A PATTERN EXTRACTION ALGORITHM FOR ABSTRACT MELODIC REPRESENTATIONS THAT ALLOW PARTIAL OVERLAPPING OF INTERVALLIC CATEGORIES

Emilios Cambouropoulos[1], **Maxime Crochemore**[2,3], **Costas Iliopoulos**[3], **Manal Mohamed**[3], **Marie-France Sagot**[4]

[1] Department of Music Studies, University of Thessaloniki, 540006, Thessaloniki, Greece
emilios@mus.auth.gr
[2] Institut Gaspard-Monge, University of Marne-la-Vallée, 77454 Marne-la-Vallée CEDEX 2, France
maxime.crochemore@univ-mlv.fr
[3] Department of Computer Science, King's College London, London WC2R 2LS, England
{mac,csi,manal}@dcs.kcl.ac.uk
[4] INRIA Rhône-Alpes, Université Claude Bernard, 43 Bd du 11 novembre 1918, 69622 Villeurbanne cedex, France
Marie-France.Sagot@inria.fr

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* ($\pm s$), a *leap* ($\pm l$) 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, δ-, γ- and $\{\delta, \gamma\}$-approximate. For example, in δ-approximate matching, equal-length strings consisting of integers match if each corresponding integer differs by not more than δ – 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

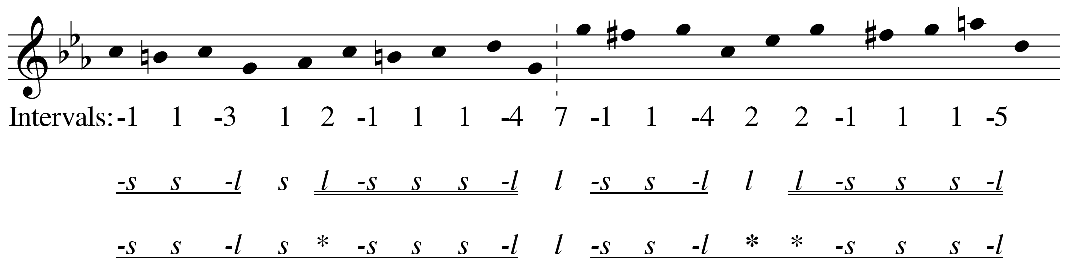

Intervals: -1 1 -3 1 2 -1 1 1 -4 7 -1 1 -4 2 2 -1 1 1 -5
-s s -l s l -s s s -l l -s s -l l l -s s s -l
-s s -l s * -s s s -l l -s s -l * * -s s s -l

Figure 1: Melodic pattern-matching example (the pitches of this example are taken from Bach's Well-Tempered Clavier, Book I, Fugue in F# major)

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 $a = 2$ interval as a member of leap can never be matched to a 'similar' $a = 1$ interval which is a step), i.e, a small leap can never be considered as 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 Cambouropoulos. 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) presented 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.

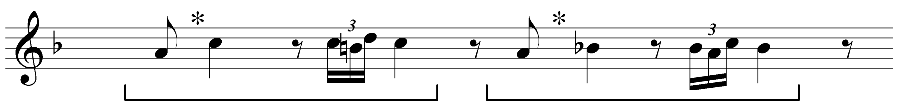
The paper is organized as follows: in Section 2, we state the preliminaries used throughout the paper. In Section 3, we define all approximate repetitions problem and describe in general how to find them. In Section 4, we detail our algorithm. Finally, in Section 5, we analyze the running time of the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

2 PRELIMINARIES

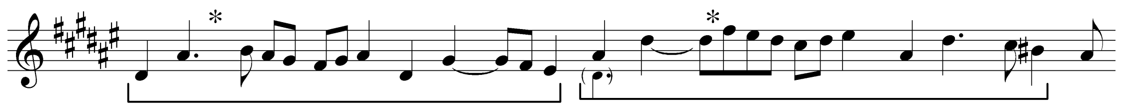
Throughout the paper, $x = x[1..n]$ denotes a *string* of *length* n over $\Sigma \cup \{*, \#\}$, where $\Sigma = \{s, -s, l, -l, u\}$. The symbols '$*$' and '$\#$' are called *binary don't care* symbols. Each binary don't care symbol *matches* itself and two different symbols, that is, $* = *, * = s, * = l, \# = \#, \# = -s$ 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 $x = uv$ 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 $1 \le j \le |y|$.

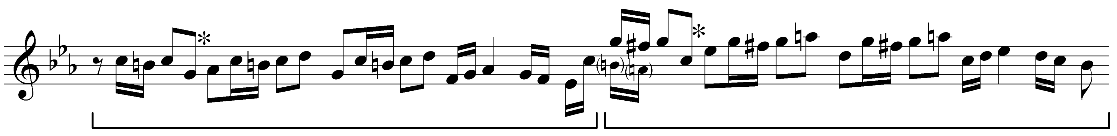
A *repeating pair* in x is represented by $(p; i, j)$ where, $x[i..i{+}p{-}1] = x[j..j{+}p{-}1]$ for some $i \neq j$. The positive integer p is called the *period* of the repeating pair. If $x[i - 1] \neq x[j - 1]$ then $(p; i, j)$ is *left-maximal*. Respectively, Opening melody of Beethoven's Piano 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

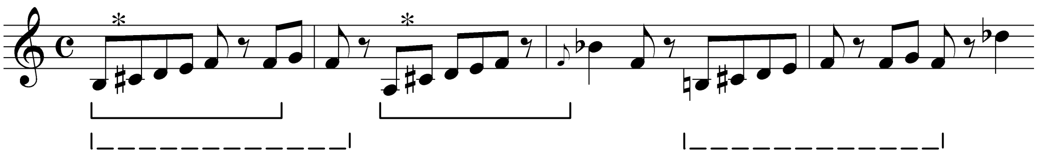


Figure 2: Melodic examples where an $a = 1$ interval (step) and an $a = 2$ (leap) should be matched (these positions are indicated by asterisks in the melodic examples). Brackets indicate extracted melodic repetitions

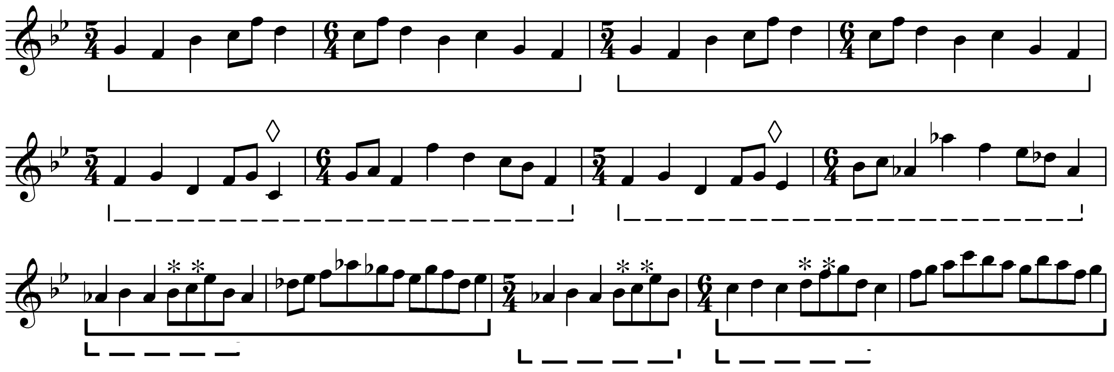


Figure 3: The opening melody of Mussorgsky's, Pictures from an exhibition, Promenade. Extracted maximal repeating pairs and repetitions with a 'hole' are indicated by brackets

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 $\mathcal{T}(x)$ of the string $x\$ = x[1..n]\$$ is a rooted directed tree with exactly n leaves numbered 1 to $n+1$, where $\$ \notin \Sigma$. 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 $\mathcal{T}(x)$ in $\Theta(n)$ time and space, assuming constant size alphabet (see for example (Crochemore and Rytter, 2002) and (Gusfield, 1997)). For any node v, the *path-label* of v is the label of the path from the root of $\mathcal{T}(x)$ 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 *leaf-list* 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 $\mathcal{T}$, the *lowest common ancestor* (LCA) of two node u and v is the deepest node in $\mathcal{T}$ 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 $\mathcal{T}(x)$, 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 sting 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 necessary 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:

$$\begin{array}{cccccccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 \\ x = & s & s & l & s & s & \# & l & s & * & -l & l & s & s \end{array}$$

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 $k = 0..|g|+1$.*
3. *If $g \leq 0$ and $j = i+1$ then some repetitions with a hole may be found in $x[i..j+p-1]$.*
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 sublists $LL_\alpha(v)$, where α is the symbol preceding the suffix associated to a leaf in the subtree rooted by v. Thus, each internal node is attached at most $|\Sigma|$ sublists. 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 necessary 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 $x = s\,s\,\#\,l\,l\,*\!-\!l\,s\,*\!-\!l\,l\,l\,s$. 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), (1;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

		1	2	3	4	5	6	7	8	9	10	11	12	13
		s	s	$\#$	l	l	$*$	$-l$	s	$*$	$-l$	l	l	s
1	s	-	**1**	0	0	0	**1**	0	1	**1**	0	0	0	**1**
2	s		-	0	0	0	1	0	**1**	2	0	0	0	**1**
3	$\#$			-	0	0	0	**2**	0	0	3	0	0	0
4	l				-	1	**1**	0	0	**1**	0	4	**1**	0
5	l					-	**2**	0	0	**1**	0	1	5	0
6	$*$						-	0	**1**	1	0	**1**	**2**	**6**
7	$-l$							-	0	0	**2**	0	0	0
8	s								-	1	0	0	0	1
9	$*$									-	0	**1**	**1**	**1**
10	$-l$										-	0	0	0
11	l											-	**1**	0
12	l												-	0
13	s													-

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 $-s$ (respectively, each $*$ by l and $\#$ by $-l$). The idea is to construct two strings both over Σ 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 new 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 concatenations of m right-maximal repeating pairs:

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

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_s then p_2 is equal to the depth of lowest common ancestor of the two leaves $i + p_1$ and $j + p_1$ in $\mathcal{T}(x_l)$. 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 $\mathcal{T}(x_s)$ and so on.

For example, if $x = s\,s\,\#\,l\,l\,*\,{-l}\,s\,*\,{-l}\,l\,l\,s$ then $x_s = s\,s\,{-s}\,l\,l\,s\,{-l}\,s\,s\,{-l}\,l\,l\,s$ and $x_l = s\,s\,{-l}\,l\,l\,l\,{-l}\,s\,l\,{-l}\,l\,l\,s$. The suffix trees of x_s and x_l are represented in Figures 4 and 5.

Consider node $u_1 \in \mathcal{T}(x_s)$. During the bottom-up traversal of $\mathcal{T}(x_s)$ and at node v_1, the maximal-pair (2;1,8) is calculated. To check whether this starting-pair can be extended to the right, that is whether $x[1 + 2]$ matches $x[8 + 2]$. Since they match, the algorithm jumps to $\mathcal{T}(x_l)$ and calculates the lowest common ancestor of leaves 1+2 and 8+2. The lowest common ancestor of these two leaves is v_2. Since $depth(v_2) = 3$, the current repeating pair is extended to the right by the right-maximal repeating pair (3;3,10). Since $x[3 + 3]$ matches $x[10 + 3]$, the algorithm jumps back to $\mathcal{T}(x_s)$ calculating the lowest common ancestor of the two leaves 3+3 and 10 +3, that is v_3. Since $depth(v_3) = 1$, the current repeating pair is extended further to the right by the right-maximal repeating pair (1;6,13). Because $x[6 + 1]$ does not match $x[13 + 1]$, no more jumps are possible. Then, the algorithm reports (6;1,8) as a maximal-pair in x. Moreover, since $g = 8 - 1 - 6 = 1$, then (6;1,8) is a repetition with a hole in x (Lemma 1).

The details of the algorithm are presented in Figures 6 and 7. For simplicity, algorithm *Find-Maximal-Pairs* assumes that both $\mathcal{T}(x_s)$ and $\mathcal{T}(x_l)$ are binary suffix trees. This is always a valid assumption since that any suffix tree can be transformed into binary one in $O(n)$ time. The function *Report*$(p; i, j)$ reports the given maximal-pair and checks according to Lemma 1 for all repetitions with a hole. Note that if $g = j - i - p \leq 0$ and $j = i + 1$ then *All-Pairs* is used to find all pairs (not necessary maximal) in $x[i..j + p - 1]$. Algorithm *All-Pairs* is a simple modification of *All-Maximal-Pairs*.

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 $\mathcal{T}(x_s)$ and $\mathcal{T}(x_l)$ 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 $\mathcal{T}(x_s)$ to $\mathcal{T}(x_l)$ 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 $-l$. Similarly, we jump back from $\mathcal{T}(x_l)$ to $\mathcal{T}(x_s)$ if and only if the current repeating pair can be extended to the right by

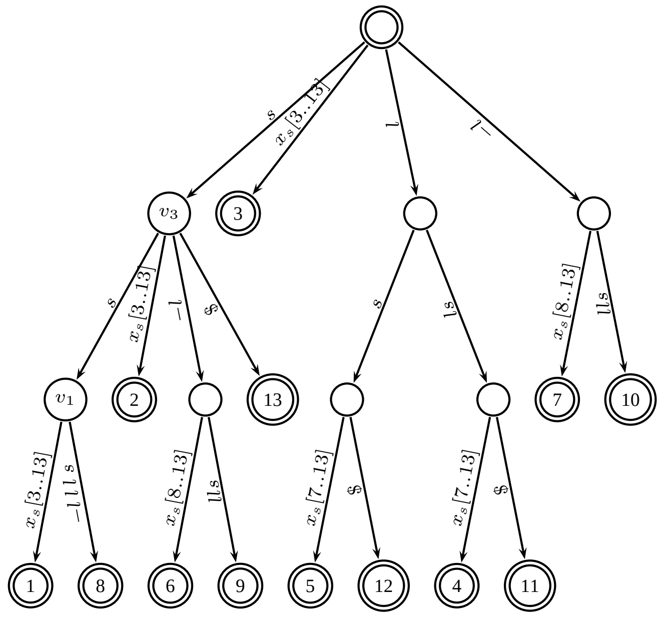


Figure 4: The suffix tree of $x_s\$$

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 $x[i] =$ ‘$*$’
3.            then $x_s[i] =$ ‘$s$’
4.            else if $x[i] =$ ‘$\#$’
5.                    then $x_s[i] =$ ‘$-s$’
6.            else $x_s[i] = x[i]$
7.  for $i = 1$ to $n$
8.          if $x[i] =$ ‘$*$’
9.            then $x_l[i] =$ ‘$l$’
10.           else if $x[i] =$ ‘$\#$’
11.                   then $x_l[i] =$ ‘$-l$’
12.           else $x_l[i] = x[i]$
13. Build the suffix trees $\mathcal{T}(x_s)$ and $\mathcal{T}(x_l)$
14. Find-Maximal-Pairs($x, \mathcal{T}(x_s)$)
15. Find-Maximal-Pairs($x, \mathcal{T}(x_l)$)
```

Figure 6: *All-Repetitions* algorithm

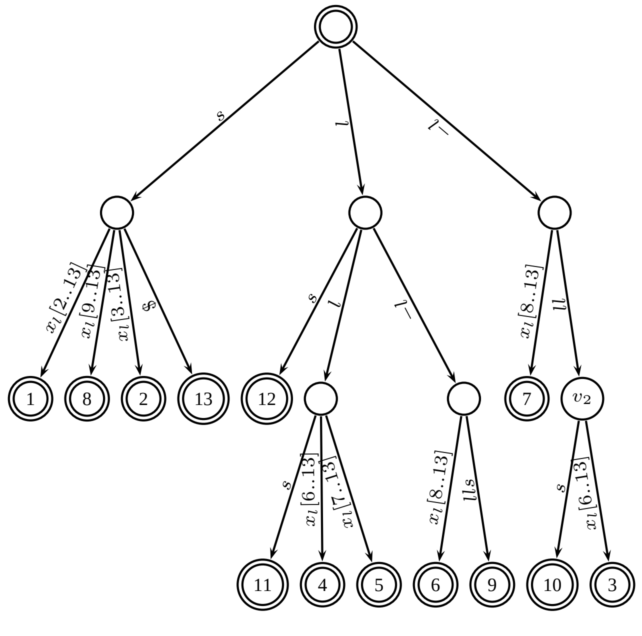


Figure 5: The suffix tree of $x_l\$$

Algorithm *Jump&Report*(x, i, j, c, d)

```
1.  $length \leftarrow d$
2.  while $x[i + length] = x[j + length]$
3.          if $c$ = ‘$s$’ then $c \leftarrow$‘$l$’
4.            else $c \leftarrow$‘$s$’
5.          $v \leftarrow \mathcal{T}(x_c).LCA(i + length, j + length)$
6.          $length \leftarrow length + depth(v)$
7.  Report$(length; i, j)$
```

Algorithm *Find-Maximal-Pairs*$(x, \mathcal{T}(x_c))$

```
1.  for each leaf node $u \in \mathcal{T}(x_c)$
2.          if $u$ represents the $i$th suffix of $x_c$
3.            then $LL_{x[i-1]}(u) \leftarrow \{i\}$
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 $j \in LL_{\alpha_2}(u_2)$)
            where $\alpha_1 \neq \alpha_2$
9.                  if ($x[i+depth(u)] = x[j+depth(u)]$)
10.                   then Jump&Report(x,i,j,c,depth(u))
11.                   else Report$(depth(u); i, j)$
12.         for each $\alpha \in \Sigma \cup \{*, \#\}$
13.                 $LL_\alpha(u) \leftarrow LL_\alpha(u_1) \cup LL_\alpha(u_2)$
```

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 $O(n + z + d(n-d))$, 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 $x = \{sl\}^{n/4} *^{n/2}$ 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 2: The values calculated and reported by *All-Repetitions* algorithm. The bold values represent the lengths of the reported maximal-pairs

		1	2	3	4	5	6	7	8	9	10	11	12	13
		s	s	$\#$	l	l	$*$	$-l$	s	$*$	$-l$	l	l	s
1	s	-	**1**				**1**			**1**				**1**
2	s		-				1		**1**	2				**1**
3	$\#$			-				**2**						
4	l				-	1	**1**			**1**			**1**	
5	l					-	**2**			**1**		1	5	
6	$*$						-		**1**			**1**	**2**	**6**
7	$-l$							-			**2**			
8	s								-	**1**				**1**
9	$*$									-		**1**	**1**	**1**
10	$-l$										-			
11	l											-	**1**	
12	l												-	
13	s													-

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

A. Amir, E. Porat, and M. Lewenstein. Approximate subset matching with don't cares. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 305–306. Society for Industrial and Applied Mathematics, 2001. ISBN 0-89871-490-7.

A. Apostolico and F. P. Preparata. Optimal off-line detection of repetitions in a string. *Theoret. Comput. Sci.*, 22: 297–315, 1983.

E. Cambouropoulos. Musical parallelism and melodic segmentation: A computational approach. music perception. *(forthcoming)*.

E. Cambouropoulos, M. Crochemore, C. Iliopoulos, L. Mouchard, and Y. Pinzon. Algorithms for computing approximate repetitions in musical sequences. *J. Computer Mathematics*, 79(11):1135–1148, 2002.

R. Clifford and C. Iliopoulos. Approximate string matching for music analusis. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 8(9), 2004.

T. Crawford, C. Iliopoulos, and R. Raman. String matching techniques for musical similarity and melodic recognition. *Computing in Musicology*, 11:73–100, 1998.

M. Crochemore and W. Rytter. *Jewels of Stringology*. World Scientific, 2002.

M. Fischer and M. Paterson. String matching and other products. In R. Karp, editor, *Complexity of Computation SIAM-AMS Proceedings*, pages 113–125, 1974.

A. Ghias, J. Logan, D. Chamberlin, and B. Smith. Query by humming: Musical information retrieval in an audio database. In *ACM Multimedia*, pages 231–236, 1995.

D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology*. Cambridge University Press, 1997.

M. Hawley. *Structure of Sound*. PhD thesis, MIT, 1993.

C. Iliopoulos, T. Lecroq, L. Mouchard, and Y. Pinzon. Computing approximate repetitions in musical sequences. In *Proc. of Prague Stringology Club Workshop (PSCW'00)*, pages 49–59, 2000.

C. S. Iliopoulos, M. Mohamed, L. Mouchard, K. Perdikuri, W. F. Smyth, and A. Tsakalidis. String regularities with don't cares. *Nordic Journal of Computing*, 10(1):40–51, 2003.

K. Lemström and P. Laine. Musical information retrieval using musical parameters. In *Proc. International Computer Music Conference (ICMC '98)*, pages 341–348, 1998.

B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. *SIAM Journal on Computing*, 17(6):1253–1262, 1988.