Journal of Algorithms **42,** 231–254 (2002)
doi:10.1006/jagm.2002.1214, available online at http://www.idealibrary.com on IDEAL®

Sparse Dynamic Programming for Longest Common Subsequence from Fragments[1]

Brenda S. Baker

Bell Laboratories, Lucent Technologies, 600-700 Mountain Avenue, Murray Hill, New Jersey
E-mail: bsb@research.bell-labs.com

and

Raffaele Giancarlo[2]

Dipartimento di Matematica ed Applicazioni, Universitá di Palermo, Via Archirafi 34, 90123 Palermo, Italy
E-mail: raffaele@altair.math.unipa.it

Received September 16, 1999

Sparse Dynamic Programming has emerged as an essential tool for the design of efficient algorithms for optimization problems coming from such diverse areas as computer science, computational biology, and speech recognition. We provide a new sparse dynamic programming technique that extends the Hunt–Szymanski paradigm for the computation of the longest common subsequence (LCS) and apply it to solve the LCS from Fragments problem: given a pair of strings X and Y (of length n and m, respectively) and a set M of matching substrings of X and Y, find the longest common subsequence based only on the symbol correspondences induced by the substrings. This problem arises in an application to analysis of software systems. Our algorithm solves the problem in $O(|M| \log |M|)$ time using balanced trees, or $O(|M| \log \log \min(|M|, nm/|M|))$ time using Johnson's version of Flat Trees. These bounds apply for two cost measures. The algorithm can also be adapted to finding the usual LCS in $O((m+n) \log |\Sigma| + |M| \log |M|)$ time using balanced trees or $O((m+n) \log |\Sigma| + |M| \log \log \min(|M|, nm/|M|))$ time using Johnson's version of Flat Trees, where M is the set of maximal matches between substrings of X and

[1] An extended abstract of this paper appeared in "Proceedings of the 6th European Symposium on Algorithms," Venice, Italy, 1998.

[2] Work supported in part by grants from the Italian Ministry of Scientific Research, Project "Bioinformatica e Ricerca Genomica," and the Italian National Research Council. Part of this work was done while the author was visiting Bell Laboratories, Lucent Technologies.

0196-6774/02 $35.00

Y and Σ is the alphabet. These bounds improve on those of the original Hunt–Szymanski algorithm while retaining the overall approach.

1. INTRODUCTION

Sparse dynamic programming [5, 6] is a technique for the design of efficient algorithms mainly with applications to problems arising in sequence analysis. As in [5, 6], here we use the term sequence analysis in a very broad sense, to include problems that share many common aspects and come from such diverse areas as computer science, computational biology, and speech recognition [8, 12, 16]. A typical problem in sequence analysis is as follows: we are given two strings and we would like to find the "distance" between those strings under some cost assumptions. The technique can be concisely described as follows. We are given a set of dynamic programming (**DP** for short) recurrences to be computed using an associated **DP** matrix. The recurrences maximize (or minimize) some objective function. However, only a *sparse* set of the entries of the **DP** matrix really matters for the optimization of the objective function. The technique takes advantage of this sparsity to produce algorithms that have a running time dependent on the size of the sparse set rather than on the size of the **DP** matrix. To the best of our knowledge, the idea of using sparsity to speed up the computation of sequence analysis algorithms can be traced back to the algorithm of Hunt and Szymanski [10] for the computation of the longest common subsequence (LCS for short) of two strings. However, the first systematic study of sparsity in **DP** algorithms for sequence analysis is due to Eppstein *et al.* [5, 6]. Over the years, additional contributions have been provided by several authors (see for instance [7, 13]).

The main contribution of this paper is to provide new tools for Sparse **DP**. Namely, we generalize quite naturally the well-known Hunt–Szymanski [2, 9, 10] paradigm for the computation of the LCS of two strings. Our generalization, called LCS from Fragments, yields a new set of Sparse **DP** recurrences and we provide efficient algorithms computing them.

As a result, we obtain a new algorithm for the computation of the LCS of two strings that compares favorably with the known algorithms in the Hunt–Szymanski paradigm and, in its most basic version, is as simple to implement as Hunt–Szymanski.

In addition, our techniques solve an algorithmic problem needed for an application to finding duplication in software systems. In this case, a database of "matching" sections of code is obtained via a program such as dup [3, 4]. The notion of "matching" code sections may be either exact textual matches or parameterized matches in the sense of [3, 4], in which parameterized matches between code sections indicate textual similarity

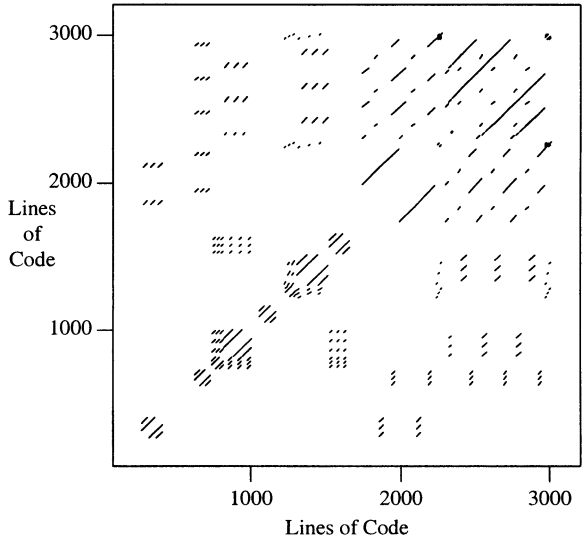

FIG. 1. A plot of parameterized duplication within a file of a production system.

except for a systematic renaming of parameters such as variable or function names. Typically, to avoid finding matches too short to be meaningful, dup is used to find maximal matches over a threshold length. An example of output from dup is shown in Fig. 1. This figure illustrates parameterized duplication within a file of a production system. Both axes represent lines of code within this file, and each diagonal line segment represents an instance in which two distinct sections of code within this file are a parameterized match of at least 10 lines. It would be convenient to have a graphical user interface that would enable a user to scroll simultaneously through two regions of code, in such a way that matching segments are aligned, to view the line-by-line differences in the source. While there is a natural line-by-line alignment within a single pair of matching sections of code, the problem is how to handle alignment upon scrolling forward or backward from one pair of matching segments to other matching segments. A solution to LCS from Fragments optimizes the alignment across multiple matching segments. A graphical user interface could allow a user to select a rectangle of Fig. 1 and scroll through the corresponding regions of code based on the alignment found by the LCS from Fragments algorithm.

We first present, at an informal level, the problem we have considered and then compare our results with the state of the art in Sparse **DP** and LCS algorithms.

Let X and Y be two strings of length n and m, respectively, over some alphabet Σ. Without loss of generality, assume that $|\Sigma| \leq n$. The LCS problem is to find the longest string $s_1 s_2 \cdots s_k$ such that $s_1, s_2, \ldots, s_k$ occur

in that order (possibly with other symbols intervening) in both X and Y. The dual of the LCS problem is to find the minimum edit distance between X and Y, where an insertion or deletion of a character has cost 1 (Levenshtein distance, see [12]). It is straightforward to transform a solution to the LCS problem into a solution to the minimum edit distance problem, and vice versa [2, 9, 14].

For the LCS from Fragments problem, we are given strings X and Y and a set M of pairs of equal-length substrings of X and Y that "match." The notion of "match" is somewhat general in the sense that the two substrings of X and Y are equal according to some definition of "equality," e.g., standard character equality or parameterized match [3, 4]. Each pair of substrings, called a "fragment," is specified in terms of the starting positions in X and Y and a length. The LCS from Fragments problem is to find a minimal-cost pair of subsequences of X and Y in which successive symbols correspond based on corresponding positions in a pair in M. For example, if $X = abcddaBC$ and $Y = ABCdabC$, and the fragments are $(1, 1, 4)$ (representing the first four characters of both strings) and $(5, 4, 4)$ (representing the last four characters of both strings), then (depending on the cost measure) subsequences *abcdaBC* of X and *ABCdabC* of Y might be a solution to the LCS from Fragments problem, since *abc* corresponds to *ABC* in the first fragment and *daBC* corresponds to *dabC* in the second fragment. We consider two cost measures based on edit distance.

The first cost measure is Levenshtein edit distance, for the case in which each pair in M consists of identical strings. The second cost measure is for the case where M represents parameterized matches, i.e., for each pair in M, the two strings exhibit a one-to-one correspondence as defined in [3, 4]. In this case, we treat the inclusion of each parameterized match in the LCS as a new "edit" operation and charge a cost of one for each insertion, deletion, or segment (of any length) of a fragment. Extension of this cost measure to deal with the full-fledged application is discussed in Section 5.

For both cost measures, LCS from Fragments can be computed in $O(|M| \log |M|)$ time. With sophisticated data structures, such as Johnson's version of Flat Trees [11], that bound reduces to $O(|M| \log \log \min(|M|, nm/|M|))$. The algorithm for the second cost measure is more complex than that for the Levenshtein cost measure.

If the set M consists of all pairs of maximal equal substrings of X and Y, and the Levenshtein cost measure is used, the solution of the LCS from Fragments problem is the usual LCS. This generalizes the Hunt–Szymanski paradigm, where the basis for the algorithm is the set of pairs of positions with the same symbols in X and Y. The LCS can be computed via LCS from Fragments in $O((m + n) \log |\Sigma| + |M| \log |M|)$ time, including the time for the computation of M. Moreover, using Johnson's version of Flat Trees [11], that bound reduces to $O((m + n) \log |\Sigma| + |M| \log \log \min(|M|, nm/|M|))$.

Since $|M| \leq nm$, our algorithm is never worse than the standard $O(nm)$ dynamic program for LCS. The time bounds that we obtain compare favorably with the best available algorithms for the computation of the LCS [2].

The next section formalizes our definitions and presents two preliminary lemmas. Section 3 describes the results for the Levenshtein cost measure, and discusses how our model differs from previous work on sparse dynamic programming. In particular, the dynamic programming recurrences we present in that section are related to those discussed in [6]. However, the differences between those families of recurrences is subtle and the algorithms in [6] do not seem to readily accommodate for those differences. A more detailed and technical discussion is given in Section 3.2, once we have presented the new recurrences. Section 4 presents the results for the second cost measure. Basically, we exhibit dynamic programming recurrences that allow to compute the LCS for the second cost measure via a reduction to a linear number of subproblems, each of which can then be handled with the algorithms presented in [5]. The last section contains a discussion and concluding remarks.

2. DEFINITIONS AND PRELIMINARY LEMMAS

Finding the LCS of X and Y is equivalent to finding the Levenshtein edit distance between the two strings [12], where the "edit operations" are insertion and deletion. Following Myers [14], we phrase the LCS problem as the computation of a shortest path in the edit graph for X and Y, defined as follows. It is a directed grid graph (see Fig. 2) with vertices (i, j), where $0 \leq i \leq n$ and $0 \leq j \leq m$. We refer to the vertices also as *points*. There is a vertical edge from each nonbottom point to its neighbor below. There is a horizontal edge from each non-right-most point to its right neighbor. Finally, if $X[i] = Y[j]$, there is a diagonal edge from $(i-1, j-1)$ to (i, j). Assume that each nondiagonal edge has weight 1 and the remaining edges weight 0. Then, the Levenshtein edit distance is given by the minimum cost of any path from $(0, 0)$ to (n, m). We assume the reader to be familiar with the notion of edit script corresponding to the min-cost path and how to recover an LCS from an edit script [2, 9, 14].

Our LCS from Fragments problem also corresponds naturally to an edit graph. The vertices and the horizontal and vertical edges are as before, but the diagonal edges correspond to a given set of fragments. Each fragment, formally described as a triple (i, j, k), represents a sequence of diagonal edges from $(i-1, j-1)$ (the *start* point) to $(i+k-1, j+k-1)$ (the *end* point). For a fragment f, the start and end points of f are denoted by $start(f)$ and $end(f)$, respectively. In the example of Fig. 3, the fragments are the sequences of at least 2 diagonal edges of Fig. 2. The LCS from

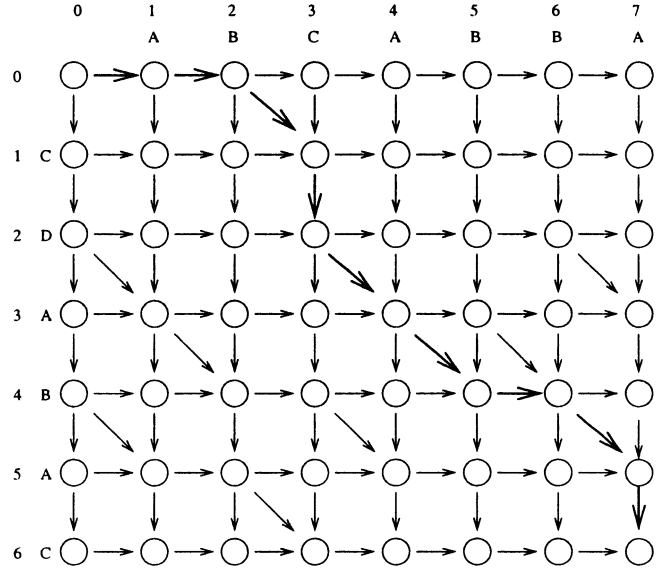

FIG. 2. An edit graph for the strings $X = CDABAC$ and $Y = ABCABBA$. It naturally corresponds to a **DP** matrix. The bold path from $(0, 0)$ to $(6, 7)$ gives an edit script from which we can recover the LCS between X and Y.

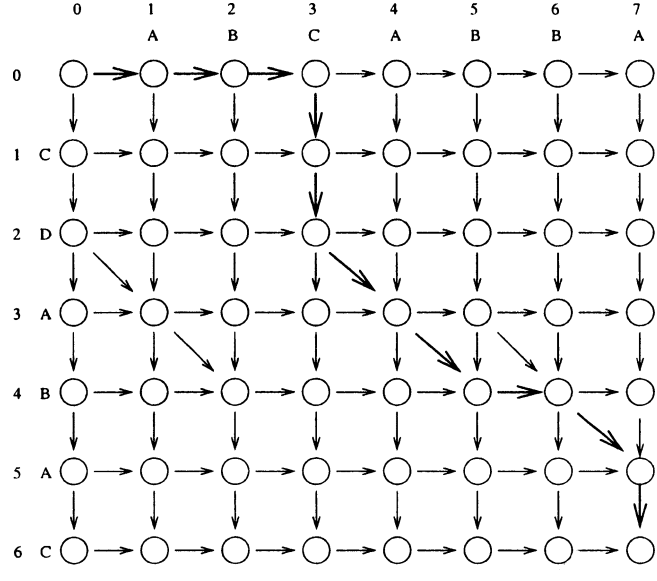

FIG. 3. An LCS from Fragments edit graph for the same strings as in Figure 2, where the fragments are the sequences of at least two diagonal edges of Figure 2. The bold path from $(0, 0)$ to $(6, 7)$ corresponds to a minimum-cost path under the Levenshtein edit distance.

Fragments problem is equivalent to finding a minimum-cost path in the edit graph from $(0, 0)$ to (n, m). We consider two cost measures. As before, in the Levenshtein cost measure, each diagonal edge has weight 0 and each nondiagonal edge has weight 1. (However, all of our results generalize to the case of weighted Levenshtein edit distance. To simplify the presentation of our ideas, we restrict attention to the unweighted case.) To suit the software application described in the Introduction, we also consider a cost measure in which a cost of 1 is incurred for traversing a horizontal edge, a vertical edge, or a segment (of any nonzero length) of a fragment.

For ease of presentation, we assume that the sequences of edges corresponding to different fragments are disjoint and do not even touch. However, the algorithms can be easily modified to handle cases in which fragments may not be disjoint, with the same time and space bounds; nondisjoint fragments actually arise commonly in the application.

For a point p, define $x(p)$ and $y(p)$ to be the x- and y-coordinates of p, respectively. We also refer to $x(p)$ as the *row* of p and $y(p)$ as the *column* of p. Define the diagonal number of f to be $d(f) = y(start(f)) - x(start(f))$.

It will be helpful to show that we need only consider paths of restricted form. To this end, we say that a segment of a fragment f is a *prefix* of f if it includes $start(f)$.

LEMMA 2.1. *Consider a cost measure in which horizontal and vertical edges have cost* 1 *and traversing a fragment segment of any length has a cost of* 0 *or* 1. *For any point p in the edit graph, there is a min-cost path from* $(0, 0)$ *to p such that every fragment segment traversed is a prefix.*

Proof. Suppose P is a min-cost path from $(0, 0)$ to p that does not satisfy the lemma. We construct another min-cost path P' from $(0, 0)$ to p in which fewer nonprefix segments are used. This argument is applied inductively to prove the lemma.

Suppose S is the last nonprefix segment used in P. Let f be the fragment containing S. Consider the rectangle R with opposite corners $(0, 0)$ and $start(f)$. P starts out at $(0, 0)$, initially is confined to R or its boundaries, and then leaves R, either through the bottom side of R or through the right side of R.

Suppose P leaves R through the bottom of R, i.e., through a point q to the left of $start(f)$, where $y(q) < y(start(f))$ and $x(q) = x(start(f))$. P has to use at least cost $d(start(f)) - d(q)$ to get from q to the start of S, since it must cross that many diagonals via horizontal edges. The new path P' follows P to q, goes right to $start(f)$, along f to the end of S, and from there follows P to p. Obviously, the cost of P' is no higher than that of P, and P' has fewer nonprefix segments than P.

The construction when P leaves R through the right edge of R is analogous. ∎

Between successive fragment segments in a min-cost path that satisfies Lemma 2.1, either there are only horizontal edges, only vertical edges, or both. In the last case, we can make a stronger statement about the preceding fragment segment.

LEMMA 2.2. *Consider a cost measure in which horizontal and vertical edges have cost* 1 *and traversing a fragment segment of any length has a cost of* 0 *or* 1, *and consider a min-cost path that satisfies Lemma* 2.1, *i.e., every fragment segment traversed is a prefix. If there are at least one horizontal edge and at least one vertical edge between two successive fragment segments in the path, the earlier fragment segment is the entire fragment.*

Proof. If the lemma fails, the path cannot be a min-cost path because a lower-cost path is obtained by following the earlier fragment for an additional diagonal edge and reducing the number of horizontal and vertical edges. ∎

3. LEVENSHTEIN COST MEASURE

In this section, we consider a cost measure corresponding to Levenshtein edit distance: diagonal edges are free, while insertions and deletions of characters have a cost of 1 each. For any point p, define $mincost_0(p)$ to be the minimum cost of any path from $(0, 0)$ to p under this cost measure. Since diagonal edges are free, proof of Lemma 2.1 yields the following corollary.

COROLLARY 3.1. *For any fragment f and any point p on f, $mincost_0(p) = mincost_0(start(f))$.*

By Corollary 3.1, it is reasonable also to define $mincost_0(f) = mincost_0(start(f))$.

3.1. *Sparse Dynamic Programming Recurrences*

We say a fragment f' is *left of* $start(f)$ if some point of f' besides $start(f')$ is to the left of $start(f)$ on a horizontal line through $start(f)$, or $start(f)$ lies on f' and $x(start(f')) < x(start(f))$. (In the latter case, f and f' are in the same diagonal and overlap.) A fragment f' is *above* $start(f)$ if some point of f' besides $start(f')$ is strictly above $start(f)$ on a vertical line through $start(f)$.

Define $visl(f)$ to be the first fragment to the left of $start(f)$ if such exists, and undefined otherwise. Define $visa(f)$ to be the first fragment above $start(f)$ if such exists, and undefined otherwise.

We say that fragment f precedes fragment f' if $x(end(f)) < x(start(f'))$ and $y(end(f)) < y(start(f'))$, i.e., if the end point of f is strictly inside the rectangle with opposite corners $(0, 0)$ and $start(f')$.

Suppose that fragment f precedes fragment f'. The shortest path from $end(f)$ to $start(f')$ with no diagonal edges has cost $x(start(f')) - x(end(f)) + y(start(f')) - y(end(f))$, and the minimum cost of any path from $(0, 0)$ to $start(f')$ through f is that value plus $mincost_0(f)$. It will be helpful to separate out the part of this cost that depends on f by the definition $Z(f) = mincost_0(f) - x(end(f)) - y(end(f))$. Note that $Z(f) \leq 0$ since $mincost_0(f) \leq x(start(f)) + y(start(f))$. The following Lemma states that we can compute LCS from fragments by considering only end-points of some fragments rather than all points in the dynamic programming matrix. Moreover, it also gives the appropriate recurrence relations that we need to compute.

LEMMA 3.1. *For a fragment f, $mincost_0(f)$ is the minimum of $x(start(f)) + y(start(f))$ and any of c_p, c_l, and c_a that are defined according to the following*:

1. *If at least one fragment precedes f, $c_p = x(start(f)) + y(start(f)) + \min\{Z(f') : f' \text{ precedes } f\}$.*
2. *If $visl(f)$ is defined, $c_l = mincost_0(visl(f)) + d(f) - d(visl(f))$.*
3. *If $visa(f)$ is defined, $c_a = mincost_0(visa(f)) + d(visa(f)) - d(f)$.*

Proof. Consider a min-cost path P to f. By Lemmas 2.1 and 2.2, we may assume that any fragment segments it traverses are prefixes, and that if such a fragment prefix is immediately followed by a sequence of horizontal and vertical edges with at least one of each, then it is a complete fragment.

Either P traverses no fragment prefixes before $start(f)$ and the cost is $x(start(f)) + y(start(f))$, or it traverses at least one prefix. In the latter case, the path from the last such prefix S to $start(f)$ has one of three forms: (a) zero or more horizontal edges, (b) one or more vertical edges, or (c) both horizontal and vertical edges, with at least one of each.

If there were both horizontal and vertical edges in the path from S to $start(f)$, then as noted above, S is an entire fragment f'. Moreover, the path to $start(f')$ must be a min-cost path. The cost to reach $start(f)$ is $x(start(f)) + y(start(f)) + Z(f')$. In addition, if there were another f'' that precedes f with $Z(f'') < Z(f')$, then P would not be optimal. Therefore, the cost of P is given by (1).

If there were only horizontal (vertical, respectively) edges in the path from S to $start(f)$, the path had to cross $visl(f)$ ($visa(f)$, respectively), and

the path must be a min-cost path to that crossing. By Corollary 3.1, the min-cost to $start(visl(f))$ is the same as the min-cost to the crossing point. The minimum additional cost to reach $start(f)$ is the difference in the diagonal numbers, as specified by (2) and (3).

We have shown that $mincost_0(f)$ must be one of the given formulas. In addition, we note that wherever $mincost_0$ is used in the formulas, there must be a corresponding path to the appropriate point, and therefore where (1)–(3) are defined, there must be paths to $start(f)$ with the corresponding costs, so $mincost_0(f)$ is the minimum of the given formulas. ∎

3.2. *Relation with Wilbur–Lipman Local Fragment Alignment Problem and with RNA Secondary Structure Prediction*

LCS from Fragments is reminiscent of the Wilbur–Lipman Local Fragment Alignment problem [17]. Indeed, also in that case, we are given a set of fragments from which we need to compute a "local alignment." Unfortunately, there are a few subtle but major differences between the two problems. Here we may need to choose segments of two overlapping fragments as part of the LCS (see Fig. 3) while in Wilbur–Lipman it is meaningful only to use whole fragments and use of partial fragments is not allowed. For the Levenshtein cost measure and its standard "edit operations" involved, the LCS privileges insertions and deletions (they are cheap compared to substitutions) while Wilbur–Lipman privileges substitutions over insertions and deletions. Technically, those differences yield to different sets of **DP** recurrences that one needs to compute (Lemma 3.1 gives the recurrences for LCS from Fragments while the recurrences associated with Wilbur–Lipman are given in [5, 6]).

As for the RNA Secondary Structure prediction, we mention that Recurrence (1) in Lemma 3.1 is essentially the computational bottleneck for the computation of the dynamic programming recurrences associated with RNA Secondary Structure prediction (see Recurrence (4) in [5]). However, here we have to consider "overlapping fragments" via Recurrences (2) and (3) in Lemma 3.1. Therefore, the system of dynamic programming recurrences we need to consider here is different than the system associated with the RNA Secondary Structure Prediction (see [5]).

As we will see, some of the techniques presented in [5] can be used also for our problem, but they do not seem to be extendable to deal with the case of "overlapping fragments."

3.3. *The Algorithm*

Based on Lemma 3.1, we now develop an algorithm. It uses a sweepline approach where successive rows are processed, and within rows, points are processed from left to right. Lexicographic sorting of (x, y)-values is

needed. The algorithm consists of two main phases, one in which we compute visibility information, i.e., $visl(f)$ and $visa(f)$ for each fragment f, and the other in which we compute Recurrences (1)–(3) in Lemma 3.1. Not all the rows and columns need to contain a start point or end point, and we generally wish to skip empty rows and columns for efficiency. For any x (y, respectively), let $C(x)$ ($R(y)$, respectively) be the i for which x is in the ith nonempty column (row, respectively). These values can be calculated in the same time bounds as the lexicographic sorting. From now on, we assume that our algorithm processes only nonempty rows and columns.

For the lexicographic sorting and both phases, we assume the existence of a data structure of type D that stores integers j in some range $[0, u]$ and supports the following operations: (1) insert, (2) delete, (3) member, (4) min, (5) successor: given j, the next larger value than j in D, (6) max: given j, find the max value less than j in D. If d elements are stored, D could be implemented via balanced trees [1] with $O(\log d)$ time per operation or via the van Emde Boas Flat Trees [15] with $O(\log\log u)$ time per operation. If we use Johnson's version of Flat Trees [11], the time for all of those operations becomes $O(\log\log G)$, where G is the length of the gap between the nearest integers in the structure below and above the priority of the item being inserted, deleted, or searched for. Moreover, the following Lemma, implicit in Johnson's paper and explicitly given in [5], will be of use:

LEMMA 3.2. *A homogeneous sequence of $k \leq u$ operations* (*i.e., all insertions, all deletions or all member*) *on Johnson's data structure requires $O(k \log\log u/k)$ time.*

With the mentioned data structures, lexicographic sorting of (x, y)-values can be done in $O(d \log d)$ time or $O(d + u)$ time for d elements in the range $[0, u]$. In our case $u \leq n + m$ and $d \leq |M|$. By using D, implemented via Johnson's version of Flat Trees as just discussed, sorting can be accomplished via a sequence of insertions, a min, and a sequence of successor operations. Due to the implementation of Johnson's data structure (see [15]), this sequence of operations reduces to an homogeneous sequence of M insertions and therefore (by Lemma 3.2) the time is $O(|M| \log\log \min(|M|, nm/|M|))$.

Visibility computation. We now briefly outline how to compute $visl(f)$ and $visa(f)$ for each fragment f via a sweepline algorithm. We describe the computation of $visl(f)$; that for $visa(f)$ is similar. For $visl(f)$, the sweepline algorithm sweeps along successive rows. Assume that we have reached row i. We keep all fragments crossing row i sorted by diagonal number in a data structure V. For each fragment f such that $x(start(f)) = i$, we record the fragment f' to the left of $start(f)$ in the sorted list of fragments; in

this case, $visl(f) = f'$. Then, for each fragment f with $x(start(f)) = i$, we insert f into V. Finally, we remove fragments $\hat{f}$ such that $y(end(\hat{f})) = i$.

If the data structure V is implemented as a balanced search tree, the total time for this computation is $O(M \log M)$. If van emde Boas Flat Trees are used, the total time is $O(M \log \log M)$ (we can "renumber" the diagonals so that the items stored in the data structure are in the range $[0, 2(n + m)]$). Even better, notice that we perform on the data structure three homogeneous sequences of operations per row: first a sequence of max operations (to identify visibility information from the start point of a fragment about to be inserted in the data structure), then a sequence of insertions, and finally a sequence of deletions. In that case, we can use Johnson's version of Flat Trees [11] to show that the sweep can be implemented to take $O(|M| \log \log \min(|M|, nm/|M|))$ time. The analysis uses Lemma 3.2 and the convexity of the log log function. The details are as in [5] (Lemma 1).

The main algorithm. Again, we use a sweepline approach of processing successive rows. It follows the same paradigm as the Hunt–Szymanski LCS algorithm [10] and the computation of the *RNA* secondary structure (with linear cost functions) [5].

We use another data structure B of type D, but this time B stores column numbers (and a fragment associated with each one). The values stored in B will represent the columns at which the minimum value of $Z(f)$ decreases compared to any columns to the left, i.e., the columns containing an end point of a fragment f for which $Z(f)$ is smaller than $Z(f')$ for any f' whose end point has already been processed and which is in a column to the left. Notice that, once we fix a row, D gives a partition of that row in terms of columns.

Within a row, first process any start points in the row from left to right. For each start point of a fragment, compute $mincost_0$ using Lemma 3.1. Note that when the start point of a fragment f is computed, $mincost_0$ has already been computed for each fragment that precedes f and each fragment that is $visa(f)$ or $visl(f)$. To find the minimum value of $Z(f')$ over all predecessors f' of f, the data structure B is used. The minimum relevant value for $Z(f')$ is obtained from B by using the max operation to find the $\max j < y(start(f))$ in B; the fragment f' associated with that j is one for which $Z(f')$ is the minimum (based on endpoints processed so far) over all columns to the left of the column containing $start(f)$, and in fact this value of $Z(f')$ is the minimum value over all predecessors of f.

After any start points for a row have been processed, process the end points. When an end point of a fragment f is processed, B is updated as necessary if $Z(f)$ represents a new minimum value at the column $y(end(f))$; successor and deletion operations may be needed to find and remove any values that have been superseded by the new minimum value.

Correctness and time analysis. Given M precomputed fragments, the above algorithm can be implemented in $O(|M| \log |M|)$ time via balanced trees, or $O(n + |M| \log\log |M|)$ time if van Emde Boas data structures are used. Moreover, using the same ideas as in Eppstein *et al.* [5], we can "reschedule" the operations on B so that, for each processed row, we perform three sequences of homogeneous operations of the type insert, delete, member. Using this fact, again Lemma 3.2 and the convexity of the log log function, we can show that this phase of the algorithm takes $O(|M| \log\log \min(|M|, nm/|M|))$ time. Again the details are as in Eppstein *et al.* [5] (Lemma 1).

We note that for each fragment f, a pointer may be kept to the fragment from which a min-cost path arrived at $start(f)$, and hence both the LCS and a minimal edit script are easily recovered in $O(|M|)$ space.

THEOREM 3.1. *Suppose $X[1:n]$ and $Y[1:m]$ are strings, and a set M of fragments relating substrings of X and Y is given. One can compute the LCS from Fragments in $O(|M| \log |M|)$ time and $O(|M|)$ space using standard balanced search tree schemes. When one uses Johnson's data structure, the time reduces to $O(|M| \log\log \min(|M|, nm/|M|))$.*

Proof. Correctness follows from Lemma 3.1 and the time analysis from the discussion preceding the statement of the theorem. ∎

Specialization to standard longest common subsequence. Two substrings $X[i : i + k - 1]$ and $Y[j : j + k - 1]$ are a *maximal match* if and only if they are equal and the equality cannot be extended to the right and to the left. A maximal match between two substrings is conveniently represented by a triple (i, j, k), corresponding to a sequence of diagonal edges in the edit graph starting at $(i - 1, j - 1)$ and ending at $(i + k - 1, j + k - 1)$.

When M is the set of maximal matches between X and Y, a solution to the LCS from Fragments problem also gives a solution to the usual LCS problem. Using techniques in [3, 4], we can compute the set of maximal matches in $O((m + n) \log |\Sigma| + |M|)$ time and $O(m + n + |M|)$ space. Thus, we obtain the following corollary.

COROLLARY 3.2. *Given two strings $X[1:n]$ and $Y[1:m]$ one can compute the LCS of those two strings in $O((m + n) \log |\Sigma| + |M| \log |M|)$ time and $O(m + n + |M|)$ space using standard balanced search tree schemes. When one uses Johnson's data structure, the time reduces to $O((m + n) \log |\Sigma| + |M| \log\log \min(|M|, nm/|M|))$.*

4. COST MEASURE WITH UNIT COST FOR FRAGMENT SEGMENTS

In this section, we consider a cost measure in which any segment of a fragment can be traversed at a cost of 1, regardless of length. Each insertion and deletion still incurs a cost of 1. For any point p, define $mincost_1(p)$ to be the minimum cost of a path from $(0, 0)$ to p under this cost measure.

The solution of the previous section breaks down under the new cost measure. The problem is that a minimum-cost path to a point on a fragment f may not be able to traverse f because of the cost of traversing f itself. The failure has ramifications because the proof of Lemma 3.1 assumed that the cost of a path through a point on $visl(f)$ could be computed from $mincost_0(visl(f))$ without concern about whether $visl(f)$ itself was used in the path, and similarly for $visa(f)$. Fortunately, a path from $(0, 0)$ to a point on a fragment f can save at most 1 by not passing through $start(f)$. Based on this observation, we can derive a dynamic programming recurrence which can then be computed via a linear number of subproblems that are essentially the same as those considered in [5].

4.1. *A Reduction*

PROPOSITION 4.1. *For a point p on a fragment f, if there is a path of cost c from $(0, 0)$ to p, there is a path of cost at most c from $(0, 0)$ to $start(f)$ and a path of cost at most $c + 1$ from $(0, 0)$ to p via $start(f)$ and a prefix of f.*

Proof. Suppose that P is a min-cost path to p and its cost is c. By Lemma 2.1, we may assume that every fragment segment P traverses is a prefix. If p passes through $start(f)$, we are done since the cost of traversing a prefix of f is 1. So assume P does not pass through $start(f)$. In this case, it cannot traverse any segment of f, and path P arrives at p either from the left or from above.

Suppose it arrives from the left. Since it is a min-cost path to p, it cannot pass through a point of f closer to $start(f)$ than p, or it would incur a cost of at least 2 in horizontal and vertical edges to reach p from that point, while the cost for traversing part of f is only 1. Consequently, it must pass through some point to the left of $start(f)$. Let q be the right-most such point. Consider a new path that follows P as far as q and then goes directly right to $start(f)$ and along f to p. Since P had to traverse just as many diagonals to get from q to p, at a cost of 1 per diagonal, the cost of reaching $start(f)$ via the new path is no higher than c. The additional cost of traversing part of f to reach p is 1.

The case in which P arrives at p from above is analogous. ∎

For any fragment f, we define $mincost_1(f) = 1 + mincost_1(start(f))$. By Lemma 2.1, this is the minimum cost of reaching any point on f (other than $start(f)$) via a path that uses a nonempty prefix of f.

We wish to consider the minimum cost of reaching $start(f)$ from a previous fragment as we did before, but will take into account the complications of the new cost measure. Define $visl(f)$ and $visa(f)$ as before. Corresponding to Z from before, we define $Z_p(f) = mincost_1(f) - x(end(f)) - y(end(f))$. Since the cost of a sequence of horizontal edges or a sequence of vertical edges is the change in diagonal number, it will be convenient to define $Z_l(f) = mincost_1(f) - d(f)$ and $Z_a(f) = mincost_1(f) + d(f)$ to separate out costs dependent only on f. The lemma that corresponds to Lemma 3.1 is the following.

LEMMA 4.1. *For a fragment f, $mincost_1(start(f))$ is the minimum of $x(start(f)) + y(start(f))$ and any of c_p, c_l, and c_a that are defined via the following*:

1. *If at least one fragment precedes f, then $c_p = x(start(f)) + y(start(f)) + \min\{Z_p(f') : f' \text{ precedes } f\}$.*

2. *If $visl(f)$ is defined, then c_l is defined as follows*: *if there exists at least one fragment f' to the left of $start(f)$ with $Z_l(f') = Z_l(visl(f)) - 1$ then $c_l = Z_l(visl(f)) - 1 + d(f)$ else $c_l = Z_l(visl(f)) + d(f)$.*

3. *If $visa(f)$ is defined then c_a is defined as follows*: *if there exists at least one fragment f' above $start(f)$ with $Z_a(f') = Z_a(visa(f)) - 1$ then $c_a = Z_a(visa(f)) - 1 - d(f)$ else $c_a = Z_a(visa(f)) - d(f)$.*

Proof. By Lemma 2.1, we may assume that a min-cost path to $start(f)$ traverses only prefixes of fragments, although it may cross a single point of a fragment. By Lemma 2.2, we may assume that if its traversal of a fragment prefix is followed by a sequence of horizontal and vertical edges, with at least one of each, that prefix must be a complete fragment.

If the path traverses no fragment prefixes, the cost is $x(start(f)) + y(start(f))$. If it traverses at least one prefix, we consider how the path arrives at $start(f)$ from the last prefix traversed. Case 1 computes the minimum cost of paths from $(0, 0)$ to $start(f)$ that arrive at $start(f)$ via a sequence of horizontal and vertical edges after the last fragment prefix traversed, with at least one of each. Case 2 computes the minimum cost of paths that arrive at $start(f)$ via zero or more horizontal edges from the last fragment prefix traversed. Case 3 computes the minimum cost of paths that arrive at $start(f)$ via one or more vertical edges from the last fragment prefix traversed. The proof of case 1 is similar to that of case (1) of Lemma 3.1. However, the proofs of cases (2) and (3) are more complicated than the corresponding cases of Lemma 3.1. We give the proof for case (2); case (3) is analogous.

Case 2 covers two types of paths that reach $start(f)$ via zero or more horizontal edges after traversing a prefix (of more than one point) of the previous fragment. In type (a), the path traverses a prefix of $visl(f)$ to the point p on f that is directly to the left of $start(f)$. In type (b), it passes through p but does not traverse a prefix of $visl(f)$; the last segment it traverses is a prefix of a fragment to the left of p. (In both types, "left" includes the situation in which p coincides with $start(f)$ and there are zero horizontal edges.) Consider a min-cost path of type (a) and a min-cost path of type (b) for which f' is the last fragment for which a prefix is traversed. The cost of reaching $start(f)$ through the min-cost path of type (a) is $Z_l(visl(f)) + d(f)$, while the cost of reaching $start(f)$ through the min-cost path of type (b) is $Z_l(f') + d(f)$. We will show that $Z_l(f') \geq Z_l(visl(f)) - 1$. Consequently, in Case 2, along with paths of type (a), we need consider only paths of type (b) for which $Z_l(f') = Z_l(f) - 1$, and we derive the formula in Case 2.

For the min-cost path of type (b), let c be its cost for reaching p. For this path, the cost of reaching $start(f)$ is $Z_l(f') + d(f) = c + d(f) - d(visl(f))$. Consequently, $Z_l(f') = c - d(visl(f))$. By Proposition 4.1, the cost of reaching p by a min-cost path of type (a) is at most $c + 1$, and the cost of reaching $start(f)$ via this path is $Z_l(visl(f)) + d(f) \leq c + 1 + d(f) - d(visl(f))$. Thus, $Z_l(visl(f)) \leq c + 1 - d(visl(f)) = Z_l(f') + 1$, and $Z_l(f') \geq Z_l(visl(f)) - 1$, as we undertook to prove.

We have shown that $mincost_1(start(f))$ has to be one of the given formulas. In addition, we note that whenever any one of c_p, c_l, and c_a is defined, there must be a path to $start(f)$ with the corresponding cost, so $mincost_1(start(f))$ is the minimum of any of these that are defined. ∎

Notice that (1) in Lemma 4.1 is the same as (1) in Lemma 3.1 and can be handled in essentially the same way. Recurrence (2) conveniently requires grouping fragments according to their Z_l values, since for a given fragment f, we are free to choose any fragment with the value of Z_l specified by the Lemma. There can be $O(n + m)$ such groups, since the range of values of Z_l is $[-(n + m), 0]$. Each of those groups of fragments must be organized so that we can properly identify a fragment to the left of $start(f)$, when needed. The same type of observation holds for (3).

However, in applying Lemma 4.1, there is a technical difficulty. If the dynamic programming matrix is swept by rows, then dynamically maintaining fragments in order to answer (1) and (2) is "easy" since the "visibility" information is nicely partitioned in columns (for (1)) and diagonal numbers (for (2)) that are invariant with respect to row changes. However, dynamically maintaining fragments needed for (3) is not as simple since we now have "visibility" limited by columns and simultaneously by diagonal numbers. In a sense, groups of fragments in (2) and (3) are "orthogonal."

Such an "orthogonality" is completely analogous to one encountered in the computation of the Wilbur–Lipman dynamic programming recurrence [5], except that for Wilbur–Lipman one can divide the fragments in only two orthogonal groups. The algorithm proposed here can be outlined as follows (we give only a brief overview for the computation of (1) and (2) in Lemma 4.1 and provide a more detailed presentation for (3) in Section 4.2):

ALGORITHM PLCS

(a) **Computation of Z_p.** First, compute $visl(f)$ and $visa(f)$ as in Section 3.3. Again, the main algorithm sweeps across successive rows and has a data structure B that keeps track of the columns in which the minimum value of $Z_p(f)$ decreases compared to columns for the left. This data structure B is used to compute (1) in Lemma 4.1 in essentially the same way in which it is used in Section 3.3 to compute (1) in Lemma 3.1.

(b) **Computation of Z_l.** During the life of the entire algorithm, a separate data structure of type D is kept for each distinct value of Z_l created by the algorithm. One possibility is to keep a pointer to each of these data structures in an array A_l that would use $O(n+m)$ space since the range of values of Z_l is $[-(n+m), 0]$. Alternatively, another instance of D can be used to store the distinct values of Z_l that occur, together with the associated pointers. In this case, the total space used is $O(|M|)$. For a particular value v of $Z_l(f')$, consider the set S of fragments f' with $Z_l(f') = v$. We need to store information about S in a way that enables answering queries about whether one of the fragments in S is left of $start(f)$ for a particular fragment f, i.e., to ask whether there exists an f' in S with $x(start(f')) < x(start(f)) \leq x(end(f'))$ and $d(f) > d(f')$. We need only keep track of the diagonal numbers of all fragments that have this value of Z_l and cross the current row, query for the minimum diagonal, and compare that diagonal number of $d(f)$. This is enough because by (2) of Lemma 4.1 we can pick any fragment f' with the given value to the left of $start(f)$. The data structure must be built dynamically as fragments are processed and the values of Z_l become available. We omit the details.

(c) **Computation of Z_a.** As for the computation of Z_a, we keep each group corresponding to a value of Z_a separate. For a particular value of Z_a, dynamic management of the fragments in it is essentially as in Algorithm `Left Influence` in Section 4 in [5]. In the next subsection, for convenience of the reader, we report a high-level description of that algorithm as well as the minor changes needed to use it here.

4.2. *Computation of Z_a*

Fix a value of Z_a, say v. We show how to dynamically maintain the set of fragments that, during the life of Algorithm PLCS, have a value of $Z_a = v$. In what follows *MDP* refers to the dynamic programming matrix swept by

Algorithm PLCS, while MDP_v refers to the matrix restricted only to fragments with value $Z_a = v$. The *left influence* of f is the region of MDP_v below the fragment and limited by the columns touching $start(f)$ and $end(f)$. Notice that the left influence of a fragment includes all possible start points of fragments that have f above them.

In order to describe Algorithm `Left Influence` in [5] and then show how it can be used to compute (2) for $Z_a = v$, it is best to extend each fragment to "touch" the bottom boundary of MDP_v (see Fig. 4). That has the effect of extending the left influence of a fragment to be a triangular region of the matrix, fitting the framework in [5]. From now on, a fragment is represented by a single point: $start(f)$. Given two points x and y, notice that their left influences may intersect. We need a criterion to establish which point is "better than the other" in the common part of their left influence. Such a criterion is application dependent and the only two important things are: (a) it can be applied in constant time to any point z of the intersection; (b) if x is better than y at z, then it will be better in any other point of the intersection. We use the following rule. Consider

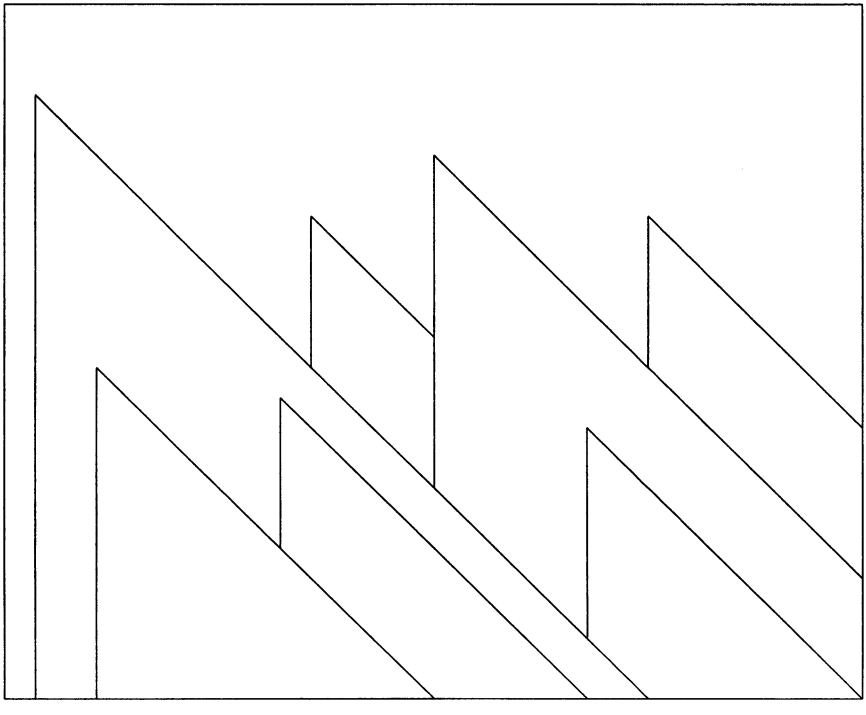

FIG. 4. The matrix MDP_v and the left influences of points in it. Part of a left influence may be hidden by another.

two points x and y with nonempty intersection of left influences: x is better than y whenever (a) the left influence of x completely contains that of y or (b) x is to the left of y, i.e., the diagonal of the left influence of x crosses the column of the left influence of y.

ALGORITHM `Left Influence`

(a) **Input Description.** We are given two sequences of batches $B_1, \ldots, B_k$ and $A_1, \ldots, A_g$. We refer to the B batches as *partition batches* and to the A batches as *query batches*. Batch B_i, $1 \leq i \leq k$, is composed of fragment starting points on the same row of MDP_v. Row numbers associated to the batches are increasing with batch numbers. Query batches satisfy the same conditions. A query batch A_i and a partition batch B_j can have equal row numbers. In that case, the query batch "precedes" the partition batch. So, the two types of batches can be organized in a unique linear ordering, which we refer to as the sequence of batches. All the batches become available on-line and must be processed in $k + g$ steps.

(b) **Processing of Query Batches.** Assume that we have reached step j and that the jth batch of the sequence is query batch A. For each point z in A return the point x in the partition batches preceding A that is best for z. That is, z is in the left influence of x and, for any other point y having it in its left influence and being in one of the partition batches preceding A, x is better than y at z.

(c) **Processing of Partition Batches.** Assume that we have reached step j and that the jth batch of the sequence is a partition batch B. Integrate B with the previous partition batches so that (b) can be properly answered for future query batches.

(d) **Preparing for the Next Batch.** Intersection points of regions are processed as necessary to update the partition for the next row containing a query batch or partition batch. The partition is updated in accordance with the criterion established above for which region is "better" than the other.

Now, if the fragments in our LCS problem were really touching the bottom boundary of the dynamic programming matrix, we could use Algorithm `Left Influence` to answer queries in (2) of Lemma 4.1 for $Z_a = v$. Indeed, as Algorithm `PLCS` sweeps *MDP* by rows, batches of points with $Z_a = v$ become available and they are interleaved with batches of points for which `PLCS` needs to query in order to find out the fragments needed in (2) of Lemma 4.1.

However, the relevant left influence of a fragment in Algorithm `PLCS` is only under the fragment, and not the entire triangular region as above. We show how to accommodate this change by hiding the irrelevant part of the left influence, and briefly discuss the "inner working" of Algorithm `Left-Influence` using the notion of left influence needed here.

We consider two types of triangular regions: *real* and *dummy*. Given a fragment f, a real region is the left influence originating at $start(f)$ while a dummy region is the left influence originating at $end(f)$. A dummy region is always worse than a real region, in their intersection, except when compared to its corresponding real region. The rule for comparing two regions is modified as follows. Let f and f' be two fragments. The real region of f is better than the real region of f' in their intersection if and only if f has its endpoint on a column of higher number than the one on which f' has its endpoint. The dummy region of f is better than the dummy region of f' in their intersection if and only if the real region of f is better than the real region of f' in their intersection.

We modify Algorithm `PLCS` to take both real and dummy regions into account in generating and handling query batches and partition batches. Partition batches will include the starting points of dummy regions as well as starting points of real regions.

We now give some details on the "internal working" of Algorithm `Left Influence`, using the notions of real and dummy regions. We assume that we need to process query batch A, corresponding to row r. We also assume that we have the partition of row r into regions (see Fig. 5) and that the next batch to process after A is a partition batch B (see Fig. 6), corresponding to the same row as A. We also have a data structure containing potential intersection points of successive boundaries in the partition, where there is a diagonal boundary to the left of a vertical boundary; these intersection points may lie in rows to be processed in the future.

Representing and querying the partition. Each region is represented by its boundaries. Each boundary can be encoded as a column number or a diagonal number (see Fig. 5). We keep those boundaries in two distinct data structures *C-Bound* and *D-Bound*. Given a query point x, we can easily find which region it belongs to in the current partition by identifying the column in *C-Bound* to the left of $y(c)$ and the diagonal in *D-Bound* below $d(x)$. Additional details are provided in [5].

Updating the partition. For each point x in B, we identify the region R in which it lies in the current partition. Assume that R belongs to y. A decision is made as to whether x's region is better or y's region is better. If x's region is better, x's region is inserted into the partition; y's region may be split in two. If y's region is better, x is discarded. Obviously, in the case where both regions are real and x loses to y because x's fragment reaches a column number lower than the one reached by the fragment originating at y, discarding of x is the appropriate action because the part of the matrix covered by the fragment at x is already covered by the fragment at y. Changes to the partition are made by properly updating *C-Bound* and

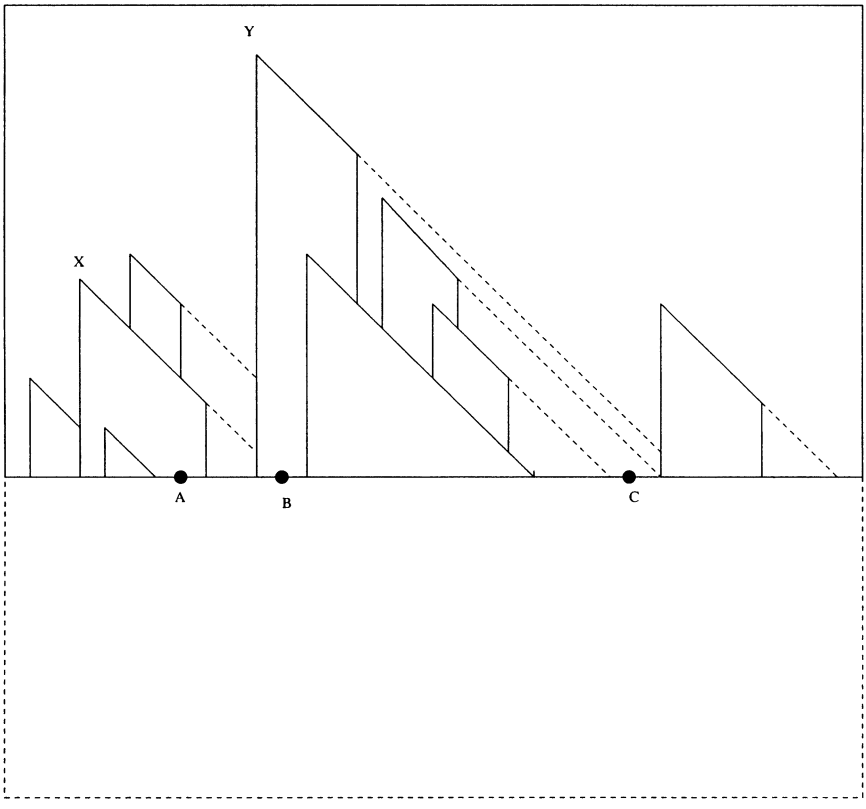

FIG. 5. A partition of row r into regions, according to which left influence is best at their intersection with row r. Solid diagonal lines show fragments, while dashed diagonal lines show dummy left influences. Notice that regions appear linearly ordered with different types of borders, e.g., column–column; column–diagonal. Each point can "own" more than one region (see point X). The query points A and B fall into regions owned by X and Y. Query point C falls into a region that belongs to a dummy left influence. Therefore, it has no fragment above it.

D-Bound. Creation of a new region may also involve the creation of new intersection points and the deletion of an old one. See Fig. 6.

Preparing for the next batch. Assume that the next batch corresponds to row $r' > r$. Independently of whether it is a partition or query batch, we need to have the correct partition when we get to row r'. That means we have to process the intersection points on rows from $r + 1$ to r'. In processing an intersection point, a region is removed, and a decision is made as to which of its neighboring regions wins. Processing an intersection point may also require adding another intersection point. See Fig. 6. Intersection points to be processed are maintained dynamically and a conflict at an intersection point is resolved only when needed.

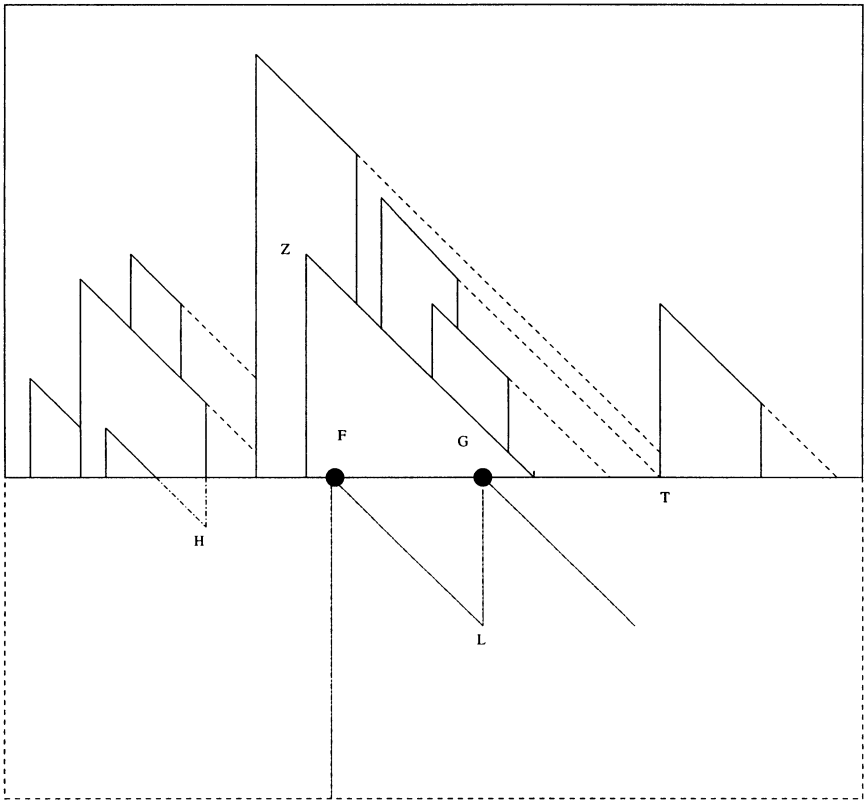

FIG. 6. Points F and G fall into a region owned by Z. In the figure, they have been inserted in the partition assuming that their fragments "touch" a column to the right of that "touched" by the fragment originating in Z. Notice that they create an intersection point, L in the figure, where we have to decide how the partition must continue. Assuming that the next batch is on a row past L, we need to resolve conflicts at intersection points T, H, and L.

Correctness and consistency of this approach with Algorithm PLCS and with the requirements of Lemma 4.1 is straightforward, given the correctness of Algorithm `Left Influence`. As before, operations of the same type may be grouped together in order to take advantage of the efficiencies of Johnson's version of Flat Trees.

LEMMA 4.2. [5] *Assume that Algorithm* `Left Influence` *has given a total of M_v query points and M'_v partition points. For each query point x, it returns the start point of a fragment above x. Then total time is $O(M_v \log\log \min(M_v, nm/M_v) + M'_v \log\log \min(M'_v, nm/M'_v))$, when D is Johnson's version of Flat Trees. When D is a balanced search tree, the time is $O(M_v \log M_v + M'_v \log M'_v)$ time.*

4.3. *Correctness and Time Analysis*

THEOREM 4.1. *Suppose* $X[1:n]$ *and* $Y[1:m]$ *are strings, and a set* M *of fragments relating substrings of* X *and* Y *is given. One can use standard balanced search trees to compute the LCS from Fragments in* $O(|M|\log|M|)$ *time and* $O(|M|)$ *space for a cost measure where each insertion, deletion, or fragment segment has cost* 1. *When one uses Johnson's data structure, the time reduces to* $O(|M|\log\log\min(|M|, nm/|M|))$.

Proof. Correctness follows from Lemma 4.1. As for the time analysis, we limit ourselves to consider the $O(n+m)$ instances of the Algorithm `Left Influence` when D is Johnson's version of Flat Trees. All other cases of Algorithm `PLCS`, i.e., (a) and (b), can be handled similarly.

Assume that we have $s \le n+m+1$ distinct values of Z_a during the entire life of Algorithm `PLCS`. Let M'_{v_i} be the number of partition points for the ith value of Z_a. Then, the time due only to partition points is, apart from multiplicative constants, $\sum_{i=1}^{n+m+1} M'_{v_i}\log\log\min(M'_{v_i}, nm/M'_{v_i})$. But for some constant c, $\sum_{i=1}^{n+m+1} M'_{v_i} \le c|M|$, since there are at most two regions per fragment, and each region has at most two intersection points at the bottom. By the convexity of the log log function we obtain that the time due only to partition points is $O(|M|\log\log\min(|M|, nm/|M|))$. An analogous reasoning holds for query points, since each fragment queries at most one value of Z_a. ∎

5. CONCLUDING REMARKS

We have shown how to compute longest common subsequence of two strings X and Y from Fragments, i.e., from a set of "matching" pairs of substrings of the two input strings. The notion of match is either exact textual match or parameterized match according to the definition given in [3, 4]. Moreover, we use two cost measures, one of which is motivated by a software application. The algorithmic techniques that we obtain complement the ones already available in [5].

The two algorithms are quite elegant in that Lemmas 3.1 and 4.1 succinctly represent the requirements for each piece of the minimum cost path through the edit graph and the algorithms cleverly store a minimal amount of information to apply the lemmas.

For the software application, the database of matches may include both exact matches (where the corresponding substrings are identical) and parameterized matches (where the corresponding substrings contain different variable names). It may be desirable to assign a cost of 0 to exact matches and a cost of 1 to parameterized matches that are not exact. It is straightforward to modify the algorithm of the previous section to allow

for this modification, without changing the time bounds. Other weights may also be used as long as the cost of a fragment is less than the cost of an insertion plus a deletion, otherwise Lemma 2.2 would fail.

ACKNOWLEDGMENT

The authors are deeply indebted to the referee for very punctual comments and suggestions that have greatly helped in the presentation of our ideas.

REFERENCES

1. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, "Data Structures and Algorithms," Addison-Wesley, Reading, MA, 1983.
2. A. Apostolico, String editing and longest common subsequence, *in* "Handbook of Formal Languages" (G. Rozenberg and A. Salomaa, Eds.), Vol. 2, pp. 361–398, Springer Verlag, Berlin, 1997.
3. B. S. Baker, Parameterized pattern matching: Algorithms and applications, *J. Comput. Syst. Sci.* **52**(1) (1996), 28–42.
4. B. S. Baker, Parameterized duplication in strings: Algorithms and an application to software maintenance, *SIAM J. Comput.* **26**(5) (1997), 1343–1362.
5. D. Eppstein, Z. Galil, R. Giancarlo, and G. Italiano, Sparse dynamic programming I: Linear cost functions, *J. Assoc. Comput. Mach.* **39** (1992), 519–545.
6. D. Eppstein, Z. Galil, R. Giancarlo, and G. Italiano, Sparse dynamic programming II: Convex and concave cost functions, *J. Assoc. Comput. Mach.* **39** (1992), 546–567.
7. M. Farach and M. Thorup, Sparse dynamic programming for evolutionary tree comparison, *SIAM J. Comput.* **26** (1997), 210–230.
8. D. Gusfield, "Algorithms on Strings, Trees and Sequences—Computer Science and Computational Biology," Cambridge University Press, Cambridge, 1997.
9. D. S. Hirschberg, Serial computations of Levenshtein distances, *in* "Pattern Matching Algorithms" (A. Apostolico and Z. Galil, Eds.), pp. 123–142, Oxford University Press, Oxford, 1997.
10. J. W. Hunt and T. G. Szymanski, A fast algorithm for computing longest common subsequences, *Comm. ACM* **20** (1977), 350–353.
11. D. B. Johnson, A priority queue in which initialization and queue operations take $O(\log \log D)$ time, *Math. Systems Theory* **15** (1982), 295–309.
12. J. B. Kruskal and D. Sankoff (Eds.), "Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison," Addison-Wesley, Reading, MA, 1983.
13. W. Miller and E. Myers, Chaining multiple alignment fragments in sub-quadratic time, *in* "Proc. of 6th ACM—SIAM SODA," 1995, pp. 48–57.
14. E. W. Myers, An O(ND) difference algorithm and its variations, *Algorithmica* **1** (1986), 251–266.
15. P. van Emde Boas, Preserving order in a forest in less than logarithmic time and linear space, *Inform. Process. Lett.* **6** (1977), 80–82.
16. M. S. Waterman, "Introduction to Computational Biology. Maps, Sequences and Genomes," Chapman Hall, Los Angeles, 1995.
17. W. J. Wilbur and D. J. Lipman, Rapid similarity searches of nucleic acid and protein data banks, *in* "Proc. Nat. Acad. Sci. USA 80," 1983, pp. 726–730.