A Sub-quadratic Sequence Alignment Algorithm for Unrestricted Scoring Matrices [*]

Maxime Crochemore [†]
University of Marne-la-Vallée
and
King's College London

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

Michal Ziv-Ukelson [§]
Haifa University
and
IBM T.J.W Research Center

Abstract

Given two strings of size n over constant alphabet, the classical algorithm for computing the similarity between two sequences [47, 50] uses a dynamic programming matrix, and compares the two strings in $O(n^2)$ time. We address the challenge of computing the similarity of two strings in sub-quadratic time, for metrics which use a scoring matrix of unrestricted weights. Our algorithm applies to both *local* and *global* similarity computations.

The speed-up is achieved by dividing the dynamic programming matrix into variable sized blocks, as induced by Lempel-Ziv parsing of both strings, and utilizing the inherent periodic nature of both strings. This leads to an $O(n^2/\log n)$ algorithm for an input of constant alphabet size. For most texts, the time complexity is actually $O(hn^2/\log n)$ where $h \leq 1$ is the entropy of the text.

We also present an algorithm for comparing two *run-length* encoded strings of length m and n, compressed into m' and n' runs respectively, in $O(m'n+n'm)$ complexity. This result extends to all distance or similarity scoring schemes which use an additive gap penalty.

Keywords: alignment, dynamic programming, text compression, run length.

1 Introduction

The rapid progress in large-scale DNA sequencing opens a new level of computational challenges involved in storing, organizing and analyzing the wealth of biological information. One of the most interesting new fields that the availability of the complete genomes has created is that of genome comparison (the genome is all of the DNA sequence passed from one generation to the next).

[*] This paper is the extended journal version of [11].

[†] Institut Gaspard-Monge, Université de Marne-la-Vallée, Cité Descartes, Champs-sur-Marne, 77454 Marne-la-Vallée Cedex 2, France, `http://www-igm.univ-mlv.fr/~mac/`; partially supported by CNRS, NATO Science Programme, and Wellcome Trust Foundation.

[‡] 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 NATO Science Programme grant PST.CLG.977017, by the Israel Science Foundation grants 173/98 and 282/01, by the FIRST Foundation of the Israel Academy of Science and Humanities, and by IBM Faculty Partnership Award.

[§] Department of Computer Science, Haifa University, Haifa 31905, Israel; On Education Leave from the IBM T.J.W. Research Center; email: `michal@cs.haifa.ac.il`; partially supported by by the Israel Science Foundation grants 173/98 and 282/01, and by the FIRST Foundation of the Israel Academy of Science and Humanities.

Comparing complete genomes can give deep insights about the relationship between organisms, as well as shedding light on the function of specific genes in each single genome. The challenge of comparing complete genomes necessitates the creation of additional, more efficient computational tools.

One of the most common problems in biological comparative analysis is that of aligning two long bio-sequences in order to measure their similarity. The alignment is classically based on the transformation of one sequence into the other via operations of substitutions, insertions, and deletions (indels). Their costs are given by a scoring matrix.

Definition 1 (Gusfield [25]) Global Alignment Problem. *Given a pairwise scoring matrix δ over the alphabet Σ, the similarity of two strings A and B is defined as the value* **maxV** *of the alignment of A and B that maximizes the total alignment value.*

- The score value $maxV$ is called the *optimal global alignment value* of A and B.
- A description of a $maxV$-scoring transformation of A into B is called a *global alignment trace.*

In many applications, two strings may not be highly similar in their entirety but may contain regions that are highly similar. The task is to find and extract a pair of regions, one from each of the two given strings, that exhibit high similarity. This is called the *local alignment* or *local similarity* and is defined formally below.

Definition 2 (Gusfield [25]) Local alignment problem. *Given two strings A and B, find substrings α and β of A and B, respectively, whose similarity (optimal global alignment value) is maximum over all pairs of substrings from A and B.*

- The score value $maxL$ of the most similar pair of substrings α and β is called the *optimal local alignment value.*
- The description of a $maxL$-scoring transformation of substring α into substring β is called a *local alignment trace.*

Given two strings of size n, both global and local similarity problems can be solved in $O(n^2)$ time by dynamic programming [25], [36], [50]. After the optimal similarity scores have been computed, both global alignment and local alignment traces can be reported in time linear with their size [10, 26, 28].

1.1 Results

In this paper data compression techniques are employed to speed up the alignment of two strings. The compression mechanism enables the algorithm to adapt to the data and to utilize its repetitions. The periodic nature of the sequence is quantified via its *entropy*, denoted by the real number h, $0 < h \leq 1$. Entropy is a measure of how “compressible” a sequence is (see [7],[13]), and is small when there is a lot of order (i.e, the sequence is repetitive and therefore more compressible) and large when there is a lot of disorder (see Section 2.2).

Our results include the following algorithms.

1.1.1 Global Alignment

- We present an $O(n^2/\log n)$ algorithm for computing the optimal global alignment value of two strings, of size n each, over a constant alphabet (see Section 3). The algorithm is even faster when the sequence is compressible. In fact, for most texts, the complexity of our algorithm is actually $O(hn^2/\log n)$.
- After the optimal score is computed, a single alignment trace corresponding to the optimal score can be recovered in time complexity that is linear with the size of the trace (see Section 4).
- For global alignment over "discrete" scoring matrices, we explain how the space complexity can be reduced to $O(h^2n^2/(\log n)^2)$, without impairing the $O(hn^2/\log n)$ time complexity (see Section 5).

1.1.2 Local Alignment

- We describe a sub-quadratic, $O(hn^2/\log n)$ algorithm for the computation of the optimal local alignment value of two strings over a constant alphabet (see Section 6.1).
- Given an index on A where substring α ends and an index on B where substring β ends, an *optimal local alignment trace* can be reported in time linear with its size (see Section 6.2).

1.1.3 Comparing Two Run-Length Encoded Strings

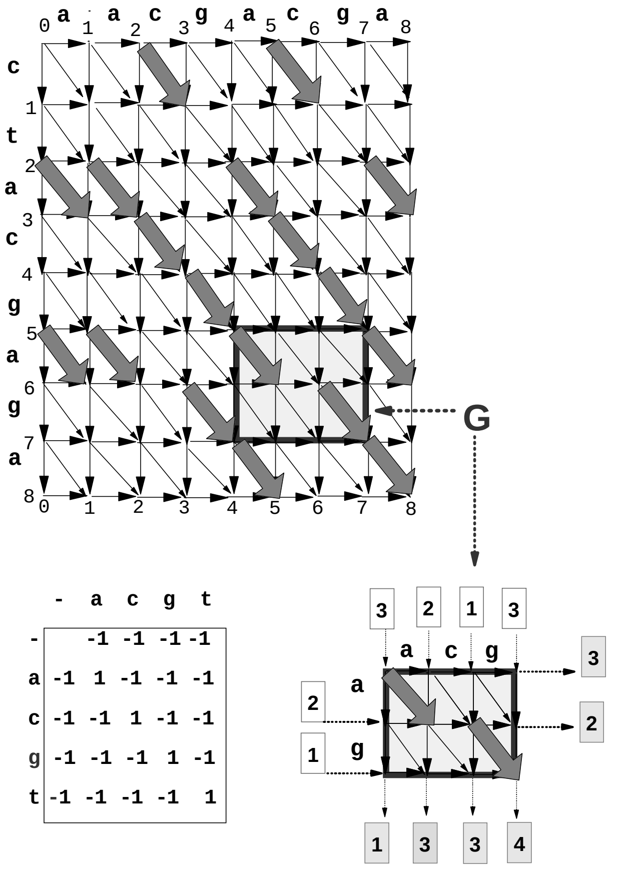
- We give an algorithm for comparing two *run-length* encoded strings of length m and n, compressed to m' and n' runs respectively, using any distance or similarity scoring scheme with additive gaps, in $O(m'n + n'm)$ complexity (see Section 7).

The algorithms described in this paper are the first to approach *fully LZ compressed* (both source and target strings are compressed) string alignment. The methods given in this paper can also be used by applications where both input strings are stored or transmitted in the form of an $LZ78$ or LZW compressed sequence, thus providing an efficient solution to the problem of how to compare two strings without having to decompress them first.

Remark:

For the sake of presentation simplicity, we assume, throughout the description of the global alignment and the local alignment solutions, that both input strings A and B are of the same size n, and that both sequences share the same entropy h. For the case of comparing string A of size m and entropy $0 < h_A \leq 1$ with string B of size n and entropy $0 < h_B \leq 1$, the results of subsections 1.1.1 and 1.1.2 are as follows.

- $O(mn(h_A/\log m + h_B/\log n))$ time and space complexity for both global alignment and local alignment, replaces the $O(hn^2/\log n)$ result.
- $O(h_A h_B mn/\log m \log n)$ space complexity for global alignment over "discrete" scoring matrices, replaces the $O(h^2n^2/(\log n)^2)$ result.



	-	a	c	g	t
-		-1	-1	-1	-1
a	-1	1	-1	-1	-1
c	-1	-1	1	-1	-1
g	-1	-1	-1	1	-1
t	-1	-1	-1	-1	1

Figure 1: The alignment graph for comparing strings A = "*ctacgaga*" and B = "*aacgacga*". The scoring scheme matrix δ is shown in the lower left corner of the figure. The highest scoring global alignment paths originate in vertex (0,0), end in vertex (8,8) and have a total weight of 3. The highest scoring local alignment path has a total weight of 5 and corresponds to the alignment of substrings a = "acgaga" and b = "acgacga". A sub-graph G corresponding to the block for comparing substrings a = "*ag*" and b = "*acg*" is shown in the lower-right corner of the figure. Also specified are the values I for the entries of the input border for G (in white-shaded rectangles), and the values O of the output border of G (in grey-shaded rectangles), as set during a local alignment computation.

1.2 Previous Results

The only previously known sub-quadratic global alignment string comparison algorithm, by Masek and Paterson [40], is based on the Four Russians paradigm. The "Four Russians" algorithm divides the dynamic programming table into uniform sized ($\log n$ by $\log n$) blocks, and uses table lookup to obtain an $O(n^2/\log n)$ time complexity string comparison algorithm, based on two assumptions. One is that the sequence elements come from a constant alphabet. The other, which they denote the "discreteness" condition, is that the weights (of substitutions and indels) are all rational numbers.

Our algorithms present a new approach and are better than the above algorithm in two aspects. First, the algorithms presented here are faster for compressible sequences. For such sequences, the complexity of our algorithms is $O(hn^2/\log n)$, where $h \leq 1$ is the entropy of the sequence.

Second, our algorithms are general enough to support scoring schemes with real number weights. For many scoring schemes, the rational number weights supported by Masek and Paterson's algorithm do not suffice. For example, the entries of PAM similarity matrices, as well as BLOSUM evolutionary distance matrices, are defined to be real numbers, computed as log-odds ratios - and therefore could be irrational.

The paper by Masek and Paterson concludes with the following statement: "The most important problem remaining is finding a better algorithm for the finite (in our terms constant) alphabet case without the discreteness condition". Here, more than twenty years later, this important open question will finally be answered!

These advantages are based on the following facts. First, our algorithm does not require any precomputation of lookup-tables, and therefore can afford more flexible weight values. Also, instead of dividing the dynamic programming matrix into uniform-sized blocks as did Masek and Paterson, we employ a variable-sized block partition, as induced by Lempel-Ziv factorization of both source and target. The common denominator between blocks, maximized by the compression technique, is then re-cycled and used for computing the relevant information for each block in time which is linear with the length of its sides. In this sense, the approach described in this paper can be viewed as another example of speeding up dynamic programming by keeping and computing only a relevant subset of important values, as demonstrated in [17], [18], [34] and [48]. A similar unbalanced strategy has been successfully used for square detection in strings [12] to speed up the original algorithm based on a divide-and-conquer approach [37].

2 Preliminaries

2.1 The Alignment Graph

The dynamic programming solution to the string comparison computation problem can be represented in terms of a weighted alignment graph [25] (See Figure 1).

The weight of a given edge can be specified directly on the grid graph, or as is frequently the case in biological applications, is given by a scoring matrix, denoted δ, which specifies the substitution score for each pair of characters and the insertion/deletion scores for each character from the alphabet.

The two widely used classes of scoring schemes are distance scoring, in which the objective is to minimize the total alignment score, and similarity scoring, in which the objective is to maximize the total alignment score. Within these classes, scoring schemes are further characterized by the treatment of gap scores. A *gap* is the result of the deletion of one or more consecutive characters in one of the sequences. *Additive* gap scores assign a constant weight to each of the consecutive characters. For other gap functions which have been found useful for biological sequences, see [25]. The solutions in this paper assume a scoring scheme with additive gap scores.

Global Alignment via Dynamic Programming The classical dynamic programming algorithm for the global comparison of two strings will set the value at each vertex (i, j) of the alignment graph, row by row in a left to right order, to the score between the first i characters of A and the first j characters of B, using the following recurrence:

$$\begin{aligned} V(i,j) = max[& V(i,j-1) + \delta(\epsilon, B_j), \\ & V(i-1,j) + \delta(A_i, \epsilon), \\ & V(i-1,j-1) + \delta(A_i, B_j)]. \end{aligned}$$

Computing and setting the values of all vertices in the alignment graph, using the above recurrence, takes $O(n^2)$ time and space. After the values at each vertex of the alignment graph have been computed and set, the optimal global alignment value $maxV$ is found at vertex (n,n) of the graph.

If each vertex in the alignment graph stores the operation (insertion, deletion, substitution) selected when its value was set, then a global alignment trace, corresponding to an optimal path in the alignment graph, can be recovered in time linear with its size, starting from vertex (n,n) which contains the maximal score, and tracing the edges back up to vertex $(0,0)$ in the graph.

Local Alignment via Dynamic Programming Smith and Waterman [50], [25] showed that essentially the same $O(|A||B|)$ dynamic programming solution can be used for computing local similarity, provided that the score of the alignment of two empty strings is defined as 0, and only pairs whose alignment scores are above 0 are of interest. The Smith-Waterman algorithm for computing local similarity computes the following recurrence, which includes 0 as an additional option, and thus restricts the scores to non-negative values:

$$\begin{aligned} L(i,j) = max[0, & L(i,j-1) + \delta(\epsilon, B_j), \\ & L(i-1,j) + \delta(A_i, \epsilon), \\ & L(i-1,j-1) + \delta(A_i, B_j)]. \end{aligned}$$

The method to compute the optimal local alignment value $maxL$ is to compute all alignment graph vertex values $L(i,j)$ in $O(n^2)$ time and space, and then find the largest value at *any* vertex on the table, say at vertex (i_{end}, j_{end}).

Given the vertex (i_{end}, j_{end}) which carries the score $maxL$, the corresponding substrings α and β giving the optimal local alignment of A and B are obtained in time linear with their size, by using the stored operations (insertion, deletion, substitution) to trace back the edges from vertex (i_{end}, j_{end}) until a vertex (i_{start}, j_{start}) is reached that has value zero. Then the optimal local alignment substrings for vertex (i_{end}, j_{end}) are $\alpha = A[i_{start} \ldots i_{end}]$ and $\beta = B[j_{start} \ldots j_{end}]$ [25].

2.2 A Block Partition of the Alignment Graph based on LZ78 Factorization

The traditional aim of text compression is the efficient use of resources such as storage and bandwidth. Here, we will compress the sequences in order to speed up the alignment process. Note that this approach, denoted "acceleration by text-compression", has been recently applied to a related problem - that of *exact string matching* [31], [39], [49].

It should also be mentioned that another related problem - that of exact string matching in compressed text without decoding it, which is often referred to as "compressed pattern matching", has been studied extensively [4], [19] [45]. Along these lines, string search in compressed text was developed for the compression paradigm of LZ78 [54], and its subsequent variant LZW [52], as described in [32], [46]. A more challenging problem is that of "fully compressed" pattern matching when both the pattern and text strings are compressed [22], [23].

For the LZ78-LZW paradigm, compressed matching has been extended and generalized to that of *approximate pattern matching* (finding all occurrences of a short sequence within a long one allowing up to k changes) in [30], [44].

The LZ compression methods are based on the idea of self reference: while the text file is scanned, substrings or phrases are identified and stored in a dictionary, and whenever, later in the process, a phrase or concatenation of phrases is encountered again, this is compactly encoded by suitable pointers [35], [53], [54].

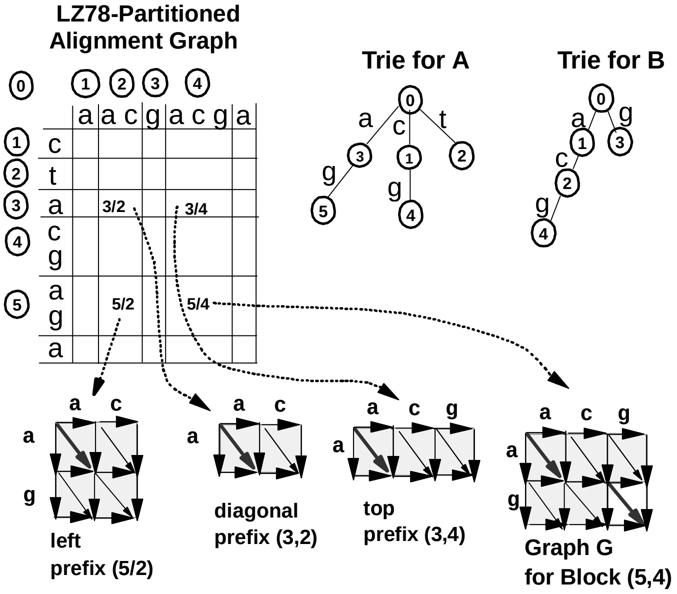


Figure 2: The block partition of the alignment graph, and the tries corresponding to $LZ78$ parsing of strings A = "*ctacgaga*" and B = "*aacgacga*". Note that for the block G in this example, α = "ag", β = "acg", $\ell_r = 2$, $\ell_c = 3$, $i = 5$ and $j = 4$. (The new cell of G, which does not appear in any of the prefix blocks, is the rightmost cell at the bottom row of G, and can be distinguished by its white color.) This figure continues Figure 1.

Of the several existing versions of the method, we will use the ones which are denoted $LZ78$ family [52], [54]. The main feature which distinguishes $LZ78$ factorization from previous LZ compression algorithms is in the choice of codewords. Instead of allowing pointers to reference any string that has appeared previously, the text seen so far is parsed into phrases, where each phrase is the longest matching phrase seen previously plus one character. For example, the string "S = aacgacg" is divided into fours phrases: a, ac, g, acg. Each phrase is encoded as an index to its prefix, plus the extra character. The new phrase is then added to the list of phrases that may be referenced.

Since each phrase is distinct from others, the following upper bound applies to the possible number of phrases obtained by $LZ78$ factorization.

Theorem 2.2.1 (Lempel and Ziv 1976 [35]) *Given a sequence S of size n over a constant alphabet. The maximal number of distinct phrases in S is $O(\frac{n}{\log n})$.*

Even though the upper bound above applies to any possible sequence over a constant alphabet, it has been shown that in many cases we can do better than that.

Intuitively, the $LZ78$ algorithm compresses the sequence because it is able to discover some repeated patterns. Therefore, in order to compute a tighter upper bound on the number of phrases obtained by $LZ78$ factorization for "compressible" sequences, the repetitive nature of the sequence needs to be quantified. One of the fundamental ideas in information theory is that of *entropy*, denoted by the real number h, $0 < h \leq 1$, which measures the amount of disorder or randomness, or inversely, the amount of order or redundancy in a sequence. Entropy is small when there is a lot of order (i.e, the sequence is repetitive) and large when there is a lot of disorder. The entropy of a sequence should ideally reflect the ratio between the size of the sequence after it has been compressed, and the length of the uncompressed sequence.

The number of distinct phrases obtained by $LZ78$ factorization has been shown to be $O(hn/\log n)$ for most texts [7], [13], [35], [54]. Note that for any text over a constant alphabet, the upper bound above still applies by setting h to 1.

3 Computing the Optimal Global Similarity Value

3.1 Definitions and Basic Observations

The alignment graph will be partitioned as follows. Strings A and B will be parsed using $LZ78$ factorization. This induces a partition of the alignment graph for comparing A with B into variable-sized blocks (see Figure 2). Each block will correspond to a comparison of an LZ phrase of A with an LZ phrase of B.

Let xa denote a phrase in A obtained by extending a previous phrase x of A with character a, and yb denote a phrase in B, obtained by extending a previous phrase of B with character b.

From now on we will focus on the computations necessary for a single block of the alignment graph.

Consider the block G which corresponds to the comparison of xa and yb. We define *input border* I as the left and top borders of G, and *output border* O as the bottom and right borders of G. (The node entries on the input border are numbered in a clockwise direction, and the node entries on the output border are numbered in a counter-clockwise direction.)

Rather than filling in the values of each vertex in G, as does the classical dynamic programming algorithm, the only values computed for each block will be those on its I/O borders (see Figures 1 and 5A). Intuitively, this is the reason behind the efficiency gain.

Let ℓ_r denote the number of rows in G, $\ell_r = |xa|$. Let ℓ_c denote the number of columns in G, $\ell_c = |yb|$. Let $t = \ell_r + \ell_c$. Clearly, $|I| = |O| = t$.

We define the following three *prefix* blocks of G.

1. The *left prefix* of G denotes the block comparing phrase xa of A and phrase y of B.

2. The *diagonal prefix* of G denotes the block comparing phrase x of A and phrase y of B.

3. The *top prefix* of G denotes the block comparing phrase x of A and phrase yb of B.

Observation 1 When traversing the blocks of an $LZ78$ parsed alignment graph in a left-to-right, top-to-bottom order, the blocks for the left prefix, diagonal prefix and top prefix of G are encountered prior to block G.

Note that the graph for the left prefix of G is identical to the subgraph of G containing all columns but the last one. More specifically, both the structure and the weights of edges of these two graphs are identical, but the weights to be assigned to vertices during the similarity computation may vary according to the input border values. Similarly, for the top prefix and diagonal prefix graphs. The only new cell in G, which does not appear in any of its prefix block graphs, is the cell for comparing a and b. This new cell consists of one new vertex and three new edges.

3.2 I/O Propagation Across G.

The work for each block consists of two stages (a similar approach is shown in [8, 29, 34]).

$DIST$ matrix

$$\begin{array}{l|rrrrrr}
I_0 = 1 & 0 & -1 & -2 & -3 & \triangle & \triangle \\
I_1 = 2 & -1 & -1 & -2 & -1 & -3 & \triangle \\
I_2 = 3 & -2 & 0 & 0 & 1 & -1 & -3 \\
I_3 = 2 & \triangle & -2 & -2 & 0 & -2 & -2 \\
I_4 = 1 & \triangle & \triangle & -2 & 0 & -1 & -1 \\
I_5 = 3 & \triangle & \triangle & \triangle & -2 & -1 & 0
\end{array}$$

OUT matrix

$$\begin{array}{rrrrrr}
1 & 0 & -1 & -2 & -\infty & -\infty \\
1 & 1 & 0 & 1 & -1 & -\infty \\
1 & 3 & 3 & 4 & 2 & 0 \\
-12 & 0 & 0 & 2 & 0 & 0 \\
-13 & -13 & -1 & 1 & 0 & 0 \\
-14 & -14 & -14 & 1 & 2 & 3
\end{array}$$

$$\begin{array}{cccccc}
O_0 & O_1 & O_2 & O_3 & O_4 & O_5 \\
1 & 3 & 3 & 4 & 2 & 3
\end{array}$$

column numbers

$$\begin{array}{cccccc}
\mathbf{0} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{5}
\end{array}$$

Figure 3: The *DIST* matrix which corresponds to the subsequences "acg", "ag", the *OUT* matrix obtained by adding the values of I to the rows of *DIST*, and the O containing the row maxima of *OUT*. This figure continues Figures 1 and 2.

1. *encoding*: Study the structure of G and represent it in an efficient way.
2. *propagation*: Given I and the encoding of G, constructed in the previous stage, compute O for G.

The structure of G is encoded by computing weights of optimal paths connecting each entry of its input border with each entry of its output border. The following $DIST$ matrix is used (see Figure 3).

Definition 3 *$DIST[i, j]$ stores the weight of the optimal path from entry i of the input border of G to entry j of its output border.*

$DIST$ matrices have also been used in [5], [8], [29], [34] and [48].

Given input row I and the $DIST$ for G, the weight of output row vertex O_j can be computed as the maximum among the sums $I_r + DIST[r, j]$, if there is indeed a path connecting input border entry r with output border entry j.

Vertex O_j is the maximum of column j of the following OUT matrix, which merges the information from input row I and $DIST$. (See Figure 3).

Definition 4 $OUT[i, j] = I_i + DIST[i, j]$.

Aggarwal and Park [3] and Schmidt [48] observed that $DIST$ matrices are Monge arrays [43].

Definition 5 *A matrix* $M[0 \ldots m, 0 \ldots n]$ *is* **Monge** *if either condition 1 or 2 below holds for all* $a, b = 0 \ldots m; c, d = 0 \ldots n$:

1. **convex condition:** $M[a, c] + M[b, d] \leq M[b, c] + M[a, d]$ *for all* $a < b$ *and* $c < d$.
2. **concave condition:** $M[a, c] + M[b, d] \geq M[b, c] + M[a, d]$ *for all* $a < b$ *and* $c < d$.

Since $DIST$ is Monge, so is OUT, which is a $DIST$ with constants added to its rows.

An important property of Monge arrays is that of being totally monotone.

Definition 6 *A matrix* $M[0 \ldots m, 0 \ldots n]$ *is* **totally monotone** *if either condition 1 or 2 below holds for all* $a, b = 0 \ldots m;\ c, d = 0 \ldots n$:

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

Note that the Monge property implies total monotonicity, but the converse is not true. Therefore, both $DIST$ and OUT are totally monotone by the concave condition.

Aggarwal et al [2] gave a recursive algorithm, nicknamed *SMAWK* in the literature, which can compute in $O(n)$ time all row and column maxima of an $n \times n$ totally monotone matrix, by querying only $O(n)$ elements of the array. Hence, one can use *SMAWK* to compute the output row O by querying only $O(n)$ elements of OUT. Clearly, if both the full $DIST$ and all entries of I are available, then computing an element of OUT is O(1) work.

For various solutions to related problems, which also utilize Monge and Total Monotonicity properties, we refer the interested reader to [15], [16], [20], [21], [33] and [34]. In order to efficiently utilize these properties here, we need to address the following two problems.

1. How to efficiently compute $DIST$ and represent it in a format which allows direct access to its entries. This will be done in Section 3.4.
2. *SMAWK* is intended for a full, rectangular matrix. However, both $DIST$ and its corresponding OUT are not rectangular. Since paths in an alignment graph can only assume a left-to-right, top-to-bottom direction, connections between some input border vertices and some output border vertices are impossible. Therefore, the matrices are missing both a lower left triangle and upper right triangle (see Figure 3). The question is addressed in Section 3.3.

3.3 Addressing the Rectangle Problem

The undefined entries of OUT can be complemented in constant time each, as follows.

1 The missing upper right triangle entries can be completed by setting the value of any entry $OUT[i, j]$ in this triangle to $-\infty$.

2 Let k denote the maximal absolute value of a score in δ. The missing lower left triangle entries can be completed by setting the value of any $OUT[i,j]$ in this triangle to $-(n+i+1)*k$.

Lemma 3.3.1 *Complementing the undefined entries as described above preserves the concave total monotonicity condition of OUT, and does not introduce new row-maxima.*

Proof:

1 Upper Right Triangle: All similarity scores in the alignment graph are finite. Therefore, no new column maxima are introduced. Suppose $OUT[a,c] \leq OUT[b,c]$, $a < b$, and $OUT[a,c]$ has been set to $-\infty$. Due to the shape of the redefined upper-right triangle, once a $-\infty$ value in row a is encountered, all future values in row a are also $-\infty$. The future values of row b could either be finite or $-\infty$. Therefore, $OUT[a,d] \leq OUT[b,d]$ for all $d > c$.

2 Lower Left Triangle: The worst score appearing in the alignment graph is lower bounded by $-nk$. Since i is always greater than or equal to zero, the complemented values in the lower left triangle are upper-bounded by $-(n+1)*k$ and no new column maxima are introduced. Also, for any complemented entry $OUT[b,c]$ in the lower left triangle, $OUT[b,c] < OUT[a,c]$ for all $a < b$, and therefore the concave total monotonicity condition holds. ▮

3.4 Incremental Update of the new *DIST* Information for G

In this section we show how to efficiently compute the new *DIST* information for G, using the *DIST* representations previously computed for its prefix blocks, plus the information of its new cell.

When processing a new block G, we compute the scores of t new optimal paths, leading from the input border to the new vertex (ℓ_r, ℓ_c) in the new cell of G in its lowest, rightmost corner. These values correspond to column ℓ_c of the *DIST* matrix for G, and can be computed as follows.

Entry $[i]$ in column ℓ_c of the *DIST* for G contains the weight of the optimal path from entry i in the input border of G to vertex (ℓ_r, ℓ_c). This path must go through one of the three vertices $(\ell_r - 1, \ell_c)$, $(\ell_r - 1, \ell_c - 1)$ or $(\ell_r, \ell_c - 1)$. Therefore, the weight of the optimal path from entry i in the input border of G to (ℓ_r, ℓ_c) is equal to the maximum among the following three values:

1 Entry $[i]$ of column $\ell_c - 1$ of the *DIST* for the left prefix of G, plus the weight of the horizontal edge leading into (ℓ_r, ℓ_c).

2 Entry $[i]$ of column $\ell_c - 1$ of the *DIST* for the diagonal prefix of G, plus the weight of the diagonal edge leading into (ℓ_r, ℓ_c).

3 Entry $[i]$ of column ℓ_c of the *DIST* for the top prefix of G, plus the weight of the vertical edge leading into (ℓ_r, ℓ_c).

3.4.1 Maintaining Direct Access to *DIST* Columns

In order to compute an entry of OUT in constant time during the execution of *SMAWK*, direct access to *DIST* entries is necessary. This is not straightforward, since as shown in the previous section, for each block only one new *DIST* column has been computed and stored. All other columns besides column ℓ_c of the *DIST* for G need to be obtained from G's prefix ancestor blocks.

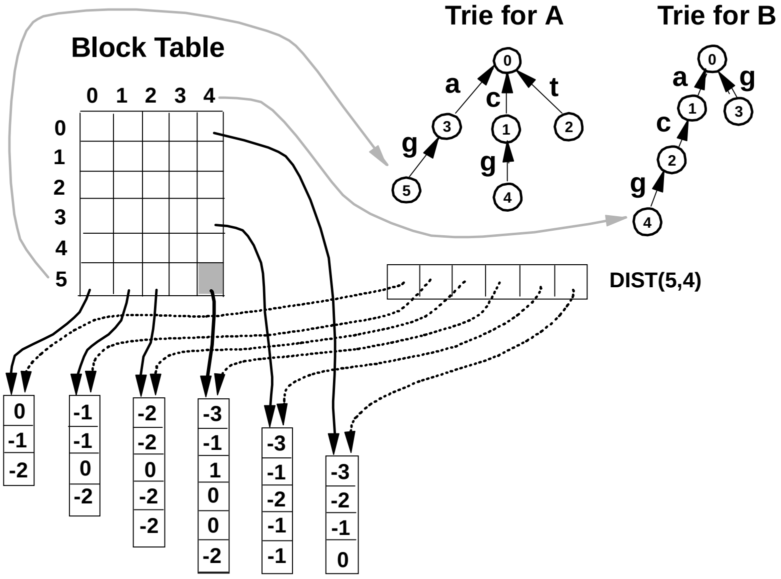


Figure 4: A table containing an entry for each block of the alignment graph. Entry (i, j) of the table represents the block which corresponds to node i in the trie for A and node j in the trie for B. The entry for each block in the table points to the start of its new $DIST$ column. Also shown is the vector which contains pointers to all columns of the $DIST$ for block $(5, 4)$, as obtained from its ancestor prefix blocks. This figure continues Figures 1, 2 and 3.

Therefore, before the execution of $SMAWK$ begins, a vector with pointers to all $t+1$ columns of the $DIST$ for G is constructed (see Figure 4). This vector is no longer needed after the computations for G have been completed, and its space can be freed.

The pointers to all columns of the $DIST$ for G are assembled as follows. Column ℓ_c is set to the newly constructed vector for G. All columns of indices smaller than ℓ_c are obtained via ℓ_c recursive calls to left prefix blocks of G. All columns of indices greater than ℓ_c are obtained via ℓ_r recursive calls to top prefix blocks of G.

3.4.2 Querying a Prefix Block and Obtaining its *DIST* Column in Constant time

The $LZ78$ phrases form a trie (see Figure 2), and the string to be compressed is encoded as a sequence of names of prefixes of the trie. Each node in the trie contains the serial number of the phrase it represents. Since each block corresponds to a comparison of a phrase from A with a phrase from B, each block will be identified by a pair of numbers, composed of the serial numbers for its corresponding phrases in the tries for A and B.

Another data structure to be constructed is a Block Table (see Figure 4), containing an entry for each partitioned block of the alignment graph. The entry for each block in the table points to the start of its new $DIST$ column, and can be directly accessed via the block's phrase number index pair.

The left prefix of G can be identified in constant time as a pair of phrase numbers, the first identical to the serial number of xa, and the second corresponding to the serial number of y, which is the direct ancestor of yb in the trie for B. Similarly, the top prefix of G can be identified in constant time. Given the pair of identification numbers for a block, a pointer to the corresponding *DIST* column can then be directly obtained from the Block Table.

Time and Space Analysis Assuming sequence size n and sequence entropy $h \leq 1$. The $LZ78$ factorization algorithm parses the strings and constructs the tries for A and B in $O(n)$ time. The resulting number of phrases in both A and B is $O(hn/\log n)$. The number of resulting blocks in the alignment graph is equal to the number of phrases in A times number of phrases in B, and is therefore $O(h^2n^2/(\log n)^2)$. For each block G, the following information (1–3) is computed, in time and space complexity linear with the size of its I/O borders:

1. Updating the Encoding Structure for G. The prefix blocks of G can be accessed in constant time. The vectors of *DIST* column pointers for the prefix blocks have already been freed. However, since each prefix block directly points to its newly computed *DIST* column, all values needed for the computations are still available. Since each entry of the new *DIST* column for G is set to the maximum among up to three sums of pairs, the new *DIST* column for G can be constructed in $O(t)$ time and space.

2. Maintaining Direct Access to *DIST* columns. Since *prefix* blocks and their *DIST* columns can be accessed in constant time, the vector with pointers to columns of the *DIST* for G can be set in $O(t)$ time.

3. Propagating I/O values across the block. Using the information computed for G, and given the I for G obtained from the O vectors for the block above G and the block to its left, the values of O for G are computed via *SMAWK* Matrix Searching in $O(t)$ time.

Total Complexity Since the work and space for each block is linear with the size of its I/O borders, the total time and space complexity is linear with the total size of the borders of the blocks. The block borders form $O(hn/\log n)$ rows of size $|B|$ each, and $O(hn/\log n)$ columns of size $|A|$ each, in the alignment graph (see Figure 2). Therefore, the total time and space complexity is $O(hn^2/\log n)$.

4 Global Similarity Optimal Alignment Trace Recovery

The recovery of an optimal global alignment trace between A and B starts at vertex (n, n). The series of block crossing paths is then traced back until vertex $(0, 0)$ is reached. For each block crossed, the internal alignment trace is reported, starting from the output border sink, and back to the optimal origin source vertex in the corresponding input border. In order to support the recovery of block-crossing paths in time linear with their size, the computation and storage of the following additional information for a given block G is required.

1. During the Propagation stage, for each entry j in the output border of G, the index of the input border entry i, which is the source of the highest scoring path to output border entry j, is saved.

2. During Encoding, an additional $O(t)$ sized vector of pointers, the *ancestors* vector, is computed for G. For any output border entry $O[j = 0 \dots t]$, $ancestors[j]$ points to the ancestor block of G for which this entry is the new vertex in its new cell. (The value of $ancestors[\ell_c]$ is set to G. All columns of indices smaller than ℓ_c are obtained via ℓ_c recursive calls to left

prefix blocks of G. All columns of indices greater than ℓ_c are obtained via ℓ_r recursive calls to top prefix blocks of G.)

3. During Encoding, G's new vertex (ℓ_r, ℓ_c) is annotated with an additional $O(t)$ sized vector of pointers, denoted *direction*. These pointers are set during the *DIST* column computation described in Section 3.4, as follows. The value of *direction*$[i]$ is set according to the direction of the last edge in the optimal path originating at entry i of G's input border and ending at vertex (ℓ_r, ℓ_c).

Given that the optimal path enters through entry j of the output border of G, the trace-back of the part of the path going through G proceeds in two stages. The first stage is a destination and origin initialization stage. This stage includes the fetching of the input row source entry i, which was stored as the origin for the highest scoring path to G's output border entry j (see 1 above). Entry i serves as the destination for the alignment trace-back. In addition, the ancestor prefix block P of G, pointed to by *ancestors*$[j]$ is fetched (see 2 above). The edge recovery begins in block P.

During the second stage, the origin and destination information computed in the first stage is used to trace back the part of the path contained in P, from entry j on P's output border (the new vertex of P), to entry i on its input border. This is done by backtracking through a dynasty of prefix ancestor blocks internal to P, using the *direction* vector computed for each of the traversed blocks (see 3 above). If *direction*$[i]$ of the traversed block specifies a horizontal edge, then the trace-back retreats to the left prefix of P, and an "insertion" operation is reported in the alignment trace. Correspondingly, "substitution" and "deletion" are reported when backtracking to diagonal and top prefix blocks. The recovery continues through a series of prefix blocks of P until the full optimal alignment trace is recovered.

Time and Space Analysis The two additional vectors for G, *direction* and *ancestors*, and the input border source entry i, can be computed and stored during the encoding and propagation stages in $O(t)$ time and space.

The work for the first stage in the trace-back can be done in constant time. In the second stage, each edge in the recovered alignment path results in a traversal to a single prefix block. Since prefix blocks and their corresponding direction vectors can be accessed in constant time, a highest scoring global alignment between strings A and B can be recovered in additional time linear in its size, using the $O(hn^2/\log n)$ storage which was allocated during the encoding and propagation stages.

5 Reducing the Space Complexity

When computing the optimal global alignment value with scoring matrices which follow the "discreteness" condition (see Section 1), the *efficient alignment stage algorithm* described in [34] can be extended to support full propagation from the leftmost and upper boundaries to the bottom and right most boundaries of G.

This extended propagation algorithm can then be used to compute the values of the global alignment O for G, given the I for G and a minimal encoding of the *DIST* for G. The advantage of this minimal encoding of *DIST* is that rather than saving an $O(t)$ sized *DIST* column per block, we only need to save a constant number of values per block. The encoding for the new *DIST* column of each block can be computed and stored in constant time and space from the information stored for the left, diagonal and top prefix blocks of G, using the technique described in Section 6 of [48].

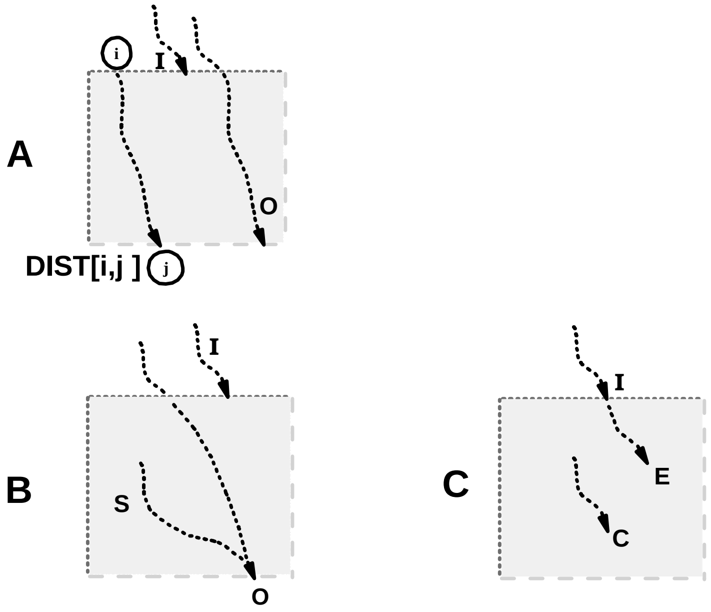


Figure 5: **A.** The I/O path weight vectors computed for each block in the global alignment solution. $DIST[i,j]$ will be set to the highest scoring path connecting vertex i in the input border with vertex j in the output border. **B,C.** The vectors of optimal path weights considered for the local alignment computation.

This reduces the space complexity to $O(h^2n^2/(\log n)^2)$, while preserving the $O(hn^2/\log n)$ time complexity.

6 The Local Alignment Algorithm

6.1 Computing the Optimal Local Similarity Value

When computing the optimal local similarity value, an optimal path could either be contained entirely in one block (type C), or could be a block-crossing path (see figure 5). A block crossing path consists of a (possibly empty) S-path, followed by any number of paths leading from the input border of a block to its output border, and ending in an E-path with a highest scoring last vertex. Since an optimal path could begin inside any block, vector O needs to be updated to consider the additional paths originating inside G. Also, since an optimal path could end inside any block, extra bookkeeping is needed in order to keep track of the highest scoring paths ending in each block.

Therefore, in addition to the $DIST$ described in Section 3, we compute for each block G the following data structures (see Figures 5B and 5C).

1. E is a vector of size t. $E[i]$ contains the value of the highest scoring path which starts at vertex i of the input border of G and ends inside G. $E[i]$ is computed as the maximum between $E[i]$ for the left prefix of G, $E[i]$ for the top prefix of G, and $DIST[i,\ell_c]$.

2. S is a vector of size t. $S[i]$ contains the value of the highest scoring path which starts inside G and ends at vertex i of the output border of G.

The only new value computed for S is the local alignment scores for the new vertex of G, $S[\ell_c]$. Given the scores $S[\ell_c - 1]$ obtained from the *diagonal prefix*, $S[\ell_c - 1]$ obtained from the *left prefix* and $S[\ell_c]$ obtained from the *top prefix* of G, and the weights of the 3 edges leading into vertex (ℓ_r, ℓ_c), $S[\ell_c]$ can be computed in $O(1)$ time complexity, using the recursion given in Section 2.1.

The values of all other entries of S are then set as follows. The first ℓ_c values of S are copied from the first ℓ_c values of the S computed for the left prefix of G. The last ℓ_r values are copied from the last ℓ_r values of the S vector for the top prefix of G.

3. C is the value of the highest scoring path contained in G, that is, the highest scoring path which originates inside G and ends inside G. C is computed as the maximum between the C value for the left prefix of G, the C value for the top prefix of G, and the newly computed $S[\ell_c]$ as described above.

The S vector computed for G is used to update the values of the output border O, while E and C will be used to compute the weight of the highest scoring path ending in G.

Vector O is first computed from the I and $DIST$ for G as described in Section 3.2. At this point entry $O[i]$ reflects the weight of the optimal path starting anywhere outside G and ending at entry i of the output border. It needs to be updated with the weights of the highest scoring paths starting inside G. This is achieved by resetting $O[i]$ to the maximum between $O[i]$ and $S[i]$.

The weight of the highest scoring path ending in G is computed as $max(Max_{i=0}^{t}\{I[i] + E[i]\},\ C)$.

After the computations for each block have been completed, the overall highest local alignment score for comparing A and B can be computed as the maximum among the values of the highest scoring path ending in each block.

Time and Space Analysis Since, as shown in Section 3.4.1, each prefix block of G can be accessed in constant time, the values of the S and E vectors for G can be computed and stored in $O(t)$ time and space, and the C value for G can be computed in constant time and space.

Given the S, E and C vectors for G, the values of O and the weight of the highest scoring path ending in G can be computed in $O(t)$ time each as described above.

The weight of the highest scoring path in the alignment graph can then be computed in an additional $O(h^2n^2/(\log n)^2)$ time as the maximum value among the best values computed for each block.

Since the work and space for each block is linear with the size of its I/O borders, the total time and space complexity of computing the optimal local alignment value is $O(hn^2/\log n)$.

6.2 Optimal Alignment Trace Recovery for the Local Alignment Solution

Similarly to the alignment trace defined in Section 4, given a $maxL$ vertex (i_{end}, j_{end}) which was obtained in the previous section, we show how to recover the optimal path ending in this vertex. by reporting a trace-back of the edges from vertex (i_{end}, j_{end}) until a start-point vertex (i_{start}, j_{start}) is reached that has value zero.

A block crossing optimal path consists of a (possibly empty) S-path, followed by any number of paths leading from the input border of a block to its output border, and ending in an E-path whose last vertex is (i_{end}, j_{end}).

The recovery starts at vertex (i_{end}, j_{end}) and continues back to the optimal path origin in three stages.

1. Recovering the E-path part.

   During encoding, whenever the $E[i]$ value of a block is updated by its new vertex, a pointer to the updating block is saved together with the new $E[i]$ value.

   During alignment recovery, given that vertex (i_{end}, j_{end}) ends an $E[i]$ path in G, the corresponding block can be fetched, and the path from its new vertex to entry i on its input border recovered, as described in Section 4.

2. Recovering all paths leading from the input border of a block to its output border.

   The part of the path contained in each one of these blocks can be recovered as described in Section 4.

3. Recovering the S-path part.

   During encoding, when computing the S-score of the new vertex of each block, the direction of the edge optimizing the score $S[\ell_c]$ of the new vertex of G, denoted $s_{direction}$, is saved with the score.

   During the termination of the propagation stage, when setting the score values for each entry in O, a field is set, indicating whether the newly set score value for this entry corresponds to a path originating inside G (an S-path), or a path crossing G. If the score corresponds to an S-path, the recovery of the S-path part utilizes the technique described in Section 4, with a slight modification. Instead of the *direction* vector, the $s_{direction}$ field is used for the edge trace-back. The recovery halts when an ancestor block is reached whose $S[\ell_c]$ value is zero.

A special case occurs when vertex (i_{end}, j_{end}) is the end point of a C-path. A C-path is, in essence, a halted S-path. During encoding, whenever the C value of a block is updated by its new vertex, a pointer to the updating block is saved together with the new C value. The recovery of the C path in G starts at the new vertex of its corresponding block and continues similarly to the S path recovery, as described in 3 above.

Time and Space Analysis In addition to the values described in Section 4, an additional $O(t)$ information (pointers to the $E[i]$ updating blocks) is computed and stored for E-paths, and an additional $O(1)$ information per block is computed and stored for C and S paths. During propagation termination, an addition $O(t)$ information is stored with the O vector.

During recovery, each edge in the recovered alignment path results in a traversal to a single prefix block, for each one of the three path parts. Both prefix blocks and their corresponding direction vectors can be accessed in constant time. Therefore, in addition to the basic $O(hn^2/\log n)$ time and space needed for computing the optimal local alignment score $maxL$, an alignment trace ending at a given $maxL$-scoring vertex can be reported in time linear with the size of the trace.

7 Applications to the Problem of Comparing Two Run Length Encoded strings

A string S is *run-length encoded* if it is described as an ordered sequence of pairs (σ, i), often denoted "σ^i," each consisting of an alphabet symbol, σ, and an integer, i. Each pair corresponds to a *run* in

S, consisting of i consecutive occurrences of σ. For example, the string $aabbbbbccc$ can be encoded as $a^2b^5c^3$. Such a run-length encoded string can be significantly shorter than the expanded string representation after efficiently encoding the integers (see [14] for example).

Run-length encoding serves as a popular image compression technique, since many classes of images (e.g., binary images in facsimile transmission or for use in optical character recognition) typically contain large patches of identically-valued pixels.

Let m and n be the lengths of two run-length encoded strings X and Y, of encoded lengths m' and n', respectively. Previous algorithms for the problem compared two run-length encoded strings using the Levenshtein Edit Distance [36] and the LCS similarity measure [26]. For the LCS metric, Bunke and Csirik [9] presented an $O(mn' + nm')$ time algorithm, while Apostolico, Landau, and Skiena [6] described an $O(m'n' \log(m'n'))$ time algorithm. Mitchell [42] has obtained an $O((d + m' + n') \log(d + m' + n'))$ time algorithm for a more general string matching problem in run-length encoded strings, where d is the number of matches of compressed characters. Both Arbell et al [1] and Mäkinen et al [38] independently obtained an $O(m'n + n'm)$ time algorithm for computing the edit distance between two run-length encoded strings for the Levenshtein distance metric.

Mäkinen et al. [38] posed as an open problem the challenge of extending these results to more general scoring schemes, since in those applications which are related to image compression, the change from a pixel value to the next is smooth. Here, we will show how to extend the results to apply them to any distance or similarity scoring scheme with additive gap scores.

In this solution, the alignment graph is also partitioned into blocks. But rather than using the $LZ78$ partition described in Section 2, each block here consists of two runs – one of X and one of Y. This results in the partition of the alignment graph into $m'n'$ blocks. The algorithm suggested also propagates accumulated scores from the left and upper boundaries of each block, to its bottom and right boundaries.

Consider the block R for comparing the run α^i of X with the run β^j of Y. An edge in R could be assigned one of three possible weight values: D(diagonal), H(horizontal) and V(vertical).

Let Δ_h and Δ_w denote the difference in row index values and column index values respectively, between entry i on the input border of R, and entry j on the output border of R.

We show how to compute $DIST[i, j]$ (which is the cost of the best scoring path from entry i in the input border of the block, to entry j in the output border of the block) in constant time, given Δ_h and Δ_w for the input and output entries, and the values D, H and V.

- $H + V \leq D$. Clearly, an optimal path from i to j can use all possible diagonal edges and only then the minimal number of remaining H and V edges necessary to reach j.

  Therefore, $DIST[i, j]$ obtains one of three values:

  1. If $\Delta_w = \Delta_h$, then $DIST[i, j] = D \times \Delta_h$.
  2. If $\Delta_w > \Delta_h$, then $DIST[i, j] = D \times \Delta_h + H \times (\Delta_w - \Delta_h)$.
  3. If $\Delta_w < \Delta_h$, then $DIST[i, j] = D \times \Delta_w + V \times (\Delta_h - \Delta_w)$.

- $H + V > D$. In this case, an optimal path never uses any diagonal edge. The path includes only the minimal number of H edges, and the minimal number of V edges necessary to reach j from i. in this case, $DIST[i, j] = H \times \Delta_w + V \times \Delta_h$.

Therefore, $DIST[i, j]$ can be easily computed in constant time when using the general scoring scheme described in Section 2.1.

Time and Space Analysis The O vector for each block is computed using *SMAWK*. Vector I for block R can be easily obtained from the O vectors for the block above R and the block to its left, in time linear with the sides of R. The "rectangle" problem can be solved similarly to Section 3.2. Therefore, any value $OUT[i, j] = I[i] + DIST[i, j]$ can be computed in constant time.

Since the work for each block is linear with the size of its I/O borders, the total time complexity is linear with the total size of the borders of the blocks, which is $O(m'n + n'm)$.

Note that alternative methods to achieve linear-time propagation across run-length compressed blocks can be obtained by adapting any of the queue algorithms described in [16, 20, 27] and [34].

Since all the relevant $DIST$ entry values are computed "on the fly" and do not need to be stored, Hirschberg's method [26] can be applied to achieve an algorithm with a space complexity which is linear with the size of the uncompressed strings.

Open Problems

The sub-quadratic sequence comparison algorithms presented in this paper are perhaps close to optimal in time complexity. However, an important concern is the space complexity of the algorithms. If only the similarity score value is required, the classical, quadratic time sequence alignment algorithm can easily be implemented to run in linear space, by keeping only two rows of the dynamic programming table alive at each step. If the recovery of either global or local optimal alignment traces is required, quadratic-time and linear-space algorithms can be obtained by applying Hirschberg's refinement to the classical sequence alignment algorithms [10, 26, 28]. We post as an open problem the challenge of further reducing the space requirement of the algorithms described in this paper, without impairing their sub-quadratic time complexity.

Acknowledgement

We are grateful to Dan Gusfield for a helpful discussion.

References

[1] O. Arbell, G. M. Landau, and J. Mitchell, Edit distance of run-length encoded strings, *accepted for publication in Information Processing Letters.*

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

[3] Aggarawal, A., and J. Park, Notes on Searching in Multidimensional Monotone Arrays, *Proc. 29th IEEE Symp. on Foundations of Computer Science*, 497-512 (1988).

[4] Amir, A., G. Benson, and M. Farach, Let sleeping files lie: Pattern matching in Z-compressed files. *J. of Comp. and Sys. Sciences*, **52**(2), 299–307 (1996).

[5] Apostolico, A., M. Atallah, L. Larmore, and S. McFaddin, Efficient parallel algorithms for string editing problems. *SIAM J. Comput.*, **19**, 968-998 (1990).

[6] Apostolico, A., G.M. Landau and S. Skiena, Matching for Run Length Encoded Strings, *Journal of Complexity*, **15**, 1, 4–16 (1999).

[7] Bell, T.C., J.C. Cleary, and I.H. Witten. *Text Compression.* Prentice Hall, (1990).

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

[9] Bunke, H., and J. Csirik. An improved algorithm for computing the edit distance of run length coded strings, *Information Processing Letters*, **54**, 93–96 (1995).

[10] Chao, K. M., R. Hardison, and W. Miller, Recent developments in linear-space alignment methods: a mini survey. *J. Comp. Biol.*, **1**, 271–291 (1994).

[11] 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).

[12] Crochemore, M., Transducers and Repetitions. *Theoret. Comput. Sci.*, **45**, 63–86 (1986).

[13] Crochemore, M., and W. Rytter, Text Algorithms, *Oxford University Press*, (1994).

[14] Elias, P., Universal Codeword Sets and Representation of Integers, *I.E.E.E. Transf. Inform. Theory*, **IT21,2**, 194–203 (1975).

[15] Eppstein, D., Sequence Comparison with Mixed Convex and Concave Costs, *Journal of Algorithms*, **11**, 85–101 (1990).

[16] Eppstein, D., Z. Galil, and R. Giancarlo, Speeding Up Dynamic Programming, *Proc. 29th IEEE Symp. on Foundations of Computer Science*, 488–296 (1988).

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

[18] Eppstein, D., Z. Galil, R. Giancarlo, and G.F. Italiano, Sparse Dynamic Programming II: Convex and Concave Cost Functions, *JACM*, **39**, 568–599 (1992).

[19] Farach, M., and M. Thorup, String matching in Lempel-Ziv compressed strings. *Algorithmica*, **20**, 388–404 (1998).

[20] Galil, Z., and R. Giancarlo, Speeding Up Dynamic Programming with Applications to Molecular Biology, *Theoretical Computer Science*, **64**, 107-118 (1989).

[21] Galil Z., and K. Park, A linear-time algorithm for concave one-dimensional dynamic programming, *Info. Processing Letters*, **33**, 309-311 (1990).

[22] Gasieniec, L., M. Karpinski, W. Plandowski, W. Rytter, Randomised efficient algorithms for compressed strings: the finger-print approach, *Proc. 7th Annual Symposium On Combinatorial Pattern Matching*, LNCS 1075, 39–49 (1996).

[23] Gasieniec, L., and W. Rytter, Almost optimal fully LZW compressed pattern matching, *Data Compression Conference*, J. Storer, ed, (1999).

[24] Giancarlo, R. , Dynamic Programming: Special Cases, *Pattern Matching Algorithms*, edited by Apostolico, A. and Z. Galil, Oxford University Press, 201-232 (1997).

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

[26] Hirschberg, D.S., A linear space algorithm for computing maximal common subsequences, *Comm. Assoc. Comput. Mach.*, **18**(6), 341-343, (1975).

[27] Hirshberg, D.S., and L.L. Larmore, The Least Weight Subsequence Problem, *SIAM J. Compt.*, **16**(4), 628–638 (1987).

[28] Huang, X., and W. Miller, A time-efficient, linear space local similarity algorithm, *Adv. Appl. Math.*, **12**, 337–357 (1991).

[29] 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).

[30] Karkkainen, J., G. Navarro and E. Ukkonen, Approximate String Matching over Ziv-Lempel Compressed Text, *Proc. 11th Annual Symposium On Combinatorial Pattern Matching*, LNCS 1848, 195–209 (2000).

[31] Karkkainen, J., and E. Ukkonen, Lempel-Ziv parsing and sublinear-size index structures for string matching, *Proc. Third South American Workshop on String Processing (WSP '96)*, 141–155 (1996).

[32] Kida, T., M. Takeda, A. Shinohara, M. Miyazaki, and S. Arikawa, Shift-And approach to pattern matching in LZW compressed text, *Proc. 10th Annual Symposium On Combinatorial Pattern Matching*, LNCS 1645, 1–13 (1999).

[33] Klawe, M., and D. Kleitman, An Almost Linear Algorithm for Generalized Matrix Searching, *SIAM Jour. Descrete Math.*, **3**, 81-97 (1990).

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

[35] Lempel, A., and J. Ziv, On the complexity of finite sequences, *IEEE Transactions on Information Theory*, **22**, 75–81 (1976).

[36] Levenshtein, V.I., Binary Codes Capable of Correcting, Deletions, Insertions and Reversals, *Soviet Phys. Dokl*, **10**, 707–710 (1966).

[37] Main, M. G., R. J. Lorentz, An $O(n \log n)$ algorithm for finding all repetitions in a string. *J. Algorithms*, **5**, 422–432 (1984).

[38] Mäkinen, V., G. Navarro, and E. Ukkonen, Approximate Matching of Run-Length Compressed Strings, *Proc. 12th Annual Symposium On Combinatorial Pattern Matching*, LNCS 1645, 1–13 (1999).

[39] Manber, U., A text compression scheme that allows fast searching directly in the compressed file, *Proc. 5th Annual Symposium On Combinatorial Pattern Matching*, LNCS 2089, 31–49 (2001).

[40] Masek, W.J., and M.S. Paterson, A faster algorithm for computing string edit distances. *J. Comput. Syst. Sci.*, **20**, 18–31 (1980).

[41] Miller, W., and E.W. Myers, "Sequence comparison with concave weighting functions". *Bull Math Biol.*, **50**(2), 97–120 (1988).

[42] Mitchell, J., A Geometric Shortest Path Problem, with Application to Computing a Longest Common Subsequence in Run-Length Encoded Strings, Technical Report, Dept. of Applied Mathematics, SUNY Stony Brook, 1997.

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

[44] Navarro G., T. Kida, M. Takeda, A. Shinohara, and S. Arikawa: Faster Approximate String Matching Over Compressed Text, *Proc. Data Compression Conference (DCC2001)*, IEEE Computer Society, 459-468 (2001).

[45] Navarro, G., and M. Raffinot, A general practical approach to pattern matching over Ziv-Lempel compressed text, *Proc. 10th Annual Symposium On Combinatorial Pattern Matching*, LNCS 1645, 14–36 (1999).

[46] Navarro, G., and M. Raffinot. Boyer-Moore string matching over Ziv-Lempel compressed text, *Proc. 11th Annual Symposium On Combinatorial Pattern Matching*, LNCS 1848, 166–180 (2000).

[47] Sankoff D., and J.B. Kruskal (editors), *Time Warps, String Edits, and Macromolecules: the Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, MA, (1983).

[48] 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).

[49] Shabita Y., T. Kida, S. Fukamachi, M. Takeda, A. Shinohara, T. Shinohara, S. Arikawa, *Speeding up pattern matching by text compression*, *CIAC 2000*, LNCS 1767, 306–315 (2000).

[50] Smith, T. F. and M. S. Waterman, Identification of common molecular subsequences, *J. Molecular Biol.*, **147**, 195–197 (1981).

[51] Szpankowski, W., and P. Jacquet. Asymptotic Behavior of the Lempel-Ziv Parsing Scheme and Digital Search Trees, *Theoretical Computer Science*, **144**, 161–197 (1995).

[52] Welch, T.A., A Technique for High Performance Data Compression, *IEEE Trans. on Computers*, **17**(6), 8–19 (1984).

[53] Ziv, J., and A. Lempel, A Universal Algorithm for Sequential Data Compression, *IEEE Transactions on Information Theory*, **IT-23**(3), 337–343 (1977).

[54] Ziv, J., and A. Lempel, Compression of individual sequences via variable rate coding, *IEEE Trans. Inform. Th.*, **24**, 530-536 (1978).