Longest Common Subsequence from Fragments via Sparse Dynamic Programming

Brenda S. Baker[1] and Raffaele Giancarlo[2*]

[1] Bell Laboratories, Lucent Technologies,
700 Mountain Avenue, Murray Hill, NJ 07974, USA
bsb@research.bell-labs.com
[2] Dipartimento di Matematica ed Applicazioni, Universitá di Palermo,
Via Archirafi, 34-90123 Palermo - ITALY
raffaele@altair.math.unipa.it

Abstract. 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 [7, 11, 15]. We provide a new Sparse Dynamic Programming technique that extends the Hunt-Szymanski [2, 9, 8] 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, resp.) 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 [10]. These bounds apply for two cost measures. The algorithm can also be adapted to finding the usual LCS in $O((m + n) \log |\Sigma| + |M| \log |M|)$ using balanced trees or $O((m + n) \log |\Sigma| + |M| \log \log \min(|M|, nm/|M|))$ using Johnson's Flat Trees, where M is the set of maximal matches between substrings of X and Y and Σ is the alphabet.

1 Introduction

Sparse Dynamic Programming [4, 5] is a technique for the design of efficient algorithms mainly with applications to problems arising in Sequence Analysis. As in [4, 5], 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 [7, 11, 15]. 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

* Work Supported in part by Grants from the Italian Ministry of Scientific Research and by the Italian National Research Council. Part of this work was done while the author was visiting Bell Laboratories of Lucent Technologies

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 [9] 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. [4, 5]. Over the years, additional contributions have been provided by several authors (see for instance [6, 12]).

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, 8] 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 within or between software systems. In this case, a database of "matching" sections of code is obtained via a program such as dup [3]. The notion of "matching" code sections may be either exact textual matches or parameterized matches in the sense of [3], in which parameterized matches between code sections indicate textual similarity 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. It would be convenient to have a graphical user interface that would enable scrolling simultaneously through two source programs, in such a way that matching sections are aligned. 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 sections to other matching sections. A solution to LCS from Fragments optimizes the alignment across multiple matching segments.

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 $s_1 s_2 \ldots s_k$ such that s_1, s_2, ... , 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). It is straightforward to transform a solution to the LCS problem into a solution to the minimum edit distance problem, and vice versa [2, 8, 13].

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]. 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.

When the notion of match is the standard one, i.e., the two strings are equal for each pair in M, we use the Levenshtein edit distance.

When the notion of match is that of parameterized match, i.e., for each pair in M, the two strings exibit a one-to-one correspondence as defined in [3], we treat the inclusion of each parameterized match in the LCS as a new "edit" operation. This naturally leads to our second cost measure: a cost of one is incurred 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 [10], 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 [10], that bound reduces to $O((m + n) \log |\Sigma| + |M| \log \log \min(|M|, nm/|M|))$. Since $|M| \leq nm$, the time bounds that we obtain compare favorably with the best available algorithms for the computation of the LCS [2].

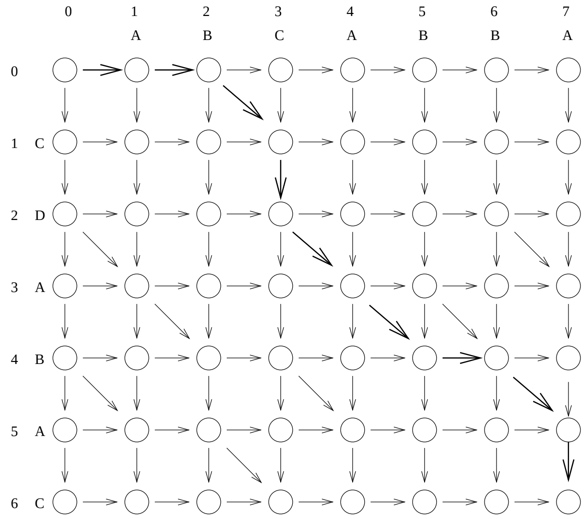

Fig. 1. An edit graph for the strings $X = CDABAC$ and $Y = ABCABBA$. It naturally corresponds to a **DP** matrix. The bold path from $(0,0)$ to $(6,7)$ gives an edit script from which we can recover the LCS between X and Y

2 Definitions and Preliminary Lemmas

Finding the LCS of X and Y is equivalent to finding the Levenshtein edit distance between the two strings [11], where the "edit operations" are insertion and deletion. Following Myers [13], 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. 1) 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 non-bottom point to its neighbor below. There is a horizontal edge from each non-rightmost 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 non-diagonal 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); the vertical and horizontal edges of this path correspond to the insertions and deletions of a minimal edit script, while the diagonal edges correspond to the LCS.

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 Figure 1, the fragments correspond to the maximal matching substrings of X and Y. In Figure 2, the fragments are the sequences of at least 2 diagonal edges of Figure 1. The LCS from Fragments problem is equivalent to finding a

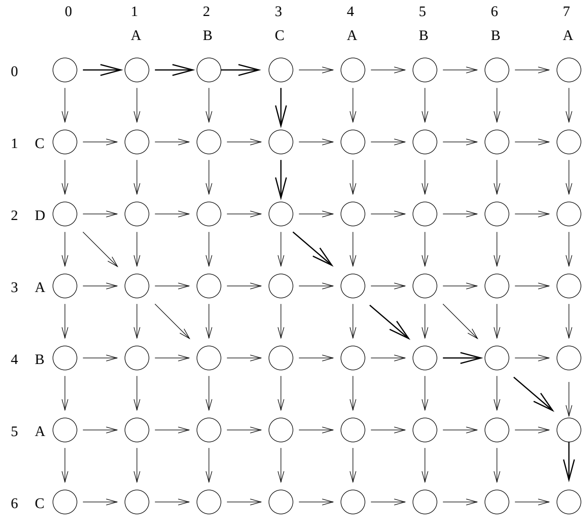

Fig. 2. An LCS from Fragments edit graph for the same strings as in Figure 1, where the fragments are the sequences of at least two diagonal edges of Figure 1. The bold path from $(0, 0)$ to $(6, 7)$ corresponds to a minimum-cost path under the Levenshtein edit distance

minimum-cost path in the edit graph from $(0, 0)$ to (n, m). A minimum-cost path may use only part of certain fragments, as happens with fragment $(3, 5, 2)$ in the two figures. We consider two cost measures. As before, in the Levenshtein cost measure, each diagonal edge has weight 0 and each non-diagonal 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, 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 overlapping fragments, which 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. We say that a segment of a fragment f is a *prefix* of f if it includes $start(f)$.

Lemma 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 a new path P' with the same cost and fewer non-prefix segments; the argument is applied inductively to prove the lemma.

If the last non-prefix segment S of P is part of fragment f, let R be a rectangle with corners $(0,0)$ and $start(f)$; P begins within R and then exits it at a point q. The new path P' follows P except between q and S, where it instead follows the side of R to $start(f)$ and a prefix of f to S. The cost does not increase because the replaced part of P had to cross the same number of diagonals via horizontal or vertical edges, and P' has fewer non-prefix segments than P. □

Between successive fragment segments in a min-cost path satisfying Lemma 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. *In a min-cost path that satisfies Lemma 1, 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, the proof of Lemma 1 yields the following corollary.

Corollary 1. *For any fragment f and any point p on f, $mincost_0(p) = mincost_0(start(f))$.*

By Corollary 1, it is reasonable to define $mincost_0(f) = mincost_0(start(f))$.

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 $start(f)$. Similarly, a fragment f' is *above* $start(f)$ if some point of f' besides $start(f')$ is 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 $mincost_0(f) \leq x(start(f)) + y(start(f))$.

Lemma 3. *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. Cases (1)-(3) represent the minimum costs of the various possible ways a path that can reach $start(f)$ from a preceding fragment: via both horizontal and vertical edges, horizontal edges only, or vertical edges only. □

Next, we develop an algorithm based on Lemma 3.

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, find the next larger value than j in D, and (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 [14] with $O(\log \log u)$ time per operation. If we use Johnson's version of Flat Trees [10], 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.

The general approach of the algorithm will be a sweepline approach where successive rows are processed, and within rows, points are processed from left to right. Lexicographic sorting of the (x, y)-values to enable this can be done in $O(|M| \log |M|)$ time. Alternatively, using data structure 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 in $O(|M| \log \log \min(|M|, nm/|M|))$ time, with analysis as in Eppstein et al. [5]. Not all the rows and columns need contain a start point or end point, and we generally wish to skip empty rows and columns for efficiency. From now on, we assume our algorithm processes only nonempty rows and columns, renumbered to exclude empty ones.

First, 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 of type D. For each fragment f such that $x(start(f)) = i$, we calculate the fragment f' to the left of $start(f)$ from 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 D 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 $[0, 2|M|]$). 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 for the start points of fragments), then a sequence of insertions, and finally a sequence of deletions. In that case, we can use Johnson's version of Flat Trees [10] to show that the sweep can be implemented to take $O(|M| \log \log \min(|M|, nm/|M|)$ time. The analysis is as in [4] (Lemma 1).

The main algorithm takes a sweepline approach of processing successive rows. It follows the same paradigm as the Hunt-Szymanski LCS algorithm [9] 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. 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.

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 "re-schedule" the operations on D so that, for each processed row, we perform four sequences of homogeneous operations of the type max, insert, delete, member. Using this fact and the analysis in Eppstein et al. [5], we can show that this phase of the algorithm takes $O(|M| \log\log \min(|M|, nm/|M|)$ time.

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 and time.

Theorem 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 and the time analysis from the discussion preceding the statement of the theorem. □

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 or 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], 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 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 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, the minimum cost of reaching a point on a fragment f cannot be more than one less than the cost of reaching it through f; this fact follows from the proof of Lemma 1.

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

Lemma 4. *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. The proof is similar to that of Lemma 3 except that cases (2) and (3) are more complicated.

In particular, (2) covers two cases for paths that reach $start(f)$ via only horizontal edges from the previous fragment. In one case, the path traverses $visl(f)$, and in the other case, it passes through a point p on $visl(f)$ but does not traverse a segment of $visl(f)$. Since the cost to p can be lower by at most one in the latter case than in the former, we need consider only the fragments to the left of $start(f)$ that will result in a cost of one less than in the former case. Since the cost of a sequence of horizontal edges is the change in diagonal number, we need consider only fragments f' to the left for which $Z_l(f') = Z_l(f) - 1$. For condition (3), the argument is similar. $\square$

Lemma 4 allows us to restrict our attention to fragments f' with particular values of Z_a and Z_l.

We describe the approach for a particular value of Z_l; for Z_a, the approach is similar. For the set S of fragments f' with this value of $Z_l(f')$, we need to store information about the relevant fragments 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 $d(f) > d(f')$. Our strategy is to keep track of the minimum diagonal number of all fragments crossing each row with this value of Z_l. To do this, we use a data structure of type D (as defined as in the previous section) to store the endpoints of intervals where the minimum diagonal number changes, and with each such endpoint, we store the diagonal number; to determine whether any fragment in the set is to the left of $start(f)$, we query to find the maximum stored value less than $x(start(f))$, retrieve the corresponding diagonal number, and compare it to $d(f)$. The construction for case (2) is similar. The data structure must be built dynamically as fragments are processed and the values of Z_l become available.

A separate data structure is kept for each distinct value of Z_l or Z_a. One possibility is to keep a pointer to each of these data structures in two arrays A_l and A_a; these would use $O(n+m)$ space since the range of values of Z_l and Z_a is $[-(n+m), 0]$. Alternatively, two instances of D can be used to store the distinct values of Z_l and Z_a that occur, together with the associated pointers. The total space used is $O(|M|)$.

The algorithm proceeds as follows. First, compute $visl(f)$ and $visa(f)$ as before. As before, the main sweepline algorithm sweeps across successive rows and has a data structure B that keeps track of the columns in which the minimum value of $Z_l(f)$ decreases compared to columns for the left. However, when a new value is to be computed for $mincost_1(start(f))$, the new algorithm applies Lemma 4, using the data structure for $z = Z_l(visl(f))$. After the values of $mincost_1(f)$ for the fragments with start points in the row have been computed, the next task is to update the data structures storing the sets of used values of $Z_l(f)$ and $Z_a(f)$ and the individual data structure for each such value. Then B is updated as before based on the end points in the row and the values of $Z_p(f)$ for the fragments containing these end points.

As before, data structures of type D can be implemented via either balanced trees or Johnson's version of van Emde Boas Flat Trees with the same time bounds as discussed in the previous section, and there are $O(|M|)$ operations to perform. Consequently, we obtain the following result.

Theorem 2. *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 and the time and space analysis in the discussion of the algorithm. □

5 Extensions to Other Cost Measures

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 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 would fail.

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 G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages, Vol. 2*, pages 361–398, Berlin, 1997. Springer Verlag.
3. B. S. Baker. A theory of parameterized pattern matching: Algorithms and applications. In *Proc. 25th Symposium on Theory of Computing*, pages 71–80. ACM, 1993.
4. D. Eppstein, Z. Galil, R. Giancarlo, and G. Italiano. Sparse dynamic programming I: Linear cost functions. *J. of ACM*, 39:519–545, 1992.
5. D. Eppstein, Z. Galil, R. Giancarlo, and G. Italiano. Sparse dynamic programming II: Convex and concave cost functions. *J. of ACM*, 39:546–567, 1992.
6. M. Farach and M. Thorup. Optimal evolutionary tree comparison by sparse dynamic programming. In *Proc. 35th Symposium on Foundations of Computer Science*, pages 770–779. IEEE, 1994.
7. D. Gusfield. *Algorithms on Strings, Trees and Sequences-Computer Science and Computational Biology.* Cambridge University Press, Cambridge, 1997.
8. D.S. Hirschberg. Serial computations of Levenshtein distances. In A. Apostolico and Z. Galil, editors, *Pattern Matching Algorithms*, pages 123–142, Oxford, 1997. Oxford University Press.
9. J.W. Hunt and T.G. Szymanski. A fast algorithm for computing longest common subsequences. *Comm. of the ACM*, 20:350–353, 1977.
10. D. B. Johnson. A priority queue in which initialization and queue operations take $O(\log \log D)$ time. *Math. Sys. Th.*, 15:295–309, 1982.
11. J.B. Kruskal and D. Sankoff, editors. *Time Wraps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison.* Addison-Wesley, 1983.
12. W. Miller and E. Myers. Chaining multiple alignment fragments in sub-quadratic time. In *Proc. of 6-th ACM-SIAM SODA*, pages 48–57, 1995.
13. E. W. Myers. An O(ND) difference algorithm and its variations. *Algorithmica*, 1:251–266, 1986.
14. P. van Emde Boas. Preserving order in a forest in less than logarithmic time. *Info. Proc. Lett.*, 6:80–82, 1977.
15. M.S. Waterman. *Introduction to Computational Biology. Maps, Sequences and Genomes.* Chapman Hall, Los Angeles, 1995.