A Sub-quadratic Sequence Alignment Algorithm for Unrestricted Cost Matrices

Maxime Crochemore *
Institut Gaspard-Monge
Université de Marne-la-Vallée

Gad M. Landau†
Haifa University
and
Polytechnic University

Michal Ziv-Ukelson ‡
Haifa University
and
IBM T.J.W Research Center

Abstract

The classical algorithm for computing the similarity between two sequences [36, 39] uses a dynamic programming matrix, and compares two strings of size n 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* alignment 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.

*Institut Gaspard-Monge, Université de Marne-la-Vallée, Cité Descartes, Champs-sur-Marne, 77454 Marne-la-Vallée Cedex 2, France, email: `mac@univ-mlv.fr`.

†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.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.

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). 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. In the *global alignment* problem [19], [29], the similarity between two strings A and B is measured. In the *local alignment* problem [39], the objective is to find substrings of A which are similar to substrings of B. Both alignment problems can be solved in $O(n^2)$ time by dynamic programming [19], [39].

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 $0 \leq h \leq 1$. Entropy is a measure of how "compressible" a sequence is [5],[7], 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).

We present an $O(n^2/\log n)$ algorithm for computing both global and local similarity between two strings over a constant alphabet. The algorithm is even faster when the sequence is compressible. In fact, for most texts, the complexity of our algorithm is actually $O(hn^2/\log n)$.

Note that the algorithm presented is the first sub-quadratic *local alignment* algorithm.

After the optimal scores are computed, an alignment trace corresponding to the optimal score can be recovered in time complexity which is linear with the size of the trace, for both the global alignment and the local alignment problems.

The algorithms described in this paper are the first to approach *fully 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 $LZ78$ or LZW compressed sequence, thus providing an efficient solution to the problem of how to compare the two strings without having to decompress them first.

The only previously known sub-quadratic global alignment string comparison algorithm, by Masek and Paterson [31], 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, 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.

- 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.
- 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 is concluded 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 in the following facts. First, our algorithm does not require any pre-computation 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 [10], [11], [27] and [37].

The remainder of this paper is organized as follows. Section 2 includes preliminaries. In section 3 we describe the global alignment solution using fully compressed string comparison. In section 4 we extend the solution to compute the highest scoring regions of local alignment. Section 5 contains a discussion of how to reduce the space complexity without impairing the time complexity, when computing global alignment over "discrete" scoring matrices.

A description of how to recover a path alignment trace in time linear with its size will be given in the journal version of the paper.

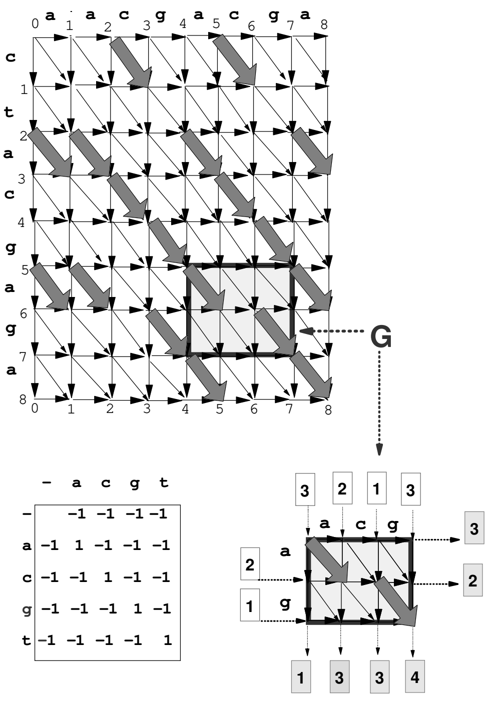

Figure 1: The alignment graph for comparing strings $A =$ "$ctacgaga$" and $B =$ "$aacgacga$". The scoring scheme matrix δ is shown in the lower left corner of the figure. The highest scoring global alignment paths originate in vertex (0,0), end in vertex (8,8) and have a total weight of 3. The highest 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.

2 Preliminaries

2.1 Highest Scoring Paths in the Alignment Graph. The dynamic programming solution to the string comparison computation problem can be represented in terms of a weighted alignment graph [19] (See Figure 1). The weight of a given edge can be specified directly in the grid graph, or as is frequently the case in biological applications, is given by a penalty matrix, denoted δ, which specifies the substitution cost for each pair of characters and the deletion cost for each character from the alphabet. Typically, in the biological domain, δ is negative for all operations except replacement of similar symbols, and the objective is to maximize the alignment score.

The classical dynamic programming algorithm for global alignment 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. $V(i,j) = max[V(i,j-1) + \delta(\epsilon, B_j),$

$$V(i-1,j) + \delta(A_i, \epsilon),$$
$$V(i-1,j-1) + \delta(A_i, B_j)]$$

Smith and Waterman [39] showed that essentially the same $O(|A||B|)$ dynamic programming solution can be used for local alignment, 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 local alignment will compute the following recurrence, which includes 0 as an additional option, and thus restricts the scores to non-negative values.

$$S(i,j) = max[0, S(i,j-1) + \delta(\epsilon, B_j),$$
$$S(i-1,j) + \delta(A_i, \epsilon),$$
$$S(i-1,j-1) + \delta(A_i, B_j)]$$

The score for the most similar substrings is found in the highest scoring nodes in the alignment graph.

2.2 A Block Partition of the Alignment Graph based on LZ78 Factorization. The traditional aim of text compression is 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* [22], [30], [38].

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 [3], [13] [34]. Along these lines, string search in compressed text was developed for the compression paradigm of LZ78 [45], and its subsequent variant LZW [43], as described in [23], [35]. A more challenging problem is that of "fully compressed" pattern matching when both the pattern and text strings are compressed [16], [17].

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 [21], [33].

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 [28], [44], [45].

Of the several existing versions of the method, we will use the ones which are denoted $LZ78$ family [43], [45]. The main feature which distinguishes $LZ78$ factorization from previous LZ compression algorithms is in the choice of code words. 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.

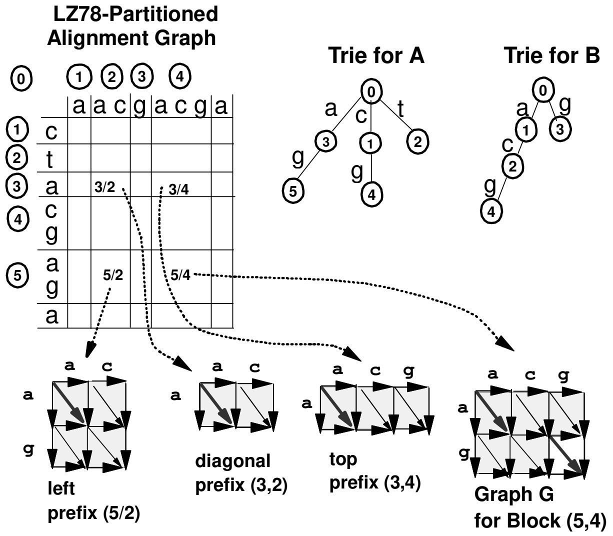

Figure 2: The block partition of the alignment graph, and the tries corresponding to LZ-78 parsing of strings $A =$ "*ctacgaga*" and $B =$ "*aacgacga*". Note that for the block G in this example, $\alpha =$ "ag", $\beta =$ "acg", $\ell_r = 2$, $\ell_c = 3$, $i = 5$ and $j = 4$. (The new cell of G, which 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.

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

THEOREM 2.1. (ZIV AND LEMPEL 1976 [28].) *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 $0 \leq h \leq 1$, which is a measure of 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 [5], [7], [28], [45]. Note that for any other text over a constant alphabet, the upper bound above still applies by setting h to 1.

3 The Global Alignment Solution

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, 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 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 the edges of these two graphs are identical, but the weights to be assigned to the 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.

3.2 I/O **Propagation Across** G**.** The work for each block will consist of two stages (a similar approach is shown in [6, 20, 26, 27]).

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 will be 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 will be used (see Figure 3).

DEFINITION 3.1. *$DIST[i,j]$ 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 [4], [6], [20], [27] and [37].

Given input row I and the $DIST$ for G, the weight of output row vertex O_j can be computed as follows.

$$O_j = \max_{r=0}^{j}\{I_r + DIST[r,j]\}$$

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

Aggarwal and Park [2] and Schmidt [37] observed that $DIST$ matrices are Monge arrays [32].

	DIST **matrix**					
$I_0 = 1$	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

OUT **matrix**

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

O_0	O_1	O_2	O_3	O_4	O_5
1	3	3	4	2	3

column numbers

0	**1**	**2**	**3**	**4**	**5**

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.

DEFINITION 3.3. *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 3.4. *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 [1] 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 could 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 [8], [9], [14], [15], [24] and [27]. 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 3.2.2.
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).

3.2.1 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 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 3.1. *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.

3.2.2 Incremental Update of the new $DIST$ Information for G.

In this section we will show how to efficiently compute the new $DIST$ info 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 will compute the scores of t new optimal paths, leading from the input border to the new vertex (ℓ_r, ℓ_c) in the lowest, rightmost corner of G. 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).

3.2.3 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.

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.

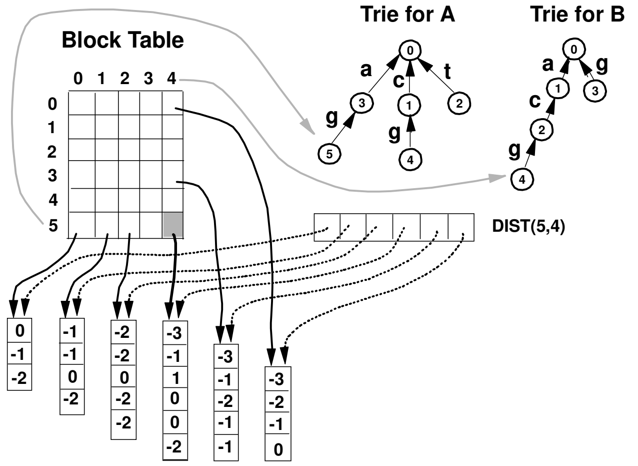

Figure 4: A table containing an entry for each block of the alignment graph. Entry (i, j) of the table corresponds to the block whose substrings are represented by node i in the trie for A and node j in the trie for B. The entry for each block in the table points to the start of its new $DIST$ column. 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.

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.

3.2.4 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.

3.3 Time and Space Analysis

Assuming sequence size n and sequence entropy $h \leq 1$. The $LZ78$ factorization algorithm will parse the strings and construct 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. Propagation for G. 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 Extensions to Local Alignment

When computing the highest local alignment score, the added challenge is that an optimal path could begin and end inside any block. Therefore, we will modify O to consider the additional paths originating inside G.

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

1. E - is a vector of size t. $E[i]$ contains the value of the highest scoring path

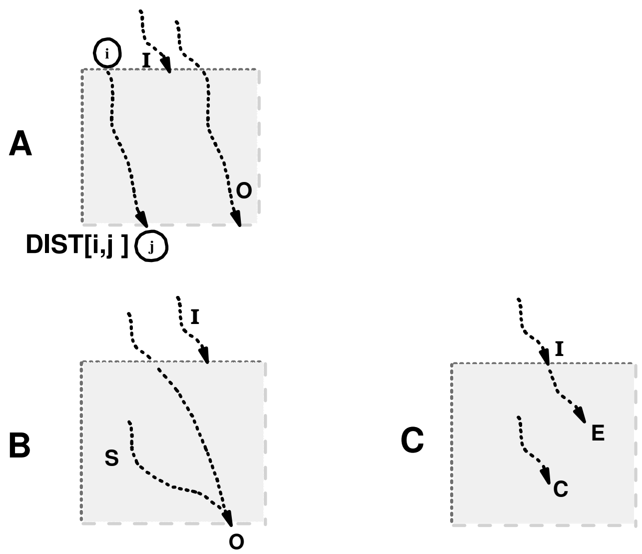

Figure 5: **A.** The I/O path weight vectors computed for each block in the global alignment solution. $DIST[i,j]$ will be set to the highest scoring path connecting vertex i in the input border with vertex j in the output border. **B,C.** The vectors of optimal path weights considered for the local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

which starts in vertex i of the input border of G and ends inside G.

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

4. F - is the weight of the highest scoring path ending in G. This path could either begin and end inside G (a C-path) or start outside G and end inside G (an I-path followed by an E-path).

The overall highest local alignment score for comparing A and B can be computed as the maximum among the F values of each block.

The two stages described in section 3.2 will be extended as follows.

4.1 Encoding. $DIST$ is computed as described in section 3.2. In addition, the values of E, S and C are computed as follows.

1. Computing the values of E. $E[i]$ 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. Computing the values of S. The only new value computed for S is the Smith-Waterman score for the new vertex (ℓ_r, ℓ_c). Given the Smith-

Waterman local alignment scores for vertices $(\ell_r - 1, \ell_c - 1)$ obtained from the *diagonal prefix*, $(\ell_r, \ell_c - 1)$ obtained from the *left prefix* and $(\ell_r - 1, \ell_c)$ obtained from the *top prefix* of G, and the weights of the 3 edges leading into vertex (ℓ_r, ℓ_c), the Smith-Waterman score for vertex (ℓ_r, ℓ_c) can be computed in $O(1)$ time complexity, using the recursion given in section 2.1. The value for entry ℓ_c of S is set to this newly computed Smith-Waterman score for vertex (ℓ_r, ℓ_c).

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. Computing the value of C. 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.

4.2 Propagation.

1. Computing the values of O. Our objective is to set $O[i]$ to the weight of the highest scoring path originating anywhere in the alignment graph and ending in entry i of the output border. Vector O will first be 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 in 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 $S[i]$.

2. Computing the values of F. F is computed as $max(Max_{i=0}^{t}\{I[i] + E[i]\},\ C)$

4.3 Time and Space Analysis

Encoding. Since, as shown in section 3.2.3, 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.

Propagation. Given the vectors computed in the encoding stage - the values of O and F 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 F values computed for each block.

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 for the local alignment solution is $O(hn^2/\log n)$.

5 Reducing the Space Complexity

When computing global alignment with scoring matrices which follow the "discreteness" condition (see Section 1), the *efficient alignment stage algorithm* described in [27] 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 [37].

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

6 Conclusions

The results demonstrated in this paper are as follows.

- The algorithm presented in this paper is the first $O(hn^2/\log n)$ string comparison algorithm.
- This is the first sub-quadratic string comparison algorithm for general scoring tables whose weights are not restricted to rational numbers.
- We showed how to extend this result to a *local alignment* $O(hn^2/\log n)$ algorithm.
- For global alignment over "discrete" scoring matrices, we explained 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.

In addition to the scores computed by dynamic programming, it is often desired to recover a meaningful trace of the optimal alignments. Optimal paths in the alignment graph (paths whose total weight is maximum) represent optimal alignments of A and B.

Without any added complexity, the current algorithmic infrastructure can be modified to support the recovery of an optimal global alignment path trace, as well as an optimal local alignment trace as defined by Erickson and Sellers [12], in time complexity which is linear with the size of the trace.

Due to lack of space, the description of how to recover the path alignment traces is reserved for the journal version of the paper.

Acknowledgement

Thanks to Dan Gusfield for a helpful discussion of the "Four Russians" algorithm.

References

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

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

[3] 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).

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

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

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

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

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

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

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

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

[12] Erickson, B.W., and P.H. Sellers, Recognition of patterns in genetic sequences, *in Time Warps, String Edits, and Macromolecules: The Theory and Practice of Sequence Comparison*, D. Sankoff and J.B. Kruskal, eds., Addison-Wesley, Reading, MA, 55–91 (1983).

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

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

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

[16] 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).

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

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

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

[20] 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).

[21] 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).

[22] 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).

[23] 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).

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

[25] Landau, G.M., E.W. Myers and J.P. Schmidt, Incremental String Comparison, *SIAM J. Comput.*, **27**(2), 557–582 (1998).

[26] Landau, G.M. and M. Ziv-Ukelson, On the Shared Substring Alignment Problem, *Proc. Symposium On Discrete Algorithms*, 804–814 (2000).

[27] Landau, G.M., and M. Ziv-Ukelson, On the Common Substring Alignment Problem, *Journal of Algorithms.*

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

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

[30] Manber, U., A text compression scheme that allows fast searching directly in the compressed file, *Proc. 5th Annual Symposium On Combinatorial Pattern Matching*, LNCS 807, 113–124 (1994).

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

[32] Monge, G., Deblai et Remblai,*Memoires del l'Academie des Sciences*, Paris (1781).

[33] 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).

[34] 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).

[35] 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).

[36] 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).

[37] 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).

[38] 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).

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

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

[41] Takeda, M., Y. Shibata, T. Matsumoto, T. Kida, A. Shinohara, S. Fukamachi, T. Shinohara, and S. Arikawa: Speeding up string pattern matching by text compression: The dawn of a new era, **42**(3), pp. 370-384 (2001).

[42] Waterman, M.S., and M. Eggert, A new algorithm for best subsequence alignment with application to tRNA-rRNA comparisons, *J. Molecular Biol.*, **197**, 723–728 (1987).

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

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

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