worst-case time algorithm.

3. The Central Theorem and Basic Algorithm. Given two strings $A = a_1a_2 \ldots a_m$ and $B = b_1b_2 \ldots b_n$, we now show how to compute the $k + 1$ waves $L^0_{new}, \ldots, L^k_{new}$ of the edit-distance matrix $D_{new}[i, j]$ of A and bB, when given the $k + 1$ waves $L^0_{old}, \ldots, L^k_{old}$ of the edit-distance matrix $D_{old}[i, j]$ of A versus B.

D_{new}, when viewed as a leftward extension of D_{old}, is an $(m+1) \times (n+2)$ matrix $D_{new}[0 \ldots m][-1 \ldots n]$, with main diagonal labeled -1. Labeling the columns in D_{new} from -1 to n, (as opposed to $0 \ldots n + 1$), has the advantage that entry $D_{new}[i, j]$ corresponds to the edit distance, $ED(A^i, bB^j)$ between the prefixes A^i and bB^j of A and bB, establishing the correspondence of the points $D_{new}[i, j]$ and $D_{old}[i, j]$. Furthermore, for any index pair (i, j) that is valid in both D_{old} and D_{new}, $Slide_{j-i}(i)$

[2] Specifically, the result of Bernoulli trials over a finite distribution.
[3] $O(n \log \Sigma)$ time if the alphabet size, Σ, is not considered fixed.

is the same in both matrices.

Since the waves for D_{new} are computed after those for D_{old} have been computed, we shall refer to L^h_{old} and D_{old} as the old h-wave and old matrix and to L^h_{new} and D_{new} as the new h-wave and matrix. We show that L^h_{new} is composed of a concatenation of a prefix of L^{h+1}_{old}, a sublist of L^h_{old}, a suffix of L^{h-1}_{old}, and at most two points p_1 and p_2, separating the sublists of L_{old}, not included in any of the old waves L^{h-1}_{old}, L^h_{old}, L^{h+1}_{old}. Furthermore, the two points p_1 and p_2 can be computed in $O(1)$ time and the entire list L^h_{new} can be pasted together from the lists L^{h+1}_{old}, L^h_{old}, L^{h-1}_{old} in $O(1)$ time.

The following five Observations define the concepts and terminology needed to formulate and prove our central theorem.

The first Observation relates the values in the matrices D_{old} and D_{new}. We note that any alignment between A^i and B^j with k differences can easily be used to obtain an alignment between A^i and bB^j with at most $k+1$ differences, and any alignment between A^i and bB^j with k differences can easily be used to obtain an alignment between A^i and B^j with at most $k+1$ differences. This implies the following.

Observation 1:

$$\forall (i,j) \in [0 \ldots m, 0 \ldots n], \;\; D_{old}[i,j] - 1 \leq D_{new}[i,j] \leq D_{old}[i,j] + 1$$
$$\forall i \in [0 \ldots m], \;\; D_{new}[i,-1] = i.$$

D_{old} is not defined on $(i,-1)$. The point following $(i,-1)$ on diagonal $-i-1$ is $(i+1,0)$ and $D_{old}[i+1,0] = i+1$.

The second observation is an immediate consequence of the recurrence for computing $L^h(d)$ from $L^{h-1}(d-1)$, $L^{h-1}(d)$, and $L^{h-1}(d+1)$ given in Lemma 2.8. Recall that we identify the points on a given diagonal by their row number. Hence an inequality $p > q$ (even if p and q are not on the same diagonal), is always interpreted as "the row number of p is higher than the row number of q".

Observation 2: If the three points of wave L^{g-1}_{old} on diagonals $d-1$, d and $d+1$ are all less than or equal to (resp. greater than or equal to) the three points on those diagonals for L^{h-1}_{new}, then $L^g_{old}(d) \leq L^h_{new}(d)$ (resp. $L^g_{old}(d) \geq L^h_{new}(d)$). If the three points on the two waves are all equal, then clearly $L^g_{old}(d) = L^h_{new}(d)$. Furthermore, if $\max\{L^{g-1}_{old}(d+1)+1, L^{g-1}_{old}(d)+1, L^{g-1}_{old}(d-1)\}$ is less than or equal to $\max\{L^{h-1}_{new}(d+1)+1, L^{h-1}_{new}(d)+1, L^{h-1}_{new}(d-1)\}$, then $L^g_{old}(d) \leq L^h_{new}(d)$.

In the sequel it will be important to distinguish how a given point $L^h(d)$ got its value in the equation of Lemma 2.8. We will say that:
$L^h(d)$ was *obtained from above* iff $L^h(d) = Slide_d(L^{h-1}(d+1)+1)$.
$L^h(d)$ was *obtained diagonally* iff $L^h(d) = Slide_d(L^{h-1}(d)+1)$.
$L^h(d)$ was *obtained from the left* iff $L^h(d) = Slide_d(L^{h-1}(d-1))$.
Note that a point can be obtained in more than one way.

Our third observation compares the scope of the new h wave with the scope of the old $h-1$, h, and $h+1$ wave.

Observation 3: Wave $L^h_{new} = < L^h_{new}(-h-1), \ldots, L^h_{new}(h-1) >$, and so it has

a point on each of diagonals $-h-1$ through $h-1$ and only these diagonals. Similarly, observe that L^g_{old} has a point on each and only the diagonals $-g$ through g. Thus the leftmost diagonal of L^h_{new}, $-h-1$, is also the leftmost diagonal of L^{h+1}_{old} and both L^h_{old} and L^{h-1}_{old} do not have a point on this diagonal or to the left of this diagonal. On the other hand, the rightmost diagonal of L^h_{new}, $h-1$, is also the rightmost diagonal of L^{h-1}_{old} and both L^h_{old} and L^{h+1}_{old} do have a point on this diagonal as well as to the right of this diagonal.

Crucial to our central result is the idea of the *key value* of a point, $p = L^h_{new}(d)$, on a new wave, which describes p's position relative to points in the old waves. Informally, if a point $p = L^h_{new}(d)$ of a new wave coincides with a point $L^g_{old}(d)$ of an old wave, then the key value of p is the wave number, g, of the old wave. The other possibility is that p is an *in-between point* that does not coincide with any old point and in this case its key value is $g - \frac{1}{2}$, where g is the smallest wave number for which $p < L^g_{old}(d)$. Note that if wave L^{g-1}_{old} has a point on diagonal d, then clearly p lies on diagonal d between the points of the old $g-1$ and g waves, hence the term *in-between* point.

DEFINITION 3.1. *Formally, for a point p on diagonal d:*

$$key(p) = \min\ \{\ g\ :\ g \in \{0, \frac{1}{2}, 1, 1\frac{1}{2}, \ldots\}\ \ and\ \ \big(p = L^{\lfloor g \rfloor}_{old}(d)\ \ or\ \ p < L^{\lfloor g+\frac{1}{2} \rfloor}_{old}(d)\big)\}.$$

In terms of key values, Observation 1 yields the following fact:

Observation 4: $\forall\ h, d, \quad h-1 \le key(L^h_{new}(d)) \le h+1.$

and in terms of key values, Observations 2 and 3 yield the following:

Observation 5: If $key(L^h_{new}(d-1))$, $key(L^h_{new}(d))$, and $key(L^h_{new}(d+1))$ are all $\le$, $=$, or $\ge g$, (for $g \in \{h-1, h, h+1\}$), then $key(L^{h+1}_{new}(d))$ is $\le$, $=$, or $\ge g+1$, respectively. However, $L^{h+1}_{new}(d)$ may exist although some of the above h-wave points do not exist, i.e. when d is in $\{-h-2, -h-1, h-1, h\}$. If those that do exist all have key values $\le g$, then it still follows that $key(L^{h+1}_{new}(d)) \le g+1$. Greater care has to be taken for the $=$ and $\ge$ case. If for all diagonals $\delta \in \{d-1, d, d+1\}$, for which $L^h_{new}(\delta)$ does not exist, $L^g_{old}(\delta)$ does not exist either, and all the other relevant h-wave points have key values $=$ or $\ge g$, then $key(L^{h+1}_{new}(d))$ will still be $=$, or $\ge g+1$, respectively.

We now have the concepts and terminology needed to proceed with our central theorem.

THEOREM 3.2. *L^h_{new} is the concatenation of (up to) five pieces: (i) a prefix of L^{h+1}_{old}, (ii) an in-between point p_1, with $key(p_1) = h + \frac{1}{2}$, (iii) a sublist of L^h_{old}, (iv) an in-between point p_2, with $key(p_2) = h - \frac{1}{2}$, and (v) a suffix of L^{h-1}_{old}. Each individual piece may be empty.*

Proof. The proof of Theorem 3.2 is essentially by induction on h, but we will first show that certain monotonicity properties imply Theorem 3.2 and then prove these properties inductively.

We will prove that the following monotonicity property on key values holds for all h waves. The key values of the points along L^h_{new} are non-increasing and strictly

decreasing around in-between points, as one proceeds from left to right, (i.e. from diagonal $-h-1$ to $h-1$). Formally,

1st Key Property: For $d \in [-h, h-1]$, $\lfloor key(L^h_{new}(d-1)) \rfloor \geq key(L^h_{new}(d))$.

Observation 4 says that all key values on L^h_{new} are between $h-1$ and $h+1$. The *first key property* says that key values are strictly decreasing along in-between points and are otherwise non-increasing. This implies that there is at most one point with key value $h+\frac{1}{2}$, (and it is to the right of any points with key value $h+1$, and to the left of any points with key value h), and at most one point with key value $h-\frac{1}{2}$, (to the right any points with key value h, and to the left of any points with key value $h-1$). It follows that the *first key property* implies that L^h_{new} is the concatenation of the (up to) five pieces given in Theorem 3.2.

We shall prove by induction on h that the *first key property* holds for all h waves.

Before proceeding with a formal proof, we provide some intuition on why the Theorem holds and at the same time point to some of the difficulties in proving it. Suppose that L^h_{new} is indeed a concatenation of a prefix $< L^{h+1}_{old}(-h-1)), \ldots, L^{h+1}_{old}(r) >$ of L^{h+1}_{old}, a point p_1 on diagonal $r+1$, a sublist $< L^h_{old}(r+2), \ldots, L^h_{old}(s) >$ of L^h_{old}, a point p_2 on diagonal $s+1$ and a suffix $< L^{h-1}_{old}(s+2), \ldots, L^h_{old}(h-1) >$ of L^{h-1}_{old}. By Observation 2, clearly L^{h+1}_{new} will be equal to L^{h+2}_{old} on diagonals $-h-2 \ldots r-1$, will be equal to L^{h+1}_{old} on diagonals $r+3 \ldots s-1$, and will be equal to L^h_{old} on diagonals $s+3 \ldots h-1$. It is hence easy to see that the Theorem can be proven by induction for "most points". One of the difficulties lies in proving that at most one of the diagonals in $\{r, r+1, r+2\}$ and at most one in $\{s, s+1, s+2\}$ can contain an in-between point. A further (more serious) difficulty comes from the fact that any individual piece (of the five pieces composing L^h_{new}) can be empty, resulting in the necessity to examine an enormous number of cases. In addition, points on extreme diagonals behave differently than points on the inner diagonals. In particular since the various scopes of the waves L^h_{new}, L^{h-1}_{new}, L^{h-1}_{old} L^h_{old} and L^{h+1}_{old} are different, special care has to be taken to cover all cases, where a diagonal is in the scope of one wave, but outside the scope of another, (see Observations 3 and 5).

Therefore, instead of proving Theorem 3.2 directly, we choose to prove that the *first key property* holds for all waves. This will allow us to reduce the number cases we need to examine significantly, although a large number still remain.

In order to prove the *first key property*, it is very helpful to have established the following second key property for wave h, which is implied by the first key property on wave $h-1$, as shown below in Lemma 3.3.

2nd Key Property: If $key(L^h_{new}(d)) = g+\frac{1}{2}$, then $L^{g+1}_{old}(d)$ was obtained from above.

In other words, the second key property asserts that if there is a new point p in diagonal d between the g- and $g+1$-points of the old waves, then the $g+1$-point must have been obtained from the g-point on diagonal $d+1$.

See Figure 4(a) for an illustration of the situation.

The *second key property* is quite intuitive, but its proof is not immediate. Key

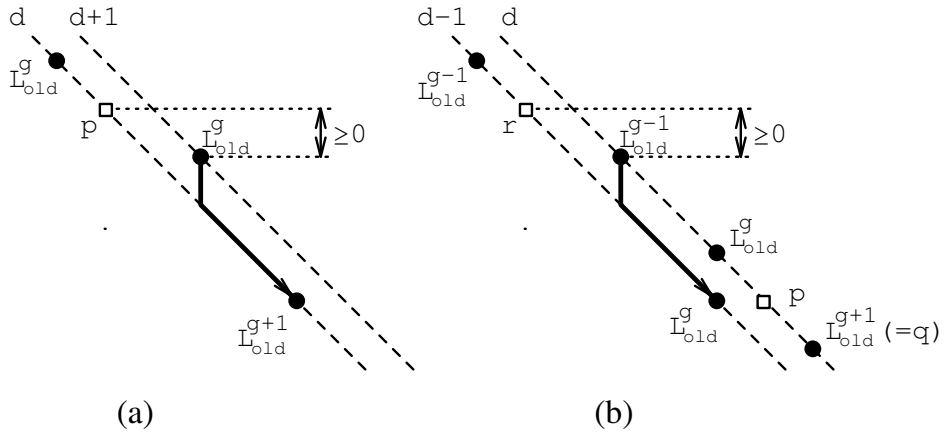

FIG. 4. *The second key property. Old points are shown as solid circles and new points as open squares. (a) If the new point p has key value $g + \frac{1}{2}$, then $L^{g+1}_{old}(d)$ was obtained from above, i.e. from $L^{g}_{old}(d+1)$. (b) Shown are new points p and r on diagonals d and $d-1$ with key values $g - \frac{1}{2}$ and $g + \frac{1}{2}$, respectively. The second key property implies that p could not have been obtained from r.*

values (on wave $h-1$) are non-increasing, hence if the old $g+1$ value was not reached on diagonal d in the new h wave, it would seem likely that this happened because the key value on diagonal $d+1$ of L^{h-1}_{new} "dropped below" g, and the old $g+1$ point was obtained from this "missing" old g point, $L^{g}_{old}(d+1)$.

The remainder of the proof of Theorem 3.2 is complex enough that we capture it in three lemmas below. First we prove in Lemma 3.3 that if the first key property holds on new wave $h-1$, then the second holds on new wave h. Then a useful corollary (Lemma 3.4) of Lemma 3.3 is given. Finally, with the aid of the second key property and its corollary, we complete the proof of Theorem 3.2, by inductively showing the first key property to hold for all waves in Lemma 3.5.

LEMMA 3.3. *If the first key property holds up to wave $h-1$ of D_{new}, then the second key property holds up to wave h of D_{new}.*

Proof. The Lemma is proven by induction on h.

BASIS: Wave L^{0}_{new} has only one point, which is on diagonal -1. By Observation 4 $key(L^{0}_{new}(-1)) \leq 1$, and since L^{0}_{old} does not have a point on diagonal -1, we also know that $key(L^{0}_{new}(-1)) > 0$. Hence $key(L^{0}_{new}(-1))$ is either 1 or $\frac{1}{2}$. $L^{1}_{old}(-1)$, (whether it collides with the new zero point or not), was obtained from the (only) old zero point $L^{0}_{old}(0)$, i.e. from above. It follows that the *second key property* holds for wave 0 of D_{new}.

INDUCTION: Assume that our claim holds for all new waves up to wave $h-1$, i.e. given an *in-between point* $L^{h-1}_{new}(d)$ with $key(L^{h-1}_{new}(d)) = g - \frac{1}{2}$, $L^{g}_{old}(d)$ was obtained from $L^{g-1}_{old}(d+1)$. As easily seen in Figure 4(a), with $p = L^{h-1}_{new}(d)$ and g replaced by $g-1$, the inductive assumption implies that:

(1) If $key(L^{h-1}_{new}(d)) = g - \frac{1}{2}$, then $L^{g-1}_{old}(d+1) \geq L^{h-1}_{new}(d)$.

Henceforward, consider an in-between point $p = L^{h}_{new}(d)$, for which

$key(L^h_{new}(d)) = g+\frac{1}{2}$. That is, $L^g_{old}(d)$(if it exists) $< p < L^{g+1}_{old}(d)$ and Observation 4 further implies that $g \in \{h-1, h\}$.

To prove the *second key property* it suffices to show that $q = L^{g+1}_{old}(d)$ could not have been obtained diagonally or from the left. If $L^g_{old}(d)$ exists, then $Slide_d(L^g_{old}(d)+1)$ is at most p as $L^g_{old}(d) < p$ and p is a furthest point. Thus since $p < q$, q could not have been obtained diagonally.

It remains to show that q could not have been obtained from the left. Again we only need to examine the case when $L^g_{old}(d-1)$ exists. (If it did not exist, then clearly q could not have been obtained from the left). This implies by Observation 3 that $d-1 \geq -g$, and since $g \leq h$, $d-1$ is at least $-h$ and $L^{h-1}_{new}(d-1)$ exists also.

The assumption that $key(p) = g+\frac{1}{2}$, $(p = L^h_{new}(d))$, implies by Observation 5 that one of the three points $L^{h-1}_{new}(d-1)$, $L^{h-1}_{new}(d)$, $L^{h-1}_{new}(d+1)$ must have a key value greater than $g-1$ and since $key(L^{h-1}_{new}(d-1)) \geq key(L^{h-1}_{new}(d))$ (if it exists) $\geq key(L^{h-1}_{new}(d+1))$ (if it exists), it must always be that the key value of $r = L^{h-1}_{new}(d-1)$ is greater than $g-1$. Hence $key(r) = g-\frac{1}{2}$ or $key(r) \geq g$.

$key(r) \geq g$ means that $r = L^{h-1}_{new}(d-1) \geq L^g_{old}(d-1)$, hence if $q = L^{g+1}_{old}(d)$ were obtained from $L^g_{old}(d-1)$ (i.e. the left), then $L^{h-1}_{new}(d-1)$ would produce a point greater than or equal to q, but $L^h_{new}(d)$, the point reached on diagonal d, is strictly less than q, a contradiction.

We now show that assuming that $key(r) = g-\frac{1}{2}$ leads to a contradiction, because none of the three points $L^{h-1}_{new}(d-1)$, $L^{h-1}_{new}(d)$ or $L^{h-1}_{new}(d+1)$ could have produced $L^h_{new}(d)$. Figure 4(b) will help in understanding the argument that follows. Since $r = L^{h-1}_{new}(d-1)$ is an in-between point in this case, it follows from (1) that $L^{g-1}_{old}(d) \geq r$, hence r can not produce $p = L^h_{new}(d)$, a point with key value (much) higher than $g-1$. On the other hand $key(L^{h-1}_{new}(d-1)) = g-\frac{1}{2}$ implies that the key values of both $L^{h-1}_{new}(d)$ and of $L^{h-1}_{new}(d+1)$ are at most $g-1$ (if they exist), hence they could not have produced $L^h_{new}(d)$ with key value $g+\frac{1}{2}$ either.

We have shown that for all legal key values of $L^{h-1}_{new}(d-1)$, q could not have been obtained from the left, and have shown earlier that q could not have been obtained diagonally. It must be that q was obtained from the right. □

The following lemma is an immediate corollary of Lemma 3.3.

LEMMA 3.4. *If the first key property holds up to wave $h-1$, then the following holds. If $key(L^{h-1}_{new}(d)) = g-\frac{1}{2}$, then $key(L^h_{new}(d+1)) \leq g$. Moreover, if $L^h_{new}(d+1)$ was obtained from $L^{h-1}_{new}(d)$, then $key(L^h_{new}(d+1)) \leq g-1$.*

Proof. Since $L^{h-1}_{new}(d)$ is an in-between point, $L^g_{old}(d)$ was obtained from $L^{g-1}_{old}(d+1)$. As observed earlier and seen in Figure 4(a), it must be the case that $L^{g-1}_{old}(d+1) \geq L^{h-1}_{new}(d)$. Hence $Slide_{d+1}(L^{h-1}_{new}(d))$ cannot be greater than $L^{g-1}_{old}(d+1)$ (a furthest point), which proves the second part of the lemma. Moreover, if the *first key property* holds for wave $h-1$, the key values of both $L^{h-1}_{new}(d+1)$ and $L^{h-1}_{new}(d+2)$ are at most $g-1$, and hence $key(L^h_{new}(d+1)) \leq g$ by Observation 2. □

	$key(L^{h-1}_{new}(d+1)) = g$	$key(L^{h-1}_{new}(d+1)) \leq g - \frac{1}{2}$ (or doesn't exist)
$key(L^{h-1}_{new}(d)) = g$	$key(L^{h}_{new}(d)) \geq g+1 \geq key(L^{h}_{new}(d+1))$	$key(L^{h}_{new}(d)) > g \geq key(L^{h}_{new}(d+1))$ or $key(L^{h}_{new}(d)) \geq g+1 \geq key(L^{h}_{new}(d+1)) > g$
$key(L^{h-1}_{new}(d)) = g + \frac{1}{2}$	$key(L^{h}_{new}(d)) \geq g+1 \geq key(L^{h}_{new}(d+1))$	$key(L^{h}_{new}(d)) \geq g + \frac{1}{2} > key(L^{h}_{new}(d+1))$

TABLE 1

The key values of the pair $L^h_{new}(d), L^h_{new}(d+1)$, as implied by the key values of the pair $L^{h-1}_{new}(d), L^{h-1}_{new}(d+1)$.

We conclude the proof of Theorem 3.2 by proving by induction that the *first key property* holds for all waves in D_{new}.

LEMMA 3.5. *The first key property holds for all waves in D_{new}.*

Proof. The proof is by induction on h.

BASIS: Wave 0 in D_{new} has only one point $L^0_{new}(-1)$. The first key property therefore trivially holds for L^0_{new}.

INDUCTION: Assume now that the *first key property* holds up to wave $h-1$ and the *second key property* holds up to wave h. We shall prove that the *first key property* also holds for wave h. To achieve this it suffices to examine all adjacent pairs of diagonals in wave h and to prove the required inequality on their key values. Henceforward consider the pair of points $L^h_{new}(d)$ and $L^h_{new}(d+1)$. The proof is divided into two cases depending on whether $L^h_{new}(d)$ is the leftmost point of the wave.

Case 1: *Diagonal d is not the leftmost (lowest) diagonal on L^h_{new}, i.e. $d \geq -h$.*

By Observation 3, L^{h-1}_{new} also has a point on diagonal d. On the other hand, we do not assume that $d < h-1$ and hence the point $L^{h-1}_{new}(d+1)$ may not exist. Let $g = \lfloor key(L^{h-1}_{new}(d)) \rfloor$ and note that by Observation 4, $g \in \{h-2, h-1, h\}$, and by definition, $key(L^{h-1}_{new}(d)) \in \{g, g+\frac{1}{2}\}$. We now further divide the proof of this case into four subcases depending on the key values of $L^{h-1}_{new}(d)$ and $L^{h-1}_{new}(d+1)$ in relation to g. Table 1 illustrates the four cases factored into two conditions on the key value of $L^{h-1}_{new}(d)$ and two conditions on the key value of $L^{h-1}_{new}(d+1)$, and for each case gives the implication on the key values of $L^h_{new}(d)$ and $L^h_{new}(d+1)$ that will be proven below. Note in each of the four cases we prove that $\lfloor key(L^h_{new}(d)) \rfloor \geq key(L^h_{new}(d+1))$.

Subcase 1.a: $key(L^{h-1}_{new}(d)) = g$ and $key(L^{h-1}_{new}(d+1)) = g$.

It suffices to show that $key(L^h_{new}(d)) \geq g+1 \geq key(L^h_{new}(d+1))$. Either $key(L^{h-1}_{new}(d+2))$ does not exist, (i.e. $d+1$ is the rightmost diagonal of L^{h-1}_{new}), or by the *first key property* on wave $h-1$ $key(L^{h-1}_{new}(d+2)) \leq g$. In either case, Observation 5 implies $key(L^h_{new}(d+1) \leq g+1$. We now show that

$key(L^h_{new}(d)) \geq g+1$. Either $key(L^{h-1}_{new}(d-1)) \geq g$, (by the *first key property* on wave $h-1$), or $key(L^{h-1}_{new}(d-1))$ does not exist because $d = -h$ is the leftmost diagonal of wave L^{h-1}_{new}, in which case by Observation 3, L^{h-2}_{old}, L^{h-1}_{old} and L^h_{old} do not have points on diagonal $d-1$ either, and d is also the leftmost diagonal on L^g_{old}. In either case, Observation 5 implies $key(L^h_{new}(d)) \geq g+1$.

Subcase 1.b: $key(L^{h-1}_{new}(d)) = g$ and $key(L^{h-1}_{new}(d+1)) \leq g - \frac{1}{2}$ (or doesn't exist).

If $key(L^h_{new}(d+1)) \leq g$ we are immediately done, as $L^h_{new}(d)$ is always greater than $L^{h-1}_{new}(d)$, and since $key(L^{h-1}_{new}(d)) = g$, we have $key(L^h_{new}(d)) > g$. So we assume $key(L^h_{new}(d+1)) \geq g + \frac{1}{2}$ and show that $key(L^h_{new}(d)) \geq g + 1 \geq key(L^h_{new}(d+1))$. $L^h_{new}(d+1)$ got its value from one of the points in $\{L^{h-1}_{new}(d), L^{h-1}_{new}(d+1), L^{h-1}_{new}(d+2)\}$. Since all three points have key values of g or less, clearly $key(L^h_{new}(d+1)) \leq g+1$. In addition, since $key(L^{h-1}_{new}(d+1))$ is either $g - \frac{1}{2}$ or $\leq g-1$, $key(L^{h-1}_{new}(d+2))$ (if it exists) $\leq g-1$, by the first key property, and can not produce $L^h_{new}(d+1)$ a point with key value greater than g. $L^{h-1}_{new}(d+1)$ (if it exists) $< L^g_{old}(d+1)$, (by definition of subcase 1.b) and hence $L^{h-1}_{new}(d+1)$ cannot produce a point beyond $L^g_{old}(d+1)$ either. It follows that $L^h_{new}(d+1)$ got its value from the left, (i.e. from $L^{h-1}_{new}(d) = L^g_{old}(d)$). Since $key(L^h_{new}(d+1))$ is also $\geq g + \frac{1}{2}$, it must be the case that

$$L^{h-1}_{new}(d) > L^g_{old}(d+1); \text{ and by subcase 1.b } L^g_{old}(d+1) > L^{h-1}_{new}(d+1).$$

It follows that $L^h_{new}(d)$ was not obtained from above. If $L^{h-1}_{new}(d-1)$ exists, then $key(L^{h-1}_{new}(d-1)) \geq g$, (by the first key property), otherwise $d = -h$ and $L^g_{old}(d-1)$ does not exist either, (since by Observation 3 none of the three relevant old waves have a point on diagonal $-h$). Thus $L^h_{new}(d)$ was obtained from the left or diagonally, and both of these source points in the L^{h-1}_{new} wave have key values of g or greater (or do not exist on both L^{h-1}_{new} and L^g_{old}). Thus by Observation 5 $key(L^h_{new}(d)) \geq g+1$.

Subcase 1.c: $key(L^{h-1}_{new}(d)) = g + \frac{1}{2}$ and $key(L^{h-1}_{new}(d+1)) = g$.

Immediately note that because $key(L^{h-1}_{new}(d)) = g + \frac{1}{2} \in [h-2, h]$ by Observation 4, it follows that $g < h$. It suffices to show that $key(L^h_{new}(d)) \geq g+1 \geq key(L^h_{new}(d+1))$. Lemma 3.4 immediately implies that $key(L^h_{new}(d+1)) \leq g+1$. $L^{h-1}_{new}(d)$ and $L^{h-1}_{new}(d+1)$, have key values g or more and the key value of $L^{h-1}_{new}(d-1)$ (if it exists) is also $> g$, (by the first key property). If $L^{h-1}_{new}(d-1)$ does not exist, then $L^g_{old}(d)$ does not exist either, (Observation 3), and it follows from Observation 5 that $key(L^h_{new}(d)) \geq g+1$.

Subcase 1.d: $key(L^{h-1}_{new}(d)) = g + \frac{1}{2}$ and $key(L^{h-1}_{new}(d+1)) \leq g - \frac{1}{2}$ (or doesn't exist).

It suffices to show that $key(L^h_{new}(d)) \geq g + \frac{1}{2} > key(L^h_{new}(d+1))$. Clearly $L^h_{new}(d) > L^{h-1}_{new}(d)$ and hence $key(L^h_{new}(d)) \geq g + \frac{1}{2}$. The first key property on wave $h-1$ implies that $key(L^{h-1}_{new}(d+2)) \leq g-1$ (if it exists) hence $key(L^h_{new}(d+1)) \leq g$ if it was obtained from above. Similarly, $key(L^h_{new}(d+1)) \leq g$ if it was obtained diagonally, as $key(L^{h-1}_{new}(d+1)) \leq g - \frac{1}{2}$. Lastly, the second part of Lemma 3.4 also implies $key(L^h_{new}(d+1)) \leq g$ if it was obtained from $L^{h-1}_{new}(d)$ whose key value is $g + \frac{1}{2}$. Thus, regardless of how it was obtained, $key(L^h_{new}(d+1)) \leq g$.

Case 2: *Diagonal d is the leftmost (lowest) diagonal on L^h_{new}, i.e. $d = -h-1$.*

This case is not covered by Table 1, since $L^{h-1}_{new}(d)$ does not exist. We need to prove

that $\lfloor key(L^h_{new}(-h-1))\rfloor \geq key(L^h_{new}(-h)$. The key value of the point $L^{h-1}_{new}(-h)$, (the leftmost point on L^{h-1}_{new}), is strictly greater than $h-1$, since L^{h-2}_{old} and L^{h-1}_{old} do not have a point on diagonal $-h$, (Observation 3), and obviously $\leq h$ by Observation 4, hence $key(L^{h-1}_{new}(-h)) \in \{h, h-\frac{1}{2}\}$.

Subcase 2.a: $key(L^{h-1}_{new}(-h)) = h$.

L^{h-1}_{new} does not have a point on either diagonal $-h-1$ or $-h-2$, hence $L^h_{new}(-h-1)$ was obtained from $L^{h-1}_{new}(-h)$ which is the same point as $L^h_{old}(-h)$ as $key(L^{h-1}_{new}(-h)) = h$. L^h_{old} does not have points on these two diagonals either, hence $L^{h+1}_{old}(-h-1)$ was obtained from $L^h_{old}(-h)$. It follows that $L^h_{new}(-h-1) = L^{h+1}_{old}(-h-1)$, or equivalently $key(L^h_{new}(-h-1)) = h+1$. Since all key values on the new h wave are less or equal to $h+1$, (Observation 4), $key(L^h_{new}(-h)) \leq h+1$, which proves the required inequality.

Subcase 2.b: $key(L^{h-1}_{new}(-h)) = h-\frac{1}{2}$.

In this case $key(L^{h-1}_{new}(-h+1))$(if it exists) $\leq h-1$, (by the first key property), (while $L^{h-1}_{new}(-h-1)$ does not exist), and hence if $L^h_{new}(-h)$ was obtained from above, then $key(L^h_{new}(-h)) \leq h$. On the other hand if $L^h_{new}(-h)$ was obtained diagonally, we also have $key(L^h_{new}(-h)) \leq h$, since $L^{h-1}_{new}(-h) < L^h_{old}(-h)$, and $L^h_{old}(-h)$ is a furthest point. On the other hand, (as noted earlier), the key value of the lowest diagonal on the new h wave ($L^h_{new}(-h-1)$) is always strictly greater than h, which proves that $key(L^h_{new}(-h-1)) > h \geq key(L^h_{new}(-h))$. This terminates the proof of this last subcase and hence the proof of Lemma 3.5.

□

In summary, Observation 4 says that all key values on L^h_{new} are between $h-1$ and $h+1$, and Lemma 3.5 shows that for all diagonals $-h-1 \leq d < h-1$, $\lfloor key(L^h_{new}(d))\rfloor \geq key(L^h_{new}(d+1))$. The key values are hence non-decreasing and L^h_{new} has at most two in-between points. It follows that Theorem 3.2 holds: L^h_{new} can be pasted together from a prefix of L^{h+1}_{old}, a sublist of L^h_{old} and a suffix of L^{h-1}_{old} and at most two additional points, which may occur between sublists. Any individual piece may be empty.

3.1. Formal Presentation of Algorithm. We now show how Theorem 3.2 and the lemmas and observations of the previous subsection directly lead to an efficient algorithm for computing $L^0_{new} \ldots L^k_{new}$ from $L^0_{old} \ldots L^k_{old}$.

Suppose the wave L^h_{new} has been constructed, and let $L^h_{new}[a \ldots b]$ denote the sublist $< L^h_{new}(a), L^h_{new}(a+1), \ldots L^h_{new}(b) >$ of L^h_{new}, (when $b < a$ $L^h_{new}[a \ldots b]$ denotes the empty list). Let $\delta_1 \in \{0,1\}$ be the number of points on L^h_{new} with key value $h+\frac{1}{2}$, and let $\delta_2 \in \{0,1\}$ be the number of points with key value $h-\frac{1}{2}$. Define p^h_1 to be the largest (rightmost) diagonal for which $key(L^h_{new}(p^h_1)) = h+1$, or let p^h_1 be $-h-2$ if no such diagonal exists. Similarly, let $p^h_2 = max\{d : (key(L^h_{new}(d)) = h)$ or $(d = p^h_1 + \delta_1)\}$. Theorem 3.2 says that $L^h_{new}[-h-1 \ldots p^h_1] = L^{h+1}_{old}[-h-1 \ldots p^h_1]$, $L^h_{new}[p^h_1 + 1 + \delta_1 \ldots p^h_2] = L^h_{old}[p^h_1 + 1 + \delta_1 \ldots p^h_2]$, and that $L^h_{new}[p^h_2 + 1 + \delta_2 \ldots h-1] = L^{h-1}_{old}[p^h_2 + 1 + \delta_2 \ldots h-1]$. Notice that the above equalities hold even if individual pieces are empty. By Observation 2 it follows that if $L^h_{new}[a \ldots b] = L^g_{old}[a \ldots b]$, then $L^{h+1}_{new}[a+1 \ldots b-1] = L^{g+1}_{old}[a+1 \ldots b-1]$. In addition, if $g = h+1$, we also have $L^{h+1}_{new}[a] = L^{g+1}_{old}[a]$, and if $g = h-1$ we also have $L^{h+1}_{new}[b] = L^{g+1}_{old}[b]$. Thus, all of L^{h+1}_{new} is determined by these equalities except for the

two or three points on the diagonals in the interval $[p_1^h, p_1^h + 1 + \delta_1]$ and the two or three points on the diagonals in the interval $[p_2^h, p_2^h + 1 + \delta_2]$. In addition, Lemma 3.4 implies that if $key(L_{new}^h(p_2^h + 1)) = h - \frac{1}{2}$, (i.e. $\delta_2 = 1$), then $key(L_{new}^{h+1}(p_2^h + 2)) \leq h$, and hence $key(L_{new}^{h+1}(p_2^h + 2)) = h$, (since all key values on L_{new}^{h+1} are at least h). Similarly, if $key(L_{new}^h(p_1^h + 1)) = h + \frac{1}{2}$, (i.e. $\delta_1 = 1$), then $key(L_{new}^{h+1}(p_1^h + 2)) \leq h + 1$, and if in addition $p_2^h > p_1^h + 2$, then $key(L_{new}^{h+1}(p_1^h + 2)) = h + 1$. Thus the only points on L_{new}^{h+1} that can not be determined by merely examining the key value of the points on L_{new}^h, are the four points on diagonals p_1^h, $p_1^h + 1$, p_2^h, and $p_2^h + 1$.

If L_{old}^g is a doubly-linked list so that given a pointer to $L_{old}^g(d)$ one can move to $L_{old}^g(d-1)$ or $L_{old}^g(d+1)$ in constant time, and each diagonal across the waves is a doubly-linked list, so that given a pointer to $L_{old}^g(d)$, one can move to $L_{old}^{g+1}(d)$ or $L_{old}^{g-1}(d)$ in constant time, we can construct L_{new}^{h+1} from L_{new}^h and the old L waves by performing at most $O(1)$ computations and pointer changes. To do so requires knowing the *break diagonals* p_1^h and p_2^h of L_{new}^h, and determining the new *break diagonals* p_1^{h+1} and p_2^{h+1} during the computation of L_{new}^{h+1}. Notice that all but $O(1)$ of the diagonal links from L_{new}^h to L_{new}^{h+1} will be imported directly from the corresponding diagonal links of L_{old}^{h-1}, L_{old}^h and L_{old}^{h+1}, so that only those for the points on diagonals p_1^h, $p_1^h + 1$, p_2^h, $p_2^h + 1$ need to be recomputed. A formal algorithmic description is given in Figures 5, 6 and 7, which shows that such a cross-linked structure can be maintained to realize an incremental update in $O(k)$ time of $L_{new}^0, \ldots L_{new}^{k-1}$.

For reasons of simplicity, the explicit algorithm in Figures 5, 6 and 7, is given as if the waves were stored in an array and $O(1)$ access to $L_{new}^h(d)$, for any d, were possible. The cross-linked structure is clearly necessary to realize an incremental update in $O(k)$ time, but the description of the algorithm is much simpler in the latter form. The reader can verify that given the cross-linked structure and pointers to $L_{new}^h(p_1^h)$ and $L_{new}^h(p_2^h)$, as well as to $L_{old}^{h+1}(p_1^h + 1)$ and $L_{old}^h(p_2^h + 1)$, every pertinent wave element in *Construct_new_wave* in Figure 6 is $O(1)$ links away from either of these four "break pointers", or the first element of a completed new wave or an as yet unused old wave. In more detail, (which is not necessary to understand the explicit algorithms in Figures 5, 6 and 7), the following elements can be accessed in $O(1)$ time.

1. The first $O(1)$ elements of L_{new}^h and L_{old}^{h+2}: $L_{new}^h(-h - 1 + O(1))$ and $L_{old}^{h+2}(-h - 2 + O(1))$.
2. All points on L_{new}^h within $O(1)$ diagonals of p_1^h and p_2^h: $L_{new}^h(p_1^h \pm O(1))$ and $L_{new}^h(p_2^h \pm O(1))$.
3. Points on old waves that are identical to the above points: $L_{old}^{h+1}[-h - 1 \ldots p_1^h] = L_{new}^h[-h - 1 \ldots p_1^h]$.
4. The first $O(1)$ points on the suffixes of the relevant old L lists: $L_{old}^{h+1}(p_1^h + 1 + O(1))$ and $L_{old}^h(p_2^h + 1 + O(1))$.
5. Points accessible from any of the above points through old diagonal links: $L_{old}^{h+2}(p_1^h)$ which is diagonally linked to $L_{old}^{h+1}(p_1^h) = L_{new}^h(p_1^h)$, as well as $L_{old}^{h+2}(p_1^h \pm O(1))$. If $L_{new}^h(p_2^h) = L_{old}^h(p_2^h)$, (which is always true when $p_2^h > p_1^h + 1$), then $L_{old}^{h+1}(p_2^h)$ is accessible through the diagonal link from $L_{old}^h(p_2^h)$, and hence $L_{old}^{h+1}(p_2^h \pm O(1))$ is also accessible in $O(1)$ time. If $L_{new}^h(p_2^h) < L_{old}^h(p_2^h)$, (and hence $p_2^h \in \{p_1^h, p_1^h + 1\}$), then $L_{old}^{h+1}(p_2^h \pm O(1))$ is still accessible in $O(1)$ steps, as $p_2^h = p_1^h + O(1)$.

Procedure Construct_new_wave($h, p_1^h, p_2^h, p_1^{h+1}, p_2^{h+1}$)

1. $L_{new}^{h+1}[-h-2 \ldots p_1^h-1] \leftarrow L_{old}^{h+2}[-h-2 \ldots p_1^h-1]$
2. Compute($L_{new}^{h+1}(p_1^h)$)
3. Compute($L_{new}^{h+1}(p_1^h+1)$)
4. $p_1^{h+1} \leftarrow \begin{cases} p_1^h - 1 & \text{if } L_{new}^{h+1}(p_1^h) < L_{old}^{h+2}(p_1^h) \\ p_1^h & \text{if } L_{new}^{h+1}(p_1^h) = L_{old}^{h+2}(p_1^h) \text{ and } L_{new}^{h+1}(p_1^h+1) < L_{old}^{h+2}(p_1^h+1) \\ p_1^h + 1 & \text{otherwise} \end{cases}$
5. **if** $p_2^h > p_1^h + 2$ **then**
6. { $L_{new}^{h+1}[p_1^h+2 \ldots p_2^h-1] \leftarrow L_{old}^{h+1}[p_1^h+2 \ldots p_2^h-1]$
7. Double_Link($L_{new}^{h+1}(p_1^h+2)$, $L_{new}^{h+1}(p_1^h+1)$)
8. }
9. **if** $p_2^h \geq p_1^h + 2$ **then** Compute($L_{new}^{h+1}(p_2^h)$)
10. **if** $p_2^h \geq p_1^h + 1$ **then** Compute($L_{new}^{h+1}(p_2^h+1)$)
11. $p_2^{h+1} \leftarrow \begin{cases} p_2^h - 1 & \text{if } L_{new}^{h+1}(p_2^h) < L_{old}^{h+1}(p_2^h) \\ p_2^h & \text{if } L_{new}^{h+1}(p_2^h) = L_{old}^{h+1}(p_2^h) \text{ and } L_{new}^{h+1}(p_2^h+1) < L_{old}^{h+1}(p_2^h+1) \\ p_2^h + 1 & \text{otherwise} \end{cases}$
12. **if** $p_2^h + 2 \leq h$ **then**
13. { $L_{new}^{h+1}[p_2^h+2 \ldots h] \leftarrow L_{old}^{h}[p_2^h+2 \ldots h]$
14. Double_Link($L_{new}^{h+1}(p_2^h+2)$, $L_{new}^{h+1}(p_2^h+1)$)
15. }

FIG. 5. *Construction of L_{new}^{h+1} from L_{new}^{h} and auxiliary pointers*

Procedure Compute($L_x^h(d)$)

1. **if** $x = new$ **then** $\delta \leftarrow 1$ **else** $\delta \leftarrow 0$
2. $L_x^h(d) \leftarrow Slide_d\left(\max \left\{ \begin{array}{ll} L_x^{h-1}(d+1)+1 & \text{if } d < h-1-\delta \\ L_x^{h-1}(d)+1 & \text{if } -h-\delta < d < h-\delta \\ L_x^{h-1}(d-1) & \text{if } d > -h+1-\delta \end{array} \right\} \right)$
3. **if** $h > 0$ **and** $d > -h-\delta$ **then**
4. { Double_Link($L_x^h(d)$, $L_x^h(d-1)$)
5. **if** $d < h-\delta$ **then** Double_Link($L_x^h(d)$, $L_x^{h-1}(d)$)
6. }

FIG. 6. *Computation of $L_{new}^h(d)$ or $L_{old}^h(d)$ and update of all relevant pointers*

Procedure New_Wave

1. $L_{new}^0(-1) \leftarrow Slide_{-1}(0)$
2. $p_2^0 \leftarrow -1$
3. **if** $L_{new}^0(-1) < L_{old}^1(-1)$ **then** $p_1^0 \leftarrow -2$ **else** $p_1^0 \leftarrow -1$
4. **for** $h \leftarrow 0$ **to** $k-2$ **do**
5. Construct_new_wave($h, p_1^h, p_2^h, p_1^{h+1}, p_2^{h+1}$)
6. **for** $d \leftarrow -k-1$ **to** p_1^k+1 **do**
7. Compute($L_{old}^{k+1}(d)$)
8. Construct_new_wave($k-1, p_1^{k-1}, p_2^{k-1}, p_1^k, p_2^k$)

FIG. 7. *Construction of $L_{new}^0 \ldots L_{new}^k$ from the corresponding old waves*

The computation of the last wave L^k_{new} and its diagonal links from and to L^{k-1}_{new} may take an additional $O(k)$ time since the required L^{k+1}_{old} piece and its diagonal links from L^k_{old} have not been computed previously and need to be computed now, (see lines 6 and 7 in Figure 7). In total the computation of the k waves requires computing $O(k)$ new points and updating $O(k)$ links. The computation of each new point is done by calling the *Slide* function, (line 2 in Figure 6) which takes $O(1)$ time. In total the computation of the k waves for D_{new} takes $O(k)$ time, when given the k waves for D, cross-linked across waves and diagonals as described above.

3.2. An Analogous Theorem for Longest Common Subsequences. As pointed out in the introduction, the model in which one allows insertions and deletions, called indels, but not mismatches is an important variation because it is dual to finding the longest common subsequence. A theorem similar to Theorem 3.2, but simpler, holds in this situation. We simply sketch the result here, since the proof method and outline remain the same.

When mismatches are not allowed, several of the preliminary lemmas of Section 2 change in small ways. In Lemma 2.1, the term δ_{a_i,b_j} should be multiplied by two as a substitution can now only be achieved by an insertion and deletion. In Lemma 2.3 values increase along a diagonal by zero or two, i.e., $D[i,j] - D[i-1,j-1] \in \{0,2\}$. Next note that with an even number of indels one must end in an even numbered diagonal, and an odd diagonal with an odd number of indels. Thus for this variation of the problem, an h-wave has $h+1$ points on every other diagonal, i.e., $L^h = < L^h(-h), L^h(-h+2), \ldots, L^h(h-2), L^h(h) >$. Finally, in Lemma 2.8, one must drop the term $L^{h-1}(d)+1$ from the 3-way maximum as it represents the contribution of extension via a substitution. As a consequence a point is either obtained from the *left* or from *above*.

We continue to define key values as in Definition 3.1. Note however that the diagonals on a new h wave do not contain old h points, and hence an in-between point p on L^h_{new} is between the old $h-1$ and the old $h+1$ wave. As a result of our definitions a key value of a point p on a new h wave will assume one of the three values in $\{h-1, h+\frac{1}{2}, h+1\}$. In spirit, all the Observations in Section 3 continue to hold with the obvious modifications that follow from the fact that points can only be obtained in two ways. Theorem 3.2 becomes:

THEOREM 3.6. *Wave h in D_{new} (L^h_{new}) is the concatenation of (up to) three pieces: (i) a prefix of L^{h+1}_{old}, (ii) an in-between point p with $key(p) = h+\frac{1}{2}$, and (iii) a suffix of L^{h-1}_{old}. Not all pieces must be present.*

The proof of Theorem 3.6 is in some sense a subset of the proof of Theorem 3.2. So rather than prove it formally, we sketch the main elements and leave the details as an exercise. The informal statement of the first key property remains the same, but its formal statement becomes:

1st Key Property: For $d \in [-h+1, h-1]$, $\lfloor key(L^h_{new}(d-2)) \rfloor \geq key(L^h_{new}(d))$.

to account for the fact that only alternate diagonals are relevant. This first key property, can similarly be proven by induction on h, using the second key property and its immediate consequence, the analog of Lemma 3.4.

2nd Key Property: If $key(L^h_{new}(d)) = h + \frac{1}{2}$, then $L^{h+1}_{old}(d)$ was obtained from above.

Because the proof in this case is simpler, we combine the proof of the second key property and of the analog of Lemma 3.4 in Lemma 3.7.

LEMMA 3.7. *If the first key property holds up to wave h, and if $key(L^h_{new}(d)) = h + \frac{1}{2}$, then (1) $key(L^{h+1}_{old}(d)$ was obtained from above, and $key(L^{h+1}_{new}(d+1)) = h$.*

Proof. (sketch) The proof is by induction:

BASIS: We need to show that if $key(L^0_{new}(-1)) = \frac{1}{2}$, then $L^1_{old}(-1)$ was obtained from above and $key(L^1_{new}(0)) = 0$. $L^1_{old}(-1)$, (whether it collides with $L^0_{new}(-1)$ or not), was obtained from the (only) old zero point $L^0_{old}(0)$, i.e. from above. In addition if $key(L^0_{new}(-1)) = \frac{1}{2}$, then $L^0_{new}(-1)$ must be $\leq L^0_{old}(0)$, hence $L^1_{new}(0)$ will be equal to $L^0_{old}(0)$.

INDUCTION: Assume that Lemma 3.7 holds up to wave h. Consider an in-between point $L^h_{new}(d)$, for which $key(L^h_{new}(d)) = h + \frac{1}{2}$. We need to show that $L^{h+1}_{old}(d)$ could not have been obtained from the left, and that $key(L^{h+1}_{new}(d+1) = h$. $key(L^h_{new}(d)) = h + \frac{1}{2}$, implies that one of the two points $L^{h-1}_{new}(d-1)$, $L^{h-1}_{new}(d+1)$ must have a key value of $h - \frac{1}{2}$ or h, and since $key(L^{h-1}_{new}(d-1)) \geq key(L^{h-1}_{new}(d+1))$, it must be that $key(L^{h-1}_{new}(d-1)) \in \{h - \frac{1}{2}, h\}$. $key(L^{h-1}_{new}(d-1)) = h - \frac{1}{2}$ can be ruled out, since it contradicts the inductive hypotheses, which says that in this case $key(L^h_{new}(d)) = h - 1$. If $L^{h+1}_{old}(d)$ was obtained from $L^{h-1}_{new}(d-1)$, (i.e. the left) and $key(L^{h-1}_{new}(d-1)) = h$, we would have $key(L^h_{new}(d)) = h + 1$, again a contradiction. Hence $L^{h+1}_{old}(d)$ was obtained from $L^h_{old}(d+1)$, (i.e. from above). This in turn implies that $L^h_{old}(d+1) \geq L^h_{new}(d)$. If $L^{h+1}_{new}(d+1)$ was obtained from $L^h_{new}(d)$, then the above inequality would imply that $L^{h+1}_{new}(d+1) \leq L^h_{old}(d+1)$, but since $L^{h+1}_{new}(d+1) \geq L^h_{old}(d+1)$, (always), it follows in this case that $L^{h+1}_{new}(d+1) = L^h_{old}(d+1)$. On the other hand by the first key property on wave h, $key(L^h_{new}(d+2)) = h - 1$, hence if $L^{h+1}_{new}(d+1)$ was obtained from $L^h_{new}(d+2)$, we also have $L^{h+1}_{new}(d+1) = L^h_{old}(d+1)$. Hence regardless of how $L^{h+1}_{new}(d+1)$ was obtained, $key(L^{h+1}_{new}(d+1)) = h$. □

Given the second key property and Lemma 3.7 the proof of the first key property, and hence Theorem 3.6, is outlined in a nutshell as follows. Given a pair of consecutive points on wave h, $L^h_{new}(d-2)$ and $L^h_{new}(d)$, we consider the point $L^{h-1}_{new}(d-1)$. If $key(L^{h-1}_{new}(d-1)) = g$, $(g \in \{h-2, h\})$, then $key(L^h_{new}(d-2)) \geq g + 1$, while $key(L^h_{new}(d)) \leq g + 1$. If $L^{h-1}_{new}(d-1)$ is an in-between point then $key(L^{h-1}_{new}(d-1)) = h - \frac{1}{2}$, and by Lemma 3.7 $key(L^h_{new}(d)) = h - 1$, while $key(L^h_{new}(d-2))$ is always greater than or equal to $h - 1$.

3.3. Explicit Algorithm for LCS. The explicit algorithm given in Figures 8, 9, 10 is similar to and simpler than the one for edit distance. Waves are still doubly linked lists, with $L^h(d)$ doubly-linked to $L^h(d-2)$ and $L^h(d+2)$). Diagonal-links will be slightly different, in that $L^h(d+1)$ is linked to $L^{h+1}(d)$ (as L^{h+1} does not have a point on diagonal $d+1$). p^h is defined as the largest (rightmost) diagonal on L^h_{new} for which $key(L^h_{new}(p^h)) = h + 1$ and is set to $-h - 2$ if there is no such diagonal.

Procedure Construct_new_LCS_wave(h, p^h, p^{h+1})

1. $L^{h+1}_{new}[-h-2 \ldots p^h - 1] \leftarrow L^{h+2}_{old}[-h-2 \ldots p^h - 1]$
2. Compute_LCS($L^{h+1}_{new}(p^h + 1)$)
3. $p^{h+1} \leftarrow \begin{cases} p^h - 1 & \text{if } L^{h+1}_{new}(p^h+1) < L^{h+2}_{old}(p^h+1) \\ p^h + 1 & \text{otherwise} \end{cases}$
4. $L^{h+1}_{new}[p^h + 3 \ldots h] \leftarrow L^h_{old}[p^h + 3 \ldots h]$
5. Double_Link($L^{h+1}_{new}(p^h + 3)$, $L^{h+1}_{new}(p^h + 1)$)

FIG. 8. *Construction of L^{h+1}_{new} from L^h_{new} and auxiliary pointers*

Procedure Compute_LCS($L^h_x(d)$)

1. **if** $x = new$ **then** $\delta \leftarrow 1$ **else** $\delta \leftarrow 0$
2. $L^h_x(d) \leftarrow Slide_d \left(\max \left\{ \begin{array}{ll} L^{h-1}_x(d+1) + 1 & \text{if } d < h - \delta \\ L^{h-1}_x(d-1) & \text{if } d > -h - \delta \end{array} \right\} \right)$
3. **if** $h > 0$ **and** $d > -h - \delta$ **then**
4. { Double_Link($L^h_x(d)$, $L^h_x(d-2)$)
5. **if** $d < h - \delta$ **then** Double_Link($L^h_x(d)$, $L^{h-1}_x(d+1)$)
6. }

FIG. 9. *Computation of $L^h_{new}(d)$ or $L^h_{old}(d)$ and update of all relevant pointers*

Procedure New_LCS_Wave

1. $L^0_{new}(-1) \leftarrow Slide_{-1}(0)$
2. **if** $L^0_{new}(-1) < L^1_{old}(-1)$ **then** $p^0 \leftarrow -2$ **else** $p^0 \leftarrow -1$
3. **for** $h \leftarrow 0$ **to** $k - 2$ **do**
4. Construct_new_LCS_wave(h, p^h, p^{h+1})
5. **for** $d \leftarrow -k - 1$ **to** $p^k_1 + 1$ **by** 2 **do**
6. Compute_LCS($L^{k+1}_{old}(d)$)
7. Construct_new_LCS_wave($k - 1, p^{k-1}, p^k$)

FIG. 10. *Construction of $L^0_{new} \ldots L^k_{new}$ from the corresponding old waves*

4. Four Applications. The power of the incremental algorithm of the proceeding section is that it delivers an encoding of the dynamic programming solution for each and every problem so obtained. In the context of the four applications of this section this feature of the method allows the algorithm to completely explore the space of solutions to each subproblem. In the case of some of the applications this is essential to their efficient solution and in others it provides leverage not found in previous algorithms that can only keep track of the best solution to a set of subproblems. For each application we show how to apply the incremental algorithm as a subprocedure and focus on how its wave structure is processed at each stage to provide the desired solutions.

Before proceeding with the description of the applications, introduce a notation for the wave structure that is different from the one introduced in Section 3, but is convenient for the discussion of applications. In Section 3 we showed how to compute the $k+1$ waves $L^0_{new}, \ldots L^k_{new}$ of the edit-distance matrix $D_{new}[i,j]$ of A and bB, when given the $k+1$ *waves* $L^0_{old}, \ldots L^k_{old}$ of the matrix $D_{old}[i,j]$ of A versus B. In all of our applications the overall outline of the algorithm is to start with a k-thresholded solution for A versus some suffix B_s of B. (Recall that B^r_l is the substring $b_{l+1}b_{l+2}\ldots b_r$, and that $B_l = B^n_l$ and $B^r = B^r_0$.) When $s = n$ this is the trivial solution of A versus the empty string, and when $s < n$ the initial solution can be computed with the standard $O(n + k^2)$ algorithm. Thereafter, the application algorithm incrementally computes the solutions to A versus B_l for $l = s-1, s-2, \ldots 0$ using the incremental algorithm of Section 3 as a subroutine.

At any moment we will have $k + 1$ waves L^0, L^1 $\ldots$ L^k modelling the solution to a comparison between some suffix B_l of B and A. Let the *origin diagonal* for this problem be l, and denote by D_l the dynamic programming matrix of the comparison of A with B_l. Further let $C^h(d) = L^h(l+d)$ denote the wave value that is at diagonal d of the matrix D_l. In this way $L^h = < C^h(-h), C^h(-(h-1)), \ldots, C^h(h-1), C^h(h) >$ regardless of the suffix of B at hand. Recall from the preliminaries that in order to be compliant with the recursion for computing L^h in terms of L^{h-1} it was convenient to sometimes set $L^h(d)$ to ∞ even though the extreme point on diagonal d in D had value h. But in the context of our applications, it is now convenient to reset these ∞ values so that C^h always holds the furthest h point on the corresponding diagonal, *within the boundary of* D, if such an h point exists. Formally if $L^h(l+d) = \infty$ while $L^{h-1}(l+d) \neq p_{last}(d)$, where $p_{last}(d) = min\{m, n-l-d\}$ is the row number of the last point on diagonal d in matrix D, then set $C^h(d) = p_{last}(d)$. Note that the D value of $p_{last}(d)$ on diagonal d is indeed equal to h in this case. It is easy to see that this "adjustment" of values can be done $O(k)$ time per D_l matrix.

Recall from the previous section that each wave is implemented as a doubly-linked list so that given a pointer to $C^h(d)$ one can move to $C^h(d-1)$ or $C^h(d+1)$ in constant time, and each diagonal across the waves is in a doubly-linked list so that given a pointer to $C^h(d)$ one can move to $C^{h+1}(d)$ or $C^{h-1}(d)$ in constant time.

4.1. Approximate String Matching and Longest Prefix. The approximate string matching problem was used in the introduction as an example of how our incremental alignments algorithm could be used to find *all* matching substrings in $O(nk)$ time. We begin by exploring in greater detail how our incremental algorithm is applied to the following slightly more general problem and the leverage it provides

over other algorithms. Consider first finding for each position $l \in [0, n]$ of a text B the length $m(l)$ of the longest prefix $A^{m(l)}$ of A that can be matched to some prefix of B_l with no more than k differences. Formally, $m(l) = max\{p \in [0, m] : \exists r \in [l, n],\ ED(A^p, B_l^r) \leq k\}$. Further consider finding for each l the set of all r such that the substring B_l^r of the text B and the prefix $A^{m(l)}$ are within edit distance k of each other. When $m(l) = m$ we call such a substring of the text a *k-match*, and otherwise a *longest prefix k-match*. In summary, the *longest prefix approximate match problem* is given strings A, B, and threshold k, find for each l the length $m(l)$ and the set of indices r such that $ED(A^{m(l)}, B_l^r) \leq k$. The problem obviously generalizes the approximate string matching problem which seeks all (l, r) for which $m(l) = m$.

We start by building the trivial k-thresholded solution for A versus B_n $(= \varepsilon)$ and then proceed incrementally. First observe that constructing the trivial solution for A versus B_n simply requires building an initial cross-linked wave structure and setting $C^h(d)$ to h if $d = -h$ and to ∞ otherwise. The algorithm then proceeds to produce solutions for A versus progressively longer suffixes of B taking $O(k)$ time per incremental shift using our central result. The only remaining detail is to show how, given the k-thresholded solution for A versus some suffix B_l, one finds the longest prefix of A and all r and h such that $ED(A^{m(l)}, B_l^r) = h \leq k$. Note that when $m(l) < m$ the corresponding edit distance is always k, since we would otherwise choose a longer prefix of A. Even when $m(l) = m$ there is at least one diagonal d for which $C^k(d)$ is exactly m. This is easily seen by observing that the least possible number of differences on diagonal $-k$ is k, hence $D_l[m, m-k]$ is always $\geq k$, and hence $C^k(-k) \leq m$. For $d < 0$, $C^k(d) < m$ implies that $C^{k-1}(d+1) < m$ and hence $C^k(d+1) \leq m$. Hence even if $C^k(d) = \infty$ for some $d \leq 0$, we have $C^k(d') = m$ for some diagaonal d' between $-k$ and d. We can therefore determine $m(l)$ by examining all non ∞ values on wave k, hence $m(l) = max_d\{C^k(d) \mid C^k(d) \leq m\}$, and we wish to find d and h such that $C^h(d) = m(l)$, for this will give us all points $(m(l), m(l)+d)$ for which $D_l[m(l), m(l)+d] = h$.

It suffices to first traverse wave C^k to determine $m(l)$ and thereafter traverse the wave structure in order of diagonals, moving up or down along a diagonal list to find the entry on that diagonal equal to $m(l)$, if any. The algorithm fragment in Figure 11 gives the details. Note that we start with looking for a match with k differences on diagonal $-k$. Thereafter d and h are advanced in unit increments to suggest the pointer-based traversal of the cross-linked structure. The traversal again takes advantage of the fact that adjacent D-values never differ by more than one implying that if we move from a best point on wave h and diagonal d to the adjacent wave point on the next diagonal, at most one move up or down to wave $h+1$ or wave $h-1$ along that diagonal will reach a point on row $m(l)$ if there is one.

Figure 11 clearly takes $O(k)$ time which is no more expensive than the time to produce the wave structure for the given index l. Thus the overall algorithm takes $O(nk)$ time. Note that a corollary is that there are at most $O(nk)$ k-matches to A. While there are other algorithms for approximate string matching that take only $O(nk)$ time, none delivers or can be trivially extended to deliver the longest prefix of A that can be matched with k differences to some substring of B and to deliver all the k-matching substrings and their associated edit distances. Finding the longest prefix that can be matched to a given string turns out to be crucial in an algorithm for finding approximate repeats, i.e. adjacent substrings whose edit distance is no

```
1.   m(l) ← max_d{C^k(d) | C^k(d) ≤ m}
2.   h ← k
3.   for d = −k to k by 1 do
4.    {   if h < k and C^{h+1}(d) ≤ m(l) then
5.            h ← h + 1
6.        else if C^h(d) = ∞ then
7.            h ← h − 1
8.        if C^h(d) = m(l) then
9.            Report A^{m(l)} matches B_l^{l+m(l)+d} with h differences
10.   }
```

FIG. 11. *Reporting Matching Substrings.*

more than k [LS-93], and is almost certain to find additional applications. Finding all matching substrings is crucial if one has some secondary criteria that is a non-linear function of match length and edit distance. In this case one needs to examine all matches and not simply the best one ending or beginning at a given index. For example, if $A = aaaacbbbbccccc$ and $B = xxxxxaaaaccccccbbbbccccccxxxx\ldots$, then both B_5^{15} and B_{14}^{24} match A (of length 14) with 4 differences but B_5^{24} matches with 5 differences. The last match is conceivably more significant than the two others, (involving 14 identical symbols versus 10), but would not be revealed by previous algorithms. These algorithms can only determine the match(es) with the minimum number of differences ending at a given character of B, or (in a backward solution) the match(es) with the minimum number of differences starting at a given character of B. That is, previous algorithms only detect the difference 4 match B_{14}^{24} to the suffix of B^{24} when run left-to-right over B, and the difference 4 match B_5^{15} to the prefix of B_5 when run right-to-left over B. The difference 5 match B_5^{24} is missed in both directions. The next application treats this issue in more detail.

4.2. Approximate Overlap. Our second example comes from a problem in molecular biology that arises in sequencing DNA. Current methods for determining sequence allow the direct determination of the DNA sequence of a string of length less than 1000. To determine the sequence of a very long DNA strand, say 50,000 symbols in length, an experimentalist employing the "shotgun" sequencing method, randomly extracts fragments of sufficiently small length from the subject strand and determines the sequence of these fragments via a direct experimental method. The resulting *fragment assembly problem* is to determine the subject strand given the collected set of fragments.

The first step in solving the fragment assembly problem is to compare every sequence against every other sequence to see if a suffix of one matches a prefix of another. [4] Such detected overlaps indicate the possibility (but not the certainty) that the two fragments came from overlapping regions of the subject stand. Detecting the overlaps is complicated by the fact that direct sequencing experiments are imperfect and so do not produce the exact sequence. Typically, the error rate runs at about a 1-10% difference between the reported string and the true fragment sequence. Thus fragments must be compared to determine if there is an *approximate overlap* between

[4] [GLS-92] describes an algorithm that finds the exact (0 differences) matches between prefixes and suffixes of a set of strings.

them. A fast method is essential for this fundamental subproblem since it must be solved for a quadratic number of fragment pairs.

More precisely given a threshold k (reflective of the length of the fragments and the error rate of the sequencing method), and two fragments A and B, we say A approximately overlaps B within threshold k if either (a) a prefix of A aligns with a suffix of B with not more than k differences, or (b) A aligns with a substring of B with not more than k differences. Matches of type (a) are called *dovetail* matches and those of type (b) are *containment* matches. Unfortunately, whenever A and B approximately overlap within k-difference, there are typically a number of possible ways to do so. One way to compare the significance of different alignments is to choose the overlap which is the least likely to occur by chance, as suggested in [KM-94]. Let $\Pr_\Sigma(m, n, k)$ be the probability, or some approximation thereof, that two strings of respective lengths m and n formed by random Bernoulli trials over a Σ symbol alphabet can be aligned with k-or-less differences. For this discussion, let us assume a precomputed table containing values $Pr_\Sigma[m, n, k]$ between 0 and 1 for the relevant range of m, n and k (e.g., $0 \leq m, n \leq 2000$ and $0 \leq k \leq 400$ for most DNA sequencing projects). Note that Pr_Σ is typically a non-linear function of m, n and k.

Given a threshold k and strings A and B of length m and n, the *approximate overlap problem* can now be stated as finding three indices l, r, and p such that:

(a) $r = n$ or $p = m$, and
(b) $ED(A^p, B_l^r) = h \leq k$, and
(c) $Pr_\Sigma[p, r - l, h]$ is minimal.

When $r = n$ the approximate overlap is of the dovetail variety (prefix A^p versus suffix B_l), and when $p = m$ it is of the containment type (A versus substring B_l^r).

As for the approximate match problem, the approximate overlap problem is solved by incrementally computing the solution for A versus B_l for $l = n, n-1, \ldots 0$. As before, the initial solution for B_n versus A is easy to compute and $O(k)$ time is spent thereafter incrementally delivering each additional k-thresholded solution as cross-threaded lists of the $k+1$ waves, $L^0, L^1, \ldots L^k$. For each suffix B_l, one traverses the wave structure finding all p and r that satisfy conditions (a) and (b) above. We term such a triple, (l, r, p), a *candidate*. As each candidate is discovered its Pr_Σ "score" is compared against the minimum scoring triple (L, R, P) of probability score S encountered thus far in the algorithm, and entered as the new best if its score is lower. Thus at the end of the algorithm the indices delimiting a minimum scoring approximate overlap and its probability score are delivered. The algorithm is shown in Figure 12.

As for approximate match, the tricky detail is the discovery of the candidates within a given solution. Suppose we have the k-thresholded solution for A versus B_l. Like approximate match, containment candidates correspond to those d and h for which $C^h(d) = m$ for this implies $ED(A, B_l^{l+m+d}) = h$. Dovetail candidates correspond to those furthest points in the structure that reach the extreme column $n - l$ of D_l, the dynamic programming matrix for the problem of comparing A and B_l. That is we seek d and h for which $C^h(d) = n - l - d$ and is therefore a point on the extreme column of D_l, for this implies $D_l[n - l - d, n - l] = h$ which in turn implies $ED(A^{n-l-d}, B_l) = h$.

```
1.  Compute initial solution for A versus B_n.
2.  (L, R, P, S) ← (n, n, 0, 1)    # B_n^n overlaps A^0 with 0 errors with probability 1.
3.  for l = n − 1 downto 0 by 1 do
4.    { Incrementally compute solution for A versus B_l.
5.      h ← k
6.      for d = −k to k by 1 do
7.        { if h < k and C^{h+1}(d) < ∞ then
8.              h ← h + 1
9.          else if C^h(d) = ∞ then
10.             h ← h − 1
11.         if C^h(d) = m or C^h(d) + d = n − l then
12.             s ← Pr_Σ[C^h(d), d + C^h(d), h]
13.             if s < S then
14.                 (L, R, P, S) ← (l, l + d + C^h(d), C^h(d), s)
15.       }
16.   }
17. Best overlap is A^P with B_L^R with score S.
```

FIG. 12. *Approximate Overlap Algorithm.*

Observe that the algorithm given in Figure 12 has exactly the same wave-traversal logic as the one given in Figure 11. The modifications are related to differences between the applications: lines 1 and 8-9 in Figure 11 versus lines 2, 11-14, and 17 in Figure 12. As a consequence, it is clear the algorithm takes $O(nk)$ time.

The algorithm can be further refined to deliver not only the indices of the best approximate overlap in $O(nk)$ time but an alignment achieving it as well. An alignment can of course be produced in all applications but it is particularly appealing here since we output only the "best matching substring" with respect to our scoring function and we can produce $O(1)$ alignments per substring B_l in the desired time bound. During the traversal of the k-thresholded solution for A versus B_l, record the best candidate encountered in the traversal and if it becomes the best candidate seen thus far in the algorithm, then take $O(k)$ additional time to record the alignment of the overlap modeled by the candidate. In order to take only $O(k)$ additional time, the alignment must be recorded as the ordered sequence of its $O(k)$ *differences*, often called a Δ-encoding of the alignment. Building this encoding simply requires tracing back from the entry $C^h(d)$ corresponding to the candidate. Specifically, from $C^h(d)$ trace back to whichever entry yields the maximum of $C^{h-1}(d-1)$, $C^{h-1}(d)+1$, or $C^{h-1}(d+1)+1$ (by Lemma 8), and then trace back from that entry recursively until $C^0(0)$ is reached. If $u = C^{h-1}(d-1)$ gave the maximum, then append "Insert b_{l+u+d}" to the Δ-encoding. If $v = C^{h-1}(d)$ gave the maximum, then append "Substitute $b_{l+v+d+1}$ for a_{v+1}". Finally, if $w = C^{h-1}(d+1)$ gave the maximum, then append "Delete a_{w+1}". Upon completion of the algorithm, the Δ-encoding of the best approximate overlap will have been recorded, and one can use it to produce a display of the alignment in $O(n+m)$ time if desired.

4.3. Cyclic String Comparison. Yet another variation of traditional string comparison involves considering *cyclic shifts* of the two strings A and B in question. Let $cycle(a_1 a_2 \ldots a_m) = a_2 \ldots a_m a_1$, and let $cycle^p(A)$ be the result of applying $cycle$

exactly p times. The *cyclic string comparison problem* is to determine p and q such that $e = ED(cycle^p(A), cycle^q(B))$ is minimal. It is quite easy to see that if the minimum is obtained for $cycle^p(A)$ and $cycle^q(B)$, then by simply cyclically shifting an alignment achieving this minimum one obtains an equally good alignment between A and $cycle^r(B)$ for some r. Thus the problem really reduces to the simpler one of finding q such that $ED(A, cycle^q(B))$ is minimal. This problem was introduced by Mathias Maes [Ma-90] and he gives an $O(mn \log m)$ algorithm for the problem that permits arbitrarily weighted edit costs. Our incremental alignment algorithm leads to a more efficient $O(ne)$ time algorithm for the case of unit cost editing operations.

First we present an $O(n^2)$ algorithm, and later show how to refine it to give an $O(ne)$ algorithm. Consider comparing A and $\overline{B} = B \cdot B$ (B concatenated with itself). Let the threshold $k = n$ so that for any threshold structure computed $C^k(n-m) \geq m$. Begin by computing the threshold structure for A versus $\overline{B}_n$ ($=B$) using the greedy algorithm in $O(n^2)$ time. Then incrementally compute the thresholded structure for A versus $\overline{B}_l$, for $l = n-1, n-2 \ldots 0$. We examine each threshold structure to find $ED(A, cycle^l(B))$ in $O(k)$ time. Specifically, it is that h for which $C^h(n-m) = m$ and this is easily found by starting with $C^k(n-m)$ and walking the diagonal list until h is encountered. Note that because k was chosen to be n it follows that h is always on the diagonal list. It takes $O(k)$ time to compute each incremental solution and $O(k)$ additional time to find the edit distance for the given cyclic shift of B. Thus the algorithm takes $O(kn) = O(n^2)$ time to find the minimum edit distance, e, over all possible cyclic shifts.

To bring the complexity down to $O(ne)$ time, consider running the algorithm with a threshold $k < n$. If for a given cyclic shift, $C^k(n-m) < m$ then the edit distance for that cyclic shift of B cannot be determined. On the other hand, if $C^k(n-m) \geq m$ then the edit distance can be computed and k upper bounds the answer e to the overall problem. So consider running the algorithm with $k = 1$, and then with $k = 2$, and $k = 4$, and so on in geometric sequence until the edit distance for at least one cyclic shift is determined in a given trial. Of course, in this last trial, the best edit distance obtained in the trial is the answer, e, to the cyclic string comparison problem. Since k is doubled with each trial, the total time complexity is bounded by the time of the last trial and $k = O(e)$ in that trial. Thus the algorithm takes $O(ne)$ time.

4.4. Text Screen Updating. Screen oriented programs maintain a representation of an object and present a view of it on the screen. For example, screen editors keep an internal edit buffer and display a block of lines from the buffer. The screen must be updated when the object is changed. In one solution, procedures that modify the object must also update the view or at least specify how the view has changed. A cleaner approach lets an autonomous screen manager module determine how to update the screen by comparing its record of the screen contents with views of the modified object. The interface to the screen manager is then a single routine, *refresh*, that updates the screen with respect to the current object. It is given no information other than the object and screen contents. Unfortunately, the simplicity of the interface requires the screen manager to solve a difficult comparison problem.

The feasibility of this design is demonstrated by the UNIX EMAC editor [Go-81] and the Maryland Window System [Ws-85]. In a two level approach, sequences of lines are compared to decide at the top level which lines to delete, insert, and replace. At

the bottom level, the sequences of characters in two lines are compared to appraise and perform line or row replacements. The approach is not guaranteed to update the screen with a minimal set of terminal commands but nonetheless performs well. However, a number of improvements are possible at both levels. At the lower level, Myers and Miller [MM-89] developed algorithms that account for the non-uniformity of terminal command costs and produce optimal update command sequences for the row replacement subproblem. At the top level, a weakness of the earlier approach is the assumption that the screen-sized segment of buffer lines that is to replace the current screen contents is known a priori. More realistically, the screen manager should determine the optimal *window position*, i.e. the screen-sized segment of the buffer that most closely resembles the current screen contents subject to the constraint that the current cursor position be in this segment. A useful approximation to the theoretically optimal choice can be computed economically with our incremental algorithm by finding a window position minimizing the number of screen rows that need to be updated, inserted, or removed.

Suppose B is the current buffer contents, c is the current cursor position, and S is the current screen contents, where B and S are viewed as strings over the infinite alphabet of lines of ASCII text. Suppose B is n lines long and that the screen displays m lines. Formally, the *window positioning problem* is to find a position $p \in [c-m, c-1]$ that minimizes $e = ED(B_p^{p+m}, S)$. Given the cursor position c it is clear that we can restrict our attention to the substring $\overline{B} = B_{c-m}^{c+m-1}$ of B because the window B_p^{p+m} must contain line c. Now observe that this problem is very similar to the one we solved for the cyclic string comparison problem. Namely we compute solutions for $\overline{B}_l$ for $l = m-1, m-2, \ldots 0$, and for each determine $ED(\overline{B}_l^{l+m}, S)$ which is the value h for which $C^h(0) = m$. Using the same geometric progression of threshold increases as in the cyclic string comparison problem, gives an $O(me)$ algorithm for the window positioning problem.

Acknowledgement

We would like to thank Esko Ukkonen for bringing the Cyclic String Comparison problem to our attention.

REFERENCES

[BV-93] O. Berkman and U. Vishkin *Recursive star-tree parallel data-structure,* SIAM J. Comput., Vol. 22, No. 2 (1993), pp. 221-242.

[GP-90] Z. Galil and Q. Park *An improved algorithm for approximate string matching,* SIAM J. Comput., Vol. 19, No. 6 (1990) pp. 989-999.

[Go-81] J. Gosling *A Redisplay Algorithm,* Proceedings of the ACM SIGPLAN/SIGOA Symposium on Text Manipulation, (1991) pp. 123-129.

[GLS-92] D. Gusfield, G. M. Landau, and B. Schieber *An efficient algorithm for the all pairs suffix-prefix problem,* Information Processing Letters, Vol. 41 (1992), pp. 181-185.

[HD-80] P.A. Hall and G.R. Dowling *Approximate String Matching,* Computing Surveys, Vol. 12, No. 4 (1980), pp. 381-402.

[HT-84] D. Harel and R.E. Tarjan *Fast algorithms for finding nearest common ancestors,* SIAM J. Comput., Vol. 13 (1984), pp. 338-355.

[Hi-77] D.S. Hirschberg *Algorithms for the Longest Common Subsequence Problem,* Journal of

ACM, Vol. 24, No. 4 (1977), pp. 664-675.

[HS-77] J.W. Hunt and T.G. Szymanski *An algorithm for differential file comparison,* Comm. of the ACM, Vol. 20, No. 5 (1977), pp. 350-353.

[KM-94] J. Kececioglu and E. Myers *Exact and Approximate Algorithms for the Sequence Reconstruction Problem,* Algorithmica, (1994), in press.

[LS-93] G.M. Landau and J.P. Schmidt *An algorithm for approximate tandem repeats,* Proc. 4th Symp. Combinatorial Pattern Matching, Spring-Verlag Lecture Notes in Computer Science, Vol. 648 (1993), pp. 120-133.

[LV-88] G.M. Landau and U. Vishkin *Fast string matching with k differences,* J. of Comp. and Sys. Sci., Vol. 37, No. 1 (1988), pp. 63-78.

[LV-89] G.M. Landau and U. Vishkin *Fast parallel and serial approximate string matching,* Journal of Algorithms, Vol. 10, No. 2 (1989), pp. 157-169.

[Ma-90] M. Maes *On a cyclic string-to-string correction problem* Info. Proc. Lett., Vol. 35 (1990), pp. 73-78.

[Mc-76] E. M. McCreight *A space-economical suffix tree construction algorithm,* J. of the ACM, Vol. 23 (1976), pp. 262-272.

[My-86a] E. Myers *An $O(ND)$ difference algorithm and its variations,* Algorithmica, Vol. 1, No. 2 (1986), pp. 251-266.

[My-86b] E. Myers *Incremental Alignment Algorithms and Their Applications,* Tech. Rep. 86-22, Dept. of Computer Science, U. of Arizona, Tucson, AZ 85721, (1986).

[MM-89] E. Myers and W. Miller *Row Replacement Algorithms for Screen Editors,* ACM Trans. Prog. Lang. and Systems, Vol. 11 (1989), pp. 33-56.

[NKY-82] N. Nakatsu, Y. Kambayashi, and S. Yajima *A longest common subsequence algorithm suitable for similar text string,* Acta Informatica, Vol. 18 (1982), pp. 171-179.

[NW-70] S.B. Needleman and C.D. Wunsch *A general method applicable to the search for similarities in the amino acid sequence of two proteins,* J. of Mol. Bio., Vol. 48 (1970), pp. 443-453.

[SV-88] B. Schieber and U. Vishkin *On finding lowest common ancestors: simplification and parallelization,* SIAM J. Comput., Vol. 17 (1988), pp. 1253-1262.

[Se-80] P.H. Sellers *The Theory and Computation of Evolutionary Distances: Pattern Recognition,* Journal of Algorithms, Vol. 1 (1980), pp. 359-373.

[SW-81] T.F. Smith and M.S. Waterman *Identification of Common Molecular Subsequences,* Journal of Molecular Biology, Vol. 147, No. 2 (1981) pp. 195-197.

[Uk-85a] E. Ukkonen *Algorithms for approximate string matching,* Information and Control, Vol. 64 (1985), pp. 100-118.

[Uk-85b] E. Ukkonen *On approximate string matching,* J. of Algorithms, Vol. 6 (1985), pp. 132-137.

[WF-74] R.A. Wagner and M.J. Fischer *The string-to-string correction problem,* J. of the ACM, Vol. 21, No. 1 (1974), pp. 168-173.

[Wn-73] P. Weiner *Linear pattern matching algorithm,* Proc. 14 IEEE Symposium on Switching and Automata Theory, (1973), pp. 1-11.

[Ws-85] M. Weiser *CWSH: The Windowing Shell of the Maryland Window System,* Software – Practice and Experience, Vol. 15 (1985), pp. 515-519.

[WM-92] S. Wu and U. Manber *Fast text searching allowing errors,* Comm. of the ACM, Vol. 35 (1992), pp. 83-91.