HAL
archives-ouvertes.fr

Algorithms for computing evolutionary chains in molecular and musical sequences

Maxime Crochemore, Costas S. Iliopoulos, Hiafeng Yu

▶ To cite this version:

Maxime Crochemore, Costas S. Iliopoulos, Hiafeng Yu. Algorithms for computing evolutionary chains in molecular and musical sequences. Proceedings of the ninth Australian Workshop on Combinatorial Algorithms AWOCA'98 (Perth, 1998), 1998, France. pp.172-184. hal-00619988

HAL Id: hal-00619988

https://hal-upec-upem.archives-ouvertes.fr/hal-00619988

Submitted on 26 Mar 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Algorithms for Computing Evolutionary Chains in Molecular and Musical Sequences

Maxime Crochemore[1*], Costas S. Iliopoulos[2**], and Hiafeng Yu[3]

[1] Institut Gaspard-Monge, Université de Marne-la-Vallée, Cité Descartes, 5 Bd Descartes, Champs-sur-Marne, F-77454 Marne-la-Vallée CEDEX 2, France.
`mac@univ-mlv.fr`
WWW home page `http://www-igm.univ-mlv.fr/~mac`

[2] Dept. Computer Science, King's College London, London WC2R 2LS, England, and School of Computing, Curtin University of Technology, GPO Box 1987 U, WA.
`csi@dcs.kcl.ac.uk`,
WWW home page `http://www.dcs.kcl.ac.uk/staff/csi`

[3] Dept. Computer Science, King's College London, London WC2R 2LS, England.
`yuh@dcs.kcl.ac.uk`
WWW home page `http://www.dcs.kcl.ac.uk/pg/yuh`

Abstract. The problem of finding evolutionary chains is defined as follows: given a string t ("the text") and a pattern p (the "motif"), find whether there exists a sequence $u_1 = p, u_2, \ldots, u_l$ occurring in the text t such that u_{i+1} occurs to the right of u_i in t and u_i and u_{i+1} are "similar" (i.e. the differ by a certain number of symbols). Here we consider several variants of the evolutionary chain problem and we present efficient algorithms for solving them.

Keywords: String algorithms, approximate string matching, dynamic programming, molecular sequences, music analysis.

1 Introduction

This paper is focused on a set of string pattern-matching problems which arise in music analysis, musical information retrieval and molecular sequence analysis. A musical score can be viewed as a string: at a very rudimentary level, the alphabet could simply be the set of notes in the chromatic or diatonic notation, or at a more complex level, we could use the GPIR representation of Cambouropoulos [3,4] as the basis of an alphabet. Approximate repetitions in musical entities play a crucial role in finding musical similarities amongst different musical entities, as well as playing a part in defining the "characteristic signature" (see [7]). Such algorithms can be used for melody identification and music retrieval e.g. audio applications on Internet systems.

* Partially supported by the C.N.R.S. Program "Génomes"
** Partially supported by the EPSRC grant GR/J 17844.

Furthermore, efficient algorithms for computing the approximate repetitions are also directly applicable to molecular biology (see [10, 15, 18] and in particular in DNA sequencing by hybridization ([24]), reconstruction of DNA sequences from known DNA fragments (see [25,26]), in human organ and bone marrow transplantation as well as the determination of evolutionary trees among distinct species ([25]).

Exact repetitions have been studied extensively. The repetitions can be either concatenated with the original substring or they may overlap or they may not. Algorithms for finding non-overlapping repetitions in a given string can be found in [1, 8, 14, 19, 18, 23] and algorithms for computing overlapping repetitions can be found in [2, 12, 13, 22]. A natural extension of the repetitions problem is to allow the presence of errors; that is, the identification of substrings that are duplicated to within a certain tolerance k (usually edit distance or Hamming distance). Moreover, the repeated substring may be subject to other constraints: it may be required to be of at least a certain length, and certain positions in it may be required to be invariant.

The problem of finding evolutionary chains is defined as follows: given a string t ("the text") and a pattern p (the "motif"), find whether there exists a sequence $u_1 = p, u_2, \ldots, u_l$ occurring in the text t such that u_{i+1} occurs to the right of u_i in t and u_i and u_{i+1} are "similar" (i.e. the differ by a certain number of symbols).

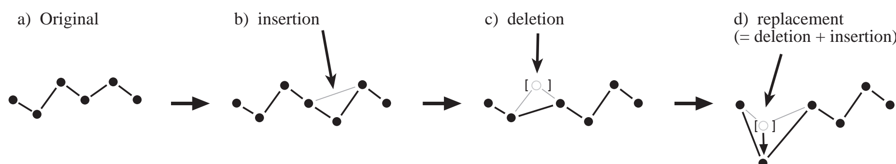


Figure 1
Local approximations in search pattern, trace gradual change ('evolution') in a motif. See Music Example in Appendix

There is no specific algorithm for the evolutionary chain problem in the literature. Landau and Vishkin [16,17] gave an algorithm (LV Algorithm) for the *string searching with k-differences* problem: given a text of length n, and 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 $O(n(\log m + \log|\Sigma|))$ time, where Σ is the alphabet used. A naive way to solve this problem is to repeatedly apply the LV algorithm to the text using u_i as the pattern, for $i = 1, 2, ...$, giving a worst-case $O(n^2(\log m + \log|\Sigma|))$ running time. Here we present a straightforward $O(nm)$ algorithm for computing non-overlapping evolutionary chains with k-differences. We also present an $O(n(\log m + \log|\Sigma|))$ algorithm for the same problem that makes use of suffix trees; this algorithm require $O(kn)$ time for fixed alphabets. Furthermore we present $O(n^2)$ algorithms for several variants of the computing overlapping evolutionary chains with k differences, where n is the size of the input string.

Here we study the computation of the longest evolutionary chain as well as the chain with least number of errors in total. Several variants to the evolutionary chain problem are still open. The choice of suitable similarity criteria in music and biology 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 [11, 17] can be adapted to solve the above problems is needed.

The paper is organised as follows. In the next section we present some basic definitions for strings and background notions for pattern-matching with k-differences. In Section 3 we describe the algorithms for non-overlapping evolutionary chains. In Section 4 we describe the algorithms for several variants of overlapping evolutionary chains. Finally in Section 6 we present our conclusions and open problems.

2 Background and basic string definitions

A *string* is a sequence of zero or more symbols from an alphabet Σ; the string with zero symbols is denoted by ϵ. The set of all strings over the alphabet Σ is denoted by Σ^*. A string x of length n is represented by $x_1 \ldots x_n$, where $x_i \in \Sigma$ for $1 \leq i \leq n$. A string w is a *substring* of x if $x = uwv$ for $u, v \in \Sigma^*$; we equivalently say that the string w occurs at position $|u| + 1$ of the string x. The position $|w| + 1$ is said to be the *starting position* of u in x and the position $|w| + |u|$ the *end position* of u in x. A string w is a *prefix* of x if $x = wu$ for $u \in \Sigma^*$. Similarly, w is a *suffix* of x if $x = uw$ for $u \in \Sigma^*$.

The string xy is a *concatenation* of two strings x and y. The concatenations of k copies of x is denoted by x^k. For two strings $x = x_1 \ldots x_n$ and $y = y_1 \ldots y_m$ such that $x_{n-i+1} \ldots x_n = y_1 \ldots y_i$ for some $i \geq 1$, the string $x_1 \ldots x_n y_{i+1} \ldots y_m$ is a *superposition* of x and y. We say that x and y *overlap*

Let x be a string of length n. A prefix $x_1 \ldots x_p$, $1 \leq p < n$, of x is a *period* of x if $x_i = x_{i+p}$ for all $1 \leq i \leq n - p$. *The period* of a string x is the shortest period of x. A string b is a *border* of x if b is a prefix and a suffix of x.

Consider the sequences $\tau_1\tau_2...\tau_l$ and $\rho_1\rho_2...\rho_l$ with $\tau_i, \rho_i \in \Sigma \cup \{\epsilon\}$, $i \in \{1..l\}$. If $\tau_i \neq \rho_i$, then we say that τ_i *differs* to ρ_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., $\tau_i \neq \epsilon$ and $\rho_i \neq \epsilon$.
2. A symbol of the first sequence corresponds to “no symbol” of the second sequence, that is $\tau_i \neq \epsilon$ and $\rho_i = \epsilon$. 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 $\tau_i = \epsilon$ and $\rho_i \neq \epsilon$. This type of difference is called an *insertion*.

As an example, let the text be $abcdefghi$ and the pattern be $bxdyegh$ (see Figure 2). 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 τ sequence to ρ by performing insertions, deletions and replacements of the mismatched symbols.

	1	2	3	4	5	6	7	8
τ	b	x	d	y	e		g	h
ρ	b	c	d		e	f	g	h

Figure 2
Types of differences: mismatch, insertion, deletion.

Let $t = t_1 t_2 \ldots t_n$ and $p = p_1 p_2 \ldots p_m$ with $m < n$. We say that p occurs at position q of t with at most k-differences if there are a sequences $a_1, ..., a_r$, $b_1, ..., b_r$ such that

1. There is a subsequence of the sequence a such that
$$a_{s_1}, a_{s_2}, ..., a_{s_r} = t_q, t_{q+1}, ...t_{q+r-1} \quad \text{with} \quad s_1 < s_2 < ... < s_r$$
and $a_i = \epsilon$ for all $i \in \{1..r\} - \{s_1, s_2, ...s_r\}$.
2. There is a subsequence of the sequence b such that
$$b_{v_1}, b_{v_2}, ...b_{v_r} = p_1, p_2, ...p_m \quad \text{with} \quad v_1 < v_2 < ... < v_r$$
and $b_i = \epsilon$ for all $i \in \{1..r\} - \{v_1, v_2, ...v_r\}$.
3. The number of differences between the sequence a and b is at most k.
4. There are no sequences that satisfy 1 and 2 and have less than k differences.

The problem of *string searching with k-differences* is defined as follows: given a text $t = t_1 t_2 \ldots t_n$, a pattern $p = p_1 p_2 \ldots p_m$ and an integer k, find all occurrences of the pattern p in the text t with at most k differences.

	1	2	3	4	5	6	7	8	9	10	11	13	14	15	16
t	x	b	c	b	b	x	d	y	e		g	h	x	y	b
p				b	c		d		e	f	g	h			
p		b	c				d		e	f	g	h			
p					b	c	d		e	f	g	h			

Figure 3
String searching with k-differences.

Let the text be $t = abcdefghi$ and the pattern be $p = bxdyegh$ (see Figure 3). The pattern p occurs at position 4 of x with at most 3 differences. The pattern p also occurs in position 2 with at most 5 differences and the pattern p occurs in position 5 with at most 3 differences.

3 Computing Non-overlapping Evolutionary Chains

The problem of *non-overlapping evolutionary chains* (abbreviated NOEC) is as follows: given a text t, an integer k and a pattern p, find whether the strings of the a sequence $u_1 = p, u_2, \ldots, u_l$ occur in the text t such that:

1. The number of differences between any two consecutive strings u_i and u_{i+1} in the evolutionary chain is at most k, for all $i \in \{1..l\}$.
2. The starting position of the string u_{i+1} in t is nearest one to the right of the end position of u_i for all $i \in \{1..l\}$.

The first condition ensures that the strings in the evolutionary chain have errors within some tolerance and the second condition enforces the strings in the chain not to overlap.

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
c	a	b	c	d	e	a	b	d	e	a	b	d	a	b	b	b	b	c	a	a	b	b
	a	b	c	d		a	b	d		a	b	d				b	b				b	b
													a	b	b							

Figure 4

The non-overlapping evolutionary chain for p = abcd with at most one difference.

The pattern p first occurs in position 2. The first re-occurrence of the pattern to the right of position 5 with at most 1-difference is at position 7. Consider the pattern *abb* in position 14. The nearest re-occurrence of *abb* with at most one difference is at position 15 (i.e. the string *bb*) but it is not part of the chain because it overlaps with *abb*; the first re-occurrence with at most one difference that is part of the chain is in position 17. The non-overlapping evolutionary chain is $\{abcd, abd, adb, abb, bb, bb\}$.

3.1 The modified dynamic programming Algorithm

First we consider an $O(nm)$ algorithm for computing the non-overlapping evolutionary chain of a text of length n and a pattern of lenght m. The algorithm NOEC presented below is based on the Dynamic-Programming procedure presented in [16,17]. The main idea is to construct a matrix $D[1..m, 1..n]$, where $D_{i,j}$ is the minimum number of differences between the prefix of the pattern $p_1...p_i$ and any contiguous substring of the text ending at t_j. The Dynamic-programming procedure below terminates when it finds the first occurrence of the pattern with at most k differences.

	G	G	G	T	C	T	A
G	0	0	0	1	1	1	1
G	1	0	0	1	2	2	2
G	2	1	0	1	2	3	3
T	3	2	1	0	1	2	3

Figure 5
The matrix $D_{i,j}$ for $p = GGGT$ and $t = GGGTCTA$

Procedure DYNAMIC-PROGRAMMING(t, p, k)
begin
$n \leftarrow |t|$; $m \leftarrow |p|$;
$D_{i,j} \leftarrow 0,\ 0 \leq i \leq m,\ 0 \leq j \leq n$;
$D_{i,0} \leftarrow i,\ 0 \leq i \leq m$;
for $i := 1$ **to** m **do**
for $j := 1$ **to** n **do**
if $p[i] = t[j]$ **then**
$D_{i,j} = \min\{D_{i-1,j} + 1, D_{i,j-1} + 1, D_{i-1,j-1}\}$
else
$D_{i,j} = \min\{D_{i-1,j} + 1, D_{i,j-1} + 1, D_{i-1,j-1} + 1\}$
if $D_{i,j} \leq k$ **then return** i, u
comment u is the suffix of $t_1 ... t_i$ is the one that achieves the score $D_{i,j}$.
od
od
end

Next, the algorithm NOEC makes repeated applications of the DYNAMIC-PROGRAMMING procedure; every time that an occurrence of the pattern is found ending at position i of the text, then we re-apply the DYNAMIC-PROGRAMMING procedure to the suffix $t_{i+1} ... t_n$.

Algorithm NOEC(t, p, k)
begin
while $i < n$ **do**
$(i, p) \leftarrow$ DYNAMIC-PROGRAMMING(t, p, k)
$t \leftarrow t_{i+1} ... t_n$;
end

Theorem 1. *Algorithm* NOEC *computes the non-overlapping evolutionary chain in $O(nm)$ time and $O(nm)$ space, where n is the length of the input text and m is the length of the pattern.*

3.2 A fast dynamic programming algorithm

The matrix D computed by the DYNAMIC-PROGRAMMING procedure above contains a lot of redundant data which are not of used by algorithm NOEC. In fact

it will suffice to find the index of the largest row of each diagonal of the matrix D, which has an entry less than k. This computation can be done in linear time with the aid of suffix trees (see [1]) . The alternative dynamic programming algorithm given in [17] can be modified as above and it will lead to the following theorem (for details and proofs see [17]).

Theorem 2. *There exists an algorithm that computes the non-overlapping evolutionary chain in $O(kn)$ time for fixed alphabets, where n is the length of the input text and k the is maximum number of differences allowed between consecutive members of the chain.*

Theorem 3. *There exits an algorithm that computes the non-overlapping evolutionary chain in $O(n(\log m + \log|\Sigma|))$ time for a general alphabet Σ, where n is the length of the input text , m is the length of the pattern and k is the maximum number of differences allowed consecutive members of the chain.*

4 Computing Overlapping Evolutionary Chains

The problem of *overlapping evolutionary chains* (abbreviated OEC) is defined as follows: given a text t, a pattern p and an integer $k < |p|/2$, find whether the strings of the a sequence $u_1 = p, u_2, \ldots, u_l$ occur in t and satisfy the following conditions:

1. The number of differences between u_i and u_{i+1} is at most k, for all $i \in \{1..l\}$.
2. Let s_i he starting position of string u_i in t for all $i \in \{1..l\}$. The starting position of u_{i+1} for all $i \in \{1..l\}$ is to the right of $s_i + |u_i|/2$.

In this case we allow the strings of the evolutionary chain to overlap. These strings have been constrained the overlap at most $|p|/2$ symbols. Without such constraint, we can obtain trivial chains such as $u_i = t_i...t_{m-1}$, where u_i and u_{i+1} have at most one difference.

First we present a method for finding all possible members of an overlapping evolutionary chain Let $D_{i,j}$ be as in section 3 but the pattern is identical to the text, i.e. $p = t$; thus D is an $n \times n$ matrix. In order to efficiently compute the matrix $D_{i,j}$ with both $i, j \in \{1..n\}$, we need to evaluate the following matrix M; we mark $M_{i,j} := \surd$ if there is the alignment of $p_1...p_i$ with $t_1...t_j$ with the least number of differences requires that p_1 matching t_l for some l; otherwise we mark $M_{i,j} := \times$.

	G	G	G	T	C	T	A
G	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\times$	$\times$
G	$\times$	$\surd$	$\surd$	$\surd$	$\times$	$\times$	$\times$
G	$\times$	$\times$	$\surd$	$\surd$	$\surd$	$\times$	$\times$
T	$\times$	$\times$	$\times$	$\surd$	$\surd$	$\surd$	$\surd$

Figure 6

The matrix $M_{i,j}$ for $t = GGGTCTA$

The computation of matrix M can easily be done using the matrix D. Consider two consecutive entries in a column of D, say $D_{i,j}$ and $D_{i+1,j}$. We have to consider two cases:

1. The case $p_{i+1} \neq t_j$. If $D_{i+1,j} \leq D_{i,j}$, then the only way that we can align $p_i ... p_{i+1}$ and $t_1 ... t_j$ and achieve $D_{i+1,j}$ differences is by aligning and matching p_1 with t_l for some l; hence $M_{i+1,j} = \surd$. Otherwise $M_{i+1,j} = \times$.
2. The case $p_{i+1} = t_j$. We have the following subcases:
   - $D_{i+1,j} = D_{i,j+1} + 1$. In this case one can see that $M_{i+1,j} = M_{i,j+1}$.
   - $D_{i+1,j} = D_{i,j-1} + 1$. In this case one can see that $M_{i+1,j} = M_{i,j-1}$.
   - $D_{i+1,j} = D_{i-1,j-1}$. In this case one can see that $M_{i+1,j} = M_{i-1,j-1}$.
   - $i = 1$. One can see that $M_{i+1,j} = \times$.

   If more than one of the above subcases hold, then we opt for the one that leads to $M_{i+1,j} = \times$. Thus the computation M can easily done in parallel with the computation of D. In order to simplify the exposition the computation of M is omitted in the pseudocode below.

	G	G	G	T	C	T	A
G	0	0	0	1	1	1	1
G	1	0	0	1	2	2	2
G	2	1	0	1	2	3	3
T	3	2	1	0	1	2	3
C	3	2	2	1	0	1	2

Figure 7
The matrix $D_{i,j}$ for $t = GGGTCTA$ using $M_{i,j}$ with $m = 3$

Let's consider Figure 7. We compute the rows 1,2,3, and 4 of the matrix D as in section 3 for the pattern $p = GGGT$. We also compute the matrix M as above. We will now proceed to compute approximate matches of $GGTC$ with $t_1 ... t_i$ for all i. Lets consider the evaluation of $D_{5,2}$; its value depends on the values of $D_{5,1}, D_{4,1}$ and $D_{4,2}$. If we were to use $D_{5,1}$, then we have to increase its value by 1 for the mismatch of $p_5 = C$ and $t_2 = G$ and decrease it by 1 because $M_{4,1} = \times$; note p_1 is no longer taking part in this alignment. If were to use $D_{4,1}$, then we have to increase its value by 1 for the mismatch of $p_5 = C$ and $t_2 = G$ and decrease it by 1 because $M_{4,1} = \times$. If were to use $D_{4,2}$, then we have to increase its value by 1 for the mismatch of $p_5 = C$ and $t_2 = G$ and decrease it by 1 because $M_{4,1} = \times$. Whenever there is a match then we only use the three neighbouring values unaltered (see Figure 6). The correctness proof will appear in the full paper. The pseudocode for the procedure is outlined below.

Procedure DYNAMIC-PROGRAMMING-II(t, p, k)
 begin
  $n \leftarrow |t|;\ \ m \leftarrow |p|;$
  $D_{i,j} \leftarrow 0,\ 0 \leq l \leq n,\ 0 \leq i \leq m;$

```
D_{i,0} ← i, 0 ≤ i ≤ m;
for i := 1 to n do
  for j := 1 to n do
    if M_{i-1,j} = √ OR i < m then q = 0;
    else q = 1;
    if p[i] = t[j] then
      D_{i,j} = min{D_{i-1,j} + 1 - q, D_{i,j-1} + 1 - q, D_{i-1,j-1}}
    else
      D_{i,j} = min{D_{i-1,j} + 1 - q, D_{i,j-1} + 1 - q, D_{i-1,j-1} + 1 - q}
    if D_{i,j} ≤ k then return i, u
  end
end
```

```
Algorithm OEC(t, p, k)
 begin
  while i < n do
   (i, p) ← DYNAMIC-PROGRAMMING-II(t, p, k)
   t ← t_{i+1}...t_n;
  end
```

Theorem 4. *Algorithm OEC computes the all possible overlapping chains in* $O(n^2)$ *time, where* n *is the length of the input text.*

The definition of the problem OEC does not specify which of the overlapping patterns is chosen as members of the overlapping evolutionary chain. The following variants of the OEC problem give three choices with different criteria.

4.1 Computing the nearest-neighbour evolutionary chain.

The problem of *nearest-neighbour overlapping evolutionary chains* (abbreviated NNOEC) is defined as follows: given a text t, a pattern p and an integer $k < |p|/2$, find whether the strings of the sequence $u_1 = p, u_2, \ldots, u_l$ occur in t and satisfy the conditions for the OEC problem and the string u_{i+1} is the nearest one to the right of $s_i + |u_i|/2$ that has at most k differences with u_i for all $i \in \{1..l\}$.

Next, the algorithm NNOEC makes repeated applications of the DYNAMIC-PROGRAMMING procedure; every time that an occurrence of the pattern is found ending at position i of the text, then we re-apply the DYNAMIC-PROGRAMMING procedure to the suffix $t_{i+1}...t_n$.

```
Algorithm NNOEC(t, p, k)
 begin
  while i < n do
   (i, p) ← DYNAMIC-PROGRAMMING(t, p, k)
   t ← t_{i+|u_i|/2+1}...t_n;
  end
```

Theorem 5. *Algorithm NNOEC computes the nearest neighbour evolutionary chain in $O(nm)$ time, where n is the length of the input text and m is the size of the input pattern.*

The NNOEC algorithm can be speeded up in a similar manner to the procedure DYNAMIC-PROGRAMMING (see Theorems 2 & 3). The details will be shown in the full paper. Hence, we have the following theorems:

Theorem 6. *There exists an algorithm that computes the nearest neighbour evolutionary chain in $O(kn)$ time over fixed alphabets, where n is the length of the input text and k is the maximum number of differences allowed between consecutive members of the chain.*

Theorem 7. *There exists an algorithm that computes the nearest neighbour evolutionary chain in $O(n(\log m + \log(|\Sigma|))$ time over an alphabet $|\Sigma|$, where n is the length of the input text , m is the length of the pattern and k is maximum number of differences allowed between consecutive members of the chain.*

4.2 Computing the maximal length evolutionary chain.

The problem of computing *longest overlapping evolutionary chains* (abbreviated LOEC) is defined as follows: given a text t, a pattern p and an integer $k < |p|/2$, find whether the strings of the a sequence $u_1 = p, u_2, \ldots, u_l$ occur in t and satisfy the conditions as in the OEC problem and maximizes l. The computation of the maximal chain requires the full matrix D as is computed in Theorem 4 and the evaluation of the following recursion

$$l_{max} = \begin{cases} l_j, & \text{if } d_{i,j} < k \text{ and } d_{i,r} > k \ \ \forall r \\ \max_r\{l_{i+j} \text{ and } d_{i,r} > k\}, & \text{otherwise.} \end{cases}$$

where $i + |p|/2 < r \leq |p|$.

4.3 Computing the minimal weight evolutionary chain

The problem of computing *minimal-weight overlapping evolutionary chains* (abbreviated WOEC) is defined as follows: given a text t, a pattern p and an integer $k < |p|$, find whether the strings of the a sequence $u_1 = p, u_2, \ldots, u_l$ occur in t and satisfy the conditions as in the OEC problem and minimize

$$d = \sum_{i=1}^{l} \delta_i + \gamma_i$$

where δ_i is the sum of the differences between u_i and u_{i+1} and $\gamma_i = f(s_{i+1} - s_i - |u_i|)$, where f is a penalty table.

The computation of the maximal chain requires the full matrix D as computed in Theorem 4 and the evaluation of the following recursion

$$w_{min}(i) = \begin{cases} j - i - |p| + w_j, & \text{if } d_{i,j} < k \text{ and } d_{i,r} > k \ \ \forall r \\ \min_r\{w_{i+j} \text{ and } d_{i,r} > k\}, & \text{otherwise.} \end{cases}$$

where $i + |p|/2 < r \leq |p|$.

5 Conclusions and Open problems

Our primary goal is to identify efficient algorithms for computational problems which arise in computer-assisted analysis of music, and to also formalise their relation to well known string pattern-matching problems. The major obstacle to applying computational and mathematical techniques developed in the context of string pattern-matching problems to problems of computer-assisted music analysis appears to be the difficulty of communication and mutual comprehension between computer scientists and musicologists.

The primary direction of this research is towards a formal definition of musical similarity between musical entities (i.e. complete pieces of music or meaningful subsets of pieces, e.g. 'themes' or 'motifs') (See [5,6,7] for details). In particular we are aiming at producing a quantitative measure or 'characteristic signature' of a musical entity. This measure is essential for melodic recognition and it will have many uses including, for example, data retrieval from musical databases.

Here we presented the practical algorithms NOEC and OEC for the computation of non-overlapping and overlapping evolutionary chains. Furthermore, we presented theoretical algorithms for the same problems with improved upper bounds on their time complexity. Additionally we presented two variants of the OEC problem, the maximal evolutionary chains and minimum-weight evolutionary chains, both of which are of practical importance.

The problems presented here need to be further investigated under a variety of *similarity* or *distance* rules (see [7,21]). 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 (to be presented in the full paper) and to those that can be obtained from software library on string algorithms (see [9]) should be drawn.

References

1. A. Apostolico, The myriad virtues of the Suffix Trees, in A. Apostolico and Z. Galil, eds, *Combinatorial Algorithms on Words*, Springer Verlag, NATO ASI Series, 1985. *Theoretical Computer Science* **22** (1983), pp. 297–315.
2. O. Berkman, C. Iliopoulos and K. Park, String covering, *Information and Computation* **123** (1996), pp. 127–137.
3. E. Cambouropoulos, A General Pitch Interval Representation: Theory and Applications, *Journal of New Music Research* **25** (1996), pp. 231–251.

4. E. Cambouropoulos, A formal theory for the discovery of local boundaries in a melodic surface, in *Proceedings of the III Journees d' Informatique Musicale, Caen, France*, 1996.
5. E. Cambouropoulos, The role of similarity in categorisation: Music as a case study. In *Proceedings of the Third Triennial Conference of the European Society for the Cognitive Sciences of Music (ESCOM), Uppsala*, 1997.
6. E. Cambouropoulos and A. Smaill, A Computational Theory for the Discovery of Parallel Melodic Passages, in *Proceedings of the XI Colloquio di Informatica Musicale, Bologna, Italy*, 1995.
7. T. Crawford, C. S. Iliopoulos, R. Raman, String Matching Techniques for Musical Similarity and Melodic Recognition, to appear in *Computing in Musicology*.
8. M. Crochemore, An optimal algorithm for computing the repetitions in a word, *Information Processing Letters* **12** (1981), pp. 244–250.
9. A. Czumaj, P. Ferragina, L. Gasieniec, S. Muthukrishnan and J. Traeff, The architecture of a software library for string processing, to be presented at *Workshop on Algorithm Engineering*, Venice, September 1997.
10. 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, 1992, pp. 111–120.
11. Z. Galil and K. Park, An improved algorithm for approximate string matching, *SIAM Journal on Computing*, **19** (1990), pp. 989–999.
12. C. S. Iliopoulos and L. Mouchard, Fast local covers, (Submitted).
13. C. S. Iliopoulos, D. W. G. Moore and K. Park, Covering a string, *Algorithmica* **16** (1996), pp. 288–297.
14. C. S. Iliopoulos, D. W. G. Moore and W. F. Smyth, A linear algorithm for computing the squares of a Fibonacci string, in P. Eades and M. Moule, eds. *Proceedings CATS'96, "Computing: Australasian Theory Symposium," University of Melbourne*, pp. 55–63, 1996.
15. S. Karlin, M. Morris, G. Ghandour, and M.-Y. Leung, Efficients algorithms for molecular sequences analysis, *Proc. Natl. Acad. Sci., USA (1988) 85:841–845*
16. 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.
17. G.M. Landau and U. Vishkin, Fast string matching with k differences, *Journal of Computer and Systems Sciences*, **37** (1988), pp. 63–78.
18. 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.
19. G. Main and R. Lorentz, An $O(n \log n)$ algorithm for finding all repetitions in a string, *Journal of Algorithms* **5** (1984), pp. 422–432.
20. A. Milosavljevic and J. Jurka, Discovering simple DNA sequences by the algorithmic significance method, *Comput. Appl. Biosci. (1993) 9:407–411*
21. M. Mongeau and D. Sankoff, Comparison of Musical Sequences, *Computers and the Humanities* **24** (1990), pp. 161–175.
22. D. W. G. Moore and W. F. Smyth, Computing the covers of a string in linear time, in *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms*, pp. 511–515, 1994.
23. E. Myers and S. Kannan, An algorithm for locating non-overlapping regions of maximum alignment score, in *Proc. Fourth Symposium on Combinatorial Pattern Matching*, Springer-Verlag Lecture Notes in Computer Science 648, 1993.

24. Pavel A. Pevzner & W. Feldman, Gray Code Masks for DNA Sequencing by Hybridization, *Genomics*, 23, 233-235 (1993).
25. Jeanette P. Schmidt, All shortest paths in weighted grid graphs and its application to finding all approximate repeats in strings, in *Proc. of the Fifth Symposium on Combinatorial Pattern Matching* CPM'94, Lecture Notes in Computer Science (1994).
26. Steven S. Skiena & Gopalakrishnan Sundaram, Reconstructing strings from substrings, *J. Computational Biol.* 2 (1995) 333-353.

Appendix: Music Example

a) Selected entries (in their original sequence):

A (meas. 4)

B (meas. 7)

C (meas. 19)

D (meas. 25)

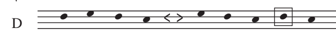
E (meas. 28)

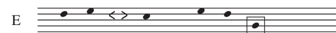
b) 'Evolution' of diatonic-pitch pattern

A

B

C

D


E


< > Deletion ◯ Insertion ☐ Replacement

Music Example Francesco da Milano, monothematic lute recercar (Cavalcanti Lutebook, f. 71v)

The five successive entries, A-E, are audibly related and can be treated as stages in the 'evolution' of a diatonic motif by a series of alterations of edit distance 2 (where the deletion, insertion, replacement and time-displacement operations each have weight 1). Thr example was taken from *Cavalcanti Lutebook*, Brussels, Belgium, Bibliothèque Royale (B-Br), MS II 275.