Sparse LCS Common Substring Alignment

Gad M. Landau[*]
Haifa University
and
Polytechnic University

Baruch Schieber [†]
IBM Research Division

Michal Ziv-Ukelson [‡]
Haifa University
and
IBM Research Division

Abstract

The "Common Substring Alignment" problem is defined as follows. The input consists of a set of strings $S_1, S_2 \ldots S_c$, with a common substring appearing at least once in each of them, and a target string T. The goal is to compute similarity of all strings S_i with T, without computing the part of the common substring over and over again.

In this paper we consider the Common Substring Alignment problem for the LCS (Longest Common Subsequence) similarity metric. Our algorithm gains its efficiency by exploiting the *sparsity* inherent to the LCS problem. Let Y be the common substring, n be the size of the compared sequences, L_y be the length of the LCS of T and Y, denoted $|LCS[T, Y]|$, and L be $\max\{|LCS[T, S_i]|\}$. Our algorithm consists of an $O(nL_y)$ time encoding stage that is executed once per common substring, and an $O(L)$ time alignment stage that is executed once for each appearance of the common substring in each source string. The additional running time depends only on the length of the parts of the strings that are not in any common substring.

1 Introduction

The problem of comparing two sequences A and B to determine their similarity is one of the fundamental problems in pattern matching. One of the basic forms of the problem is to determine the *longest common subsequence (LCS)* of A and B. The LCS string comparison metric measures the subsequence of maximal length common to both sequences [1]. Longest Common Subsequences have many applications, including sequence comparison in molecular biology as well as the widely used *diff* file comparison program. The LCS problem can be solved in $O(mn)$ time, where m and n are the lengths of strings A and B, using dynamic programming [8]. The dynamic programming creates an $m \times n$ "DP Table" that contains in its (i, j) entry the LCS of the prefix of A of size i and the prefix of B of size j.

[*]Department of Computer Science, Haifa University, Haifa 31905, Israel, phone: (972-4) 824-0103, FAX: (972-4) 824-9331; Department of Computer and Information Science, Polytechnic University, Six MetroTech Center, Brooklyn, NY 11201-3840; email: `landau@poly.edu`; partially supported by NSF grant CCR-0104307, by the Israel Science Foundation grant 282/01, by the FIRST Foundation of the Israel Academy of Science and Humanities, and by IBM Faculty Partnership Award.

[†]IBM T.J. Watson Research Center, P.O. Box 218 Yorktown Heights, NY 10598; Tel: (914) 945-1169; Fax: (914) 945-3434; e-mail: `sbar@watson.ibm.com`;

[‡]Department of Computer Science, Haifa University, Haifa 31905, Israel; On Education Leave from the IBM T.J. Watson Research Center; email: `michal@cs.haifa.ac.il`; partially supported by the Israel Science Foundation grant 282/01, and by the FIRST Foundation of the Israel Academy of Science and Humanities.

More efficient LCS algorithms, which are based on the observation that the LCS solution space is highly redundant, try to limit the computation only to those entries of the DP Table which convey essential information, and exploit in various ways the *sparsity* inherent to the LCS problem. Sparsity allows us to relate algorithmic performances to parameters other than the lengths of the input strings. Most LCS algorithms that exploit sparsity have their natural predecessors in either Hirshberg [8] or Hunt-Szymanski [9].

All Sparse LCS algorithms are preceded by an $O(n \log |\Sigma|)$ preprocessing [1]. The Hirshberg algorithm uses $L = |LCS[A, B]|$ as a parameter, and achieves an $O(nL)$ complexity. The Hunt-Szymanski algorithm utilizes as parameter the number of matches between A and B, denoted r, and achieves an $O(r \log n)$ complexity. Apostolico and Guerra [2] achieve an $O(L \cdot m \cdot \min(\log |\Sigma|, \log m, \log(2n/m))$ algorithm, where $m \leq n$, and another $O(m \log n + d \log(nm/d))$ algorithm, where $d \leq r$ is the number of dominant matches (as defined by Hirschberg [8]). This algorithm can also be implemented in time $O(d \log \log \min(d, nm/d))$ [6]. Note that in the worst case both d and r are $\Omega(n^2)$, while L is always bounded by n.

The *Common Substring Alignment Problem* is defined in [12] as follows: The input consists of a set of one or more strings $S_1, S_2 \ldots S_c$ and a target string T. It is assumed that a common substring Y appears in all strings S_i at least once. Namely, each S_i can be decomposed in at least one way to $S_i = B_i Y F_i$. (See Figure 1.) The goal is to compute the similarity of all strings S_i with T, without computing the part of Y over and over again. It is assumed that the locations of the common subsequence Y in each source sequence S_i are known. However, the part of the target T with which Y aligns, may vary according to each B_i and F_i combination.

T	=	"BCBADBDCD"	Y	=	*"BCBD"*			
S_1	=	"BC *BCBD* C"	B_1	=	"BC"	F_1	=	"C"
S_2	=	"E *BCBD* D*BCBD* A"	B_{2a}	=	"E"	F_{2a}	=	"D*BCBD* A"
			B_{2b}	=	"E*BCBD* D"	F_{2b}	=	"A"

Figure 1: An example of two different source strings S_1, S_2 sharing a common substring Y, and a target T.

More generally, the common substring Y could be shared by different source strings competing over similarity with a common target, or could appear repeatedly in the same source string. Also, in a given application, we could of course be dealing with more than one repeated or shared subcomponent. (See Figure 1.)

Common Substring Alignment algorithms are usually composed of a pre-processing stage that depends on data availability, an *encoding* stage and an *alignment* stage. During the encoding stage, a data structure is constructed which encodes the comparison of Y with T. Then, during the alignment stage, for each comparison of a source S_i with T, the pre-compiled data structure is used to speed up the part of aligning each appearance of the common substring Y.

In most of the applications for which Common Substring Alignment is intended, the source sequence database is prepared *off-line*, while the target can be viewed as an "unknown" sequence which is received *online*. The source strings can be pre-processed *off-line* and parsed into their optimal common substring representation. Therefore, we know well beforehand where, in each S_i, Y begins and ends. However, the comparison of Y and T can not be computed until the target is received. Therefore, the encoding stage, as well as the alignment stage, are both *online* stages.

Even though both stages are *online*, they do not bear an equal weight on the time complexity of the algorithm. The efficiency gain is based on the fact that the encoding stage is executed only once per target, and then the encoding results are used, during the alignment stage, to speed up the alignment of each appearance of the common substring in any of the source strings.

To simplify notation we assume from now on that all compared strings are of size n. Our results can be extended easily to the case in which the strings are of different length. We use L to denote $\max\{|LCS[T, S_i]|\}$, and L_y to denote $|LCS[T, Y]|$. (Note that $L_y \leq |Y|$, $L_y \leq L$, and $L \leq n$.)

Results. In this paper we address the following challenge: can a more efficient common substring alignment algorithm, which exploits the *sparsity* inherent to the LCS metric, be designed for the LCS metric. We show how to exploit sparsity, by replacing the traditional matrix which is used to encode the comparison of Y and T, with a smaller matrix. We show that this smaller matrix can be computed using Myers' Consecutive Suffix Alignments algorithm [14]. We also prove that this smaller matrix preserves the Total Monotonicity condition, thus enabling efficient adaptation of a matrix searching algorithm of Aggarwal *et al.* [3].

Our algorithm consists of an $O(nL_y)$ encoding stage, and an $O(L)$ alignment stage. When the problem is sparse ($L_y << |Y|$, $L << n$), our time bounds are better than those of previous algorithms. Even when the data is dense, our solution for the problem is no worse than the best-known algorithms.

The first Common Substring Alignment algorithm for the LCS metric was given in [11]. It presents an $O(n^2 + n|Y|)$ encoding stage, and an $O(n)$ alignment stage. In [12] a Common Substring Alignment algorithm for the LCS, Edit Distance and more extended metrics is given. This algorithm consists of an $O(n|Y|)$ encoding stage, and an $O(n)$ alignment stage.

The remainder of this paper is organized as follows. Section 2 contains Common Substring Alignment preliminaries. The new algorithm is described in section 3. Section 4 contains an analysis and assertion of some of the properties of the new data representation which allow for the efficiency gain. Conclusions and open problems are given in Section 5.

2 Common Substring Alignment Preliminaries

In the literature the DP Table used for computing the alignment is also viewed as a directed acyclic graph (DAG), called the Dynamic Programming (DP) Graph [7]. The DP Graph for S and T, contains $(|S|+1)(|T|+1)$ vertices, each labeled with a distinct pair $(x, w)(0 \leq x \leq |S|, 0 \leq w \leq |T|)$. The vertices are organized in a matrix of $(|S| + 1)$ rows and $(|T| + 1)$ columns. (See Figure 2.)

When using the LCS metric, the DP Graph contains a directed edge with weight zero from each vertex (x, w) to each of the vertices $(x, w+1)$, $(x+1, w)$. It also contains a directed edge from each vertex (x, w) to vertex $(x + 1, w + 1)$. This diagonal edge has weight one if $S[x + 1] = T[w + 1]$, and zero otherwise. Maximal weight paths in the Dynamic Programming graph represent optimal alignments of S and T. The Dynamic Programming algorithm will set the value of vertex (i, j) in the graph to the total weight of the highest scoring path which originates in vertex $[0, 0]$ of the graph and ends in vertex $[i, j]$.

The Dynamic Programming Graph used for computing the similarity between a source string $S_i = B_iYF_i$ and a target string T can be viewed as a concatenation of 3 sub-graphs, where the first graph represents the similarity between B_i and T, the second graph represents the similarity between Y and T, and the third graph represents the similarity between F_i and T.

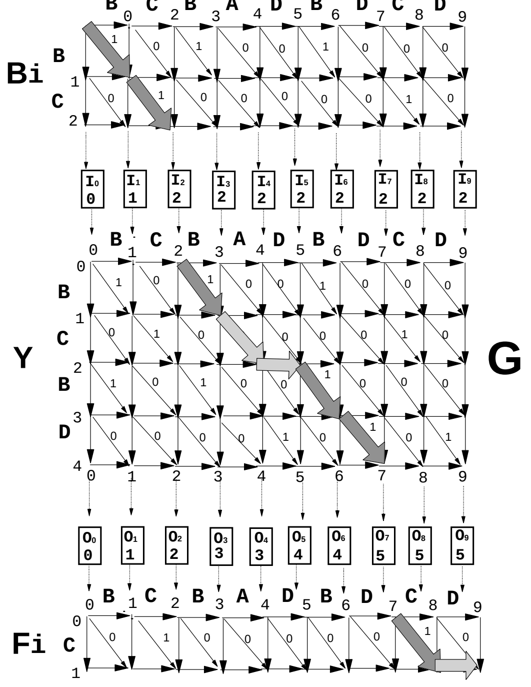

Figure 2: The LCS Dynamic Programming Graph for the comparison of S_1 = "$BCBCBDC$" with T = "$BCBADBCDC$". The highlighted path in the graph corresponds to the common subsequence "$BCBBDC$".

In this partitioned solution, the weights of the vertices in the last row of the first graph serve as input to initialize the weights of the vertices in the first row of the second graph. The weights of the last row of the second graph can be used to initialize the first row of the third graph.

The motivation for breaking the solution into 3 sub-graphs is that the second sub-graph, which represents the comparison of Y with T, is identical in each of its appearances in all DP graphs comparing any of the strings S_i with T. More specifically, both the structure and the weights of the edges of all DP sub-graphs comparing Y with T are identical, but the weights to be assigned to the vertices during the LCS computation may vary according to the prefix B_i which is specific to the source string. Therefore, an initial investment in the learning of this graph as an encoding stage, and in its representation in a more informative data structure, may pay off later on.

For a string A of length n and for $1 \leq u \leq z \leq n$, let A_u^z denote the substring of A from index u (inclusive) up to index z (inclusive). We define the following notations with respect to the Dynamic Programming Graph used for computing the LCS of $S_i = B_iYF_i$ and T. Let G denote the second sub-graph which represents the comparison of Y with T. Let I denote the first row of G. Namely, $I[j] = |LCS[T_1^j, B_i]|$, for $j = 1, \ldots, n$. Let O denote the last row of G. Namely, $O[j] = |LCS[T_1^j, B_iY]|$, for $j = 1, \ldots, n$.

Our solution will focus on the work necessary for an appearance of a given common substring: Given a target string T, a source substring Y and an input row I, compute the output row O.

As described above the online work for each common substring consists of two stages.

1. *Encoding Stage*: Study the structure of G and represent it in an efficient way.

2. *Alignment Stage*: Given I and the encoding of G, constructed in the previous stage, compute O.

Observe that, due to the monotonicity and unit-step properties of LCS, both I and O are monotone staircases with at most $L+1$ unit steps. This enables the following definitions of the vectors PI and PO.

Definition 1 *For $k = 0, \ldots, L$, the entry $PI[k]$ contains the smallest index in the input row I whose value is k, if such exists.*

Definition 2 *For $k = 0, \ldots, L$, the entry $PO[k]$ contains the smallest index in the output row O whose value is k, if such exists.*

We claim that PI is sufficient for the computation of PO. To see this consider some $k \in [0..L]$ and assume that $i_1 = PI[k]$ is defined. Let i_3 be $PI[k+1]$ if it is defined or $n+1$ otherwise. Note that for any index i_2, where $i_1 < i_2 < i_3$, $I[i_1] + |LCS[T^j_{i_1+1}, Y]| \geq I[i_2] + |LCS[T^j_{i_2+1}, Y]|$, for any $j \in [i_2..n]$. This is true since $I[i_2] = I[i_1] = k$ and $|LCS[T^j_{i_1+1}, Y]| \geq |LCS[T^j_{i_2+1}, Y]|$ for any $i_1 < i_2 < i_3$. (The second inequality follows since the concatenation of prefix $T^{i_2}_{i_1+1}$ to the string $T^j_{i_2+1}$ can either increase the size of its common subsequence with Y or leave it unchanged.)

3 The Algorithm

The objective of the algorithm is to compute $PO[k]$, for $k = 1, \ldots, L$ given the vector PI. (Note that $PO[0] = 0$.) Recall that $PO[k]$ is the smallest index of an entry in O with a value of k. In terms of optimal paths in the alignment graph, this means that $PO[k]$ is the index of the leftmost vertex in the output border of G to end a path of weight k, which originates in vertex $(0,0)$ of the alignment graph. Note that such a path could enter G through any one of its input border vertices whose value is $\leq k$. However, in section 2 we have shown that the PI indices are sufficient for representing all the potential entry points of I, and therefore only the values $PI[0 \ldots k]$ will be considered as relevant representative entry point indices for the sought k-path.

Now, consider any path of total weight k, which originates in vertex $(0,0)$ of the alignment graph, enters G through a given input border vertex $PI[r]$, and ends in some vertex j on the output border of G. This path could be decomposed into two parts: the sub-path connecting vertex $(0,0)$ with the selected input border entry $PI[r]$, followed by the sub-path from the selected input border entry to vertex j on the output border. The weight of the sub-path from vertex (0,0) to the selected input border entry is the value $I[PI[r]] = r$. The weight of the sub-path from vertex $PI[r]$ on the input border to vertex j on the output border is $|LCS[T^j_{PI[r]+1}, Y]|$. By definition, the sum of the weights of these two sub-paths, whose concatenation gives the total k-path, must be k.

Now recall that among all such potential k-paths, which could enter G through any of the $PI[r]$, $r = 0 \ldots k$ input border vertices, we are actually interested in the ones which are optimal in the sense that they end in the leftmost possible output border vertex. Therefore, the value of $PO[k]$, for $1 \leq k \leq L$, is computed as follows.

$$PO[k] = \min_{r=0}^{k} \{ j \mid r + |LCS[T^j_{PI[r]+1}, Y]| = k \} \tag{1}$$

We show how to obtain PO in two stages.

1. The encoding stage which is executed only once for each common substring Y, in which we compute $\min\{j \mid |LCS[T_{i+1}^{j}, Y]| = k\}$ for each pair (i,k), $i = 0 \dots n$ and $k = 1 \dots L_y$.
2. The alignment stage in which we compute PO, using Equation 1, given PI and the values computed in the encoding stage.

3.1 The Encoding Stage

In the encoding stage we compute the $n \times L_y$ table S, where $S[i,k] = \min\{j \mid |LCS[T_{i+1}^{j}, Y]| = k\}$, if such an index exists. In other words, $S[i,k]$ contains the smallest index of a vertex in the last row of the DP Graph defined by the LCS of T and Y that can be reached from vertex i in the first row of this graph via a path of weight k. We observe that Myers' Consecutive Alignments algorithm [14] can be used to construct the table S.

Complexity. Given two strings Y and T over a constant alphabet Myers [14] constructs the table S for the comparison of Y versus T in $O(nL_y)$ time and space.

3.2 The Alignment Stage

During each execution of the alignment stage the objective is to compute PO from PI, using the table S computed in the encoding stage. Recall that $PO[0] = 0$ and for $k > 0$, $PO[k] = \min_{r=0}^{k}\{S[PI[r], k-r]\}$.

Note that when the alignment stage is executed the L values of the PI for this specific alignment stage are known, and S has already been computed. It follows that the representation of the competing PI's can be reduced to the $(L+1) \times L$ matrix OUT in which $OUT[r,k] = S[PI[r], k-r]$, for $1 \le k \le L$ and $0 \le r \le k$.

Note that in the DP Graph of S_i and T the entry $OUT[r,k]$ is the smallest index of a vertex in row O that is an endpoint of a path of weight k that starts in vertex $(0,0)$, and goes through vertex $PI[r]$ in row I.

Clearly, for $k \in [1..L]$, $PO[k]$ is the minimum of the k-th column in the above OUT matrix. In Section 4 we prove that OUT is convex totally monotone. Hence, a recursive algorithm by Aggarwal *et al.* [3], nicknamed $SMAWK$ in the literature, can be used to compute the column minima of OUT.

Complexity. Given S and PI, computing an element of OUT requires O(1) time and space. The $SMAWK$ algorithm computes the column minima of the $(L+1) \times L$ totally monotone matrix OUT in $O(L)$ time and space, by querying only $O(L)$ entries of the array. Hence, PO is computed during the alignment stage in $O(L)$ time and space.

4 *OUT* as a Totally Monotone Rectangular Matrix

Definition 3 *A matrix $M[0 \dots m, 0 \dots n]$ is* **totally monotone** *if either condition 1 or 2 below holds for all $a, b = 0 \dots m$; $c, d = 0 \dots n$:*

1. **convex condition:** $M[a,c] \ge M[b,c] \Longrightarrow M[a,d] \ge M[b,d]$ *for all $a < b$ and $c < d$.*
2. **concave condition:** $M[a,c] \le M[b,c] \Longrightarrow M[a,d] \le M[b,d]$ *for all $a < b$ and $c < d$.*

In this section we prove that OUT can be safely transformed into a full, rectangular, convex totally monotone matrix, as needed for the implementation of $SMAWK$ algorithm in the alignment stage. We start by noting that, originally, OUT is not a full, rectangular matrix, since some its entries are undefined. Recall that $OUT[r,k]$ is defined as $S[PI[r], k-r]$. Since I is a monotone staircase, $PI[r]$ is defined for $0 \leq r \leq L_I$, where L_I denotes $|LCS[T, B_i]|$, and undefined for $r > L_I$. We consider only the first L_I rows of OUT and thus we may assume that $PI[r]$ is always defined. It follows that an entry $OUT[r,k]$ is undefined whenever the entry $S[PI[r], k-r]$ is undefined.

Definition 4 *Let k_r, for $0 \leq r \leq L_I$, denote the greatest column index of an entry in row r of OUT whose value is defined.*

Lemma 1 *Each row in OUT consists of a (possibly empty) span of undefined entries, called the undefined prefix, followed by a span of defined entries, and by a (possibly empty) span of undefined entries, called the undefined suffix. For a given row r of OUT, the span of defined entries starts in entry $OUT[r,r]$ and ends in some entry $OUT[r,k_r]$, such that $r \leq k_r \leq L$.*

Proof: Suppose that for $c < e$, both $OUT[r,c]$ and $OUT[r,e]$ are defined. From the definition of the table S it follows that all the entries $S[PI[r], d-r]$, for $c \leq d \leq e$ are also defined and thus also $OUT[r,d]$. This means that the defined entries in each row of OUT form a consecutive interval. Following the definition of the S and OUT tables, $S[PI[r], 0]$ is always defined and therefore $OUT[r,r] = S[PI[r], r-r]$ is always defined. $S[PI[r], -1]$, on the other hand, is never defined and therefore $OUT[r, r-1]$ is never defined. Thus we conclude that the span of consecutive defined entries in row r of OUT begins in $OUT[r,r]$ and ends in $OUT[r,k_r]$. ▮

Our next goal is to prove that the defined entries of OUT follow the convex total monotonicity property. Later, we show how to complement the undefined entries of OUT, without changing its column minima, and still maintain this property.

Lemma 2 *For any a, b, c, such that $a < b$ and all four entries: $OUT[a,c]$, $OUT[b,c]$, $OUT[a,c+1]$, and $OUT[b,c+1]$ are defined, if $OUT[a,c] \geq OUT[b,c]$, then $OUT[a,c+1] \geq OUT[b,c+1]$.*

Proof: The proof is based on a crossing paths contradiction [4, 10, 13, 15].

We consider two paths in the DP Graph of B_iY and T. Let path A_{c+1} denote an optimal path (of weight c+1) connecting vertex $(0,0)$ of the graph with vertex $OUT[a, c+1]$ of O and going through vertex $PI[a]$ of I. Note that by our definition such a path always exists. Similarly, let path B_c denote an optimal path connecting vertex $(0,0)$ of the graph with vertex $OUT[b,c]$ of O and going through vertex $PI[b]$ of I. Figure 3 shows paths B_c and A_{c+1}. Note that, since $OUT[b,c] \leq OUT[a,c] < OUT[a,c+1]$, the two paths B_c and A_{c+1} must intersect at some column of the DP Graph before or at column $OUT[b,c]$. Let X and Y be the prefixes of A_{c+1} and B_c up to the intersection point, and let W and Z be their suffixes from this intersection point.

There are two cases to be addressed, depending on the outcome of the comparison of the weights of X and Y, denoted $|X|$ and $|Y|$.

Case 1. $|X| \leq |Y|$. Then $|Y| + |W| \geq c+1$. By monotonicity of LCS this implies that $OUT[a, c+1] \geq OUT[b, c+1]$.

Case 2. $|X| > |Y|$. Then $|X| + |Z| > c$. By monotonicity of LCS this implies that $OUT[a,c] < OUT[b,c]$, in contradiction to the assumption of the proof. ▮

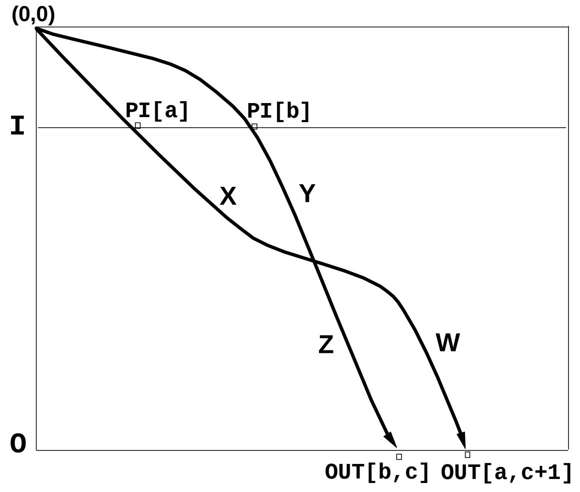

Figure 3: Optimal paths which must cross.

Note that the two assertions of Lemmas 1 and 2 lead, by definition, to the following conclusion.

Conclusion 1 *The defined entries of OUT follow the convex total monotonicity property.*

We now turn to handle the undefined entries of OUT. By Lemma 1 we know that the span of defined entries in a given row r starts in $OUT[r,r]$ and ends in $OUT[r,k_r]$. This implies, by definition, that the undefined prefix of row r consists of its first $r-1$ entries, and thus the undefined prefixes form a lower left triangle in OUT.

The next Lemma assists in defining the shape formed by the undefined suffixes of the rows of OUT.

Lemma 3 *For any two rows a, b of OUT, where $a < b$, if $k_a > k_b$, then $OUT[a,\ell] \leq OUT[b,\ell]$, for all the defined elements in rows a and b of OUT.*

Proof: Since $k_a > k_b$, it follows that $OUT[a,k_b]$, $OUT[b,k_b]$ and $OUT[a,k_b+1]$ are defined, yet $OUT[b,k_b+1]$ is undefined. Note that for every index $b \leq j \leq k_b$ both $OUT[b,j]$ and $OUT[a,j]$ are defined. Suppose there was some index $b \leq j \leq k_b$ such that $OUT[a,j] > OUT[b,j]$.

We consider two paths in the DP Graph of B_iY and T. Let path A_{k_b+1} denote an optimal path (of weight k_b+1) connecting vertex $(0,0)$ of the graph with vertex $OUT[a,k_b+1]$ of O and going through vertex $PI[a]$ of I. Note that by our definition such a path always exists. Similarly, let path B_j denote an optimal path (of weight j) connecting vertex $(0,0)$ of the graph with vertex $OUT[b,j]$ of O and going through vertex $PI[b]$ of I.

Similarly to the proof of Lemma 2, one can show that these two paths cross and that since $OUT[a,j] > OUT[b,j]$ there must be a path of weight at least k_b+1 that originates in vertex $(0,0)$, ends in O and goes through vertex $PI[b]$ of I, in contradiction to the assumption that $OUT[b,k_b+1]$ is undefined. We conclude that, for all the defined elements in rows a and b of OUT, $OUT[a,j] \leq OUT[b,j]$. ▮

Lemma 3 implies that, for any row r of OUT, if there exists another row r' in OUT, such that $r' < r$ and $k_r < k_{r'}$, then row r can be skipped in the search for column minima. After removing all such rows we are guaranteed that the undefined suffixes form an upper right triangle in OUT.

The undefined entries of OUT can be complemented in constant time each, similarly to the solution described in [5], as follows.

1 Lower Left Triangle: These entries can be complemented by setting the value of any $OUT[r,j]$ in the missing lower-left triangle to $(n+r+1)$.

2 Upper Right Triangle: The value of any undefined entry $OUT[r,j]$ in this triangle can be set to ∞.

In the next lemma we prove that OUT can be safely converted into a full, rectangular, convex totally monotone matrix.

Lemma 4 *The OUT matrix can be transformed into a full, rectangular matrix that is totally monotone, without changing its column minima.*

Proof: We have shown (Conclusion 1) that the defined entries of OUT follow the convex total monotonicity property. It remains to show that complementing the undefined entries of OUT, as described above, preserves its column minima, and still maintains this property.

1 Lower Left Triangle: The greatest possible index value in OUT is n. Since r is always greater than or equal to zero, the complemented values in the lower left triangle are lower-bounded by $(n+1)$ and no new column minima are introduced. Also, for any complemented entry $OUT[b,c]$ in the lower left triangle, $OUT[a,c] < OUT[b,c]$ for all $a < b$, and therefore the convex total monotonicity condition holds.

2 Upper Right Triangle: All scores in OUT are finite. Therefore, no new column minima are introduced by the re-defined entries.

Due to the upper right corner triangular shape of this ∞-patched area, for any two rows a, b, where $a < b$, if $OUT[b,c] = \infty$, then surely $OUT[a,c] = \infty$. Let d be the smallest index such that $OUT[a,d] = \infty$. It follows that $OUT[a,e] \geq OUT[b,e]$ for all $e \geq d$, and the convex total monotonicity property is preserved. ▮

Time Complexity We show that the overhead associated with the marking of the undefined elements in OUT, as well as the removal of redundant rows in OUT in order to obtain the crisp undefined upper right triangle, does not slow down the time complexity of the suggested Common Substring Alignment algorithm.

1. **Encoding Stage.** During the encoding stage, $LCS[T^n_{r+1}, Y]$, for $r = 0, \dots, n-1$ is computed and stored. This information will be used later, during the alignment stage, to mark the beginning of the undefined suffix in each row. $LCS[T^n_{r+1}, Y]$ corresponds to the greatest column index of a defined entry in row r of S. Therefore, all n values of $LCS[T^n_{r+1}, Y]$, for $r = 0, \dots, n-1$, can be queried from the constructed S table without changing the original $O(nL_y)$ complexity of the encoding stage algorithm.

2. **Alignment Stage.**

   (a) **Marking the Defined and Undefined Intervals in Each Row of OUT.** Given a row index r and the value $LCS[T^n_{r+1}, Y]$ which was computed in the encoding stage, the undefined prefix and suffix of row r of OUT can each be identified and marked in a constant time. Since there are $O(L)$ rows in OUT, this work amounts to an additional $O(L)$ time.

(b) **Removing the Redundant Rows to Create an Upper Right Triangle.** Since PI is available in the beginning of the alignment stage, the $O(L)$ representative rows of OUT can be scanned in increasing order as a first step in the alignment stage, prior to the activation of $SMAWK$, and the rows which become redundant identified and removed.

Altogether, this additional $O(L)$ work does not change the original $O(L)$ time complexity of the alignment stage.

5 Conclusions and Open Problems

The Sparse LCS Common Substring Alignment algorithm described in this paper consists of an $O(nL_y)$ time encoding stage and an $O(L)$ time alignment stage. It is intended for those applications where the source strings contain shared and repeated substrings. Note that we just leverage on the appearance of common substrings in the source strings. It is an open problem whether a more efficient algorithm exists when the target string contains encoded repetitions as well as the source strings.

Another remaining open challenge is to try to extend the solutions presented in this paper to more general metrics, such as Edit Distance.

References

[1] A. Apostolico, String editing and longest common subsequences. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, Vol. 2, 361–398, Berlin, 1997. Springer Verlag.

[2] Apostolico A., and C. Guerra, The longest common subsequence problem revisited. *Algorithmica*, **2**, 315–336 (1987).

[3] Aggarwal, A., M. Klawe, S. Moran, P. Shor, and R. Wilber, Geometric Applications of a Matrix-Searching Algorithm, *Algorithmica*, **2**, 195-208 (1987).

[4] Benson, G., A space efficient algorithm for finding the best nonoverlapping alignment score, *Theoretical Computer Science*, **145**, 357–369 (1995).

[5] Crochemore, M., G.M. Landau, and M. Ziv-Ukelson, A Sub-quadratic Sequence Alignment Algorithm for Unrestricted Cost Matrices, *Proc. Symposium On Discrete Algorithms*, 679–688 (2002).

[6] Eppstein, D., Z. Galil, R. Giancarlo, and G.F. Italiano, Sparse Dynamic Programming I: Linear Cost Functions, *JACM*, **39**, 546–567 (1992).

[7] Gusfield, D., Algorithms on Strings, Trees, and Sequences. *Cambridge University Press*, (1997).

[8] Hirshberg, D.S., "Algorithms for the longest common subsequence problem", *JACM*, **24**(4), 664–675 (1977).

[9] Hunt, J. W. and T. G. Szymanski. "A fast algorithm for computing longest common subsequences." *Communications of the ACM* , **20**, 350–353 (1977).

[10] Kannan, S. K., and E. W. Myers, An Algorithm For Locating Non-Overlapping Regions of Maximum Alignment Score, *SIAM J. Comput.*, **25**(3), 648–662 (1996).

[11] Landau, G.M., and M. Ziv-Ukelson, On the Shared Substring Alignment Problem, *Proc. 11th Annual ACM-SIAM Symposium on Discrete Algorithms*, 804-814 (2000).

[12] Landau, G.M., and M. Ziv-Ukelson, On the Common Substring Alignment Problem, *Journal of Algorithms*, **41**(2), 338–359 (2001)

[13] Monge, G., Déblai et Remblai, *Mémoires de l'Academie des Sciences*, Paris (1781).

[14] Myers, E. W., "Incremental Alignment Algorithms and their Applications," *Tech. Rep. 86-22, Dept. of Computer Science, U. of Arizona.* 1986.

[15] Schmidt, J.P., All Highest Scoring Paths In Weighted Grid Graphs and Their Application To Finding All Approximate Repeats In Strings, *SIAM J. Comput*, **27**(4), 972–992 (1998).