On the Common Substring Alignment Problem [1]

Gad M. Landau [*]

Department of Computer Science, Haifa University, Haifa 31905, Israel, phone: (972-4) 824-0103, FAX: (972-4) 824-9331; and Department of Computer and Information Science, Polytechnic University, Six MetroTech Center, Brooklyn, NY 11201-3840
E-mail: landau@poly.edu

Michal Ziv-Ukelson [†]

Department of Computer Science, Haifa University, Haifa 31905, Israel; On Education Leave from the IBM T.J.W. Research Center;
E-mail: michal@cs.haifa.ac.il

The *Common Substring Alignment Problem* is defined as follows: Given a set of one or more strings $S_1, S_2 \ldots S_c$ and a target string T. Y is a common substring of all strings S_i, that is $S_i = B_i Y F_i$. The goal is to compute the similarity of all strings S_i with T, without computing the part of Y again and again. Using the classical dynamic programming tables, each appearance of Y in a source string would require the computation of all the values in a dynamic programming table of size $O(n\ell)$ where ℓ is the size of Y.

Here we describe an algorithm which is composed of an encoding stage and an alignment stage. During the first stage, a data structure is constructed which encodes the comparison of Y with T. Then, during the alignment stage, for each comparison of a source S_i with T, the pre-compiled data structure is used to speed up the part of Y.

We show how to reduce the $O(n\ell)$ alignment work, for each appearance of the common substring Y in a source string, to $O(n)$ - at the cost of $O(n\ell)$ encoding work, which is executed only once.

Key Words: design and analysis of algorithms, dynamic programming, sequence comparison, repeated substrings, shared substrings, Monge arrays, candidate lists.

[1] This paper continues work from [19]. The efficiency of the solutions, as well as the range of scoring schemes to which they apply, have been further enhanced in this paper.

[*] partially supported by NSF grant CCR-9610238, by NATO Science Programme grant PST.CLG.977017, and by the Israel Science Foundation, founded by the Israel Academy of Sciences and Humanities.

[†] partially supported by the Israel Science Foundation founded by the Israeli Academy of Sciences and Humanities.

1. INTRODUCTION

The *Common Substring Alignment Problem* is defined as follows: Given a set of one or more strings $S_1, S_2 \dots S_c$ and a target string T. Y is a common substring of all strings S_i, that is $S_i = B_i Y F_i$, and Y may be repeated several times in any of the source strings (see Figure 1). The goal is to compute the similarity of all strings S_i with T, without computing the part of Y again and again. We know the locations where the common subsequence Y starts and ends in each source sequence S_i. The part of the target T with which Y will align, however, will vary according to each B_i and F_i combination.

T	=	"DCBADBDC"	Y	=	*"DCBD"*			
S_1	=	"E *DCBD DCBD"*	B_1	=	"E"	F_1	=	"DCBD"
S_2	=	"CBA *DCBD* C"	B_2	=	"CBA"	F_2	=	"C"

FIG. 1. An example of two different source strings (S_1, S_2) sharing a common substring Y. Note that Y is repeated twice in S_1.

More generally, the sequence sub-component Y could be shared by different source sequences competing over similarity with a common target, or could appear repeatedly in the same source string. Also, in a given application, we could of course be dealing with more than one repeated or shared sub-component.

In this paper, we will describe an algorithm which is composed of an encoding stage and an alignment stage. During the first stage, a data structure is constructed which encodes the comparison of Y with T. Then, during the second stage, for each comparison of a source S_i with T, the pre-compiled data structure is used to speed up the part of aligning each appearance of the common substring Y.

A clear distinction should be made between the *off-line* pre-processing work and the *online* encoding stage. In the applications for which our algorithm is intended, the source sequence database is prepared *off-line*, while the target can be viewed as an "unknown" sequence which is received *online*. The source strings can be pre-processed *off-line* and parsed into their optimal common substring representation. Therefore, we know well beforehand where, in each S_i, Y begins and ends.

However, the comparison of Y and T can not be computed until the target is received. Therefore, the encoding stage, as well as the alignment stage - are both *online* stages, and the tradeoff between the two must be cleverly minimized in order to maximize the efficiency gain by the suggested two-stage scenario.

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

We will show how to reduce the $O(n\ell)$ runtime work for each appearance of a repeated substring Y in a source sequence to $O(n)$, at the cost of a single execution of the $O(n\ell)$ time encoding work, where n is target size and ℓ is the size of Y.

For source sequences with two or more common factors, the time complexity of the encoding stage is further reduced to $O(nD)$, where D is the number of nodes in the dictionary trie for the common factors.

The remainder of this paper is organized as follows. In section 2 we present some applications which can be cast as Common Substring Alignment problems. Section 3 contains a background overview, including a description of the scoring schemes to which the algorithm applies. The notation, as well as a general description of our approach to solving the Common Substring Alignment problem, is given in Section 4. In section 5 we describe the first algorithm, which encodes a common substring in $O(n^2 + n\ell)$ work, and then uses an $O(n)$ time complexity recursive algorithm for the alignment stage. In section 6 we present a more efficient, $O(n\ell)$ time encoding stage algorithm, and an algorithm which utilizes the results of the efficient encoding for a non-recursive, linear time alignment stage.

2. APPLICATIONS

There are various applications which can be cast as Common Substring Alignment problems. The applications differ by the pattern in which the common subcomponents are repeated or shared by the source strings, and therefore may vary in the potential combinatorial gain by applying Common Substring Alignment algorithms to their solution.

2.1. Template Matching Applications

In Template Matching Applications, the data is viewed as a set of many competing source sequences to be compared to a common target. The template source sequences are usually known well in advance, and the target is given online. The objective is to classify the target by finding the source string whose alignment with the target gives the highest similarity score. Very often, the competing source strings are variations of a similar signal, or different combinations of a common set of subcomponents. Common Substring Alignment can be used to speed up the comparison of each common subcomponent, rather than comparing it again and again for each template source string containing the common subcomponent.

Intelligent Tutoring.

In the Intelligent Tutoring application [8], the alphabet for each sequence are all possible computer interface artifacts (keyboard and mouse input combinations). The student is given the task, and the resulting events are recorded as the student tries to solve the problem. The new stream of user input events is then compared to various templates, which represent different solutions to the given exercise. The result of the comparison between the student's input sequence and the most similar template solution can be used to provide the proper feedback to the student. For many problems, the various template solutions are variations of a common theme, and share common substrings of artifacts.

Electronic Commerce.

Another example of a potential application domain for the problem is in Electronic Commerce [5], [6], [20]. In an attempt to improve both merchandise and marketing aspects of the system, logging can be employed to record the sequences of site traversal actions of potential customers from the minute they enter the commerce site until they exit. Accumulated server logs can be mined [5], in order to provide the system with a prototype set of sequences of site traversal actions known to have led to purchase. A new site traversal sequence which did not result in purchase will be compared against all prototypes in an attempt to find the most similar sequence of actions which did lead to a purchase. The resulting alignment between the two can then be used to study what went wrong with the potential purchase. (For example, a shopping cart may have been filled, and then the customer left without completing the purchase order, due to difficulties in a specific part of the purchase form - in which case improving the user interface of that part of the form may result in better sales.) Various subsets of the prototype-set sequences may share long similar subcomponents representing common protocols, such as typical shopping cart routes, or sequences of actions required to fill a purchase form.

Network Security.

Another potential application is in Intrusion Detection Systems (IDSs) [28]. An "attack" is a sequence of audit trail log entries leading to a break-in. System security would like to spy on users who attempt to access a site, in order to detect aggressive users and block their entrance before they break in. An audit trial sequence is labeled as a potential threat if it is similar enough to one of the known attacks. Audit trail sequences of known attacks may share long subsequences of common security breech protocols.

2.2. Alignment of Repetitive Sequences

Here, each repeated factor is, in essence, a common substring which we would like to compare against the target only once during an encoding stage, rather than comparing it again and again for each appearance of the repeated factor in the source string, during the matching stage.

Especially interesting are those applications where the repetitions are such that one common subcomponent can be derived from another common subcomponent via minor modifications. For example, each repeated factor may be obtained from a smaller repeated factor plus one character, such as in the application of approximate string matching over L-Z compressed text [17]. Another example is in genomic data [31], such as DNA sequences, where repetitions can be grouped into families of similar subcomponents. DNA has a small alphabet, and repetitions belonging to one family form hierarchies of subsequences which evolved from a common core and from one another. Therefore, common subcomponents belonging to one family tend to form a compact keyword trie.

The fact that the factors form a compact trie allows for an even more efficient encoding, where a prefix common to one or more factors can be encoded once, instead of redoing the encoding work for each factor sharing the prefix.

2.3. Subcomponent Concatenation With Preserved Order

In application belonging to this category, the source string is segmented into many subparts, and the target string is matched against different concatenations of these source substrings. The concatenations preserve the ordering of the subsegments in the source string. Gene Prediction Via Spliced Alignment is an example of an application from this category.

Gene Prediction Via Spliced Alignment

Recognition of genes in eukaryotic DNA is seriously complicated by noisy regions (*introns*) that interrupt the coding regions (*exons*) of genes. The *gene-prediction via spliced alignment* approach, due to Gelfand, Mironov and Pevsner [11, 23, 29] incorporates similarity analysis into gene prediction by attempting to find a set of potential exons in a genomic sequence whose concatenation is highly similar to one of the already known gene sequences in the database.

The task of gene prediction is generally divided into two stages. The first task is that of finding *candidate exons* in a long DNA sequence believed to contain a gene. A candidate exon is a sequence fragment whose left boundary is an acceptor site or a start codon, and the right boundary is a donor site or a stop codon. The nucleotide sequence in Figure 2 contains marked sites where a candidate exon may begin and end. Uppercase A-E mark identified sites where an exon is likely to begin (start/acceptor sites), and lowercase f-j mark sites where exons are likely to end (stop/donor sites). Candidate exons are A-f, A-g, A-h, A-i, A-j, B-f, B-g, B-h, B-i, B-j, C-g, etc. This set of derived candidate exons should include all true exons, but could contain any number of false exons, depending on the filtration degree used in the preprocessing stage. The second task is that of selecting the best subset of nonoverlapping candidate exons to cover the sequence of the predicted gene. (Two of the many possible assemblies of candidate exons as candidate genes are shown in the figure: $S_1 = \{A-f, C-h, E-i\}$ and $S_2 = \{B-g, D-j\}$.) Each candidate gene (a concatenation of non-intersecting candidate exons which satisfy some natural consistency conditions [26]) is compared against the target sequence, which is an already known gene from a homologous species. An interesting combinatorial approach, using Network Alignment, which explores all possible exon assemblies in polynomial time, is described in [11].

Dominant portions of each of the competing candidate gene assemblies are segments common to other candidates, since the candidate exons overlap in the genomic source sequence. (The two source strings in the figure, S_1 and S_2, share

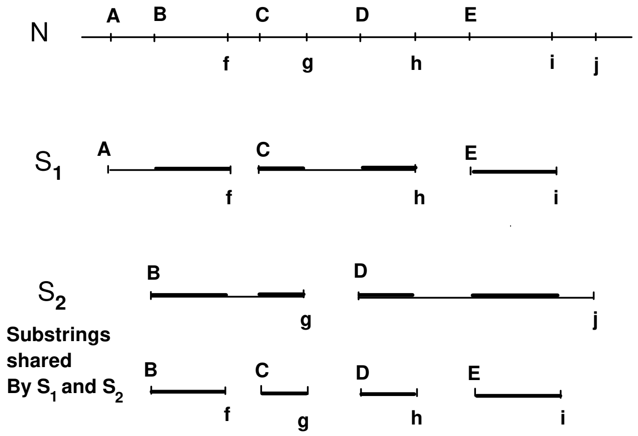

FIG. 2. A nucleotide sequence ($LineN$) and two of its derived candidate genes (S_1, S_2).

the substrings B-f, C-g, D-h and E-i.) Therefore, casting this application as a "Common Substring Alignment" problem, would enable us to compare each of the shared segments only once against the target, instead of having to match it again and again for each candidate gene in which it is included.

3. BACKGROUND

When formalizing the relatedness between two sequences, one could measure either their $similarity$ or their $distance$. An example of a basic $similarity$ metric is LCS [14], which measures the subsequence of $maximal$ length common to both sequences, where a subsequence is defined as any series of elements which can be obtained from a given sequence by deleting some of its elements. The Edit Distance metric [21], on the other hand, measures the $minimal$ number of substitutions, insertions and deletions required to transform one sequence into another. Each mismatched aligned pair and unaligned symbol is called a *difference* and scores 1. All pairs of equal aligned characters score 0. One seeks an alignment which *minimizes* the score or number of differences and this minimal score is called the $distance$ between S and T. The distance and similarity perspectives are complementary, and any distance problem can be translated into a similarity problem.

From now on we will describe the solutions in terms of distance minimization. (For the sake of simplicity, we will restrict our examples to the Edit Distance measure [21].) However, the solutions can easily be translated to a score maximization problem, in order to apply to string comparison metrics which measure similarity, rather than distance.

The Operation Weight Edit Distance problem [13] is a generalization of Edit Distance which allows an arbitrary weight to be associated with every edit operation, as well as with a match. Thus, any insertion or deletion has a weight denoted $indel$, a substitution has a weight s, and a match has a weight m.

An even more general scoring scheme is that of Alphabet Weight Edit Distance [13], in which the scoring scheme matrix δ contains for each character c a value $\delta(c, -)$ for deleting the character, and a value $\delta(-, c)$ for inserting the character. For a pair of characters a and b, $\delta(a, b)$ is the score obtained by aligning character a against character b. Given two strings A, B and the scoring scheme matrix δ, the objective is to compute the minimal score for an alignment of A and B. The solutions described in this paper will apply to all string comparison metrics for which the range of values in δ is bounded by a constant.

The distance between strings A and B can be computed via the dynamic programming algorithm, using the given score matrix, as described in [25]. The dynamic programming solution to the string comparison computation problem can be represented in terms of a weighted dynamic programming graph [13] (See Figure 3). A DP Graph for A and B is a directed, acyclic, weighted graph containing $(|A| + 1)(|B| + 1)$ nodes, each labeled with a distinct pair $(x, w)(0 \leq x \leq |A|, 0 \leq w \leq |B|)$. The nodes are organized in a matrix of $(|A| + 1)$ rows and $(|B| + 1)$ columns. The DP Graph contains a directed edge with a weight of $\delta(-, b_{w+1})$ from each node (x, w) to each of the nodes $(x, w + 1)$, and a weight of $\delta(a_{x+1}, -)$ from each node (x, w) to $(x + 1, w)$. Node (x, w) will contain a diagonal edge with a weight of $\delta(a_{x+1}, b_{w+1})$ to node $(x + 1, w + 1)$, where δ is the scoring scheme matrix for the problem. Upon completion, the value at vertex (i, j) of the DP Graph will be set to the score between the first i characters of A and the first j characters of B. The optimal score between A and B, and the weights for entire graph, can be obtained in $O(|A||B|)$ time.

Optimal paths in the DP Graph (paths whose total weight is minimum) represent optimal alignments of A and B. In particular, the score for comparing A and B is equivalent to the total weight of the optimal path connecting the leftmost vertex in the first row of the DP Graph for A and B with the rightmost vertex in the last row of the graph.

4. THE COMMON SUBSTRING ALIGNMENT APPROACH

The DP Graph used for computing the distance between a source string $S_i = B_i Y F_i$ and a target string T can be viewed as a concatenation of 3 sub-graphs, where the first graph represents the distance between B_i and T, the second graph represents the distance between Y and T, and the third graph represents the distance between F_i and T. (See Figure 3.)

In this partitioned solution, the weights of the vertices in the last row of the first graph serve as input to initialize the weights of the vertices in the first row of the

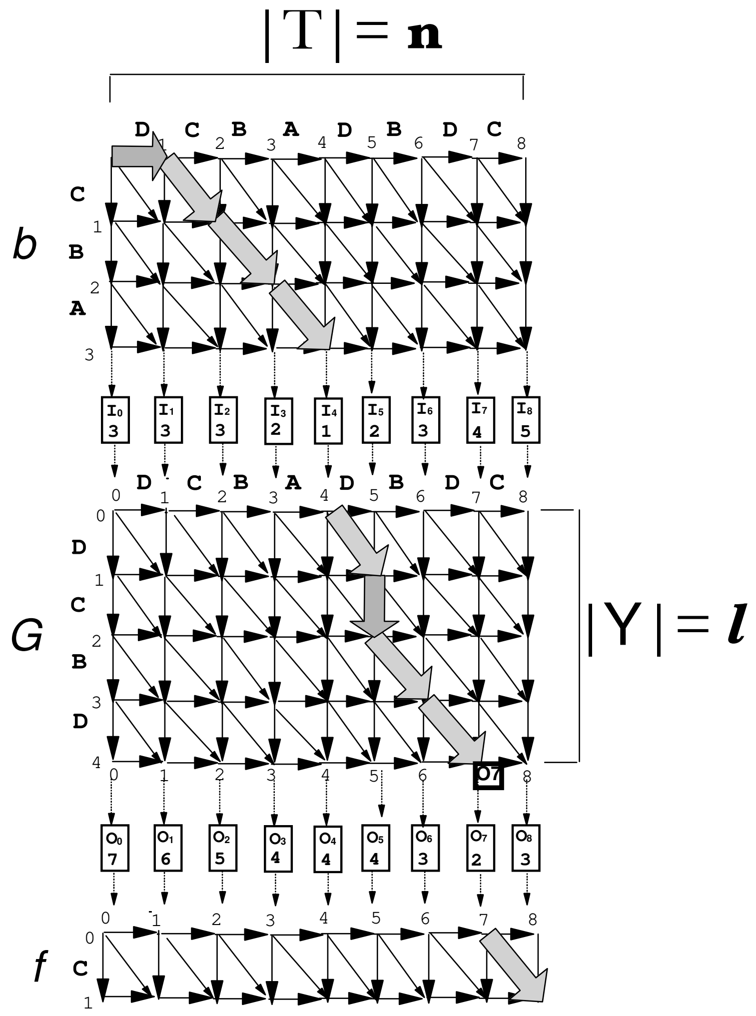

FIG. 3. The DP graph for computing the distance between $T = "DCBADBDC"$, and $S_2 = "CBADCBDC"$. S_2 contains the common substring $Y = "DCBD"$. This figure continues Figure 1.

second graph. The weights of the last row of the second graph can be used to initialize the first row of the third graph.

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

Notation

Throughout this paper, we use the following notation.

- ℓ - denote the size of the subsequence Y which is common to all S_i.
- n - denote the size of target string T.
- C_u^z - denote the substring of string C from index u up to index z, where indices are numbered from 1 to $|C|$. (C_{j+1}^j - denote the empty string.)
- G - denote the second sub-graph comparing Y and T, which is shared by all DP graphs comparing a source string S with T.
- I - denote the series of weights of the vertices in the first row of G.
- O - denote the series of weights of the vertices of the last row of G.

Algorithm Framework

The Common Substring Alignment solutions described in this paper comply by the following 2-stage approach.

Encoding Stage: Given Y and T, encode G in a format which can be efficiently used by the alignment stage.

Alignment Stage: Given the output of the encoding stage and input row I - compute the output row O.

A similar approach was introduced by [16], as a procedure in an algorithm for finding the best non-overlapping repeats in a sequence. They presented an $O(n^2 \log n)$ time complexity algorithm for the encoding stage, followed by an $O(n \log n)$ alignment stage. A more space efficient algorithm is given in [4].

5. THE FIRST ALGORITHM

5.1. The Encoding Stage.

The following $DIST$ matrix will be computed. (See Figure 4.)

DEFINITION 5.1. $DIST[i, j]$, for $j = 0 \dots n, i = 0 \dots j$, stores the weight of the shortest path from the vertex in column i of the first row of the graph G to the vertex in column j of the last row of the graph G.

A similar encoding of a graph has been used in [2, 3, 4, 16, 27].

$DIST$ can be constructed in $O(n^2+n\ell)$ time by using the algorithm of [27]. For the LCS and Edit Distance metrics, $DIST$ can also be constructed in $O(n^2 + n\ell)$

I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	
3	3	3	2	1	2	3	4	5	*input row*
$DIST$ **matrix**									
4	3	2	1	1	1	2	3	4	$Edit[T_1^{j=0...9}, Y]$
-	4	3	2	2	2	3	4	5	$Edit[T_2^{j=0...9}, Y]$
-	-	4	3	3	3	4	3	4	$Edit[T_3^{j=0...9}, Y]$
-	-	-	4	4	3	3	2	3	$Edit[T_4^{j=0...9}, Y]$
-	-	-	-	4	3	2	1	2	$Edit[T_5^{j=0...9}, Y]$
-	-	-	-	-	4	3	2	3	$Edit[T_6^{j=0...9}, Y]$
-	-	-	-	-	-	4	3	2	$Edit[T_7^{j=0...9}, Y]$
-	-	-	-	-	-	-	4	3	$Edit[T_8^{j=0...9}, Y]$
-	-	-	-	-	-	-	-	4	$Edit[T_9^{j=0...9}, Y]$
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	*column numbers*
OUT **matrix:**									
7	6	5	4	4	4	5	6	7	$OUT[0, j = 0 \dots 9]$
-	7	6	5	5	5	6	7	8	$OUT[1, j = 0 \dots 9]$
-	-	7	6	6	6	7	6	7	$OUT[2, j = 0 \dots 9]$
-	-	-	6	6	5	5	4	5	$OUT[3, j = 0 \dots 9]$
-	-	-	-	5	4	3	2	3	$OUT[4, j = 0 \dots 9]$
-	-	-	-	-	6	5	4	5	$OUT[5, j = 0 \dots 9]$
-	-	-	-	-	-	7	6	5	$OUT[6, j = 0 \dots 9]$
-	-	-	-	-	-	-	8	7	$OUT[7, j = 0 \dots 9]$
-	-	-	-	-	-	-	-	9	$OUT[8, j = 0 \dots 9]$
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	*iteration numbers*
O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_8	
7	6	5	4	4	4	3	2	3	*output row*

FIG. 4. The $DIST$ and OUT matrices which correspond to the sequences $T =$ "$DCBADBDC$", $Y =$ "$DCBD$" and the input row I for $B_i =$ "CBA". The output row O is the series of column minima of $OUT[i, j] = I_i + DIST[i, j]$. This figure continues the example of Figures 1 and 3.

time by employing [18]. Alternatively, one could use the algorithm from [3] to construct $DIST$ in $O(n^2 \log n)$ time.

5.2. The Alignment Stage.

Given input row I and sub-graph G, the weight of output row vertex O_j can be computed as follows.

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

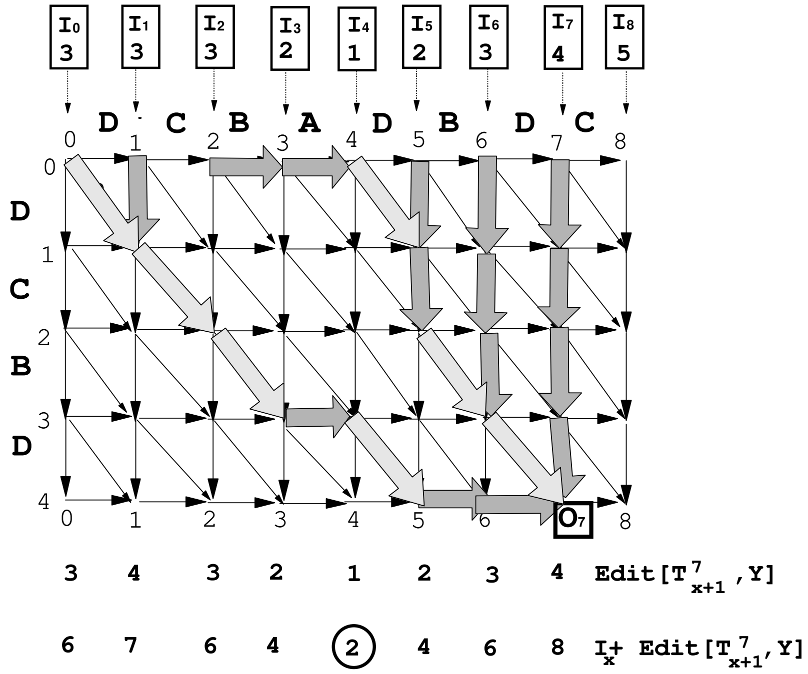

FIG. 5. The computation of output entry O_7 for $T = "DCBADBDC"$, $B_i = "CBA"$, and $Y = "DCBD"$. The minimal output at O_7, $\min_{x=0}^{7}\{I_x + Edit[T_{x+1}^{7}, Y]\} = 2$, is achieved by the path originating at column 4 and receiving input I_4. This figure continues the example of Figures 1, 3 and 4.

This computation entails selecting a minimum among up to n sums for each of the n output sources. (Figure 5 demonstrates an example of an output entry computation.) The above formulation was first presented in [16]. It is, in essence, a static version of the 1D dynamic programming problem [12]:

$$E[j] = \min_{i=0}^{j}\{D[i] + w(i, j)\}$$

-in which all values of $D[i]$ are specified before any value of $E[j]$ is computed, and the values of function $w(i, j)$ are precomputed for all integers $j = 0 \ldots n, i = 0 \ldots j$.

O_j is the minimum of column j of the following OUT matrix, which merges the information from input row I and $DIST$. (See Figure 4).

DEFINITION 5.2. $OUT[i,j] = I_i + DIST[i,j]$ for $j = 0 \dots n, i = 0 \dots j$.

Aggarwal and Park [2] and Schmidt [27] observed that $DIST$ matrices are Monge arrays [24].

DEFINITION 5.3. A matrix $M[0 \dots m, 0 \dots n]$ is **Monge** if either condition 1 or 2 below holds for all $i = 0 \dots m, j = 0 \dots n$:

1. $M[i,j] - M[i,j-1] \leq M[i-1,j] - M[i-1,j-1]$.
2. $M[i,j] - M[i,j-1] \geq M[i-1,j] - M[i-1,j-1]$.

It is easy to see that OUT matrices also follow the Monge properties.

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

DEFINITION 5.4. A matrix $M[0 \dots m, 0 \dots n]$ is **totally monotone** if either condition 1 or 2 below holds for all $a, b = 0 \dots m;\ c, d = 0 \dots n$:

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

Note that the Monge property implies total monotonicity, but the converse is not true.

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 accessing an element of OUT is O(1) work.

One obstacle which comes up during this implementation is that $DIST$ is not rectangular. Only the values in the upper triangle are defined. However, this can be resolved by setting the undefined values in the lower triangle to ∞.

5.3. Time Analysis of the First Algorithm.

$DIST$ is constructed during the encoding stage in $O(n^2 + n\ell)$ time. The alignment stage is done in $O(n)$ time, for each appearance of the common substring Y in a source sequence.

$DELTA$ **matrix:**									
–	-1	-1	-1	0	0	1	1	1	$OUT[0,j]-OUT[0,j-1]$
-	–	-1	-1	0	0	1	1	1	$OUT[1,j]-OUT[1,j-1]$
-	-	–	-1	0	0	1	-1	1	$OUT[2,j]-OUT[2,j-1]$
-	-	-	–	0	-1	0	-1	1	$OUT[3,j]-OUT[3,j-1]$
-	-	-	-	–	-1	-1	-1	1	$OUT[4,j]-OUT[4,j-1]$
-	-	-	-	-	–	-1	-1	1	$OUT[5,j]-OUT[5,j-1]$
-	-	-	-	-	-	–	-1	-1	$OUT[6,j]-OUT[6,j-1]$
-	-	-	-	-	-	-	–	-1	$OUT[7,j]-OUT[7,j-1]$
-	-	-	-	-	-	-	-	–	$OUT[8,j]-OUT[8,j-1]$
Borderline Points :									
-	-	-	-	-	**2**	**2**	**1**	**5**	$BorderlinePoint[1,j]$
-	-	-	-	-	-	**3**	**1**	**5**	$BorderlinePoint[2,j]$

FIG. 6. An example of a $DELTA$ matrix and its Borderline Points. Note that for the Edit Distance metric, which is used in this example, $\psi = 2$, and therefore the number of Borderline Points for $DELTA$ is bounded by $2n$. This figure continues the example of Figure 4.

6. A MORE EFFICIENT, NON RECURSIVE ALGORITHM

6.1. The Encoding Stage.

A more efficient encoding can be achieved, by utilizing the fact that the number of relevant changes, from one column of both OUT and $DIST$ to the next, is constant. This property, also discussed in [27], allows for a representation of $DIST$ via an $O(n)$ number of "relevant" points. The importance of this property will become clearer in section 6.2.

The $DELTA$ matrix is defined as follows.

DEFINITION 6.1. $DELTA[i,j] = OUT[i,j] - OUT[i,j-1] = DIST[i,j] - DIST[i,j-1]$ for $j = 1 \ldots n, i = 0 \ldots j-1$.

The range of possible values for $DELTA[i,j]$ depends on the scoring scheme which is used for the string comparison, and is actually the upper bound for the value difference between two consecutive elements in the dynamic programming table. (For an Example of a $DELTA$ matrix, see figure 6.)

We will use the term ψ to denote the range bound for $DELTA[i,j]$ values. As an example, if the similarity metric used is LCS, the only possible values for $DELTA$ will be either 1 or 0, and ψ assumes a value of 1. For the Edit Distance metric, on the other hand, ψ is 2, since $DELTA$ can only assume one of the 3 values: -1, 0, 1 [30]. Our algorithm applies to all scoring scheme metrics for which ψ is a constant.

Note that the following 2 observations apply to any column in $DELTA$.

Observation 1
Since $DIST$ is a Monge array - each column in $DELTA$ is a series of monotonically non-increasing values.

Observation 2
Since the range of distinct values which $DELTA$ may assume is bounded by a constant (ψ) - the number of "steps" (row indices in which the series of column entries increases in value) in each column of $DELTA$ is constant.

As a result - $DIST$ can be represented via an $O(n)$ size set of relevant "step" points collected from all columns of $DELTA$. (See Figure 6.)

DEFINITION 6.2. Let $Borderline[\alpha, j]$ for $\alpha = 0 \ldots \psi, j = 0 \ldots n$ - denote a row index of a "step" of size 1 in the series of monotonically non-increasing values of column j of $DELTA$. (A step of size k is represented by k different Borderline Points).

Clearly, $DELTA$ has up to ψn Borderline Points.

Observation 3
For any two rows x_1, x_2 where $x_1 \leq Borderline[\alpha, j] < x_2$.

$$OUT[x_2, j] - OUT[x_2, j-1] < OUT[x_1, j] - OUT[x_1, j-1]$$

During the encoding stage, the $O(n\ell)$ time complexity algorithm of [[27], section 6] is used to compute the Borderline Points of $DELTA$.

6.2. The Alignment Stage.

We will now present an algorithm which uses the pre-compiled *Borderline Points* and input row I to compute output row O. The new algorithm will employ the Candidate List concept ([7], [10], [15], [22]).

DEFINITION 6.3. The **Candidate List** is a subset of the rows of OUT, which includes only those rows which are candidates to contain future OUT column minima. The Candidate List is updated from one iteration of the alignment stage algorithm to the next. It is sorted by increasing OUT value and increasing row index. At iteration j of the alignment stage algorithm, the candidate of smallest row index in the list bears the minimal value at column j of OUT.

A high-level outline of the alignment stage algorithm is given below.

Procedure Alignment Stage:
input: The set of Borderline Points for the $DELTA$ matrix, and input row I
output: The output row O

for $j := 0$ to $|T|$ *do*
1 Append row j to the Candidate List.
2 Update the contents of the Candidate List.
3 Report output entry O_j.

The list contents are updated at each iteration by removing rows which are no longer candidates to produce future column minima. We will denote such rows as *extinct*, according to total monotonicity condition 1.

DEFINITION 6.4. A row x_1 is **extinct** if $\{\exists\ x_2 > x_1 \mid OUT[x_2, j] \leq OUT[x_1, j]\}$.

Hence, for any output value achieved during the computation of O_j, we only need to keep one representative.

A candidate becomes $extinct$, by definition 6.4, as a result of two possible events.

Event 1. A $Bordeline[\alpha, j]$. Let x_1, x_2 denote two rows which appear sequentially on the candidate list at iteration j, where $x_1 \leq Bordeline[\alpha, j] < x_2$. The Candidate List is sorted, and therefore, by Observation 3, a $Bordeline[\alpha, j]$ could result in x_2 reaching an identical value to x_1 at column j of OUT. As a result, row x_1 will be removed from the list.

Event 2. At iteration j, row j is appended to the list. At this point, j is the highest row index in the list, and therefore all the elements with an OUT value which is higher than or equal to that of candidate j will be removed from the list.

Note that two technical challenges need to be met, in order to implement a list manipulation engine, which updates the contents of the Candidate List in linear time.

1. Computing the OUT value of a candidate.
2. Efficiently accessing the rows to be removed from the Candidate List.

In the next two subsections we will show how to overcome these technical challenges, while maintaining the linearity of the alignment stage algorithm.

6.2.1. Supplementing the Unavailable $DIST$ Values

The values of OUT are needed for two purposes. One is the comparison of two adjacent candidates. The other is for reporting the OUT value O_j.

We will keep track of the difference in OUT values between the members of the Candidate List.

DEFINITION 6.5. Let $gap[x_1]$ denote the difference in OUT values between candidate x_1 and the candidate which immediately follows x_1 on the Candidate List.

DEFINITION 6.6. Let $Offset_j$ denote the difference in OUT values between the candidate of lowest row index and the candidate of highest row index on the Candidate List, at the end of iteration j of the alignment stage algorithm.

DEFINITION 6.7. Let $DeleteY$ denote the total weight of deleting the whole string Y. (For all $0 \leq j \leq n$, $DIST[j, j] = DeleteY$.)

$DeleteY$ can be computed once in the encoding stage in $O(\ell)$ time.

The candidate of highest row index on the list at iteration j is row j, and $OUT[j, j] = I_j + DeleteY$. Hence, the value of the candidate of lowest row index on the list at iteration j, which bears the minimum for column j of OUT, is

$$O_j = I_j + DeleteY - Offset_j$$

It remains to show how to update the value of $Offset_j$ from the value of $Offset_{j-1}$. During the first iteration of the alignment stage algorithm, the list contains only one candidate, and hence $Offset_0$ is initialized to zero (see Figure 7). Given the information from iteration $j - 1$, the alignment stage algorithm proceeds at iteration j as follows.

Event 1. A $Borderline[\alpha, j]$.

Let x_1, x_2 denote two rows which appear sequentially on the Candidate List at iteration j, where $x_1 \leq Borderline[\alpha, j] < x_2$.

$gap[x_1]$ is reduced by one, as a result of $Borderline[\alpha, j]$. Correspondingly, $Offset_j$ is reduced by one as well.

- If $gap[x_1] > 0$, row x_1 remains on the list.
- If $gap[x_1] = 0$, row x_1 is removed from the list.

Event 2. Candidate j joins the list.

At the end of iteration $j - 1$, row $j - 1$ is the candidate of highest row index on the list. It was appended at the end of iteration $j - 1$, and could not have been removed by any of the events 1,2 of iteration $j - 1$. Row j is appended to the end of the Candidate List at iteration j, as the candidate of highest row index.

As a result, $gap[j - 1]$ is set as follows.

$$gap[j-1] = OUT[j,j] - OUT[j-1,j] = (I_j + DeleteY) - (I_{j-1} + DIST[j-1,j]).$$

Input:

I_0	I_1	I_2	I_3	I_4	I_5	I_6	I_7	I_8	
3	3	3	2	1	2	3	4	5	*input row*
Borderline Points :									
-	-	-	-	-	**2**	**2**	**1**	**5**	$BorderlinePoint[1, j]$
-	-	-	-	-	-	**3**	**1**	**5**	$BorderlinePoint[2, j]$

Run-time variable trace:

Candidate List contents ($row\ index/gap[row\ index]$**) at the end of each iteration:**

0/0	0/1	0/1	0/1	0/1	4/2	4/2	4/2	4/2
	1/0	1/1	1/1	4/0	5/0	5/2	5/2	6/2
		2/0	3/0			6/0	6/2	7/2
							7/0	8/0

Maintained $Offset_j$ **values:**

0	1	2	2	1	2	4	6	6

Output:

O_0	O_1	O_2	O_3	O_4	O_5	O_6	O_7	O_8	
7	6	5	4	4	4	3	2	3	*output row*

FIG. 7. A trace of the contents of the Candidate List and maintained variables, during iterations 0 to 8 of the Alignment Stage Algorithm, as generated while computing the distance between T = "DCBADBDC" and Y = "DCBD", given the input row I for $B_i = "CBA"$. Note that $DeleteY = 4$. This figure continues the example from Figures 4 and 6.

Correspondingly, $Offset_j$ is increased by $gap[j-1]$.

- If $gap[j-1] > 0$, row $j-1$ remains on the list.
- If $gap[j-1] \leq 0$, candidate $j-1$ is removed from the list.

If $gap[j-1] < 0$, we continue popping candidates off the list and correcting gap values, until the candidate of highest row index preceding j is reached, whose gap value remains positive.

When computing $gap[j-1]$, the terms I_j, I_{j-1} and $DeleteY$ are available from the input to the efficient alignment stage. It remains to show how to obtain $DIST[j-1, j]$, under the constraint that the entries of $DIST$ are unavailable during the efficient alignment stage. $DIST[j-1, j] = Edit(T_j^j, Y)$, and all values $Edit(T_j^j, Y)$ for $j = 1 \ldots n$ can be computed during the encoding stage in

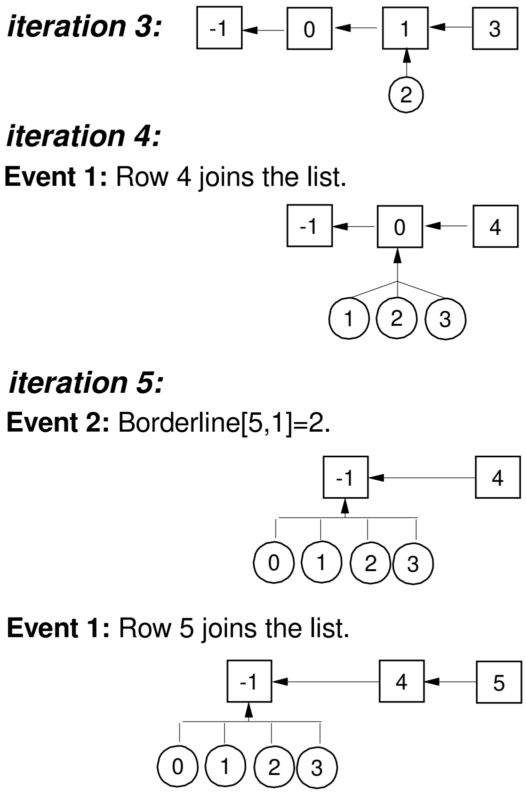

FIG. 8. The Set Union implementation of the Candidate List, as traced through iterations 4 and 5 of the efficient alignment stage algorithm. The first candidate is a "fake" row of value -1, which is intended to represent all rows which are smaller than the candidate of lowest row index in the list. This figure follows the example of Figure 7.

a total of $O(n + \ell \times \min(n, \Sigma))$ time, where Σ denotes the size of the sequence alphabet.

6.2.2. A Candidate List Implementation Using a Disjoint Set Union Algorithm.

Since not all rows of OUT appear in the Candidate List, finding the row to be removed as a result of a Borderline Point is not trivial (see Event 1). We propose to implement the Candidate List by employing the *incremental tree set union* algorithm described in [[9], pp. 216], for the special case in which the union tree is a path.

In this implementation, each row in the Candidate List will serve as the appointed representative of its set, which includes all rows up to and excluding the next candidate of higher row index on the list (see Figure 8). A new candidate is appended as the representative of a one-row set to the end of the list.

A $Find(Borderline[\alpha, j])$ operation will query representative candidate x_1, and $gap[x_1]$ will then be reduced by one. If this results in $gap[x_1]$ reaching a value of zero - x_1 will be removed from the Candidate List, and its set will be united with the set represented by the previous candidate of lower row index on the list.

6.3. Time Analysis of the Efficient Algorithm.

In the encoding stage the Borderline Points are computed in $O(n\ell)$ time using [27].

We can now state and prove the following time complexity bound on the alignment stage algorithm.

THEOREM 6.1. *The alignment stage of the efficient algorithm computes output row O, from input row I and the Borderline Points for the comparison of Y with T, in $O(n)$ time.*

Proof. The alignment stage algorithm iterates n times. The cost of list access operations is as follows.

Event 1: Using the Disjoint Set Union algorithm from [9], the Candidate List can be maintained to support operations of candidate access, due to a Borderline Point, in $O(1)$ amortized time.

Event 2: Since the list is sorted, the rows removed as a result of the addition of row j to the end of the list are sequential candidates, and therefore can be accessed in $O(1)$ time.

Each candidate row is added once to the list, and hence is removed from the list at most once. As a separate category, we will count those candidates which are examined without being removed. This can happen once per Borderline Point (Event 1), and once per addition of a new candidate to the list (Event 2). The number of Borderline Points is at most ψn, and n candidates are added to the list throughout the execution of the alignment stage algorithm. Therefore, the total complexity of the alignment stage algorithm is $O(n)$. ∎

Note that during the encoding stage, the borderline points for the comparison of the prefix Y_1^j with T, can be incrementally computed in $O(n)$ time from the borderline points for the comparison of Y_1^{j-1} with T, using [27]. Hence, for source sequences with two or more common factors, the time complexity of the encoding stage is further reduced to $O(nD)$, where D is the number of nodes in the dictionary trie for the common factors.

7. CONCLUSIONS AND OPEN PROBLEMS

Two algorithms were described for the Common Substring Alignment problem. The second algorithm, which requires an $O(n\ell)$ time encoding stage and has a non-recursive, linear alignment stage, is more applicable to the typical Common Substring Alignment applications then the first algorithm, which requires an $O(n^2 + n\ell)$ time encoding stage and has a recursive, linear alignment stage.

The solutions presented in this paper are intended for those applications where the source strings contain shared and repeated substrings. A special challenge is presented when the target strings contain encoded repetitions as well as the source strings.

Another challenge is to try to extend the solutions presented in this paper to support affine or concave gap costs.

8. ACKNOWLEDGEMENTS

The authors are grateful to the referees for their helpful comments.

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. Apostolico, A., M. Atallah, L. Larmore, and S. McFaddin, Efficient parallel algorithms for string editing problems. *SIAM J. Comput.*, **19**, 968-998 (1990).
4. Benson, G., A space efficient algorithm for finding the best nonoverlapping alignment score, *Theoretical Computer Science*, **145**, 357-369 (1995).
5. Buechner, A.G., and M. Mulvenna, Discovering Internet Marketing Intelligence through Online Analytical Web Usage Mining, *SIGMOD Record*, **27**, 4, 54–61 (1998).
6. Chen, M.S., J.S. Park, and P.S. Yu, Data mining for path traversal patterns in a web environment, *16th International Conference on Distributed Computing Systems*, 385–392 (1996).
7. Eppstein, D., Z. Galil, and R. Giancarlo, Speeding Up Dynamic Programming, *Proc. 29th IEEE Symp. on Foundations of Computer Science*, 488-296 (1988).
8. Farrell, R., P. Fairweather, and E. Breimer, A Task-based Architecture for Application-aware Adjuncts, *Proceedings of the 2000 International Conference on Intelligent User Interfaces*, ACM Press, 82–85 (2000).
9. Gabow, H.N., and R.E. Tarjan, A Linear Time Algorithm for a Special Case of Disjoint Set Union. *J. Comput. Syst. Sci*, **30**, 209-221 (1985).
10. Galil, Z., and R. Giancarlo, Speeding Up Dynamic Programming with Applications to Molecular Biology, *Theoretical Computer Science*, **64**, 107-118 (1989).

11. Gelfand, M.S., A.A. Mironov, and P.A. Pevzner, Gene Recognition Via Spliced Sequence Alignment, *Proc. Natl. Acad. Sci. USA*, **93**, 9061–9066 (1996).

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

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

14. Hirschberg, D.S., A Linear Space Algorithm for Computing Maximal Common Subsequences, *Communications of the ACM*, **18**, 6, 341–343 (1975).

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

16. 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).

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

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

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

20. Lee, J., M. Podlaseck, E. Schonberg, and R. Hoch, Visualization and Analysis of Clickstream Data of Online Stores for Understanding Web Merchandising, *Journal of Data Mining and Knowledge Discovery*, **5**, 1/2, 59–84 (2001).

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

22. Miller, W., and E.W. Myers, Sequence Comparison with Concave Weighting Functions, *Bull. Math. Biol.*, **50**, 97-120 (1988).

23. Mironov, A.A., M.A. Roytberg, P.A. Pevzner, and M.S. Gelfand, Performance-Guarantee Gene Predictions Via Spliced Alignemnt, *Genomics 51 A.N. GE985251*, 332–339 (1998).

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

25. Myers, E.W., Seeing Conserved Signals: Using Algorithms to Detect Similarities Between Biosequences, *Calculating the Secrets Of Life*, Lander and Waterman Editors, National Academy Press, 56–89 (1995).

26. Roytberg, M.A., T.V. Astakhova, and M.S. Gelfand, Combinatorial Approaches to Gene Recognition, *Computers Chemistry*, **21**, 4, 229–235 (1997).

27. 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).

28. Snapp, S.R., J. Brentano, G.V. Dias, T.L. Goan, T. Grance, L.T. Heberlein, C. Ho, K.N. Levitt, B. Mukerjee, D.L. Mansur, K.L. Pon, and S.E. Smaha, A System for Distributed Intrusion Detection, *COMPCON Spring 91 - the 36th IEEE International Computer Conference*, 170–176 (1991).

29. Sze, S-H., and P.A. Pevzner, Las Vegas Algorithms for Gene Recognition: Suboptimal and Error-Tolerant Spliced Alignment, *J. Comp. Biol.* **4**, 3, 297–309 (1997).

30. Ukkonen, E., Finding Approximate Patterns in Strings, *J. Algorithms* **6**, 132–137 (1985).

31. Ziv-Ukelson, M., Y. Horesh, G.M. Landau, R. Unger, Using Repeats to Speed-Up DNA Sequence Alignment, *private communication*.