J Intell Inf Syst (2011) 36:99–115
DOI 10.1007/s10844-010-0122-7

Mining transposed motifs in music

Aída Jiménez · Miguel Molina-Solana · Fernando Berzal · Waldo Fajardo

Received: 27 November 2009 / Revised: 23 February 2010 / Accepted: 21 March 2010 /
Published online: 13 April 2010

Abstract The discovery of frequent musical patterns (motifs) is a relevant problem in musicology. This paper introduces an unsupervised algorithm to address this problem in symbolically-represented musical melodies. Our algorithm is able to identify transposed patterns including exact matchings, i.e., null transpositions. We have tested our algorithm on a corpus of songs and the results suggest that our approach is promising, specially when dealing with songs that include non-exact repetitions.

Keywords Musical mining · Motifs · Frequent pattern mining

1 Introduction

The discovery of frequent musical patterns (motifs) is a relevant problem in musicology. In music, we can find several entities that can be repeated such as notes, intervals, rhythms, and harmonic progressions. In other words, music can be seen as a string of musical entities such as notes or chords on which pattern recognition techniques can be applied.

We can define a music motif as *the smallest meaningful melody element*. As a rule, motifs are groups of notes no longer than one measure. In human speech, a motif is a word. In the same way that sentences consist of words, motifs form musical phrases. A melody is formed by several main motifs, which are repeated, developed, and opposed one against another within the melody evolution.

A. Jiménez (✉) · M. Molina-Solana · F. Berzal · W. Fajardo
Centro de Investigación en Tecnologías de la Información y las Comunicaciones,
University of Granada, Granada, Spain
e-mail: aidajm@ugr.es

M. Molina-Solana
e-mail: miguelmolina@ugr.es

F. Berzal
e-mail: berzal@acm.org

W. Fajardo
e-mail: aragorn@ugr.es

When analyzing a music work, musicians carry out a deep analysis of the musical material. This analysis includes motif extraction as a basic task. Musician studies include contextual information (such as the author, the aim, or the period) but also morphological data from the music itself. Looking for the motifs that build the whole work is the first step that a musician takes when faced with a music sheet.

Audio-thumbnailing (i.e., summarizing or abstracting) is another interesting application in the musical domain that is related to motif extraction. It provides the user with a brief excerpt of a song that (ideally) contains the main features of the work. Before hearing or purchasing a whole song, it would be useful to hear a representative thumbnail of the whole work. This technique is also important for indexing large datasets of songs, which can be browsed more quickly and searched more efficiently if indexed by those small patterns instead of being indexed by the whole song.

One of the most fundamental ways to classify MIR methods is to divide them into those that process audio signals using signal processing methods and those that process symbolic representations. We have decided to work with a symbolic representation instead of an audio one because it is closer to the original sheet of music. In other words, the main difficulty with audio representation is that the transformation from audio signals to symbolic data is far from being accurate. This fact makes the pattern recognition problem much more difficult and it requires completely new techniques to deal with signals.

Using the algorithm we present in this paper, we are able to find frequent melodic and rhythmical patterns in music starting from the *MusicXML* representation of the song (www.wikifonia.org). We first transform this symbolic representation into a sequence of notes. These notes are defined at their lowest level (i.e., pitch and duration) and in an absolute, not relative, way.

According to the above considerations, we have developed a TreeMiner-based (Zaki 2005b) algorithm to discover frequent subsequences in music files. Our algorithm is able to identify sequences even when they are transposed. It can be used to find common motifs in several songs and also find repetitions within a song. In this paper, we present its application to the discovery of long motifs that are repeated within a single song. Our hypothesis is that those patterns probably correspond to the chorus or the more significant part of the song.

Our paper is an extended version of a paper presented at the ISMIS'09 conference (Berzal et al. 2009) and is organized as follows. In Section 2, we provide some background on musical data mining and introduce some relevant terms. Section 3 formally defines our sequence pattern mining problem and describes the algorithm we have devised to solve it. In Section 4, we explain the way our algorithm works by means of a particular example. Some experimental results are presented in Section 5, whereas in Section 6 we draw some conclusions.

2 Background

Although it is almost impossible to be exhaustive in analyzing the state of the art in musical pattern identification, we survey the most relevant works in this field in Section 2.1. As our approach is based on sequences, we introduce some standard terms and review some sequence mining algorithms proposed in the literature in Section 2.2.

2.1 Data mining in music

Pattern processing techniques have been applied to musical strings. A complete overview can be found, for instance, in the paper by Cambouropoulos et al. (2001). Those algorithms can be divided into those that deal with audio signals (using signal processing methods) and those that use symbolic representations.

2.1.1 Dealing with audio signals

There are several researchers that have addressed the problem of pattern induction in an acoustic signal. For instance, Aucouturier and Sandler (2002) proposed an algorithm to find repeated patterns in an acoustic signal by focusing on timbre; whereas Chu and Logan (2002) proposed a method to find the most representative pattern in a song using Mel-spectral features.

Recently, some works have gone further in this direction by trying to identify the sectional form of a musical piece from an acoustic signal. For example, Paulus and Klapuri (2009) address this task using a probabilistic fitness measure based on three acoustic features; whereas Levy and Sandler (2008) use clustering methods to extract this sectional structure.

Solving the problem of identifying the structure of a musical piece is key for audio-thumbnailing (i.e., finding a short and representative sample of a song). Zhang and Samadani (2007) addressed this problem by detecting paragraphs in the song with repeated melody in a first step and then identifying vocal portions in the song. With such information, the structure of the song is derived. Another approach, by Bartsch and Wakefield (2005), developed a chroma-based system that searches for structural redundancy within a given song with the aim of identifying something like a chorus or refrain.

2.1.2 Using symbolic representations

There are certain similarities in the use of text and musical data which also allow the application of text mining methods to process musical data. Both have a hierarchical structure and the relative order among the elements is of importance. For that reason, researchers have proposed many different meaningful ways of representing a piece of music as a string, but all of them use either event strings (where each symbol represents an event) or interval strings (where each symbol represents the transformation between events).

Most of the proposed techniques start from a symbolic transcription of music. For example, Hsu et al. (1998) used a dynamic programming technique to find repeating factors in strings representing monophonic melodies; whereas Rolland (1998) recursively computed the distances between large patterns from the distances between smaller patterns. Meredith et al. (2002) proposed a geometric approach to repetition discovery in which the music is represented as a multidimensional dataset. Pienimäki (2002) introduced a text mining based indexing method for symbolic representation of musical data that extracts maximal frequent phrases from musical data and sorts them by their length, frequency and personality.

Finally, the paper by Grachten et al. (2004) is of particular relevance because it represents melodies at a higher level than notes but lower enough to capture the essence of the melody. This level is the ‘Narmour patterns’ level, based on Narmour’s I/R model (1992), which is well-known in musicology.

Our algorithm is also based in a symbolic representation of the song: MusicXML.

2.2 Sequence mining

In our approach for musical motif extraction, we first transform a song into a sequence of notes. There is a rich variety of sequence types, ranging from simple sequences of letters to complex sequences of relations.

A *sequence* over an element type τ is an ordered list $S = s_1 ... s_m$, where:

- each s_i (which can also be written as $S[i]$) is a member of τ, and is called an element of S;
- m is referred to as the length of S and is denoted by $|S|$;
- each number between 1 and $|S|$ is a position in S.

$T = t_1 .... t_n$ is called a *subsequence* of the sequence $S = s_1 ... s_m$ if there exist integers $1 < j_1 < j_2 < ... < j_n < m$ such that $t_1 = s_{j_1}$, $t_2 = s_{j_2}$, and in general, $t_n = s_{j_n}$.

Sequences have been used to solve different problems in the literature (Dong and Pei 2007; Han and Kamber 2005):

- *String matching problems*: Several sequence mining techniques have been used, for instance, in Bioinformatics to find some structures in a DNA sequence (Böckenhauer and Bongartz 2007):
  - Exact string matching: Given two strings, finding the occurrence of one as a substring of the other one.
  - Substring search: Finding all the sequences in a sequence database that contain a particular string as a subsequence.
  - Longest common substring: Finding the substring with maximum length that is common to all the sequences in a given set.
  - String repetition: Finding substrings that appear at least twice in a sequence.
- *Periodic pattern discovery*: A traditional periodic pattern consists of a tuple of k components, each of which is either a literal or '*', where k is the period of the pattern and '*' can be substituted for any literal and is used to enable the representation of partial periodicity (Wang et al. 2001; Yang et al. 2001).
- *Sequence motifs:* A motif is essentially a short distinctive sequential pattern shared by a number of related sequences. There are four main problems in this area (Dong and Pei 2007): motif representation (i.e., designing the proper motif representation for the different applications), motif finding (i.e., finding the motifs shared by several sequences), sequence scoring (i.e., computing the probability of a sequence to be generated by a motif—using Markov models, for example), and sequence explanation (i.e., given a sequence and a motif with hidden states, providing the most likely state path that produced that sequence).
- *Sequential pattern mining in transactional databases:* Sequential patterns have been used for predicting the behavior of individual customers. Each customer is typically modeled by a sequence of transactions containing the set of items he has bought. Several algorithms address this kind of problems, the most common being AprioriAll (Agrawal and Srikant 1994), SPADE (Zaki 2001) GSP (Srikant and Agrawal 1996), and PrefixSpan (Pei et al 2001).

- *Sequential pattern mining in sequence databases*: Some algorithms, such as the one developed by Jiang and Hamilton (2003), look for subsequences that have a larger frequency (or number of repetitions) than an user-defined threshold, which is established beforehand. Jiang's algorithm, for example, uses a tree-based structure called *trie*, that preserves the number of times a subsequence is present in the sequence. Three different versions of his algorithm can be devised:
  - A breadth-first search algorithm passes K times through the data sequence, counting the sequences of size i in its i-th iteration. At the end of each iteration, infrequent subsequences are pruned.
  - A depth-first search algorithm passes just one time through the data using a window of size K which is moved one position at a time. The sequence in the window is preserved in the *trie* structure as well as its prefix. The way this *trie* structure is completely built is very memory consuming without pruning.
  - A heuristic-first search algorithm is a variation of the depth-first algorithm. The number of occurrences of the prefixes of a subsequence is compared to the threshold before inserting the subsequence in the *trie*. If any prefix of the subsequence has not yet been shown to be frequent, then occurrences of the subsequence itself are not counted. This algorithm is more efficient in time and space but it is not able to find all the frequent subsequences.

Our problem of motif extraction in a piece of music could be seen as a particular case of the 'sequential pattern mining in sequence databases' problem. However, the algorithms proposed by Jiang et al. have the drawback that they are limited by the size of the alphabet (i.e., the element type τ according to our notation). This value can be very high in our particular domain, provided that we take different note pitches and durations into account.

Furthermore, we consider the presence of similar sequences (transposed motifs in our problem) that should be counted as if they were exact repetitions. In our case, it is also interesting to know where these repetitions appear in the sequence, specially when considering these similar repetitions. This information would not be given by Jiang's algorithms, as they only count the number of repetitions.

In the following section, we propose a novel TreeMiner-based algorithm to find motifs in a sequence in order to solve the problem of motif extraction in a piece of music. It should be noted, however, that our algorithm can be applied to several sequences at a time, as well as to different kinds of sequence databases (not just musical ones).

3 Our sequence pattern mining algorithm

The goal of frequent sequence pattern mining is the discovery of all the frequent subsequences in a large database of sequences D or in an unique large sequence.

Let $\delta_T(S)$ be the occurrence count of a subsequence S in a sequence T and d_T a variable such that $d_T(S) = 0$ if $\delta_T(S) = 0$ and $d_T(S) = 1$ if $\delta_T(S) > 0$. We define the *support* of a subsequence as $\sigma(S) = \sum_{T \in D} d_T(S)$, i.e., the number of sequences in D that include at least one occurrence of the subsequence S. Analogously, the *weighted support* of a subsequence is defined as $\sigma_w(S) = \sum_{T \in D} \delta_T(S)$, i.e., the total number of occurrences of S within all the sequences in D.

We also consider the occurrences of a pattern that approximately match (i.e., those occurrences that are very similar but are not exactly the same). We define the *exact support* of a subsequence as the number of occurrences that are exactly equal to the pattern, whereas the *transposed support* includes both exact and similar occurrences.

We say that a subsequence S is *frequent* if its support is greater than or equal to a predefined minimum support threshold. We define L_k as the set of all frequent k-subsequences (i.e., subsequences of size k).

3.1 SSMiner

Our algorithm, called SSMiner (Similar Sequence Miner), is based on the POTMiner (Jimenez et al. 2009) frequent tree pattern mining algorithm, a TreeMiner-like algorithm for discovering frequent patterns in trees (Zaki 2005b). POTMiner and its antecessor follow the Apriori (Agrawal and Srikant 1994) iterative pattern mining strategy, where each iteration is broken up into two distinct phases:

- *Candidate Generation*: A candidate is a potentially frequent subsequence. In Apriori-like algorithms, candidates are generated from the frequent patterns discovered in the previous iteration. Most Apriori-like algorithms, including ours, generate candidates of size $k+1$ by merging two patterns of size k having $k-1$ elements in common.
- *Support Counting*: Given the set of potentially frequent candidates, this phase consists of determining their actual support and keeping only those candidates whose support is above the predefined minimum support threshold (i.e., those candidates that are actually frequent).

The pseudo-code of our algorithm is shown in Fig. 1 and its implementation details will be discussed in Sections 3.2 through 3.4.

The sequence of a song is scanned twice by our algorithm, in the process of obtaining the frequent elements of size 1. The first scan is needed to save the occurrences of each note and the second one is employed to detect the transposed occurrences of each note. Then, the infrequent notes are pruned and we are ready to apply the two phases of the SSMiner algorithm without checking the original sequence any more.

Fig. 1 SSMiner: our sequence mining algorithm

```
algorithm
    Obtain frequent elements (frequent patterns of size 1)
    Build candidate classes C1 from the frequent elements
    for k=2 to MaxSize
        for each class P ∈ Ck−1
            for each element p ∈ P.
                Compute the frequency of p
                if p is frequent
                then
                    Create a new class P′ from p.
                    Add P′ to Ck
```

3.2 Candidate generation

We use an equivalence class-based extension method to generate candidates (Zaki 2005a). This method generates $(k+1)$-subsequence candidates by joining two frequent k-subsequences with $k-1$ elements in common.

Two k-subsequences are in the same equivalence class $[P]$ if they share the same prefix string until their $(k-1)$th element. Each element of the class can then be represented as x, where x is the k-th element label.

Elements in the same equivalence class are joined to generate new candidates. This join procedure, called extension in the literature, works as follows. Let (x) and (y) denote two elements in the same class $[P]$, and $[P_x]$ be the set of candidate sequences derived from the sequence that is obtained by adding the element (x) to P. The join procedure results in attaching the element (y) to the sequence generated by adding the element (x) to P, i.e, $(y) \in [P_x]$. Likewise, $(x) \in [P_y]$

3.3 Occurrence lists

Once we have generated the potentially frequent candidates, it is necessary to determine which ones are actually frequent.

The support counting phase in our algorithm follows the strategy of AprioriTID (Agrawal and Srikant 1994). Instead of checking the presence of each candidate in the sequence (which would entail $O(|S|)$ operations), special lists are used to preserve the occurrences of each pattern in the database, thus facilitating the support counting phase.

Each occurrence list contains tuples (t, m, p, d, Θ) where t is the sequence identifier, m stores the elements of the sequence which match those of the $(k-1)$ prefix of the pattern X, p is the position of the last element in the pattern X, d is a position-based parameter used for guaranteeing that elements in the pattern are contiguous within the sequence and Θ indicates the similarity between the occurrence and the original pattern.

When building the scope lists for patterns of size 1, m is empty and the element d is initialized with the position of the pattern only element in the original database sequence. In the first pass through the sequence, exact patterns of size 1 are collected, being its Θ parameter initialized as "=". When similar pattern occurrences are collected in the second pass through the sequence, the parameter Θ is initialized with a value that indicates the similarity between the original pattern and the actual occurrence.

We obtain the occurrence list for a new candidate of size k by joining the lists of the two subsequences of size $k-1$ that were involved in the generation of the candidate. Let $(t_x, m_x, p_x, d_x, \Theta_x)$ and $(t_y, m_y, p_y, d_y, \Theta_y)$ be two tuples to be joined. The join operation proceeds as follows:

if

1. $t_x = t_y = t$ **and**
2. $m_x = m_y = m$ **and**
3. $d_x = 1$ (only if $k \neq 2$) **and**
4. $p_x < p_y$ **and**
5. Θ_x=Θ_y

then add $[t, m \bigcup \{p_x\}, p_y, d_y - d_x, \Theta_y]$ to the occurrence list of the generated candidate.

3.4 Support counting

Checking if a pattern is frequent consists of counting the elements in its occurrence list. The counting procedure is different depending on whether the weighted support σ_w is considered or not.

- If we count occurrences using the weighted support, all the tuples in the lists must be taken into account.
- If we are not using the weighted support, the support of a pattern is the number of different sequence identifiers within the tuples in the occurrence list of the pattern.

It should be noted that d represents the distance between the last node in the pattern and its prefix m. Therefore, we only have to consider the elements in the scope lists whose d parameter equals 1 for guaranteeing that elements in the pattern are contiguous within the sequence. It is important to remark that the remaining elements in the lists cannot be eliminated because they are needed to build the occurrence lists of larger patterns.

3.5 Representative patterns

Our algorithm returns all the frequent patterns of the maximum size indicated by the user (or smaller ones if there are no patterns of such size). As musical motifs are generally no longer than a measure, a value of ten is typically used by default. Nevertheless, this limit can be easily modified since our algorithm can return all the frequent patterns that exist in the song regardless of their size. The resulting output will be the set of frequent patterns that represent the song. The algorithm also returns the positions of the different occurrences of the patterns within the song (including transposed occurrences if needed).

3.6 SSMiner complexity

SSMiner starts by computing the frequent patterns of size 1. This step is performed by obtaining the vertical representation of the sequence database, i.e., the individual notes that appear in the sequences with their occurrences represented as scope lists. This representation is obtained in linear time with respect to the number of sequences in the database just by scanning it and building the scope lists for patterns of size 1. We then discard the patterns of size 1 that are not frequent. This results in L scope lists corresponding to the L frequent notes in the sequence database and each frequent label leads to a candidate class of size 1.

Let $c(k)$ be the number of classes of size k, which equals the number of frequent patterns of size k, and $e(k)$ the number of elements that might belong to a particular class of size k (i.e., the number of patterns of size $k+1$ that might be included in the class corresponding to a given pattern of size k).

In SSMiner, each sequence pattern grows only by adding an element at the end of the sequence pattern. The number of different sequences of size $k+1$ that can be obtained by the extension of a sequence of size k is $L \cdot k$. Hence, the number of elements in a particular class, $e(k)$, is $O(L)$.

The number of classes of size 1 equals L, the number of frequent labels, so that $c(1) = L$. The classes of size $k+1$ are derived from the frequent elements in classes of size k. In the worst case, when all the $e(k)$ elements are frequent, $c(k+1) = c(k) \cdot e(k)$. Solving the recurrence, we obtain $c(k+1) = c(k) \cdot L = O(L^k)$.

For each considered pattern of size $k+1$, SSMiner must perform a join operation to obtain its scope list from the scope lists of the two patterns of size k that led to it.

The size of the scope list for a pattern of size k is $O(t \cdot e)$ while the computational cost of a scope-list join operation is $O(t \cdot e^2)$, where t is the number of sequences in the database and e is the average number of embeddings of the pattern in each sequence (Zaki 2005a).

In the worst case, the number of embeddings $s(k-1)$ of a pattern of size $k-1$ in a sequence of size S equals the number of subsequences of size $k-1$ within the sequence of size S. This number is bounded by $S-k+1$.

Hence, the cost of the join operation needed for obtaining the scope list of a pattern of size k, is $j(k) = O(t \cdot s(k-1)^2) = O(t \cdot (S-k+1)^2)$.

The cost of obtaining all the frequent patterns of size k will be, therefore, $O(c(k) \cdot j(k)) = O(L^k \cdot t \cdot (S-k+1)^2)$

The total cost of executing the SSMiner algorithm to obtain all the frequent patterns up to $k = MaxSize$ is $\sum_{k=1 \ldots MaxSize}(L^k \cdot t \cdot (S-k+1)^2)$. Since the running time of our algorithm is dominated by the time needed to discover the largest patterns (i.e., $k = MaxSize$), SSMiner is $O(L^{MaxSize} \cdot t \cdot (S - MaxSize + 1)^2)$.

Therefore, our algorithm execution time is proportional to the number of sequences in the sequence database ($t = 1$ in our motifs identification problem), and to the number of patterns than can be identified (L^k). Finally, its execution time is quadratic with respect to the size of the sequences (S).

4 An example

In this section, we present an example to help the reader understand the way our algorithm identifies frequent subsequences in a sequence. In order to facilitate the understanding of the procedure, we are not considering the duration of notes. Furthermore, we only take into account those transpositions of fifth.

We will use in this paper the scientific pitch notation which combines a letter-name, accidentals (if any) and a number identifying the pitch's octave. This notation is the most common in English written texts.

Let's suppose we have the following piece of a song: G4 A4 G4 E4 D5 E5 D5 B4 G4 A4 G4 E4 A4 G4 B4 G4 A4 G4 E4 (see Fig. 2), and we want to extract those subsequences that appear at least four times in it.

The first step of our algorithm is scanning the sequence to obtain all the occurrences of each note. Then, the occurrence lists of each note are built as indicated in Section 3.3. Results are shown in Fig. 3.

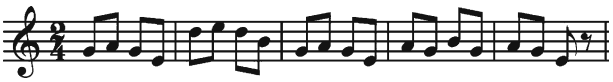

Fig. 2 Sample piece: G4 A4 G4 E4 D5 E5 D5 B4 G4 A4 G4 E4 A4 G4 B4 G4 A4 G4 E4

The first element is 1 in all the tuples because we only have one sequence (i.e., only one song) in our example. The second one is the prefix of the substring (empty in patterns of size 1). The third element indicates the position of the last element of the pattern in the sequence. The fourth element is the distance between the last element of the pattern and its prefix (or the position of the element when there is no prefix, as this is the case). Finally, the last element indicates if the occurrence is exactly equal to the pattern ('=') or if it is transposed.

In this example, only those transpositions of up one fifth are being taken into account, so that '+5' is the only alternative for this element in our example. It should be noted, however, that all possible transpositions—distances between two notes—could be taken into account. In any case, we need only to compare notes in one direction, as we will always find at least a version of the pattern that summarizes all its transpositions.

Going back to our example, the note G4 is transposed up one fifth as D5. Therefore, there are 9 tuples in the occurrence list of G4: 7 as itself, 2 as D5.

The next step is checking if all the notes are frequent. In this case, only G4, A4, and E4 have at least four occurrences. Therefore, only these patterns will be kept.

Figure 4 shows the extension of the element G4. This element is extended with all the frequent patterns of size 1 including itself, and the occurrence lists of each candidate pattern of size 2 are obtained by joining the lists of the elements that generated it, as explained in Section 3.3.

Figure 4 shows, with bold letters, the tuples where $d = 1$. That means that these are contiguous occurrences of the pattern. In our example, only the patterns G4 A4 and G4 E4 appear as contiguous subsequences in our song. Furthermore, they have at least four occurrences—our minimum support threshold—and they will be extended to generate candidates of size 3. It should be noted that the pattern G4 G4

G4	**A4**	**E4**	**D5**	**E5**	**B4**
{1,_,1,1,=}	{1,_,2,2,=}	{1,_,4,4,=}	{1,_,5,5,=}	{1,_,6,6,=}	{1,_,8,8,=}
{1,_,3,3,=}	{1,_,10,10,=}	{1,_,12,12,=}	{1,_,7,7,=}		{1,_,15,15,=}
{1,_,9,9,=}	{1,_,13,13,=}	{1,_,19,19,=}			
{1,_,11,11,=}	{1,_,17,17,=}	{1,_,8,8,+5}			
{1,_,14,14,=}	{1,_,6,6,+5}	{1,_,15,15,+5}			
{1,_,16,16,=}					
{1,_,18,18,=}					
{1,_,5,5,+5}					
{1,_,7,7,+5}					

Fig. 3 Occurrence lists of the elements of the following sequence: G4 A4 G4 E4 D5 E5 D5 B4 G4 A4 G4 E4 A4 G4 B4 G4 A4 G4 E4

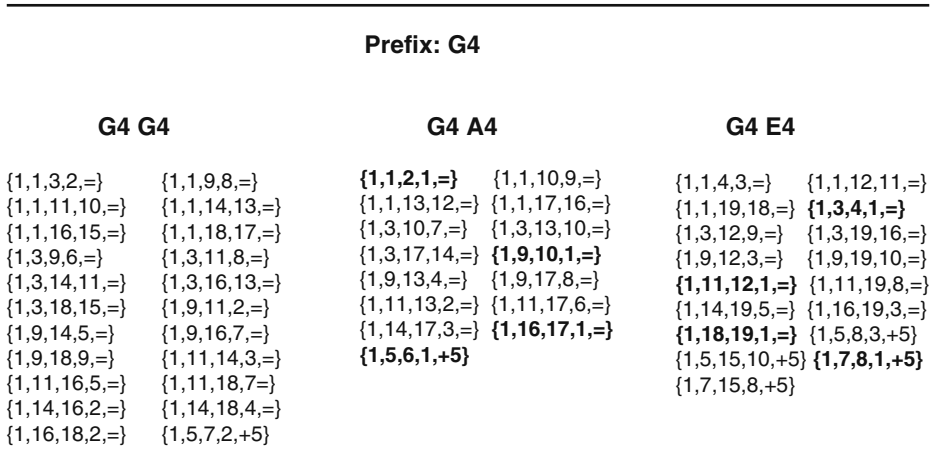

Fig. 4 Extension of the element G4 in Fig. 3

is not contiguous and will not be extended. However, it is preserved to perform the extension of G4 A4 with G4 E4.

After two more extensions, which are done in the same way, we obtain the pattern G4 A4 G4 E4 with a *support* of 4 and an *exact support* of 3. This is be the pattern we would use to characterize our example song.

5 Experiments

We have tested our algorithm using a corpus of 44 songs. This set includes songs from a wide variety of authors. The first column in Table 1 shows the songs used in our experiments.

We have performed 4 experiments with different constraints:

- Exact pitch and duration (**pitch-duration**)
- Exact pitch and any rhythm (**pitch**)
- Transpositions but exact duration (**transposition-duration**)
- Transpositions and any duration (**transposition**)

Pitch-duration is the most restrictive one, whereas **transposition** is the experiment with a lower number of constraints. All these configurations are representative when looking for musical motifs, as they can be modified in tempo or in pitch. Unlike the example in the former section, all the possible transpositions are taken into account in these experiments.

Table 1 summarizes the results of our experiments. Each row corresponds to one of the songs in the corpus. The second column (*'notes'*) indicates the number of notes in each song.

The *'Max Size'* column indicates the size of the longest pattern(s) found in each song. Patterns of this size are the only ones that are finally returned to the user. As our aim in this paper is searching for repeating motifs, and not whole repeating sections, we have to introduce an upper limit to the size of the patterns. Preliminary

Table 1 Corpus of songs and results for each song in our four series of experiments

Song	Notes	Pich-duration			Pich			Transposition-duration				Transposition			
		Max S	# pat	r_sup	Max S	# pat	r_sup	Max S	# pat	r_sup	t_sup	Max S	# pat	r_sup	t_sup
Alfie	192	7	1 / 41	4	7	1 / 45	5	7	1 / 120	4	4	7	1 / 135	5	5
All good things	248	9	2 / 97	[4–5]	9	2 / 103	[4–5]	9	2 / 151	[4–5]	[4–5]	9	2 / 183	[4–5]	[4–5]
Angels	388	8	1 / 98	4	10	3 / 195	4	8	1 / 232	4	4	10	3 / 267	4	4
Angie	530	6	2 / 106	[4–5]	10	4 / 188	4	6	2 / 401	[4–5]	[4–5]	10	4 / 455	4	4
Apologize	384	10	9 / 217	4	10	10 /209	4	10	9 / 318	4	4	10	10 / 283	4	4
Bach	182	5	2 / 38	4	5	2 / 44	4	10	2 / 178	2	5	9	1 / 189	2	4
Ballade pour Adeline	453	10	33 / 376	[4–12]	10	36 / 353	[4–30]	10	33 / 566	[4–18]	[4–27]	10	36 / 528	[4–36]	[4–52]
Beat it	171	4	5 / 42	[4–5]	6	1 / 51	4	4	5 / 62	[4–5]	[4–6]	7	1 / 68	2	4
Beautiful	332	9	1 / 82	4	10	1 / 112	4	9	1 / 331	4	4	10	1 / 265	4	4
Beautiful that way	268	10	20 / 243	4	10	20 / 235	[4–5]	10	20 / 316	4	4	10	20 / 356	4	4
Bleeding love	539	9	1 / 148	4	9	2 / 185	[4-5]	9	1 / 277	4	4	9	2 / 320	[4–5]	[4–5]
Brown eyed girl	138	3	3 / 25	[4–7]	5	1 / 34	4	4	1 / 54	3	4	5	2 / 71	[3–4]	[4–5]
Crazy	307	5	2 / 81	4	6	2 / 85	4	6	2 / 285	[1–3]	4	6	3 / 201	[3–4]	4
Don't speak	334	9	1 / 97	4	10	2 / 145	4	9	1 / 244	4	4	10	2 / 217	4	4
Every breath you take	295	6	1 / 56	7	6	1 / 68	4	8	1 / 241	3	5	8	4 / 265	[3–4]	[4–5]
Everybody hurts	271	4	1 / 46	4	8	3 / 83	4	4	2 / 121	[1–4]	4	6	1 / 149	4	4
Feel	394	10	9 / 159	[5–6]	10	13 / 172	4	10	9 / 393	[5–6]	[5–6]	10	13 / 364	[4–7]	[4–7]
Fever	121	8	1 / 31	4	8	1 / 37	[4–7]	8	1 / 45	4	4	8	1 / 54	4	4
Fur Elise	267	10	16 / 202	[4–6]	6	16 / 77	4	10	16 / 396	[4–6]	[4–6]	10	16 / 391	[4–6]	[4–6]
Hero	259	5	1 / 56	4	10	1 / 141	5	6	1 / 220	1	4	6	1 / 214	5	5
How to save a life	448	7	1 / 102	5	10	3 / 100	5	7	1 / 251	5	5	10	1 / 226	5	5
I hate this part	231	10	3 / 94	4	10	6 / 103	[4–7]	10	6 / 301	[1–4]	[4–5]	10	6 / 333	[1–4]	[4–5]
I say a little prayer	172	4	2 / 32	4	4	3 / 39	4	5	2 / 95	3	[4–5]	5	1 / 92	2	5
Imagine	188	5	1 / 44	5	6	1 / 49	[4–5]	6	1 / 93	1	4	10	6 / 150	[4–5]	[4–5]

Let it be	179	5	1 / 32	4	10	1 /49	4	5	1 / 84	4	4	6	1 / 129	4	4
Livin la Vida Loca	434	10	19 / 342	[4–6]	5	22 /345	[4–8]	10	19 / 507	[4–6]	[4–6]	10	23 / 455	[2–8]	[4–8]
Love me tender	53	2	2 / 7	[4–6]	4	1 / 33	4	5	1 / 28	1	4	5	2 / 33	[1–4]	4
Paint it black	129	2	3 / 19	[4–6]	10	16 / 205	5	3	4 / 51	2	[4–6]	5	1 / 48	2	4
Quizas quizas quizas	156	8	1 / 61	6	9	1 / 70	[4–6]	8	1 / 146	6	6	9	1 / 135	6	6
Roxanne	396	10	18 / 213	[4–10]	10	20 / 226	6	10	18 / 270	[4–10]	[4–10]	10	20 / 256	[4–10]	[4–10]
Smile	105	4	2 / 30	[4–6]	4	4 / 33	[4–10]	10	4 / 211	2	4	10	1 / 91	2	4
Sunshine of my life	352	10	26 / 308	[5–6]	10	27 / 305	[5–6]	10	26 / 419	[5–6]	[5–6]	10	27 / 409	[5–6]	[5–6]
The nearness of you	125	4	1 / 24	4	5	3 / 36	4	5	1 / 82	1	4	5	4 / 90	[1–4]	4
Torn	388	10	11 / 194	[4–5]	10	11 / 200	[4–5]	10	11 / 275	[4–5]	[4–5]	10	11 / 271	[4–5]	[4–5]
Under the bridge	541	10	9 / 246	4	10	14 / 291	4	10	9 / 661	4	4	10	14 / 889	[4–6]	[4–6]
Underneath your clothes	260	5	1 / 55	4	5	3 / 67	[4–6]	5	1 / 206	4	4	7	1 / 224	2	4
Viva la vida	444	10	1 / 165	4	10	13 / 235	[4–5]	10	1 / 366	4	4	10	13 / 476	[4–5]	[4–5]
We're the champions	303	5	1 / 62	4	8	1 / 93	4	5	1 / 206	4	4	8	2 / 256	[2–4]	4
What a wonderful world	179	7	1 / 54	4	8	1 / 61	6	7	1 / 149	4	4	8	1 / 109	6	6
When I'm sixtyfour	185	6	1 / 30	4	7	1 / 41	4	7	1 / 116	1	4	7	2 / 186	[1–4]	[4–5]
With or without you	236	5	1 / 44	4	5	1 / 57	4	5	1 / 204	4	4	5	2 / 165	[1–4]	[4–5]
Wonderwall	184	9	1 / 68	4	9	1 / 75	4	9	1 / 121	4	4	9	1 / 135	4	4
Your song	236	5	1 / 43	4	5	1 / 51	4	5	1 / 118	4	4	5	1 / 96	4	4
Zombie	239	10	3 / 92	4	10	3 /100	4	10	3 / 130	4	4	10	3 / 141	4	4

tests with our song corpus have shown that a value of ten is adequate for this parameter providing that musical motifs are generally no longer than a measure.

The *'Patterns'* column indicates the number of patterns our algorithm finds. The first number is the amount of patterns of maximum size (i.e. the size indicated on the previous column); whereas the second number is the total amount of patterns, regardless of their size. As the reader can see, the total amount of patterns is pretty high. However, many of these patterns are not musically relevant since they are part of bigger ones. Even more, patterns of size 1 and 2, which can hardly be considered as motifs, are also included in this set.

The *'exact support'* column is the exact support of the returned patterns. Intervals appear in some cases due to the fact that not all the returned patterns necessarily appear the same number of times.

Finally, *'transposed support'* indicates the support of patterns including transpositions. This column is only relevant for the experiments **transposition-duration** and **transposition**, when transpositions are taken into account. As expected, values in this column are always equal or greater than those in the corresponding *'exact support'* column.

For us, the minimum support required for a pattern to be considered frequent is four. Any pattern with a lower number of repetitions will be deleted. This value has been manually set regarding the size of the evaluating songs and some preliminary tests. However, it can be easily adjusted when new datasets require it.

The reader can also notice that, in some cases (namely, when transpositions are taken into account), patterns with *exact support* lower than four can be found. Those patterns do not have enough exact repetitions as themselves, but they are frequent when transpositions are taken into account. Hence, transpositions are important because, without considering them, some patterns would have not been discovered, as they do not reach the minimum support threshold just by exact repetitions. That happens, for instance, with the 'Crazy' and 'Hero' songs.

Regarding the length of the excerpts that have been tested, three notes can hardly represent any meaningful motif. However, almost anyone could identify Beethoven's Fifth Symphony by just four notes. Hence, a minimum length of four seems adequate.

It should be noted that, in some situations, there is another transposition of the pattern that has greater *exact support* than the one returned by our algorithm as described in Section 3. As we mentioned earlier, our algorithm looks for transposed motives only in one direction and this suffices to guarantee that it will find all the relevant occurrences; however, the returned patterns are not necessarily the most frequent exact ones. Given that our algorithm keeps track of the occurrences of a given pattern and all of its transpositions, it is trivial to obtain the most frequent exact pattern just by looking at the corresponding scope list. This pattern will correspond to the most common Θ in the scope list.

Table 2 Percentage of songs that include at least one identified pattern within their chorus

	Pitch-duration	Pitch	Transposition-duration	Transposition
%Yes	63.64	68.18	63.64	72.73
%No	29.55	25.00	29.55	20.45
%Without chorus	6.82	6.82	6.82	6.82

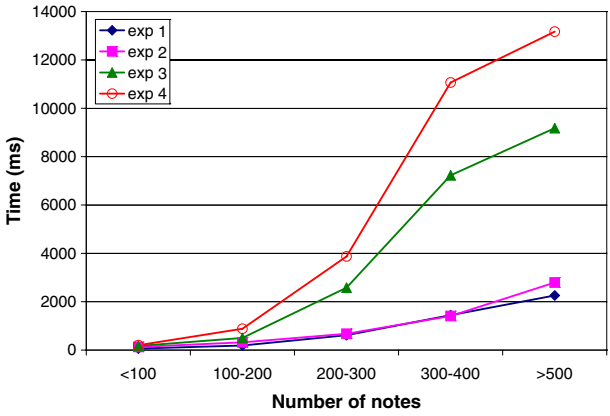

Fig. 5 SSMiner execution time

It should also be observed that, in some songs, the number of patterns of *MaxSize* elements is pretty high (e.g. 'Ballade pour Adeline' or Beethoven's 'Für Elise'). This is due to the fact that many of those are still subpatterns of bigger ones. For instance, a pattern of size 15 includes six sequential subpatterns of size 10 (starting from the 1st, 2nd, 3rd, 4th, 5th and 6th note, respectively).

In order to evaluate the goodness of our method, we have checked whether or not the discovered frequent patterns belong to the chorus of the song—in our experiments, 6.82% of the songs do not have a clear chorus. Table 2 shows the percentage of songs which have at least one identified pattern within their chorus. As can be seen, above 60% of the songs fulfill this requirement. Also, it is remarkable that not considering the rhythm results in more patterns belonging to the chorus of the songs. This fact indicates that patterns are not always exactly repeated as themselves, but slightly modified. Although the chorus-belonging criterion appears to be a valid and obvious one, it should be noted that some songs are better identified by patterns which do not belong to the chorus.

Regarding SSMiner computation time, Fig. 5 shows the time consumed in each experiment with respect to the number of notes in the melody. The chart groups the songs into five groups according to their lengths and displays the average execution time for each subset of melodies. These execution times are quadratic with respect to the number of notes in the melodies, as explained in Section 3.6.

6 Conclusions

We have presented the application of frequent pattern mining to the discovery of musical motifs in a piece of music. *MusicXML* files, which can be easily collected, are transformed into sequences of notes, defined at their lower level. Our algorithm, SSMiner, is able to efficiently identify frequent subsequences in a sequence.

The matching between the patterns does not need to be exact. Our algorithm is able to identify transposed patterns including exact matchings, i.e., null transpositions. Our experiments suggest that our approach performs well in a set of randomly-selected songs.

In the future, we intend to employ interval strings to represent notes rather than the absolute pitches we have used in the experiments reported in this paper. We will also consider more abstract representations of melodies, such as the one proposed by Narmour. Finally, we plan to study the parallelization of our algorithm implementation in order to improve its execution time, which is already asymptotically optimal.

Acknowledgements F. Berzal and A. Jiménez are supported by the projects TIN2006-07262 and TIN2009-08296, whereas W. Fajardo and M. Molina-Solana are supported by the research project TIN2006-15041-C04-01.

References

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules in large databases. In *20th int. conf. on very large data bases* (pp. 487–499).

Aucouturier, J. J., & Sandler, M. (2002). Finding repeating patterns in acoustic musical signals: Applications for audio thumbnailing. In *Audio engineering 22nd int. conf. on virtual, synthetic and entertainment audio (AES22)* (pp. 412–421).

Bartsch, M., & Wakefield, G. (2005). Audio thumbnailing of popular music using chroma-based representations. *IEEE Transactions on Multimedia, 7*(1), 96–104.

Berzal, F., Fajardo, W., Jiménez, A., & Molina-Solana, M. (2009). Mining musical patterns: Identification of transposed motives. In *18th Int. symposium of foundations of intelligent systems*. Lecture Notes in Computer Science, vol. 5722, pp. 271–280.

Böckenhauer, H. J., & Bongartz, D. (2007). *Algorithmic aspects of bioinformatics*. New York: Springer.

Cambouropoulos, E., Crawford, T., & Iliopoulos, C. S. (2001). Pattern processing in melodic sequences: Challenges, caveats and prospects. *Computers and the Humanities, 35*(1), 9–21.

Chu, S., & Logan, B. (2002). Music summary using key phrases. In *IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP-00)* (pp. 749–752).

Dong, G., & Pei, J. (2007). *Sequence data mining (advances in database systems)*. New York: Springer.

Grachten, M., Arcos, J. L., & de Mantaras, R. L. (2004). Melodic similarity: Looking for a good abstraction level. In *5th Int. Conf. on Music Information Retrieval (ISMIR 2004)* (pp. 210–215).

Han, J., & Kamber, M. (2005). *Data mining: Concepts and techniques*. Denver: Morgan Kaufmann.

Hsu, J. L., Liu, C. C., & Chen, A. (1998). Efficient repeating pattern finding in music databases. In *ACM 7th int. conf. on information and knowledge management* (pp. 281–288).

Jiang, L., & Hamilton, H. J. (2003). Methods for mining frequent sequential patterns. In *Advances in artificial intelligence, Lecture of Notes in Computer Sciences* (Vol. 2671/2003, pp. 486–491). Berlin: Springer.

Jimenez, A., Berzal, F., & Cubero, J. C. (2009). Mining induced and embedded subtrees in ordered, unordered, and partially-ordered trees. *Knowledge and Information Systems, 4994/2008*, 111–120. doi:10.1007/s10115-009-0213-3.

Levy, M., & Sandler, M. (2008). Structural segmentation of musical audio by constrained clustering. *IEEE Transactions on Audio, Speech, and Language Processing, 16*(2), 318–326.

Meredith, D., Lemström, K., & Wiggins, G. A. (2002). Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music. *Journal of New Music Research, 31*(4), 321–345

Narmour, E. (1992). *The analysis and cognition of melodic complexity: The implication realization model*. Chicago: Univ. Chicago Press.

Paulus, J., & Klapuri, A. (2009). Music structure analysis using a probabilistic fitness measure and a greedy search algorithm. *IEEE Transactions on Audio, Speech, and Language Processing, 17*(6), 1159–1170.

Pei, J., Han, J., Asl, M. B., Pinto, H., Chen, Q., Dayal, U., et al. (2001). Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *5th int. conf. on extending database technology* (pp. 215–224).

Pienimäki, A. (2002). Indexing music databases using automatic extraction of frequent phrases. In *3rd int. conf. on music information retrieval* (pp. 25–30).

Rolland, P. Y. (1998). Discovering patterns in musical sequences. *Journal of New Music Research, 28*(4), 334–350

Srikant, R., & Agrawal, R. (1996). Mining sequential patterns: Generalizations and performance improvements. *Extending Database Technology, 1057*, 3–17.
Wang, W., Yang, J., & Yu, P. S. (2001). Meta-patterns: Revealing hidden periodic patterns. In *IBM research report* (pp. 550–557).
Yang, J., Wang, W., & Yu, P. S. (2001). Infominer: mining surprising periodic patterns. In *7th ACM int. conf. on knowledge discovery and data mining (SIGKDD)* (pp. 395–400). New York: ACM
Zaki, M. J. (2001). Spade: an efficient algorithm for mining frequent sequences. *Machine Learning, 42*, 31–60.
Zaki, M. J. (2005a) Efficiently mining frequent embedded unordered trees. *Fundamenta Informaticae, 66*(1–2), 33–52
Zaki, M. J. (2005b). Efficiently mining frequent trees in a forest: Algorithms and applications. *IEEE Transactions on Knowledge and Data Engineering, 17*(8), 1021–1035.
Zhang, T., & Samadani, R. (2007). Automatic generation of music thumbnails. In *Proceedings of the 2007 IEEE int. conf. on multimedia and expo* (pp. 228–231).