Computing Evolutionary Chains in Musical Sequences

Maxime Crochemore
Institut Gaspard-Monge
Université de Marne-la-Vallée
E-mail: mac@univ-mlv.fr

Costas S. Iliopoulos[*]
King's College London
Dept. Computer Science
csi@dcs.kcl.ac.uk

Yoan J. Pinzon[†]
King's College London
Dept. Computer Science
pinzon@dcs.kcl.ac.uk

Submitted: March 23, 2000; Accepted: May 22, 2000

Abstract

Musical patterns that recur in approximate, rather than identical, form within the body of a musical work are considered to be of considerable importance in music analysis. Here we consider the "evolutionary chain problem": this is the problem of computing a chain of all "motif" recurrences, each of which is a transformation of ("similar" to) the original motif, but each of which may be progressively further from the original. Here we consider several variants of the evolutionary chain problem and we present efficient algorithms and implementations for solving them.

Keywords: String algorithms, approximate string matching, dynamic programming, computer-assisted music analysis.

1 Introduction

This paper is focused on string-matching problems which arise in computer-assisted music analysis and musical information retrieval. In a recent article ([4]), a number of string-matching problems as they apply to musical situations were reviewed, and in particular the problem of "Evolution Detection" was introduced and discussed. It was pointed out that no specific algorithms for this problem, either in music or in string-matching in general, exist in the literature. However, it seems that musical patterns, or "motifs" actually 'evolve' in this manner in certain types of composition; an actual case is shown by the successive thematic entries shown in the appended Music Example. A more recent example, from Messiaen's piano work, *Vingt Regards sur L'Enfant Jésus*, is given in [3].

A musical score can be viewed as a string: at a very rudimentary level, the alphabet (denoted by Σ) could simply be the set of notes in the chromatic or diatonic notation, or

[*]Partially supported by the Royal Society grant CCSLAAR.
[†]Partially supported by an ORS studentship.

at a more complex level, we could use the GPIR representation of Cambouropoulos [2] as the basis of an alphabet. Although a musical pattern-detection algorithm using approximate matching (allowing the normal edit operations, insertion, deletion and replacement) will detect the occurrence of an evolving pattern in the early stages of its history, once it becomes too different from the original form (past whatever threshold is set by the algorithm or its parameters) it will naturally be rejected. To detect a musical motif which undergoes continuing "evolutionary" change is a more challenging proposition, and is the object of this paper. Musical patterns that recur in approximate, rather than identical, form within a composition (or body of musical work) are considered to be of considerable importance in music analysis. Simple examples are the familiar cases of the standard "tonal" answer in a conventional fugue, or the increasingly elaborated varied reprises of an 18th-century rondo theme; on a more subtle level, the idée fixe in Berlioz's *Symphonie Fantastique* recurs in a wide variety of different forms throughout the four movements of the symphony. In all these cases, each recurrence can be seen as a transformation of the original motif, and each is roughly equivalently "similar" to the original; a measure of this "similarity" will be preset in an algorithm intended to detect the recurrence of the pattern:

$$A \cdots A' \cdots A'' \cdots A''' \cdots \qquad (a)$$

where each of the strings $A', A'', A''', \ldots$ is similar to A within the maximum edit distance preset in the algorithm.

In this paper we are considering the case where each new recurrence of the pattern is based on the previous one rather than on the original form, somewhat in the manner of a "chain":

$$A \cdots (A)' \cdots ((A)')' \cdots (((A)')')' \cdots \qquad (b)$$

(See Figure 1), where $(X)'$ denotes a string similar to a given string X within the maximum edit distance preset in the algorithm. These two types of pattern-repetition may in practice, of course, be indistinguishable in certain circumstances; in case (b), a variant of the pattern may actually cancel out the effect of a previous variant, so the overall distance from the original may remain within the bounds allowed by an algorithm for detecting patterns in case (a).

This class of musical pattern-repetition is not extremely common, but it does exist, as the musical examples given above demonstrate. As well as the obvious musical-analytical interest in detecting such evolutionary pattern-chains, they have importance in any application where they might be missed in detecting approximate repetitions of a pattern (case (a)). These would include automated music-indexing systems for data-retrieval, in which each variant of a motif needs to be detected for efficient indexing; for obvious reasons, it would be desirable for the original pattern, rather than arbitrarily-selected successive variants, to appear as a term in the index table.

Approximate repetitions in musical entities play a crucial role in finding musical similarities amongst different musical entities. The problem of finding a new type of repetitions in a musical score, called *evolutionary chains* is formally defined as follows: given a string t (the "text") and a pattern p (the "motif"), find whether there exists a sequence

$u_1 = p, u_2, \ldots, u_\ell$ occurring in the text t such that, for all $i \in \{1, \ldots, \ell - 1\}$, u_{i+1} occurs to the right of u_i in t and u_i and u_{i+1} are "similar" (i.e. they differ by a certain number of symbols).

There was no specific algorithm for the evolution chain problem in the literature. Landau and Vishkin [12] gave an algorithm (LV Algorithm) for the *string searching with k-differences* problem: given a text of length n over an alphabet Σ, an integer k and a pattern of length m, find all occurrences of the pattern in the text with at most k-differences; the LV algorithm requires $\mathcal{O}(n^2(\log m + \log|\Sigma|))$ running time. The LV method uses a complicated data structure (the suffix tree) that makes their algorithm unsuitable for practical use. Furthermore algorithms for exact repetitions are in [1, 6, 15], approximate repeats treated in [8, 13] and quasiperiodicities in [9, 10]

Here we present an $\mathcal{O}(n^2m/w)$ algorithm for several variants of the problem of computing overlapping evolutionary chains with k differences, where n is the length of the input string, m is the length of the motif and w the length of the computer word. Our methods are practical as well as theoretically optimal. Here we have also studied and implemented the computation of the longest evolutionary chain as well as the chain with least number of errors in total; both algorithms also require $\mathcal{O}(n^2m/w)$ operations.

Several variants to the evolutionary problem are still open. The choice of suitable similarity criteria in music is still under investigation. The use of penalty tables may be more suitable than the k-differences criterion in certain applications. Additionally, further investigation whether methods such as [12] can be adapted to solve the above problems is needed.

2 Basic definitions

Consider the sequences $t_1, t_2, \ldots, t_r$ and $p_1, p_2, \ldots, p_r$ with $t_i, p_i \in \Sigma \cup \{\epsilon\}, i \in \{1 \ldots r\}$, where Σ is an *alphabet*, i.e. a set of symbols and ϵ is the empty string. If $t_i \neq p_i$, then we say that t_i *differs* to p_i. We distinguish among the following three types of differences:

1. A symbol of the first sequence corresponds to a different symbol of the second one, then we say that we have a *mismatch* between the two characters, i.e., $t_i \neq p_i$.

2. A symbol of the first sequence corresponds to "no symbol" of the second sequence, that is $t_i \neq \epsilon$ and $p_i = \epsilon$. This type of difference is called a *deletion*.

3. A symbol of the second sequence corresponds to "no symbol" of the first sequence, that is $t_i = \epsilon$ and $p_i \neq \epsilon$. This type of difference is called an *insertion*.

As an example, see Figure 1; in positions 1 and 3 of t and p we have no differences (the symbols "match") but in position 2 we have a mismatch. In position 4 we have a "deletion" and in position 5 we have a "match". In position 6 we have an "insertion", and in positions 7 and 8 we have "matches". Another way of seeing this difference is that one can transform the t sequence to p by performing insertions, deletions and replacements of

```
            1  2  3  4  5  6  7  8
String t:   B  A  D  F  E  ε  C  A
            |     |     |     |  |
String p:   B  C  D  ε  E  F  C  A
```

Figure 1: Types of differences: mismatch, deletion, insertion

mismatched symbols. (Without loss of generality, in the sequel we omit the empty string ϵ from the sequence of symbols in a string).

Let $t = t_1 t_2 \dots t_n$ and $p = p_1 p_2 \dots p_m$ with $m < n$. We say that p occurs at position q of t with at most k-differences (or equivalently, a *local alignment of p and t at position q with at most k differences*), if $t_q \dots t_r$, for some $r > q$, can be transformed into p by performing at most k of the following operations: inserting a symbol, deleting a symbol and replacing a symbol. Furthermore we will use the function $\delta(x, y)$ to denote the minimum number operations (deletions, insertions, replacements) required to transform x into y.

```
            1  2  3  4  5  6  7  8  9  10 11 12 13
String t:   A  B  C  B  B  A  D  F  E  ε  F  E  A
                     |        |     |                  Alignment 1
String p:            B  C  ε  D  ε  E  F  A  F

            1  2  3  4  5  6  7  8  9  10 11 12 13
String t:   A  B  C  B  B  A  D  F  E  ε  F  E  A
               |  |           |     |                  Alignment 2
String p:      B  C  ε  ε  ε  D  ε  E  F  A  F

            1  2  3  4  5  6  7  8  9  10 11 12 13
String t:   A  B  C  B  B  A  D  F  E  ε  F  E  A
                        |     |     |                  Alignment 3
String p:               B  C  D  ε  E  F  A  F

            1  2  3  4  5  6  7  8  9  10 11 12 13
String t:   A  B  C  B  B  A  D  ε  F  E  F  E  A
                        |     |     |     |            Alignment 4
String p:               B  C  D  E  F  A  F
```

Figure 2: String searching with k-differences.

Let the text $t = ABCBBADFEFEA$ and the pattern $p = BCDEFAF$ (see Figure 2). The pattern p occurs at position 4 of t with at most 6 differences. The pattern p also occurs at position 2 with 7 differences and position 5 with 5 or 4. The alignment (or alignments) with the minimum number of differences is called an *optimal alignment*.

In the sequel we also make use of the following graph-theoretic notions: A *directed graph* $G = (V, E)$ consist of a set V of vertices (nodes) and a set E of edges (arcs). Let $u, v \in V$, then (u, v) denotes the edge between node u and v. A *path* P from v_1 to v_k is a sequence of nodes $P =< v_1, v_2, \dots, v_k >$. P is said to be *simple* iff the nodes are unique. A *cycle* in G is a path such that $v_1 = v_k$. A *directed acyclic graph (DAG)* is a directed graph without cycles. The *in-degree* d_i^{in} of node i is the number of incoming edges to i.

		0	1	2	3	4	5	6	7
		ϵ	G	G	G	T	C	T	A
0	ϵ	0	0	0	0	0	0	0	0
1	G	1	0	0	0	1	1	1	1
2	G	2	1	0	0	1	2	2	2
3	G	3	2	1	0	1	2	3	3
4	T	3	2	1	1	0	1	2	3
5	C	3	2	2	2	1	0	1	2
6	T	3	3	3	3	2	1	0	1
7	A	3	3	3	3	2	2	1	0

Table 1: The evolutionary matrix D for $t = GGGTCTA$ and $m = 3$.

The *out-degree* d_i^{out} of node i is the number of outgoing edges from i. Let $v_s \in V$ be the *source* node and $v_t \in V$ be the *target* node.

Let $c : E \rightarrow \mathbb{Z}$ be a *cost* function on the edges of G. We will also say *weight* instead of cost. We will write $c(v, u)$ to denote the cost of the edge (v, u). The cost of a path $P =< v_1, v_2, \ldots, v_k >$ is defined to be $c(P) = c(v_1, v_2) + \ldots + c(v_{k-1}, v_k)$. The *shortest path* from a node v_s to a node v_t is said to be the minimum $c(P)$ over all possible paths from v_s to v_t.

3 The Evolutionary Matrix

In this section we present a new efficient algorithm for computing the $n \times n$ *evolutionary matrix* D: for a given text t of length n and a given integer m, we define $D(i, j)$ to be the minimum number of differences between $t_{max(1,i-m+1)}, \ldots, t_i$ and any substring of the text ending at position j of t. Informally, the matrix D contains the best scores of the alignments of all substrings of t of length m and any substring of the text. Table 1 shows the evolutionary matrix for $t = GGGTCTA$ and m=3.

One can obtain a straightforward $O(n^2m)$ algorithm for computing the evolutionary matrix D by constructing matrices $D^{(s)}[1..m, 1..n]$, $1 \leq s \leq n - m$, where $D^{(s)}(i, j)$ is the minimum number of differences between the prefix of the pattern $t_{max(1,s-m+1)}, \ldots, t_s$ and any contiguous substring of the text ending at t_j; its computation can be based on the Dynamic-Programming procedure presented in [14]. We can obtain D by collating $D^{(1)}$ and the last row of the $D^{(s)}$, $2 \leq s \leq n - m$.

Here we will make use of word-level parallelism in order to compute the matrix D more efficiently, similar to the manner used by Myers in [16] and Iliopoulos-Pinzon in [11]. But first we need to compute the $n \times n$ *tick-matrix* M: if there is an optimal alignment of $t_{max(1,i-m+1)}, \ldots, t_i$ and any contiguous substring of the text ending at t_j with the property that there is a difference (i.e. insertion, deletion or mismatch) for $t_{max(1,i-m+1)}$, then we

```
EVOLUTIONARY-DP($t$, $m$, $M$)    $\triangleright$ $n = |t|$, $\times = 1$, $\checkmark = 0$
begin
  $D(0..n, 0) \leftarrow min(i, m)$; $D(0, 0..n) \leftarrow 0$   $\triangleright$ initialization
  for $i \leftarrow 1$ until $n$ do
    for $j \leftarrow 1$ until $n$ do
      if $i < m$ then
        $D(i,j) \leftarrow min\{D(i-1,j)+1, D(i,j-1)+1,$
                          $D(i-1,j-1)+\delta(t_i,t_j)\}$
      else
        $D(i,j) \leftarrow min\{D(i-1,j)+1-M(i,j-1), D(i,j-1)+1,$
                          $D(i-1,j-1)+\delta(t_i,t_j)-M(i-1,j-1)\}$
end
```

Figure 3: EVOLUTIONARY-DP algorithm

set

$$M(i,j) \leftarrow \times$$

otherwise we set

$$M(i,j) \leftarrow \checkmark.$$

		0	1	2	3	4	5	6	7
		ϵ	G	G	G	T	C	T	A
0	ϵ	×	×	×	×	×	×	×	×
1	G	×	✓	✓	✓	×	×	×	×
2	G	×	×	✓	✓	✓	×	×	×
3	G	×	×	×	✓	✓	✓	×	×
4	T	×	×	✓	✓	✓	✓	×	×
5	C	×	✓	✓	✓	✓	✓	✓	×
6	T	×	×	×	×	×	✓	✓	✓
7	A	×	×	×	×	×	×	✓	✓

Table 2: The tick-matrix M for $t = GGGTCTA$ and $m = 3$.

Assume that the tick-matrix $M[0..n, 0..n]$ is given. We can use M as an input for the EVOLUTIONARY-DP algorithm (see Fig. 3) to compute the evolutionary matrix $D[0..n, 0..n]$ as follows:

Theorem 3.1 *Given the text t, the motif length m and the tick-matrix M, the* EVOLUTIONARY-DP *algorithm correctly computes the matrix D in $\mathcal{O}(n^2)$ units of time.*

Proof. The computation of $D(i,j)$ for all $1 \le i,j \le m$, in line 7, is done using straightforward dynamic programing (see [14]) and thus their values are correct.

Let's consider the computation of $D(i,j)$ for some $i,j > m$. The value of $D(i,j)$ denotes the minimum number of differences in an optimal alignment of a contiguous substring of the text $t_q \dots t_j$ (for some $1 \le q < j$) and $t_{i-m+1} \dots t_i$; in that alignment t_i can be either to the right of t_j or to the left of t_j or aligned with t_j. It is clear that for $q < j$:

$$D(i,j) = \delta(t_{i-m+1} \dots t_i, t_q \dots t_j) \tag{1}$$

Now, we will consider all three cases. In the case that the symbol t_i is aligned to the right of t_j, for some $q' < j$, we have

$$\delta(t_{i-m+1} \dots t_i, t_q \dots t_j) = \delta(t_{i-m+1} \dots t_{i-1}, t_{q'} \dots t_j) + 1 \tag{2}$$

Let's consider, for some $\hat{q} < j$

$$D(i-1,j) = \delta(t_{i-m} t_{i-m+1} \dots t_{i-1}, t_{\hat{q}} \dots t_j) \tag{3}$$

$$\delta(t_{i-m} t_{i-m+1} \dots t_{i-1}, t_{\hat{q}} \dots t_j) = \delta(t_{i-m}, t_{\hat{q}}) + \delta(t_{i-m+1} \dots t_{i-1}, t_{q'} \dots t_j) \tag{4}$$

Note that

$$\delta(t_{i-m}, t_{\hat{q}}) = M(i, j-1) \tag{5}$$

From equations 1-5 follows that

$$D(i,j) = D(i-1,j) + 1 - M(i,j-1) \tag{6}$$

If $\hat{q} \le q'$, then t_{i-m} is either aligned with $t_{\hat{q}}$ or with ϵ in an optimal alignment of score $\delta(t_{i-m} t_{i-m+1} \dots t_{i-1}, t_{\hat{q}} \dots t_j)$. Thus we have either

$$\delta(t_{i-m} t_{i-m+1} \dots t_{i-1}, t_{\hat{q}} \dots t_j) = \delta(t_{i-m}, t_{\hat{q}}) + \delta(t_{i-m+1} \dots t_{i-1}, t_{\hat{q}-1} \dots t_j) \tag{7}$$

or

$$\delta(t_{i-m} t_{i-m+1} \dots t_{i-1}, t_{\hat{q}} \dots t_j) = \delta(t_{i-m}, \epsilon) + \delta(t_{i-m+1} \dots t_{i-1}, t_{\hat{q}} \dots t_j) \tag{8}$$

It is not difficult to see that

$$\begin{aligned} \delta(t_{i-m+1} \dots t_{i-1}, t_{q'} \dots t_j) &= \delta(t_{i-m+1} \dots t_{i-1}, t_{\hat{q}-1} \dots t_j) \\ &= \delta(t_{i-m+1} \dots t_{i-1}, t_{\hat{q}} \dots t_j) \end{aligned} \tag{9}$$

From 1-3, 5, 7 or 8 and 9, we also derive 6 in this subcase.

In the case that the symbol t_i is aligned to the left of t_j (as above), we have

$$\delta(t_{i-m+1} \dots t_i, t_q \dots t_j) = \delta(t_{i-m+1} \dots t_i, t_{q'} \dots t_{j-1}) + 1 = D(i, j-1) + 1$$

which implies that

$$D(i,j) = D(i,j-1) + 1 \tag{10}$$

In the case that the symbol t_i is aligned with t_j (as above), we have

$$\delta(t_{i-m+1}\dots t_i, t_q \dots t_j) = \delta(t_{i-m+1}\dots t_{i-1}, t_{q'}\dots t_{j-1}) + \delta(t_i, t_j) \tag{11}$$

In a similar manner as in 2-5 we can show that

$$\delta(t_{i-m+1}\dots t_{i-1}, t_{q'}\dots t_{j-1}) = D(i-1, j-1) - M(i-1, j-1) \tag{12}$$

and from 11-12 follows that

$$D(i,j) = D(i-1, j-1) + \delta(t_i, t_j) - M(i-1, j-1) \tag{13}$$

Equations 6, 10 and 13 show that line 9 of the algorithm correctly compute $D(i,j)$ and the algorithm's correctness follows.

The running time of the Evolutionary-DP algorithm can easily be shown to be $O(n^2)$. □

The key idea behind the computation of M is the use of *bit-vector* operations that gives us a theoretical speed up factor of w in comparison to the method presented in [14], where w is the compiler word length; thus on a "64-bit computer word" machine one can obtain a speed up of 64. We maintain the bit-vector

$$B(i,j) = b_\ell ... b_1$$

where $b_r = 1,\ r \in \{1 \dots \ell\},\ \ell < 2m$, if and only if there is an alignment of a contiguous substring of the text $t_q \dots t_j$ (for some $1 \le q < j$) and $t_{i-m+1}\dots t_i$ with $D(i,j)$ differences such that

- The leftmost $r-1$ pairs of the alignment have $\Sigma_\ell^{\ell-r-2} b_j$ differences in total.
- the r-th pair of the alignment (from left to right) is a difference: a deletion in the pattern, an insertion in the text or a replacement.

Otherwise we set $b_r = 0$. In other words $B(i,j)$ holds the binary encoding of the path in D to obtain the optimal alignment at i, j with the differences occurring as leftmost as possible.

Given the restraint that the length m of the pattern is less than the length of the computer word, then the "bit-vector" operations allow to update each entry of the matrix M in constant time (using "shift"-type of operation on the bit-vector). The maintenance of the bit-vector is done via operations defined as follows.

- The $shift$ operation moves the bits one position to the left and enter zeros from the right, i.e. $shift(b_\ell ... b_1) = b_{\ell-1} ... b_1 0$
- The $shiftc$ operation shifts and truncates the leftmost bit, i.e.
  $shift(b_\ell ... b_1) = b_\ell ... b_1 0$

- Given the integers x, y, z, the function $bitmin(x, y, z)$ returns r one of the integers $\{x, y, z\}$ with the property that r has the least number of 1's (bits set on), and if there is a draw then it returns the one with the leftmost bits (i.e. the maximum of the two when they are viewed as a decimal integer).
- The *lastbit* operation returns the leftmost bit, i.e. b_ℓ
- The *or* operation correspond to the bitwise-or operator

The $shift$, $shiftc(x)$, *bitmin*, *lastbit* and *or* operations can be done in $\mathcal{O}(m/w)$ time with $\{|x|, |y|, |z|\} < 2m$.

The algorithm in Fig. 4 computes the matrix $M[0..n, 0..n]$.

```
TICK-MATRIX($t$, $m$)   ▷ $n = |t|$
1   begin
2    $B[0..n, 0] \leftarrow max(i, m)$ 1's; $B[0, 0..n] \leftarrow \epsilon$   ▷ initialization
3    for $i \leftarrow 1$ until $n$ do
4       for $j \leftarrow 1$ until $n$ do
5          if $i < m$ then
6             $B(i,j) \leftarrow bitmin\{shift(B(i-1,j))$ or $1, shift(B(i,j-1))$ or $1,$
                                 $shift(B(i-1,j-1))$ or $\delta(t_i,t_j)\}$
7          else
8             $B(i,j) \leftarrow bitmin\{shiftc(B(i-1,j))$ or $1, shift(B(i,j-1))$ or $1,$
                                 $shiftc(B(i-1,j-1))$ or $\delta(t_i,t_j)\}$
9          if $lastbit(B(i,j))$=1 then $M(i,j) \leftarrow \times$ else $M(i,j) \leftarrow$ ✓
10   return $M$
11  end
```

Figure 4: TICK-MATRIX algorithm

Example. Let the text t be $GGGTCTA$ and m=3, the matrix B (Table 3) is computed to generate the tick-matrix M (Table 2). Notice that $M(i,j) \leftarrow \times$ if and only if the $lastbit(B(i,j)) = 1$ and $M(i,j) \leftarrow$ ✓ otherwise.

Theorem 3.2 *The procedure* TICK-MATRIX *correctly computes the tick-matrix M in $\mathcal{O}(n^2m/w)$ units of time.*

Proof. Lines 6 and 8 of the TICK-MATRIX are binary encodings of lines 6 and 8 respectively of the EVOLUTIONARY-DP procedure. The correctness follows from Theorem 1. □

Theorem 3.3 *The* EVOLUTIONARY-DP *matrix D can be computed in $\mathcal{O}(n^2m/w)$ units of time .*

		0	1	2	3	4	5	6	7
		ϵ	G	G	G	T	C	T	A
0	ϵ								
1	G	1	0	0	0	1	1	1	1
2	G	11	10	00	00	01	11	11	11
3	G	111	110	100	000	001	011	111	111
4	T	111	101	001	001	000	0001	110	111
5	C	111	011	011	011	001	000	0001	101
6	T	111	111	111	111	110	001	000	0001
7	A	111	111	111	111	101	101	001	000

Table 3: The Bit-Vector Matrix B for $t = GGGTCTA$ and $m = 3$.

Proof. The computation of the evolutionary matrix D can be done concurrently with the computation of the matrices M and B. □

Hence, this algorithm runs in $\mathcal{O}(n^2)$ under the assumption that $m \leq w$, where w is the number of bits in a machine word, i.e., in practical terms the running time is $\mathcal{O}(n^2)$. Also, the space complexity can be reduced to $\mathcal{O}(n)$ by noting that each row of B, M and D depends only on the one immediately preceding row of B, M and D respectively.

4 Computing the Longest Non-Overlapping Evolutionary Chain

The problem of the *longest non-overlapping evolutionary chain* (LNOEC) is as follows: given a text t of length n, a pattern p of length m and an integer $k < m/2$, find whether the strings of the sequence $u_1 = p, u_2, \ldots, u_l$ occur in t and satisfy the following conditions:

1. $\delta(u_i, u_{i+1}) \leq k$ for all $i \in \{1, \ldots, \ell - 1\}$
2. Let s_i be the starting position of u_i in t. Then $s_{i+1} - s_i \geq m$ for all $i \in \{1, \ldots, \ell - 1\}$
3. Maximizes ℓ

The method for finding the LNOEC is based on the construction of the evolutionary matrix D presented in the previous section and the graph G defined as follows

Let $G = (V, E)$ be a directed graph where

$$V = \{v_m, \ldots v_n\} \;\cup\; \{v_s, v_t\}$$

$$\begin{aligned} E = \;& \{(v_i, v_j) : D(i,j) \leq k, \;\; i \geq m, \;\; j - i \geq m\} \\ & \cup \{(v_s, v_i) : d_i^{in} = 0, d_i^{out} > 0 \text{ for each } v_i \in V\} \\ & \cup \{(v_i, v_t) : d_i^{in} > 0, d_i^{out} = 0 \text{ for each } v_i \in V\} \end{aligned}$$

```
LNOEC($t$, $m$, $k$)    ▷ $n = |t|$, $G(V, E), V = \{v_m, \ldots, v_n\} \cup \{v_s, v_t\}$
1  begin
2     $D[0..n, 0..n] \leftarrow$ EVOLUTIONARY-DP$(t, m, M)$
3     for $i \leftarrow m$ until $n - m$ do
4        for $j \leftarrow i + m$ until $n$ do
5           if $D(i, j) < k$ then $G.add\_edge(v_i, v_j, -1)$
6     for $i \leftarrow m$ until $n$ do
7        if $d_i^{in} = 0$ and $d_i^{out} > 0$ then $G.add\_edge(v_s, v_i, 0)$
8        if $d_i^{in} > 0$ and $d_i^{out} = 0$ then $G.add\_edge(v_i, v_t, 0)$
9     $P$= SHORTEST-PATH-DAG$(G)$
10 return $P - \{v_s, v_t\}$
11 end
```

Figure 5: LNOEC algorithm

To complete the construction, we define the cost of an edge as follows

$$c(v_i, v_j) = \begin{cases} 0 & , \text{ if } v_i, v_j \in \{v_s, v_t\} \\ -1, & \text{otherwise} \end{cases} \tag{14}$$

The problem of finding the LNOEC is equivalent to the problem of finding a shortest (i.e., least-cost) source-to-sink path in G. Let us denote the shortest path of G ($s \rightsquigarrow t$) by $P =< v_s, u_1, \ldots, u_\ell, v_t >$. Then we say that the length of the LNOEC is ℓ.

The time complexity for finding the shortest path of a graph G is known to be $\mathcal{O}(|V| \times |E|)$ in the general case. However, our graph G does not have cycles and all the edges are forward (i.e. for each edge $(u, v) \in V$, u appears before v), so G is a *topologically sorted* DAG. Hence, we can compute the shortest path of G in $\mathcal{O}(|V| + |E|)$ time.

Fig. 8 shows the algorithm to compute the LNOEC. Note that the function $G.add\_edge(v_i, v_j, k)$ adds the edge (v_i, v_j) to the graph G, assigning $c(v_i, v_j) = k$.

Example. Let the text t be $ABCDADCBAD$, m=3 and k=1. Table 4 shows the evolutionary matrix for the given input. Fig. 6 contains the resulting topologically sorted DAG, the shortest path $P =< v_s, 3, 7, 10, v_t >$ (shaded edges), spell out the longest non-overlapping evolutionary chain, which is $\{ABC, ADC, AD\}$.

4.1 Running time

Assuming that $m \leq w$, the time complexity of the algorithm LNOEC is easily seen to be dominated by the complexity of the EVOLUTIONARY-DP algorithm (see Fig. 5 line 2). Hence, the overall complexity for the LNOEC problem will be $\mathcal{O}(n^2)$. Fig. 7 shows the timing[1] for different values of m and n. This algorithm can be generalized for any value

[1] Using a SUN Ultra Enterprise 300MHz running Solaris Unix.

		1	2	3	4	5	6	7	8	9	10
		A	B	C	D	A	D	C	B	A	D
1	A	0	1	1	1	0	1	1	1	0	1
2	B		0	1	2	1	1	2	1	1	1
3	C			0	1	2	2	**1**	2	2	2
4	D				0	1	2	2	2	2	**1**
5	A					0	1	2	2	**1**	2
6	D						0	1	2	2	**1**
7	C							0	1	2	**1**
8	B								0	1	2
9	A									0	1
10	D										0

Table 4: The Evolutionary Matrix D for $t = ABCDADCBAD$ and $m = 3$.

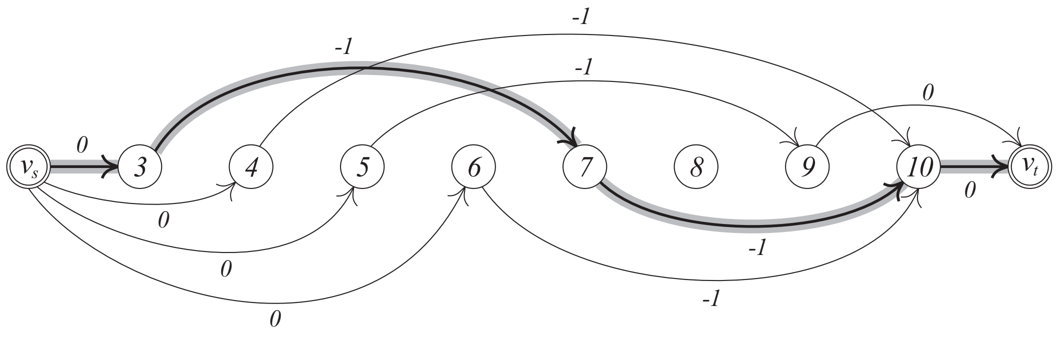

Figure 6: Graph G for $t = ABCDADCBAD$ and $m = 3$.

of m in which case the overall complexity will be $\mathcal{O}(n^2/m)$. The space complexity will be $\mathcal{O}(n + |V| + |E|)$.

5 Computing the Longest Nearest-Neighbor Non-Overlapping Evolutionary Chain

The problem of the *longest nearest-neighbor non-overlapping evolutionary chain* (LNN-NOEC) is as follows: given a text t of length n, a pattern p of length m and an integer $k < m/2$, find whether the strings of the sequence $u_1 = p, u_2, \ldots, u_\ell$ occur in t and satisfy

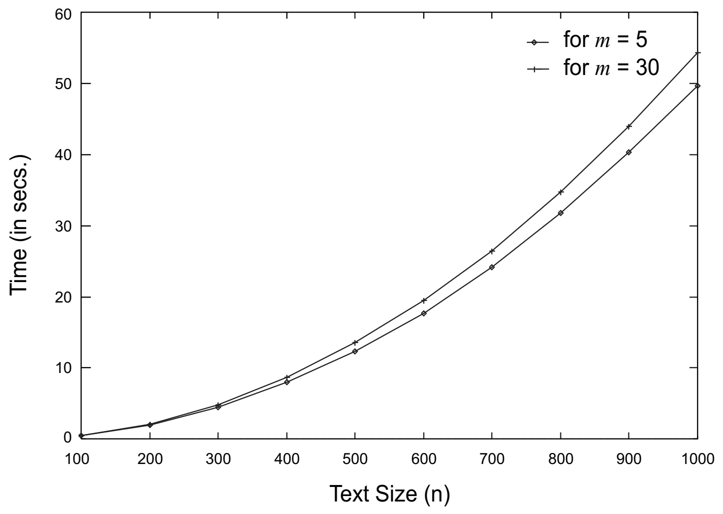

Figure 7: Timing curves for the LNOEC Procedure.

the conditions for LNOEC and minimizes

$$d = \sum_{i=1}^{\ell-1} \gamma_i$$

where γ_i is usually the length of the substring (gap) between motif occurrences in the evolutionary chain, i.e. $\gamma_i = f(s_{i+1} - s_i - m)$, where f is a penalty table.

The computation of the LNNNOEC can be accomplished by redefining equation 14 as follows

$$c(v_i, v_j) = \begin{cases} 0 & , \text{ if } v_i, v_j \in \{v_s, v_t\} \\ -n + f(s_{i+1} - s_i - m), & \text{otherwise} \end{cases} \tag{15}$$

For simplicity, let us assume $f(x) = x$. Fig. 8 shows the algorithm to compute the LNNNOEC.

6 Computing the Longest Minimum-Weight Non-Overlapping Evolutionary Chain

The problem of the *longest minimum-weight non-overlapping evolutionary chain* (LMWNOEC) is as follows: given a text t of length n, a pattern p of length m and an integer $k < m/2$, find whether the strings of the sequence $u_1 = p, u_2, \ldots, u_\ell$ occur in t and satisfy the conditions for LNOEC and minimizes

$$e = \sum_{i=1}^{\ell-1} \delta(u_i, u_{i+1})$$

```
LNNNOEC($t$, $m$, $k$)    $\triangleright$  $n = |t|$, $G(V, E), V = \{v_m, \dots, v_n\} \cup \{v_s, v_t\}$
1  begin
2     $D[0..n, 0..n] \leftarrow$ EVOLUTIONARY-DP$(t, m, M)$
3     for $i \leftarrow m$ until $n - m$ do
4        for $j \leftarrow i + m$ until $n$ do
5           if $D(i, j) < k$ then $G.add\_edge(v_i, v_j, -n + j - i - m)$
6     for $i \leftarrow m$ until $n$ do
7        if $d_i^{in} = 0$ and $d_i^{out} > 0$ then $G.add\_edge(v_s, v_i, 0)$
8        if $d_i^{in} > 0$ and $d_i^{out} = 0$ then $G.add\_edge(v_i, v_t, 0)$
9     $P$= SHORTEST-PATH-DAG$(G)$
10 return $P - \{v_s, v_t\}$
11 end
```

Figure 8: LNNNOEC algorithm

A slightly modification of the LNOEC cost function (Equation 14) will solve the problem

$$c(v_i, v_j) = \begin{cases} 0 & , \text{ if } v_i, v_j \in \{v_s, v_t\} \\ -n + \delta(u_i, u_{i+1}), & \text{otherwise} \end{cases} \tag{16}$$

Fig. 9 shows the algorithm for the LMWNOEC problem.

```
LMWNOEC($t$, $m$, $k$)   ▷ $n = |t|, G(V, E), V = \{v_m, \ldots, v_n\} \cup \{v_s, v_t\}$
1  begin
2    $D[0..n, 0..n] \leftarrow$ EVOLUTIONARY-DP$(t, m, M)$
3    for $i \leftarrow m$ until $n - m$ do
4      for $j \leftarrow i + m$ until $n$ do
5        if $D(i, j) < k$ then $G.add\_edge(v_i, v_j, -n + D(i, j))$
6    for $i \leftarrow m$ until $n$ do
7      if $d_i^{in} = 0$ and $d_i^{out} > 0$ then $G.add\_edge(v_s, v_i, 0)$
8      if $d_i^{in} > 0$ and $d_i^{out} = 0$ then $G.add\_edge(v_i, v_t, 0)$
9    $P=$ SHORTEST-PATH-DAG$(G)$
10 return $P - \{v_s, v_t\}$
11 end
```

Figure 9: LMWNOEC algorithm

7 Conclusion and Open problems

Here we presented practical algorithms for the computation of several variants of the evolutionary chain problem. The the Longest Non-Overlapping Evolutionary Chain, Computing the Longest Nearest-Neighbor Non-Overlapping Evolutionary Chain and Computing the Longest Minimum-Weight Non-Overlapping Evolutionary Chain, which are of practical importance.

The problems presented here need to be further investigated under a variety of *similarity* or *distance* rules (see [5]). For example, *Hamming distance* of two strings u and v is defined to be the number of substitutions necessary to get u from v (u and v have the same length).

Finally comparisons of the empirical results obtained and to those that can be obtained from software library of string algorithms (see [7]) should be drawn.

References

[1] A. Apostolico and F. P. Preparata, Optimal Off-line Detection of Repetitions in a String, *Theoretical Computer Science*, 22 3, pp. 297–315 (1983).

[2] E. Cambouropoulos, A General Pitch Interval Representation: Theory and Applications, *Journal of New Music Research* 25, pp. 231–251 (1996).

[3] E. Cambouropoulos, T. Crawford and C.S. Iliopoulos, (1999) Pattern Processing in Melodic Sequences: Challenges, Caveats and Prospects. In Proceedings of the AISB'99 Convention (Artificial Intelligence and Simulation of Behaviour), Edinburgh, U.K., pp. 42–47 (1999).

[4] T. Crawford, C. S. Iliopoulos, R. Raman, String Matching Techniques for Musical Similarity and Melodic Recognition, *Computing in Musicology*, Vol 11, pp. 73–100 (1998).

[5] T. Crawford, C. S. Iliopoulos, R. Winder, H. Yu, Approximate musical evolution, in the Proceedings of the 1999 Artificial Intelligence and Simulation of Behaviour Symposium (AISB'99), G. Wiggins (ed), The Society for the Study of Artificial Intelligence and Simulation of Behaviour, Edinburgh, pp. 76–81 (1999).

[6] M. Crochemore, An optimal algorithm for computing the repetitions in a word, *Information Processing Letters* 12, pp. 244–250 (1981).

[7] M. Crochemore, C.S. Iliopoulos and H. Yu, Algorithms for computing evolutionary chains in molecular and musical sequences, *Proceedings of the 9–th Australasian Workshop on Combinatorial Algorithms* Vol 6, pp. 172–185 (1998).

[8] V. Fischetti, G. Landau, J.Schmidt and P. Sellers, Identifying periodic occurences of a template with applications to protein structure, *Proc. 3rd Combinatorial Pattern Matching*, Lecture Notes in Computer Science, vol. 644, pp. 111–120 (1992).

[9] C. S. Iliopoulos and L. Mouchard, An $O(n \log n)$ algorithm for computing all maximal quasiperiodicities in strings, *Proceedings of CATS'99: "Computing: Australasian Theory Symposium"*, Auckland, New Zealand, Lecture Notes in Computer Science, Springer Verlag, Vol 21 3, pp. 262–272 (1999).

[10] C. S. Iliopoulos, D. W. G. Moore and K. Park, Covering a string, *Algorithmica* 16, pp. 288–297 (1996).

[11] C. S. Iliopoulos and Y.J. Pinzon, The Max-Shift Algorithm, submitted.

[12] G. M. Landau and U. Vishkin, Fast parallel and serial approximate string matching, in *Journal of Algorithms* 10, pp. 157–169 (1989).

[13] G. M. Landau and J. P. Schmidt, An algorithm for approximate tandem repeats, in *Proc. Fourth Symposium on Combinatorial Pattern Matching*, Springer-Verlag Lecture Notes in Computer Science 648, pp. 120–133 (1993).

[14] G. M. Landau and U. Vishkin, Introducing efficient parallelism into approximate string matching and a new serial algorithm, in *Proc. Annual ACM Symposium on Theory of Computing*, ACM Press, pp. 220–230 (1986).

[15] G. Main and R. Lorentz, An $O(n \log n)$ algorithm for finding all repetitions in a string, *Journal of Algorithms* 5, pp. 422–432 (1984).

[16] E. W. Myers , A Fast Bit-Vector Algorithm for Approximate String Matching Based on Dynamic Progamming, in *Journal of the ACM* 46 3, pp. 395–415 (1999).