Evolution of Musical Motifs in Polyphonic Passages

Costas S. Iliopoulos[⋆†]; Kjell Lemström[⋄]; Mohammed Niyad[⋆]; Yoan J. Pinzón[‡]

[⋆] Department of Computer Science, King's College, London.
[†] School of Computing, Curtin University of Technology, WA.
[⋄] Department of Computer Science, University of Helsinki, Finland.
[‡] Facultad de Ingeniería de Sistemas, Universidad Autonoma de Bucaramanga, Colombia
{csi,niyadm}@dcs.kcl.ac.uk;klemstro@cs.helsinki.fi;ypinzon@bumanga.unab.edu.co

Abstract

In this paper we consider the problem of *motif evolution in polyphonic musical sequences*. A related problem, where a set of sequences of notes (one sequence for a voice) and a pattern is given, is to find whether approximate occurrences of the pattern occur distributed across the sequences (Holub et al., 1999; Lemström and Tarhio, 2000). Formally, this related problem is as follows: given a set t of h strings (each representing a *voice*) $t^i = t^i_1, \ldots, t^i_n,\ \ i \in \{1..h\}$, for some constant h and a pattern $p = p_1, \ldots, p_m$, we say that p occurs at position j of t if $p_1 = t^{i_1}_j, p_2 = t^{i_2}_{j+1}, \ldots, p_m = t^{i_m}_{j+m-1}$ for some $\{i_1, \ldots, i_m\} \in \{1..h\}$. Our problem of finding evolutionary chains is defined as follows: given a set t of h strings t^i (the *target*), for some constant h and a *motif* p, find whether there exists a sequence $u_1 = p, u_2, \ldots, u_\ell$ occurring in the target t such that u_{j+1} occurs to the right of u_j in t and for any given $j \in \{1..\ell-1\}$, u_j and u_{j+1} are similar enough, i.e., they do not differ more than by a certain number of basic operations — insertions, deletions and substitutions. In this paper, we consider several variants of the evolutionary chain problem and present efficient algorithms solving them.

1 Introduction

This paper is focused on a set of string pattern-matching problems which arise in music analysis (Mongeau and Sankoff, 1990; Stech, 1981), musical information retrieval (Lemström, 2000) and molecular sequence analysis (Gusfield, 1997). A musical score can be viewed as a string: at a very rudimentary level the alphabet could simply comprise the pitch names of the western music notation, or at a more complex level, we could use the GPIR representation of Cambouropoulos (1996a,b) as the basis of an alphabet. It is generally agreed in musicology and music psychology, that one of the most important phases in understanding a musical work is to identify the significant repetitions. The capability of identifying such repetitions would have a direct impact on automated music analysis and music information retrieval. The definition of musical repetition, however, is very vague. In its most restricted form, repetitions may be defined as excerpts being exactly similar (by considering intervals instead of absolute pitch values, repetitions that are merely transposed fall into this category of repetitions, as well).

Exact repetitions have been studied extensively. Such repetitions may either appear as distinct substrings (Apostolico, 1983; Iliopoulos et al., 1996b; Landau and Schmidt, 1993; Main and Lorentz, 1984; Myers and Kannan, 1993) or they may overlap (Berkman et al., 1996; Iliopoulos and Mouchard, 1999; Iliopoulos et al., 1996a; Moore and Smyth, 1994). A natural extension of the repetition problem is to allow the presence of errors. Although this makes the task more challenging, it is usually much more pertinent approach, which is the case, especially, when dealing with music. In approximate pattern matching, error tolerance is achieved by introducing a similarity measure (such as *edit* or *Hamming distance*, see e.g. (Crochemore and Rytter, 1994)) and a threshold variable k indicating the allowed tolerance to errors. A repeated substring may be subject to other constraints (e.g., it may be required to be of at least a certain length) and some invariances, as well.

Efficient algorithms for computing the approximate repetitions are not only relevant to our music task at hand. Instead, they are directly applicable also, for instance, to molecular biology (Fischetti et al., 1992; Karlin et al., 1988; Milosavljevic and Jurka, 1993) and in particular in DNA sequencing by hybridization (Pevzner and Feldman, 1993), reconstruction of DNA sequences from known DNA fragments (Schmidt, 1994; Skiena and Sundaram, 1995), in human organ and bone marrow transplantation as well as the determination of evolutionary trees among distinct species (Schmidt, 1994).

In this paper, we study a certain modification of the approximate repetition problem, namely the evolution of

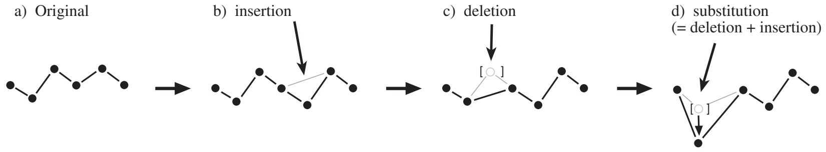

Figure 1: *An example of an evolving motif and of local editions required to trace the gradual changes.*

a monophonic[1] musical motif in a target of polyphonic music[2]. A simplistic case of the problem, when the target is also monophonic, is defined as follows: Given a string t representing the target (in combinatorial pattern matching often called the *text*) and a motif p, find whether there exists a sequence $u_1 = p, u_2, \ldots, u_\ell$ occurring in the target t such that u_{i+1} occurs to the right of u_i in t and for any given $i \in \{1..\ell - 1\}$, u_i and u_{i+1} are similar enough, i.e., they differ by at most k *editing operations*. The editing operations considered in this paper are insertions, deletions and substitutions - see Fig. 1.

Crochemore et al. (1998) presented an algorithm for computing non-overlapping evolutionary chains in monophonic targets. Their algorithm run in time $O(nm)$, where n and m denote the length of the target and the motif, respectively. They also presented an $O(n(\log m + \log |\Sigma|))$ theoretical version algorithm for the same problem that makes use of suffix trees and another version that requires $O(kn)$ time for fixed alphabets. Here Σ and k denote the size of the underlying alphabet and the approximation threshold, respectively. Furthermore, they considered also several variants of overlapping evolutionary chains for which they presented $O(n^2)$ algorithms.

Such algorithms are of great use in music analyses, for musical motifs may actually evolve this way. One actual case is shown by the successive thematic entries present in Messiaen's piano work, *Vingt Regards sur L'Enfant Jesus*. Other 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 idee 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 repetition can be seen as a transformation of the original motif. However, in these cases a repetition of generation r is often more similar to the repetition of generation $r - 1$ than to the original motif; a measure of this similarity has to be preset in an algorithm intended to detect all such repetitions of the motif.

In practice, the music under consideration is usually polyphonic (as in the examples above) in which case the algorithms by Crochemore et al. (1998) cannot be applied. However, when dealing with polyphony, problems become more challenging. For instance, for traditional pattern matching problem in music numerous algorithms for the monophonic case have been suggested (see e.g. (Ghias et al., 1995; Lemström and Laine, 1998; McNab et al., 1997; Pollastri, 1999; Rolland et al., 1999; Shmulevich et al., 2001)) but only a few for the polyphonic case (Holub et al., 1999; Lemström and Tarhio, 2000; Meredith et al., 2001). The problem of *distributed pattern matching in polyphonic musical sequences* is as follows: given a set of sequences of notes (one sequence for each voice) and a pattern, find whether occurrences of the pattern occur distributed across the sequences. More formally, given a set t of strings $t^i = t_1^i ... t_n^i, \; i \in \{1..h\}$, for some constant h and a pattern $p = p_1 ... p_m$, we say that p occurs at position j of t if $p_1 = t_j^{i_1}, p_2 = t_{j+1}^{i_2}, ..., p_m = t_{j+m-1}^{i_m}$ for some $\{i_1 ... i_m\} \in \{1..h\}$. Although the techniques solving this problem (Holub et al., 1999; Lemström and Tarhio, 2000) may also be adapted to approximate matching, they are not applicable to the problem considered here.

Our problem of *motif evolution in polyphonic music* is as follows: given a set t of strings $t^i = t_1^i ... t_n^i, \; i \in \{1..h\}$, for some constant h [3] and a motif p, find whether there exists a sequence $u_1 = p, u_2, \ldots, u_\ell$ occurring in the target t such that u_{j+1} occurs to the right of u_j in t and for any given $j \in \{1..\ell - 1\}$ u_j and u_{j+1} are k-similar (i.e. they differ by at most k insertions, deletions and substitutions). Moreover, every u_j occurs within one voice of t (any pair (u_j, u_k), however, may occur in distinct voices).

This paper is organized as follows. The next Section presents basic definitions for strings and background notions for string pattern-matching. Section 3 and 4 show how *evolution trees* representing non-overlapping and overlapping chains, respectively, can be computed. In Section 5 we show how the solutions to the three variants of our problem, that are *the longest evolutionary chain, the nearest neighbour evolutionary chain, and the minimal weight chain*, can be induced out of the evolution trees. Finally, Section 6 presents conclusions and some open problems.

2 String Combinatorics

A *string* is a sequence of zero or more symbols from an alphabet Σ; the "no symbol", that is, the string with zero symbols, is denoted by ϵ. The set of all strings over the

[1] In monophonic music, there is only one note played at a time.

[2] In polyphonic music, at times, several notes are played simultaneously.

[3] Note that $h = 1$ represents the degenerated, monophonic case.

	1	2	3	4	5	6	7	8
ρ	B	A	D	F	E	ϵ	C	A
τ	B	C	D	ϵ	E	F	C	A
	Mat	Mis	Mat	Del	Mat	Ins	Mat	Mat

Figure 2: *Types of differences: Mismatch, Insertion, Deletion.*

	1	2	3	4	5	6	7
τ	B	A	D	F	E	C	A
ρ	B	C	D	E	F	C	A
	Mat	Mis	Mat	Mis	Mis	Mat	Mat

Figure 3: *An alternative solution.*

alphabet Σ is denoted by Σ^*. A string x of length n ($|x| = n$, for short) is represented by $x_1 \dots x_n$, 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 $|u| + 1$ is said to be the *starting position* of w in x and the position $|u| + |w|$ the *ending position* of w 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 $u \in \Sigma^*$.

Consider two sequences $\tau = \tau_1\tau_2 \dots \tau_r$ and $\rho = \rho_1\rho_2 \dots \rho_r$ with $\tau_i, \rho_i \in \Sigma, i \in \{1 \dots r\}$. If $\tau_i = \rho_i$, then we say that τ_i and ρ_i *match*, otherwise τ_i *differs* from ρ_i. We distinguish among the following three types of differences:

1. Neither of these two symbols correspond to "no symbol" and the symbol of the first sequence corresponds to a different symbol of the second one, that is $\tau_i \neq \rho_i$, $\tau_i \neq \epsilon$ and $\rho_i \neq \epsilon$. This type of difference is a *mismatch* (or a substitution).

2. The symbol of the first sequence corresponds to "no symbol" of the second sequence, i.e., $\rho_i \neq \epsilon$ and $\tau_i = \epsilon$. This type of difference is called a *deletion*.

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

To give an example, let $\rho = BADFECA$ and $\tau = BCDEFCA$ (see Fig. 2). If we consider the alignment above, matches occur at positions 1, 3, 5, 7 and 8, while there is a mismatch at position 2, a deletion at position 4 and an insertion at position 6. Another way of seeing this difference is that one can transform the sequence ρ to τ by using the basic operations: insertions, deletions and substitutions. In this example, it means that we need three basic operations to transform $BADFECA$ into $BCDEFCA$: one mismatch (position 2 (A, C)), one deletion (position 4 (F, ϵ)) and one insertion (position 6 (ϵ, F)).

Note that we can also use three substitutions (position 2 (A, C), position 4 (F, E) and position 5 (E, F), see Fig. 3) without using the insertion and deletion. Therefore an "optimal alignment" is not necessarily unique. Nevertheless, we can always compute the minimal number of operations to transform one string into the other. If ρ is obtainable from τ (or vice versa) by using k editing operations, we say that ρ and τ are k similar and write $\rho =_k \tau$.

3 Computing Evolution Tree for Non-Overlapping Chains

Let p be a motif to be searched for in a polyphonic target t of h parallel voices. We aim at finding whether there exists a chain of monophonic strings $\mathcal{U} = \{u_1 = p, u_2, \dots, u_\ell\}$ occurring in the target.

Consider an occurrence of u_i in t. Function $ind(u_{i_j})$ shows the index (subscript) of the matching position of u_{i_j} in t. For example, let $h = 1$, $t =$ `combinatorics`, and $u_2 =$ `tori`. Then $ind(u_{2_3}) = 10$. A chain of monophonic strings $\mathcal{U} = \{u_1, \dots, u_\ell\}$ is called *Non-Overlapping Evolutionary Chain (NOEC)* if and only if it satisfies the conditions:

1. $u_1 = p$;

2. each u_i occurs in some t^{k_i}, $i \in \{1 \dots \ell\}, k \in \{1 \dots h\}$;

3. $ind(u_{i+1_1}) > ind(u_{i_{|u_i|}})$, for $1 \leq i < l$; and

4. $u_i =_k u_{i+1}$, for $1 \leq i < l$.

Our idea is to compute an evolution tree representing all non-overlapping evolutionary chains of the motif. In Section 5, we show how solution to different variants of the problem can be induced out of the tree. The main idea of the algorithm it to maintain the scores of differences between every prefix of the motif and the target. We keep on calculating the scores until we found an occurrence of p in a voice of t with at most k-differences. Once an occurrence of the motif has been found at position j on the target, the match would be extracted out of the target and this match will be used in a new query that starts at position $j + 1$.

The algorithm presented below uses the notion of recursion to compute NOECs.

1. *Initialization:*

2. *Main step:*
   At step j we have computed the number of differences between $t^i_1 \dots t^i_j, 1 \leq j \leq n,\ i \in \{1..h\}$ and p.

3. *Check for match*
   If the number of differences between p and $t^i_1 \dots t^i_j$ is at most k for some pair (i, j), then let u be the suffix of that prefix having k differences with p. We then add u to the chain and recursively find

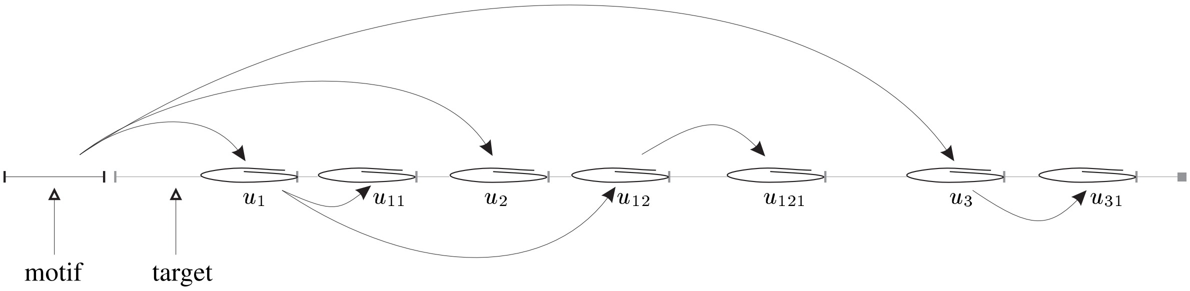

Figure 4: *The original motif and how it evolves.*

an occurrence of u with at most k-differences in $t_{j+1} \ldots t_n$.

4. *Output chain*
   If u is a leaf of the tree (see Fig. 5) then output the current chain and continue the previous recursive call.

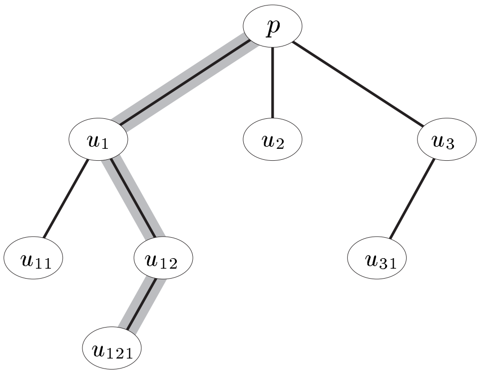

Figure 5: *The tree representation. The longest chain is emphasized by shading.*

Fig. 4 shows how the algorithm above works. The first match with the original motif is u_1 and this becomes the next motif. The position of the match becomes the starting position for the next recursive call. On continuing the search, u_1 is matched with u_{11} and u_{12}. However, any occurrences for u_{11} is not found in t, hence $\{p \rightsquigarrow u_1 \rightsquigarrow u_{11}\}$ is a chain, so is $\{p \rightsquigarrow u_1 \rightsquigarrow u_{12} \rightsquigarrow u_{121}\}$. Once the search for u_1 further in the target has been exhausted, the next match with the original motif is taken, i.e., u_2. The same procedure is repeated and the following chains are found: $\{p \rightsquigarrow u_2\}, \{p \rightsquigarrow u_3 \rightsquigarrow u_{31}\}$.

The same figure can be rearranged as a tree (see Fig. 5) which gives a more accurate representation of how the Evolution Tree is produced. All our discussions will be based on this tree representation and a leaf of this tree always completes a chain.

3.1 Pseudo-Code

Let us consider the straightforward NOEC's main routine (see Fig. 6).

NOEC(t, p, k)
1 $EvoChain \leftarrow \{\emptyset\}$
2 ETREE$(1, p, EvoChain)$

Figure 6: *Algorithm for computing non-overlapping evolutionary chains.*

The input for NOEC are target (t), motif (p) and error tolerance (k). To produce the evolution tree, NOEC calls ETREE (the first parameter gives the starting position for the search).

ETREE$(start, p, EvoChain)$
1 **if** $start > n - \lfloor m/2 \rfloor$
2 **then** PRINT$(EvoChain)$
3 **return**
4 *IsLeaf* $\leftarrow$ TRUE
5 **for** $j \leftarrow start$ **to** n
6 **do if** $D(p, t_{start..j}) \leq k$
7 **then** *IsLeaf* $\leftarrow$ FALSE
8 $p' \leftarrow$ BACKTRACK$(p, t_{start..j})$
9 ▷ the recursive call
10 ETREE$(j+1, p', EvoChain \cup \{start\})$
11 **if** (*IsLeaf*)
12 **then** PRINT$(EvoChain)$

Figure 7: *Recursive function that computes the evolution tree.*

Consider the pseudo-code for the function ETREE that is given in Fig. 7. At line 9, the all important recursive call with the new motif p' can be found. At line 8, BACKTRACK is the process of retrieving the match out of the target. Lines 1,2,3 outputs the chain since it will not be

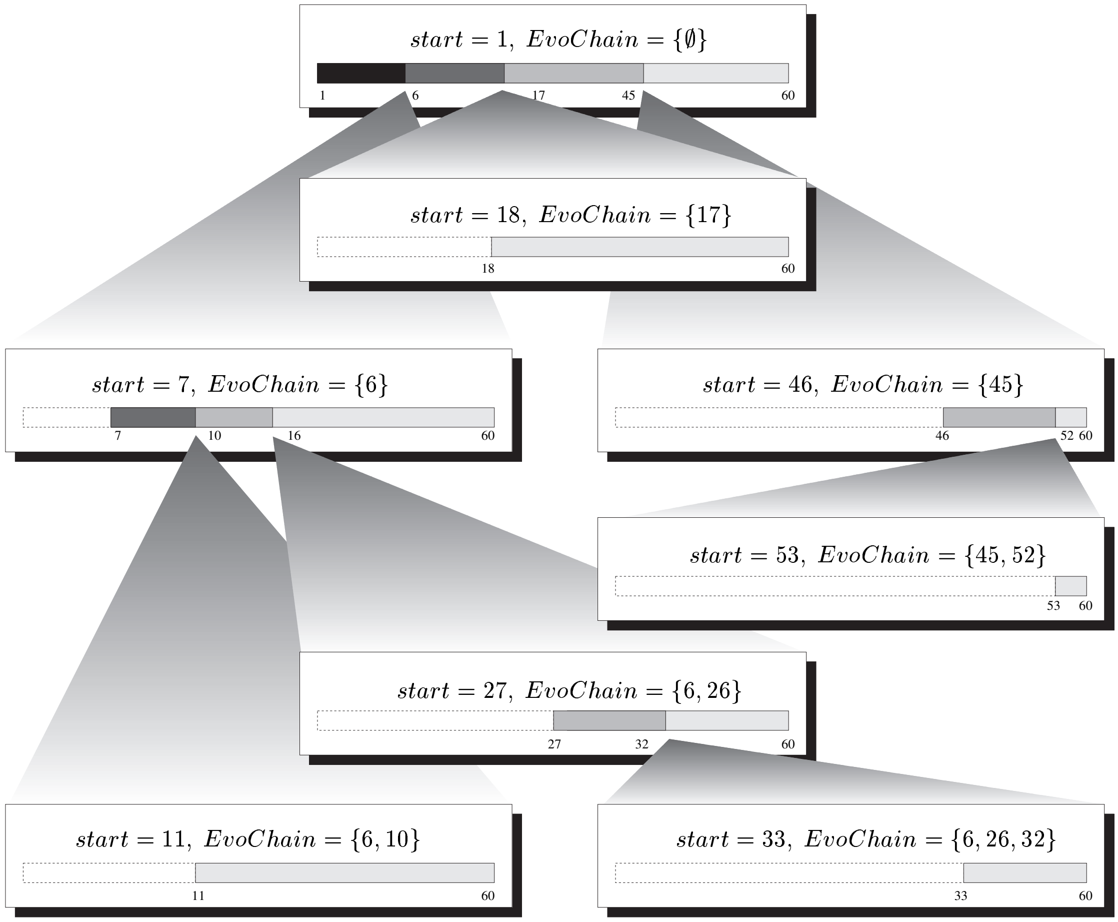

Figure 8: *Example showing how the recursive calls from the tree.*

possible to find a match after position $n - \lfloor m/2 \rfloor$. Thus, we have reached a leaf, i.e., end of a chain. Similarly line 11 outputs the chain if a match is not found in the current iteration, again this is a leaf of the evolution tree.

Referring to the tree representation shown in Fig. 5, assume $u_1 = 6,\ u_2 = 17,\ u_3 = 45,\ u_{11} = 10,\ u_{12} = 26,\ u_{31} = 52$ and $u_{121} = 32$ with $n = 60$. Fig. 8 shows how the function ETREE recursively computes the chains.

Table 1 is the tracing of the algorithm NOEC. The first entry in the table is the initial call to ETREE from algorithm NOEC. Entry 1 indicates that the first recursive call after a match (with at most k-differences) was found ending at position 6. Therefore, 6 was added to the $EvoChain$ and the new task is to find "$ABCD$" starting from position 7. This makes sure the matches do not overlap. Entry 2 denotes that a k-similar match was found ending at position 10, which is then added to the chain. Now the new recursive query is to find "$ABDD$" starting from position 11. This query, however, does not yield any results. Therefore, we must have reached a leaf on the evolution tree (see Fig. 5), and the evolutionary chain $EvoChain$ is output. Having finished with this instance of ETREE, we go back to entry 1 and continue from position 6 (which is where we left and spawned entry 2) with the old motif "$AACD$" until another k-similar match is found. Ending at position 26 is an approximate match which is added to $EvoChain$. Now ETREE is recursively called to look for "$ACCB$" starting from position 27 as depicted by entry 3 on the table. This process of recursively calling ETREE with a new query each time is repeated until the first call to ETREE (entry *init* in Table 1) is exhausted. Table 1 is the tracing of the recursive calls.

3.2 Running Time

The computation of a single non-overlapping evolutionary chain requires $O(mn)$ time and $O(mn)$ space. The computation of all non-overlapping evolutionary chains can require exponential time (in terms of nm) in pathological worst-case scenarios like this one:

$$p = A^m, t = A^n \text{ and } k = 0. \tag{1}$$

The worst case happens, when we have a match at every position. The best case happens when there is no match at all; In this case $O(mn)$ time and $O(m)$ space is needed. But for practical purposes the behaviour of this

call	$start$	$EvoChain$	p
init	1	$\{\emptyset\}$	"$AACD$"
1	7	$\{6\}$	$ABCD$
2	11	$\{6, 10\}$	$ABDD$
Output $EvoChain$ $\looparrowright$ $\{ABCD \rightsquigarrow ABDD\}$			
3	27	$\{6, 26\}$	$ACCB$
4	33	$\{6, 26, 32\}$	$BCCB$
Output $EvoChain$ $\looparrowright$ $\{ABCD \rightsquigarrow ACCB \rightsquigarrow BCCB\}$			
5	18	$\{17\}$	$BACD$
Output $EvoChain$ $\looparrowright$ $\{BACD\}$			
6	46	$\{45\}$	$AABD$
7	53	$\{45, 52\}$	$AABB$
Output $EvoChain$ $\looparrowright$ $\{AABD \rightsquigarrow AABB\}$			

Table 1: *Recursion parameters and order.*

algorithm for computing all non-overlapping evolutionary chains is quadratic $O((mn)^2)$, requiring $O(mn)$ space.

A practical speed-up for the algorithm is to mark the nodes that have already been used in a chain, and only unmarked nodes can be selected as nodes in any chain. In this way, even the exponential worst case time complexity of the pathologic case becomes polynomial: $O(n^4)$.

4 Computing Evolution Tree for Overlapping Chains

In this section we present the other variation of evolutionary chains, namely Overlapping Evolutionary Chains (OEC, for short). The problem is defined as follows: given a set t of strings $t^i = t^i_1 \dots t^i_n$, $i \in \{1..h\}$, a motif p and an integer $k < |p|/2$, find whether there exists a sequence $u_1 = p$, $u_2, \dots, u_\ell$ occurring in the target t such that the following conditions are satisfied (note the difference between the items 3 here and of that given in Section 3):

1. $u_1 = p$;
2. each u_i occurs in some t^{k_i}, $i \in \{1 \dots \ell\}, k \in \{1 \dots h\}$;
3. $ind(u_{i+1_1}) > \lceil ind(u_{i_1}) + \frac{|u_i|}{2})\rceil$, for $1 \leq i < l$;
4. $u_i =_k u_{i+1}$, for $1 \leq i < l$.

These strings have been constrained to overlap at most $|p|/2$ symbols. Without such a constraint, we can obtain trivial chains such as $ind(u_{i_1}) = s$, $ind(u_{i+1_1}) = s + 1$, and obviously u_i and u_{i+1} have at most one difference.

Let us now introduce the pseudo-code for the OEC algorithm (see Fig. 9). The function ETREE$'$ (Fig. 10) is a slightly modified version of the one that was used with NOEC algorithm.

```
OEC(t, p, k)
EvoChain ← {∅}
ETREE'(1, p, EvoChain)
```

Figure 9: *Algorithm for computing overlapping evolutionary chains.*

Note that the two versions, ETREE and ETREE$'$, are very similar: the only difference is the addition of the three lines at the beginning. Line 1 makes sure the overlapping cases are taken into account but with the constraint $(m/2)$ as explained earlier. The **if** statement in Line 2 is for the case when the initial call from NOEC is met with $start = 1$ and having executed Line 1, $start$ will be negative (thus, $start$ is fixed to be positive).

Obviously, the time complexity of this algorithm is of the same order as that of NOEC. Since we allow for overlapping, $m/2$ symbols are included again in each recursive call. More precisely, the practical time complexity for OEC is $O((mn)^2)$ and space complexity $O(mn)$.

5 Inducing Solutions out of the Evolution Trees

Let us now define three specific problems of evolutionary chain computing and show how solutions for these problems can be induced out of the evolution trees.

Longest Evolutionary Chain (LEC). LEC is the simplest form of these problems; it is the chain $\mathcal{U} = u_1 \dots u_\ell$ that maximizes ℓ. Once the evolution tree has been computed, the length of the LEC is the height of the evolution tree. Therefore every time a new chain is being output, ETREE'$(start, p, EvoChain)$

$start \leftarrow start - m/2$
if $start < 1$
  then $start \leftarrow 1$
if $start > n - \lfloor m/2 \rfloor$
  then PRINT$(EvoChain)$
    return
IsLeaf $\leftarrow$ TRUE
for $j \leftarrow start$ **to** n
  do if $D(p, t_{start..j}) \leq k$
    then *IsLeaf* $\leftarrow$ FALSE
      $p' \leftarrow$ BACKTRACK$(p, t_{start..j})$
      ▹ the recursive call
      ETREE'$(j+1, p', EvoChain \cup \{start\})$
if (*IsLeaf*)
  then PRINT$(EvoChain)$

Figure 10: *Recursive function to compute the evolution tree for OEC.*

the length of that chain is compared with the height of the tree, and if it longer then that must be the longest chain so far. This procedure is repeated for all the chains and in the end we output the length of the longest chain together with the indices of the chain elements u_i found in the target. In the example given in Fig. 5 LEC is emphasized using shading.

Nearest neighbour Evolutionary Chain (NEC). NEC is the chain $\mathcal{U} = \{u_1, \ldots, u_\ell\}$ that minimizes e in the following equation:

$$e = \sum_{i=1}^{l-1} f(ind(u_{i+1_1}) - ind(u_{i_{|u_i|}})), \qquad (2)$$

where f is some increasing function on positive integers. The simplest function of that form (that is also considered here) is the identity function; $f(x) = x$.

When solving NEC, we associate values with the edges of the tree. These values are the gaps between the two joining nodes of the tree. In the case of overlapping matches, the gap is taken to be zero. Once this is completed, the total gap of each chain is compared to find the minimum chain and the total gap together with the chain index is output.

Minimal weight Evolutionary Chain (MEC). MEC minimizes d in:

$$d = \sum_{i=1}^{l-1} D(u_i, u_{i+1}), \qquad (3)$$

where $D(u_i, u_{i+1})$ is the Edit Distance between u_i and u_{i+1}.

In this case, we attach values to the edges, as well. This time, the values are the the number of differences, d', between the two joining nodes of the tree. Once this is completed, the total difference d of each chain is summed up and compared to find the chain with minimum d. The output is the chain giving the minimum to d and the value d, itself.

6 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 formalize their relation to well known string pattern-matching problems. 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 (Cambouropoulos, 1997; Cambouropoulos and Smaill, 1995; Crawford et al., 1998; Lemström, 2000) 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.

We presented practical algorithms, NOEC and OEC, for computing non-overlapping and overlapping evolutionary chains. Furthermore, we presented three variants of these problem, the longest evolutionary chain, the nearest neighbour evolutionary chain, and the minimal weight evolutionary chain, each of which are of practical importance.

The problems presented here need to be further investigated under a variety of *similarity* or *distance* rules (see (Crawford et al., 1998; Mongeau and Sankoff, 1990; Lemström, 2000)). 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). 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.

Further investigation whether methods such as (Galil and Park, 1990; Landau and Vishkin, 1988) can be adapted to solve the problems considered here is needed. Further-

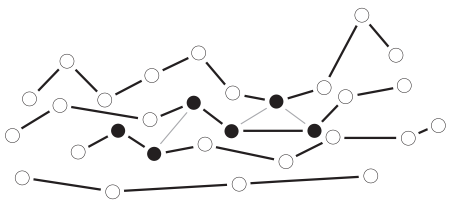

Figure 11: *Motif distributed across voices.*

more, modifications to our algorithms in order to find evolutionary occurrences that are distributed across voices (see Fig. 11) are left to future studies; the algorithms that have been presented to find distributed occurrences of a monophonic motif in a polyphonic target (Holub et al., 1999; Lemström and Tarhio, 2000) are not applicable to the problem at hand. This is because these bit-parallel algorithms are only able to locate the positions of occurrences, but they cannot, however, extract the matching substring out of the target. Therefore, they cannot be applied in a recursive manner as the algorithms presented here.

Acknowledgements

Costas Iliopoulos was partially supported by a Marie Curie Fellowship and Royal Society, Wellcome Foundation and NATO grants. Kjell Lemström was partially supported by the grants #48313 from the Academy of Finland and research grant GR/R25316 from EPSRC. Jose Pinzón was partially supported by an ORS studentship and EPSRC Project GR/L92150.

References

A. Apostolico. The myriad virtues of the suffix trees. *Theoretical Computer Science*, 22:297–315, 1983.

O. Berkman, C. Iliopoulos, and K. Park. String covering. *Information and Computation*, 123:127–137, 1996.

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, 1996a.

E. Cambouropoulos. A general pitch interval representation: Theory and applications. *Journal of New Music Research*, 25:231–251, 1996b.

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.

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.

T. Crawford, C.S. Iliopoulos, and R. Raman. String matching techniques for musical similarity and melodic recognition. *Computing in Musicology*, 11: 71–100, 1998.

M. Crochemore, C.S. Iliopoulos, and H. Yu. Algorithms for computing evolutionary chains in molecular and musical sequences. In *Proceedings of the 9–th Australasian Workshop on Combinatorial Algorithms*, volume 6, pages 172–185, 1998.

M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

V. Fischetti, G. Landau, J.Schmidt, and P. Sellers. Identifying periodic occurences of a template with applications to protein structure. In *Proc. 3rd CPM*, volume 644, pages 111–120. Lecture Notes in Computer Science, 1992.

Z. Galil and K. Park. An improved algorithm for approximate string matching. *SIAM Journal on Computing*, 19:989–999, 1990.

A. Ghias, J. Logan, D. Chamberlin, and B.C. Smith. Query by humming - musical information retrieval in an audio database. In *ACM Multimedia 95 Proceedings*, pages 231–236, San Francisco, CA, 1995.

D. Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology*. Cambridge University Press, Cambridge, 1997.

J. Holub, C.S. Iliopoulos, B. Melichar, and L. Mouchard. Distributed string matching using finite automata. In *Proceedings of the 10th Australasian Workshop On Combinatorial Algorithms*, pages 114–128, Perth, 1999.

C.S. Iliopoulos, D.W.G. Moore, and K. Park. Covering a string. *Algorithmica*, 16:288–297, 1996a.

C.S. Iliopoulos, D.W.G. Moore, and W.F. Smyth. A linear algorithm for computing the squares of a fibonacci string. In *Proceedings CATS'96, Computing: Australasian Theory Symposium*, pages 55–63, 1996b.

C.S. Iliopoulos and L. Mouchard. An $o(n \log n)$ algorithm for computing all maximal quasiperiodicities in strings. In *Proceedings of CATS'99: Computing: Australasian Theory Symposium*, volume 21, pages 262–272, Auckland, New Zealand, 1999. Lecture Notes in Computer Science.

S. Karlin, M. Morris, G. Ghandour, and M.Y. Leung. Efficients algorithms for molecular sequences analysis. In *Proc. Natl. Acad. Sci.*, volume 85, pages 841–845, 1988.

G.M. Landau and J.P. Schmidt. An algorithm for approximate tandem repeats. In *Proc. Fourth Symposium on Combinatorial Pattern Matching*, volume 648, pages 120–133. Lecture Notes in Computer Science, 1993.

G.M. Landau and U. Vishkin. Fast string matching with k differences. *Journal of Computer and Systems Sciences*, 37:63–78, 1988.

K. Lemström. *String Matching Techniques for Music Retrieval*. PhD thesis, University of Helsinki, Department of Computer Science, 2000. Report A-2000-4.

K. Lemström and P. Laine. Musical information retrieval using musical parameters. In *Proceedings of the 1998 International Computer Music Conference*, pages 341–348, Ann Arbor, MI, 1998.

K. Lemström and J. Tarhio. Detecting monophonic patterns within polyphonic sources. In *Content-Based Multimedia Information Access Conference Proceedings (RIAO'2000)*, volume 2, pages 1261–1279, Paris, 2000.

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

R.J. McNab, L.A. Smith, D. Bainbridge, and I.H. Witten. The New Zealand digital library MELody inDEX. *D-Lib Magazine*, 1997.

D. Meredith, G.A. Wiggins, and K. Lemström. Pattern induction and matching in polyphonic music and other multi-dimensional datasets. In *the 5th World Multi-Conference on Systemics, Cybernetics and Informatics (SCI'2001)*, volume X, pages 61–66, Orlando, FLO, July 2001.

A. Milosavljevic and J. Jurka. Discovering simple dna sequences by the algorithmic significance method. *Comput. Appl. Biosci.*, 9:407–411, 1993.

M. Mongeau and D. Sankoff. Comparison of musical sequences. *Computers and the Humanities*, 24:161–175, 1990.

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*, pages 511–515, 1994.

E. Myers and S. Kannan. An algorithm for locating non-overlapping regions of maximum alignment score. In *Proc. Fourth Symposium on Combinatorial Pattern Matching*, volume 648. Lecture Notes in Computer Science, 1993.

P.A. Pevzner and W. Feldman. Gray code masks for dna sequencing by hybridization. *Genomics*, 23:233–235, 1993.

E. Pollastri. Melody-retrieval based on pitch-tracking and string-matching methods. In *Proceedings of the XIIth Colloquium on Musical Informatics*, 1999.

P.Y. Rolland, G. Raskinis, and J.G. Ganascia. Musical content-based retrieval: an overview of the melodiscov approach and system. In *ACM Multimedia 99 Proceedings*, Orlando, FLO, 1999.

J.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.

I. Shmulevich, O. Yli-Harja, E. Coyle, D.J. Povel, and K. Lemström. Perceptual issues in music pattern recognition - complexity of rhythm and key finding. *Computers and the Humanities*, 35(1):23–35, 2001.

S.S. Skiena and G. Sundaram. Reconstructing strings from substrings. *J. Computational Biol.*, 2:333–353, 1995.

D.A. Stech. A computer-assisted approach to microanalysis of melodic lines. *Computers and the Humanities*, 15:211–221, 1981.