A Pattern Extraction Algorithm for Abstract Melodic Representations that Allow Partial Overlapping of Intervallic Categories

Emilios Cambouropoulos, Maxime Crochemore, Costas S. Iliopoulos, Manal Mohamed, Marie-France Sagot

To cite this version:

HAL Id: hal-00619902
https://hal-upec-upem.archives-ouvertes.fr/hal-00619902
Submitted on 25 Mar 2013
A PATTERN EXTRACTION ALGORITHM FOR ABSTRACT MELODIC REPRESENTATIONS THAT ALLOW PARTIAL OVERLAPPING OF INTERVALVIC CATEGORIES

Emilios Cambouropoulos1, Maxime Crochemore2, Costas Iliopoulos3, Manal Mohamed3, Marie-France Sagot4

1 Department of Music Studies, University of Thessaloniki, 54006, Thessaloniki, Greece
2 Institut Gaspard-Monge, University of Marne-la-Vallée, 77454 Marne-la-Vallée CEDEX 2, France
3 Department of Computer Science, King’s College London, London WC2R 2LS, England
4 INRIA Rhône-Alpes, Université Claude Bernard, 43 Bd du 11 novembre 1918, 69622 Villeurbanne cedex, France

ABSTRACT

This paper proposes an efficient pattern extraction algorithm that can be applied on melodic sequences that are represented as strings of abstract intervallic symbols; the melodic representation introduces special “don’t care” symbols for intervals that may belong to two partially overlapping intervallic categories. As a special case the well established “step-leap” representation is examined. In the step-leap representation, each melodic diatonic interval is classified as a step (±s), a leap (±l) or a unison (u). Binary don’t care symbols are introduced to represent the possible overlapping between the various abstract categories e.g. ∗ = s, ∗ = l and # = −s, # = −l. For such a sequence, we are interested in finding maximal repeating pairs and repetitions with a hole (two matching subsequences separated with an intervening non-matching symbol). We propose an \( O(n + d(n - d) + z) \)-time algorithm for computing all such repetitions in a given sequence \( x = x[1..n] \) with \( d \) binary don’t care symbols, where \( z \) is the output size.

Keywords: string, don’t care, repetitions, suffix tree, lowest common ancestor.

1 INTRODUCTION

Recently, there have been different proposals in the literature to develop an effective music information retrieval system. The goal of these proposals is to take advantage of appropriate computer science techniques. For example, representing the musical surface as a string or set of strings may make it possible in some cases to apply existing algorithms from the field of stringology. For instance, in order to discover similarities between different musical entities or to establish motivic “signatures”, music analysts may use algorithms that extract repetitions from strings. Such similarities often involve finding approximate repetitions (Crawford et al., 1998). This requires developing new approximation measures that meet musicians’ needs.

One commonly used representation for music is the numeric representation MIDI. For such a representation, different approximation measures have been developed, such as, \( \delta \), \( \gamma \) and \{\( \delta \), \( \gamma \}\}-approximate. For example, in \( \delta \)-approximate matching, equal-length strings consisting of integers match if each corresponding integer differs by not more than \( \delta \) – e.g. a C-major \{60, 64, 65, 67\} and a C-minor \{60, 63, 65, 67\} sequence can be matched if a tolerance \( \delta = 1 \) is allowed in the matching process. Using these approximation measures, algorithms for finding approximate repetitions in musical sequences have been developed (Iliopoulos et al., 2000; Cambouropoulos et al., 2002). These algorithms are based on approximate pattern matching techniques. For an overview refer to Clifford and Iliopoulos (2004).

Although MIDI is the most common representation in the computational domain, it has certain well-known shortcomings, for instance, many important musical properties are not explicitly represented (e.g. note durations, accidentals etc.) and almost all information on musical structure is lost. Therefore, different representations have been proposed in the literature. For example, Hawley (1993) proposed representing the musical signal as a sequence of pitch intervals. In order to allow tolerance in interval matching, Ghias et al. (1995) used the reduced interval alphabet of the “melodic contour” representation. Lemström and Laine (1998) proposed classifying the intervals into seven partially overlapping classes: small, medium and large, up- or downwards, and prime.

In this paper, we propose an alternative method to using approximate pattern matching techniques for finding approximate repetitions in a musical string. Our approach is based on using exact pattern matching techniques to extract repetitions from an abstract level of a musical sequence. As an abstract representation, we will use the “refined contour” (or step-leap) representation - see, for instance, application of this representation in <http://www.themefinder.org/>. In the step-leap representation, intervals are classified into five distinct equivalence classes: up- or downwards step and leap, and unison. An interval with magnitude \( a = 0 \) is a unison (u), \( a < 2 \) is a step (s), and any other interval \( a \geq 2 \) is a leap (l); the direction of intervals is preserved – see second
string of symbols in Figure 1.

In the second string of symbols in Figure 1, two repeated substrings are found: \(-s\ s\ -l\) and \(l\ s\ s\ s\ -l\) each occurring twice. However, for a listener/musician, the second half of this string of intervals is an approximate repetition of the first half (two approximately matching substrings separated by a “hole” of size one) because intervals \(a = 1\) and \(a = 2\) are considered similar (i.e. a step is similar to a small leap). This is not simply some rare exception in music. It is a rather common phenomenon especially when themes appear in their dominant form (see, for instance, the tonal answers of almost half of Bach’s fugue themes from the two books of the Well-Tempered Clavier). In Figures 2 and 3 some melodic examples are presented.

The problem in the step-leap representation is that the abstract interval classes \((u, s, l)\) have sharp boundaries and no diatonic pitch interval instance may belong to more than one class. In other words, borderline members can never be matched to other ‘similar’ members of other classes (e.g. an \(a = 2\) as member of leap can never be matched to a ‘similar’ \(a = 1\) interval which is a step). A way to overcome this problem is to allow partial overlapping between the various classes (this is also suggested in Lemström and Laine (1998)). For instance, an interval \(a = 2\) may be classified as either step or leap. A special “binary don’t care” symbol \(*\) that matches either \(s\) or \(l\) is used (see third string of symbols in Figure 1). Similarly, another don’t care symbol \#\ that matches both \(-s\) and \(-l\) is also used. Note that this idea can be easily extended to any constant number of partially overlapping classes, such as those proposed in Lemström and Laine (1998).

A pattern extraction algorithm may find a large number of repeating patterns. Many of these are not musically or perceptually important. A mechanism, therefore, for selecting important patterns is required. A relatively sophisticated method for finding beginnings of potentially salient repetitions is proposed by Cambourooulos. Following this proposal, for each pattern a prominence value is calculated based on frequency of occurrence, pattern length and degree of pattern overlapping; these prominence values contribute to establishing a segmentation prominence profile for a melody whereby the most likely positions of important repetitions are highlighted. In this paper, where the aim is not segmentation but the extraction of interesting patterns, simpler criteria are set: extraction of maximal repeating pairs and repetitions with a ‘hole’. The later involves extracting immediately repeated non-overlapping melodic patterns - in order that the patterns do not overlap over one note a ‘hole’ is necessary in the pitch interval representation (for example, in Figure 1, the second half of the melody is an approximate repetition of the first half - in the interval representations beneath the melody it is necessary to skip one symbol in the middle so that the repetition is consecutive and non-overlapping at the note level).

For strings with don’t cares, several string matching algorithms have been proposed (see (Fischer and Paterson, 1974; Apostolico and Preparata, 1983; Amir et al., 2001)). Recently, Iliopoulos et al. (2003) proposed algorithms for computing typical regularities in strings with don’t cares. Here, we consider binary don’t care symbols that each matches beside itself two additional symbols. For string \(x = x[1..n]\) with \(d\) binary don’t cares, we propose an algorithm for computing special kinds of repetition that we refer to as “maximal-pairs” and “repetitions with a hole”. The proposed algorithm uses \(O(n + d(n - d) + z)\) time, where \(z\) is the output size.

The paper is organized as follows: in Section 2, we state the preliminaries used throughout the paper. In Section 3, we define all approximate repetitions problem and describe in general how to find them. In Section 4, we detail our algorithm. Finally, in Section 5, we analyze the running time of the algorithm.

## 2 Preliminaries

Throughout the paper, \(x = x[1..n]\) denotes a string of length \(n\) over \(\Sigma \cup \{\ast, \#\}\), where \(\Sigma = \{\ast, s, l, \ast\}\). The symbols \(\ast\) and \(\#\) are called binary don’t care symbols. Each binary don’t care symbol matches itself and two different symbols, that is, \(\ast = \ast, \ast = s, \ast = l, \# = \ast, \# = s\) and \(\# = l\).

We use \(x[i]\), for \(i = 1, 2, \ldots, n\), to denote the \(i\)-th symbol of \(x\), and \(x[i..j]\) as a notation for the substring \(x[i]x[i+1] \cdots x[j]\) of \(x\). If \(x = \varepsilon\) then \(x\) is said to be the concatenation of the two strings \(u\) and \(v\). A string \(y\) is said to occur in \(x\) at position \(i\) if \(y[j] = x[i + j - 1]\), for \(1 \leq j \leq |y|\).

A repeating pair in \(x\) is represented by \((p; i, j)\) where, \(x[i..i+p-1] = x[j..j+p-1]\) for some \(i \neq j\). The positive integer \(p\) is called the period of the repeating pair. If \(x[i-1] \neq x[j-1]\) then \((p; i, j)\) is left-maximal.
Opening melody of Beethoven’s Piano Sonata Op.10, No.2.

Upper voice ‘stream’ (theme and tonal answer) from the opening of Bach’s Well-Tempered Clavier, Book I, Fugue in F# major.

Upper voice ‘stream’ (theme and tonal answer) from the opening of Bach’s Well-Tempered Clavier, Book I, Fugue in C minor.

Second theme from Shostakovich’s String Quartet No 4 in D major, Op. 83, Mov. 4

Figure 2: Melodic examples where an $a = 1$ interval (step) and an $a = 2$ (leap) should be matched (these positions are indicated by asterisks in the melodic examples). Brackets indicate extracted melodic repetitions

Figure 3: The opening melody of Mussorgsky’s, Pictures from an exhibition, Promenade. Extracted maximal repeating pairs and repetitions with a ‘hole’ are indicated by brackets
If \( x[i + p] \neq x[j + p] \) then \( (p; i, j) \) is right-maximal. If \( (p; i, j) \) is both left- and right-maximal then it is maximal-pair. A repetition with a hole is a repeating pair \( (p; i, j) \) such that \( j = i + p + 1 \).

Here, we present a method for finding all maximal-pairs and all repetitions with a hole in a given string \( x \), where \( x \) may have occurrences of binary don’t cares. Our method uses the suffix tree of \( x \) as a fundamental data structure. A complete description of suffix trees is beyond the scope of this paper, and can be found in (Gusfield, 1997) or (Crochemore and Rytter, 2002). However, for the sake of completeness, we will briefly review the notion.

**Definition 1 (Suffix tree)** A suffix tree \( T(x) \) of the string \( x \) is a rooted directed tree with exactly \( n \) leaves numbered 1 to \( n + 1 \), where \( \Sigma \notin \$. Each internal node, other than the root, has at least two children and each edge is labelled with a non-empty substring of \( x \). No two edges out a node can have edge-labels beginning with the same symbol. The key feature of the suffix tree is that for any leaf \( i \), the concatenation of the edge-labels on the path from the root to leaf \( i \) exactly spells out \( i \)-th suffix of \( x \), with \( n + 1 \) denotes the empty suffix.

Several algorithms construct the suffix tree \( T(x) \) in \( O(n) \) time and space, assuming constant size alphabet (see for example (Crochemore and Rytter, 2002) and (Gusfield, 1997)). For any node \( v \), the label of \( v \) is the label of the path from the root of \( T(x) \) to \( v \); it is denoted by \( label(v) \). The string-depth of \( v \) is the number of symbols in \( v \)’s path-label; it is denoted by \( depth(v) \). The leaflist of \( v \) is the set of the leaf numbers in the subtree rooted at \( v \); it is denoted by \( LL(v) \).

Our method makes use of the Schieber and Vishkin *(1988)*’s **Lowest Common Ancestor** algorithm. For a given rooted tree \( T \), the **lowest common ancestor** \( (LCA) \) of two nodes \( u \) and \( v \) is the deepest node in \( T \) that is ancestor of both \( u \) and \( v \). After a linear amount of preprocessing of a rooted tree, any two nodes can be specified and their lowest common ancestor is found in constant time. That is, a rooted tree with \( n \) nodes is first preprocessed in \( O(n) \) time, and thereafter any lowest common ancestor query takes only a constant time to be solved, independent of \( n \).

In the context of suffix trees, the situation commonly arises that both \( u \) and \( v \) are leaves in \( T(x) \), where \( x[i..n] \) and \( x[j..n] \) are the suffixes represented by \( u \) and \( v \) respectively, for integers \( i \) and \( j \) in the range \( 1..n + 1 \). In this case, the node \( w = LCA(u, v) \) is the root of the minimum size subtree contains \( u \) and \( v \). Note that the path-label of \( w \) \( label(w) \) is the longest common prefix of \( x[i..n] \) and \( x[j..n] \). The ability of finding such longest common prefix is an important primitive in many string problems.

### 3 FINDING ALL REPETITIONS PROBLEM

Here, we study the problem of finding in a given string \( x \) over \( \Sigma \cup \{*, \#\} \), the following two kinds of repetitions: all maximal-pairs and all repetitions with a hole, where each repetition with a hole is a repeating pair (not necessarily maximal) in which an intervening symbol separates the two matching substrings. In fact, there is a very close relation between these two kinds of repetitions. For example, if a given string \( x \) is as follows:

\[
\begin{array}{cccccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\
\text{s} & \# & \text{s} & \text{s} & \text{l} & \text{s} & \# & \text{l} & \text{s} & \ast & \text{l} & \text{l} & \text{s} \text{s}
\end{array}
\]

Then the maximal-pair (6;3,7) –which represents two overlapping matching substrings– represents four repetitions with a hole: (3;3,7)(3;4,8), (3;5,9), (3;6,10).

The relation between maximal-pairs and repetitions with a hole is presented in the following lemma:

**Lemma 1** Let \( (p; i, j) \) be a maximal-pair in \( x \) and let \( g = j - i - p \).

1. If \( g = 1 \) then \( (p; i, j) \) is a repetition with a hole.
2. If \( g \leq 0 \) and \( j \neq i + 1 \) then \( (p'; i + k, j + k) \) are repetitions with a hole, where \( p' = j - i - 1 \) and \( k = \lfloor g/|g| \rfloor + 1 \).
3. If \( g \leq 0 \) and \( j = i + 1 \) then some repetitions with a hole may be found in \( x[i..i + p - 1] \).
4. If \( g > 1 \) then \( x[i,i + p - 1] \) and \( x[j,j + p - 1]g \) are two matching substrings separated by intervening \( g \) symbols.

In the following we will introduce Gusfield’s algorithm for finding all maximal-pairs in a given string without don’t cares. The basic tool behind Gusfield’s algorithm is the suffix tree. The algorithm starts by constructing the suffix tree for a given string. The algorithm then uses a bottom-up approach (from leaves to root) to report for each internal node the maximal-pairs associated with it. This is accomplished by maintaining the leaf list \( LL(v) \) of each internal node \( v \) as a collection of disjoint subtrees \( LL_\alpha(v) \), where \( \alpha \) is the symbol preceding the suffix associated to a leaf in the subtree rooted by \( v \). Thus, each internal node is attached at most \( |\Sigma| \) subtrees. Reporting the maximal-pairs is accomplished by the cartesian product of a leaf-sublist with all the leaf-sublists of its brothers that correspond to different symbols. The algorithm runs in \( O(n + z) \), where \( z \) is the number of reported maximal-pairs. Gusfield’s algorithm can be easily modified to find all repeating pairs (not necessarily maximal) in \( x \).

The presence of the binary don’t care symbols in \( x \) complicates the construction of the suffix tree. Hence, Gusfield’s algorithm cannot be used directly to solve the problem. The dynamic programming seems to be an obvious solution. The cost of this method is quadratic. For example, if \( x = \text{ss} \# \ast \text{l l s s} \ast \text{l l l s} \). Then using dynamic programming, the following maximal-pairs can be found: (1;1,2), (1;1,6), (6;1,8), (1;1,9), (1;1,13), (2;2,6), (2;2,8), (1;2,13), (2;4,5), (1;4,6), ..., (1;11,12). Note that only (6;1,8), (1;4,6), (1;6,8), (1;9,11) and (1;4,6) are the only repetitions with a hole in \( x \) (see Table 1).

In the next section, we will explain how the suffix tree can be used to speed up the dynamic programming calculations. Independently of the size of the alphabet, our algorithm works for any string that have occurrences of finite number of binary don’t cares.
Table 1: Using dynamic programming to find all repetitions. The bold values represent the lengths of all maximal-pairs

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
</tr>
<tr>
<td>3</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
</tr>
<tr>
<td>4</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
</tr>
<tr>
<td>5</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
</tr>
<tr>
<td>6</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
</tr>
<tr>
<td>7</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
</tr>
<tr>
<td>8</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
</tr>
<tr>
<td>9</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
</tr>
<tr>
<td>10</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
</tr>
<tr>
<td>11</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
</tr>
<tr>
<td>12</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>s</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
<td>l</td>
</tr>
</tbody>
</table>

4 ALGORITHM

Given a string $x$ over $\Sigma \cup \{*, \#\}$, we construct two new strings $x_s$ and $x_l$. Where string $x_s$ (respectively, $x_l$) is obtained by substituting each $*$ by $s$ and $\#$ by $l$ (respectively, each $*$ by $l$ and $\#$ by $s$). The idea is to construct two strings both over $\Sigma$ each is a complement of the other in a sense that for each binary don’t care symbol in the original string each of the two constructed strings contains one of the two possible matching symbols. Note that the suffix trees of the two constructed strings can be built in linear time.

Given $x_s$ and $x_l$, each maximal-pair $(p; i, j)$ in $x$ can be considered as the concatenation of $m$ right-maximal repeating pairs:

$$(p_1; i, j), (p_2; i+p_1, j+p_1), \ldots, (p_m; i+\sum_{k=0}^{m-1} p_k, j+\sum_{k=1}^{m-1} p_k),$$

where

1. the starting-pair $(p_1; i, j)$ is a maximal-pair (i.e. left- and right-maximal) in either $x_s$ or $x_l$,
2. the collection of these right-maximal repeating pairs is distributed between $x_s$ and $x_l$ i.e. one right-maximal repeating pair is in $x_s$ and the following repeating pair is in $x_l$,
3. $p = \sum_{k=1}^{m} p_k$.

The above states the main idea of our algorithm. The algorithm iterates twice. In the first iteration, all maximal-pairs in $x$ whose starting-pairs are in $x_s$ are calculated. In the second iteration, all maximal-pairs in $x$ whose starting-pairs are in $x_l$ are calculated.

Recall that the starting-pair needs to be maximal. Thus, each iteration starts by calculating all maximal-pairs using the suffix tree (as in Gusfield). Then, each maximal-pair is extended to the right by a sequence of right-maximal pairs using a series of jumps from one suffix tree to another. In each attempt of jump we calculate the depth of the lowest common ancestor of two nodes. For example, if the starting-pair $(p_1; i, j)$ is a maximal-pair in $x_l$, then $p_2$ is equal to the depth of lowest common ancestor of the two leaves $i + p_1$ and $j + p_1$ in $T(x_l)$.

Similarly, $p_3$ is the depth of the lowest common ancestor of leaves $i + p_1 + p_2$ and $j + p_1 + p_2$ in $T(x_s)$ and so on.

For example, if $x = \ldots l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s l l l s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s s l l l s$. Then the collection of all maximal-pairs in $x$ is distributed between $x_s$ and $x_l$.

5 RUNNING TIME

In this section, we analyze the running time of All-Repetitions algorithm. Recall that, for constant size alphabet, a suffix tree can be built in $O(n)$-time. Thus, creating both $T(x_s)$ and $T(x_l)$ costs $O(n)$-time. Creating the leaf-lists of all leaves costs $O(n)$-time. At every internal node, the algorithm reports the repetitions associated with this node and constructs the leaf-sublists by concatenating the leaf-sublists of the children of this node. The total cost for creating the leaf-lists over all internal node is $O(n)$-time.

For each possible starting-pair the algorithm performs series of jumps form one tree to another. Each jump costs constant time which is the cost of the lowest common ancestor (LCA) query (Schieber and Vishkin, 1988). In the following we will estimate an upper bound for the number of jumps performed by the algorithm.

Observe that, we jump from $T(x_s)$ to $T(x_l)$ to extend the current repeating pair to the right by a right-maximal repeating pair in $x_l$. This only possible if and only if the first two symbols of both two copies of this new right-maximal repeating pair are either $*$ and $l$ or $\#$ and $s$.
Algorithm All-Repetitions($x$)
Input: A string $x[i..n]$ over $\Sigma \cup \{*, \#\}$
Output: All maximal-pairs and all repetitions with a hole in $x$
1. for $i = 1$ to $n$
2.  if $x[i] = \text{"*"}$
3.    then $x_s[i] = \text{"s"}$
4.  else if $x[i] = \text{"\#"}$
5.    then $x_s[i] = \text{"-s"}$
6.  else $x_s[i] = x[i]$
7. for $i = 1$ to $n$
8.  if $x[i] = \text{"*"}$
9.    then $x_t[i] = \text{"t"}$
10.  else if $x[i] = \text{"\#"}$
11.    then $x_t[i] = \text{"-t"}$
12.  else $x_t[i] = x[i]$
13. Build the suffix trees $T(x_s)$ and $T(x_t)$
14. Find-Maximal-Pairs($x, T(x_s)$)
15. Find-Maximal-Pairs($x, T(x_t)$)

Figure 6: All-Repetitions algorithm

Algorithm Jump&Report($x, i, j, c, d$)
1. $\text{length} \leftarrow d$
2. while $x[i + \text{length}] = x[j + \text{length}]$
3.   if $c = \text{"s"}$ then $c \leftarrow \text{"t"}$
4.     else $c \leftarrow \text{"s"}$
5.     $v \leftarrow T(x_c).\text{LCA}(i + \text{length}, j + \text{length})$
6.     $\text{length} \leftarrow \text{length} + \text{depth}(v)$
7. Report(length; $i, j$)

Algorithm Find-Maximal-Pairs($x, T(x_c)$)
1. for each leaf node $u \in T(x_c)$
2.   if $u$ represents the $i$th suffix of $x_c$
3.     then $LL_{i-1}(u) \leftarrow \{i\}$
4.     for each $\alpha \in \Sigma \cup \{*, \#\}$ and $\alpha \neq x[i-1]$
5.         $LL_{\alpha}(u) \leftarrow \emptyset$
6.     for each internal node $u \in (x_c)$ in bottom-up (depth-first) manner
7.         $u_1, u_2 \leftarrow$ the left and the right children of $u$
8.     for each $(i \in LL_{\alpha_1}(u_1)$ and $j \in LL_{\alpha_2}(u_2))$
9.         where $\alpha_1 \neq \alpha_2$
10.        if $(x[i] + \text{depth}(u_1) = x[j] + \text{depth}(u_2))$
11.          then Jump&Report($x, i, j, c, d$)
12.          else Report(depth($u$); $i, j$)
13.         for each $\alpha \in \Sigma \cup \{*, \#\}$
14.             $LL_{\alpha}(u) \leftarrow LL_{\alpha}(u_1) \cup LL_{\alpha}(u_2)$

Figure 7: Jump&Report and Find-Maximal-Pairs subroutines
a right-maximal repeating pair in $x_s$, where the first two symbols of both copies of this repeating pair are either $*$ and $s$ or $\#$ and $-s$. Thus, the total number of jumps is $O(d(n - d))$, where $d$ is the total number of don’t cares in $x$. Summing the above gives that the total running time is as follows:

**Theorem 1** Given string $x[1..n] \in \{\Sigma \cup \{*,\#\}\}^*$, algorithm All-Repetitions reports all maximal-pairs and all repetitions with a hole in $x$ in space $O(n)$ and time $O(n^2 + d(n - d))$, where $z$ is the output size and $d$ is the total number of binary don’t cares.

Clearly, the algorithm might have a quadratic running time if the input string has $n/2$ binary don’t care symbols. For example, finding all repetitions in string $x = \{sl\}^{n/4} s^{n/2}$ will cost $O(n^2)$-time. This is asymptotically equal to the running time of the dynamic programming. In practice, we expect our algorithm to have a better performance. Table 2 shows the values in the dynamic programming matrix that are calculated using All-Repetitions algorithm to compute all maximal-pairs and all repetitions with a hole in string $x = s s \# l l * l s * -l l l s$. Note that, in addition to the 22 reported repetitions, only 4 intermediate values have been calculated by our algorithm.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
<th>13</th>
<th>14</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
<td>*</td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>*</td>
</tr>
</tbody>
</table>

### 6 CONCLUSIONS

In this paper we have presented an algorithm that enables extraction of melodic patterns from abstract strings of symbols; this abstract representation allows partial overlapping between the various abstract symbolic classes. As a special case, we have applied the proposed algorithm on the commonly used “step-leap” interval representation.

In terms of melodic representation, it is suggested that a more refined representation that comprises of a larger number of abstract interval classes (e.g. unison, step, small leap, medium leap, large leap) may actually enable the extraction of better melodic patterns from the standpoint of musical analysis or, even, music perception. Additionally, the use of rhythmic ‘contour’, in terms of rhythmic abstract classes (e.g. equal, slightly larger, larger, much larger), may improve results further. Such representations have yet to be studied, implemented and tested.

The proposed algorithm requires extensive testing on pattern extraction tasks, and its performance has yet to be compared with other similar algorithms. This study, however, has presented a novel problem in terms of melodic representation and pattern extraction, and has attempted to provide an efficient solution to it that can be used for further testing and evaluation.

### ACKNOWLEDGEMENTS

Maxime Crochemore is partially supported by CNRS, Wellcome Foundation, and Nato grants. Costas S. Iliopoulos is partially supported by a Marie Curie fellowship, Wellcome Foundation, Nato and Royal Society grants. Manal Mohamed is supported by an EPSRC studentship. Marie-France Sagot is partially supported by French Programme BioInformatique Inter EPST, Wellcome Foundation, Royal Society and Nato grants.

### REFERENCES


