Available online at www.sciencedirect.com
ScienceDirect
Information Sciences 177 (2007) 1954–1962
INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL
www.elsevier.com/locate/ins

All maximal-pairs in step–leap representation of melodic sequence ☆

Emilios Cambouropoulos [a], Maxime Crochemore [b,c,1], Costas S. Iliopoulos [c,2], Manal Mohamed [c,*], Marie-France Sagot [d,3]

[a] *Department of Music Studies, University of Thessaloniki, 540006 Thessaloniki, Greece*
[b] *Institut Gaspard-Monge, University of Marne-la-Vallée, 77454 Marne-la-Vallée Cedex 2, France*
[c] *Department of Computer Science, King's College London, London WC2R 2LS, England, United Kingdom*
[d] *Inria Rhône-Alpes, Laboratoire de Biométrie et Biologie Évolutive, Université Claude Bernard, 69622 Villeurbanne Cedex, France*

Received 6 April 2005; received in revised form 23 November 2006; accepted 25 November 2006

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 "binary 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 used to represent the possible overlapping between the various abstract categories e.g. $* = s$, $* = l$ and $\# = -s$, $\# = -l$. We propose an $O(n + d(n - d) + z)$-time algorithm for computing all maximal-pairs in a given sequence $x = x[1..n]$, where x contains d occurrences of binary don't cares and z is the number of reported maximal-pairs.

Keywords: String; Don't care; Maximal-pair; Suffix tree; Lowest common ancestor; Combinatorial algorithm; Music retrieval; Repetitions

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

☆ A preliminarily version of this paper appeared as [7].

* Corresponding author. Tel.: +44 2078482492.
E-mail addresses: emilios@mus.auth.gr (E. Cambouropoulos), maxime.crochemore@univ-mlv.fr, mac@dcs.kcl.ac.uk (M. Crochemore), csi@dcs.kcl.ac.uk (C.S. Iliopoulos), manal@dcs.kcl.ac.uk (M. Mohamed), Marie-France.Sagot@inria.fr (M.-F. Sagot).

[1] Partially supported by CNRS, Wellcome Foundation and NATO grants.
[2] Partially supported by a Marie Curie fellowship, Wellcome Foundation, NATO and Royal Society grants.
[3] Partially supported by French Programme BioInformatique Inter EPST, Wellcome Foundation, Royal Society and NATO grants.

0020-0255/$ - see front matter © 2006 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2006.11.012

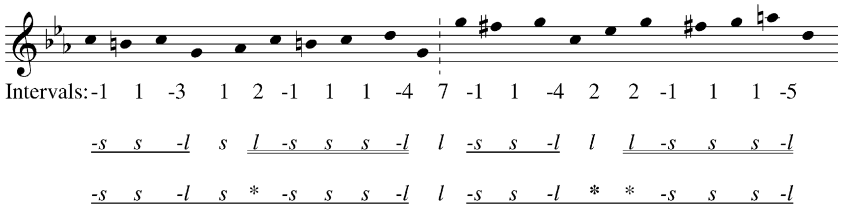


Fig. 1. Melodic pattern-matching example (the pitches of this example are taken from Bach's Well-Tempered Clavier, Book I, Fugue in F# major).

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 [11]. This requires developing new approximation measures that meet musicians' needs.

One commonly used representation for music is the numeric representation MIDI.[4] For such a representation, different approximation measures have been developed, such as, δ-, γ- and $\{\delta,\gamma\}$-approximate. For example, in the δ-approximate measure, 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 [1,8,9,18]. These algorithms are based on approximate pattern matching techniques. For an overview refer to [10].

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 [17] proposed representing the musical signal as a sequence of pitch intervals. In order to allow tolerance in interval matching, Ghias et al. [15] used the reduced interval alphabet of the "melodic contour" representation. Lemström and Laine [20] proposed classifying the intervals into seven partially overlapping classes: small, medium and large, up- or down-wards, and prime.

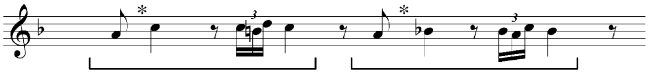
In this paper, we propose an alternative method to using approximate pattern matching techniques for finding approximate repetitions in a musical string. Our approach is based on using exact pattern matching techniques to extract repetitions from an abstract level of a musical sequence. As an abstract 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 *down-wards 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 $a \geqslant 2$ is a leap (l); the direction of intervals is preserved – see second string of symbols in Fig. 1.

In the second string of symbols in Fig. 1, two repeated substrings are found: $-ss-l$ and $l-sss-l$ 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 Figs. 2 and 3 some melodic examples are presented.

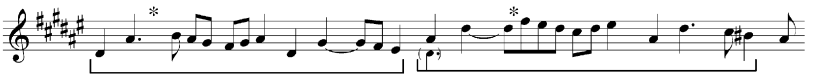
The problem in the step–leap representation is that the abstract interval classes (u, s, l) have sharp boundaries and no diatonic pitch interval instance may belong to more than one class. In other words, borderline members can never be matched to other 'similar' members of other classes (e.g. an $a = 2$ 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

[4] Musical Instrument Digital Interface.

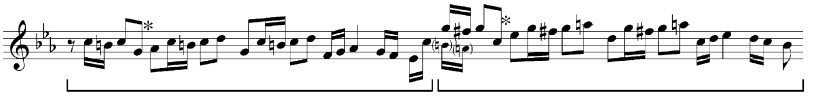
Opening melody of Beethoven's Piano Sonata Op.10, No.2.



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.

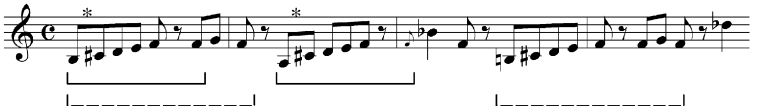


Fig. 2. Melodic examples where an $a = 1$ interval (step) and an $a = 2$ (leap) should be matched (these positions are indicated by asterisks in the melodic examples). Brackets indicate extracted melodic maximal-pairs.

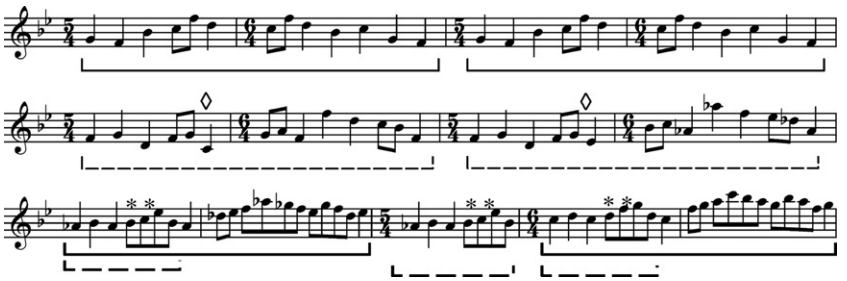


Fig. 3. The opening melody of Mussorgsky's, pictures from an exhibition, Promenade. Extracted maximal-pairs are indicated by brackets.

(this is also suggested in [20]). 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 Fig. 1). Similarly, another binary 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 [20].

For strings with don't cares, several string matching algorithms have been proposed (see [3,2,14]). Iliopoulos et al. [19] presented algorithms for computing typical regularities in strings with don't cares. Recently, approximate matching in the presence of don't cares has been studied [4]. 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 a special kind of repetition that we refer to as maximal-pair. The proposed algorithm uses $O(n + d(n - d) + z)$ time, where z is the number of reported maximal-pairs.

The paper is organized as follows: in Section 2, we state the preliminaries used throughout the paper. In Section 3, we define all maximal-pairs problem and describe in general how to calculate them. In Section 4, we give details of our algorithm. In Section 5, we analyse the running time of the algorithm. Conclusion is drawn in Section 6.

2. Preliminaries

Throughout the paper, $x = x[1..n]$ denotes a *string* over $\Sigma \cup \{*, \#\}$, where $\Sigma = \{s, -s, l, -l, u\}$. The *length* of x is denoted by $|x|$. The symbols '$*$' and '#' are called "*binary don't care*" symbols. Each binary don't care *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 ith 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 $1 \leqslant j \leqslant |y|$.

A *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 pair. If $x[i-1] \neq x[j-1]$ then $(p; i, j)$ is *left-maximal*. Respectively, 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*.

Here, we present a method for finding all maximal-pairs in a given string x, where x may have occurrences of binary don't cares. This 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 [12] or [16]. However, for the sake of completeness, we will briefly review the notion.

Definition 1 (*Suffix tree*). A suffix tree $\mathscr{T}(x)$ of the string $x\$ = x[1..n]\$$ is a rooted directed tree with exactly $n + 1$ leaves numbered 1 to $n + 1$, where $\$ \notin \Sigma$. Each internal node, has at least two children and each edge is labelled with a non-empty substring of x. No two edges descending of 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 the ith suffix $x[i..n]$ of x, with $n + 1$ denoting the empty suffix.

Several algorithms construct the suffix tree $\mathscr{T}(x)$ in $\Theta(n)$ time and space – see for example [21,23,24] for constant size alphabet, and [13] for general alphabet. For any node v, the *path-label* of v is the concatenation of the edge-labels on the path from the root to v; it is denoted by *label*(v). The *depth* of v is the length of the path-label of v; 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 relies on efficient calculation of the *lowest common ancestor* [5,6,22]. For a given rooted tree $\mathscr{T}$, the *lowest common ancestor* (LCA) of two nodes u and v is the deepest node in $\mathscr{T}$ that is an 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 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 $\mathscr{T}(x)$, 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 $w = \mathrm{LCA}(u, v)$ is the root of the minimum size subtree that 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 capability to find a longest common prefix is an important primitive in many string problems.

3. All maximal-pairs problem

In this section, we start by introducing 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 $\mathrm{LL}_\alpha(v)$, 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($n + z$) time, where z is the number of reported maximal-pairs.

Here, we want to find all maximal-pairs in a given string x, where x over $\Sigma \cup \{*, \#\}$. The presence of the binary don't care symbols complicates the extraction of the maximal-pairs. Useful data structure such as suffix

Table 1
Using dynamic programming to find all maximal-pairs

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

The bold values represent the periods of the reported maximal-pairs.

tree cannot be directly used. However, dynamic programming seems to be an obvious method for solving such problems. 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: $(1;1,2)$, $(1;1,6)$, $(6;1,8)$, $(1;1,9)$, $(1;1,13)$, $(2;2,6)$, $(1;2,8)$, $(1;2,13)$, $(2;4,5)$, $(1;4,6),\ldots,(1;11,12)$ (see Table 1).

In the next section, we will explain how we can still use the suffix tree to speed up the dynamic programming calculations. Independently of the size of the alphabet, our algorithm works for any strings that have occurrences of a finite number of binary don't cares.

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 replacing each $*$ by s and # by $-s$ (respectively, each $*$ by l and # by $-l$). The idea is to construct two new strings both over Σ where each is a complement of the other in a sense that for each binary don't care in the original string x each of the two new constructed strings contains one of the two 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 pairs:

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

where

1. the *starting-pair* $(p_1;i,j)$ is maximal (i.e. left- and right-maximal) in either x_s or x_l,
2. the collection of these right-maximal pairs is distributed between x_s and x_l i.e. one right-maximal pair is in x_s and the following 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 the lowest common ancestor of the two leaves $i + p_1$ and $j + p_1$ in $\mathscr{T}(x_l)$. 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 $\mathscr{T}(x_s)$ 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 Figs. 4 and 5.

Consider node $v_1 \in \mathscr{T}(x_s)$. During the bottom-up traversal of $\mathscr{T}(x_s)$ 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, the algorithm have to check whether $x[1+2]$ matches $x[8+2]$. Since they match, the algorithm jumps to $\mathscr{T}(x_l)$ 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 pair is extended to the right by the right-maximal pair $(3;3,10)$. Since $x[3+3]$ matches $x[10+3]$, the algorithm jumps back to $\mathscr{T}(x_s)$ 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 pair is extended further to the right by the right-maximal pair $(1;6,13)$. Because $x[6+1]$ does not match $x[13+1]$, no more jumps are possible. So, the algorithm reports $(6;1,8)$ as a maximal-pair in x.

The details of the algorithm are presented in Figs. 6 and 7. For simplicity, algorithm *Find-Maximal-Pairs* assumes that both $\mathscr{T}(x_s)$ and $\mathscr{T}(x_l)$ are binary suffix trees. This is always a valid assumption since that any suffix tree can be transformed into a binary one in $O(n)$ time.

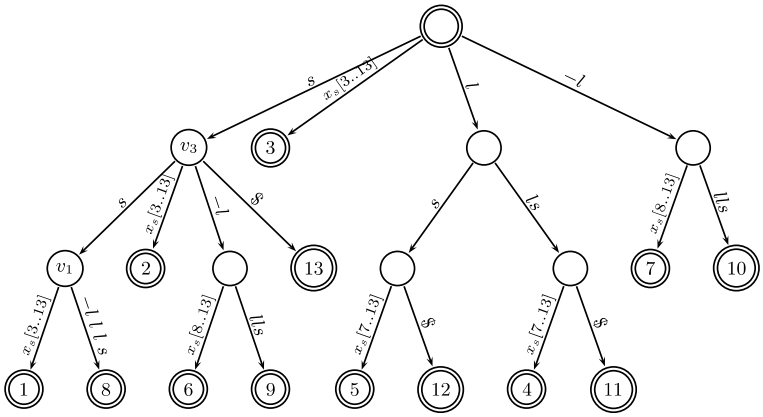


Fig. 4. The suffix tree of $x_s = s\ s\ -s\ l\ l\ s\ -l\ s\ s\ -l\ l\ l\ s\ \$$.

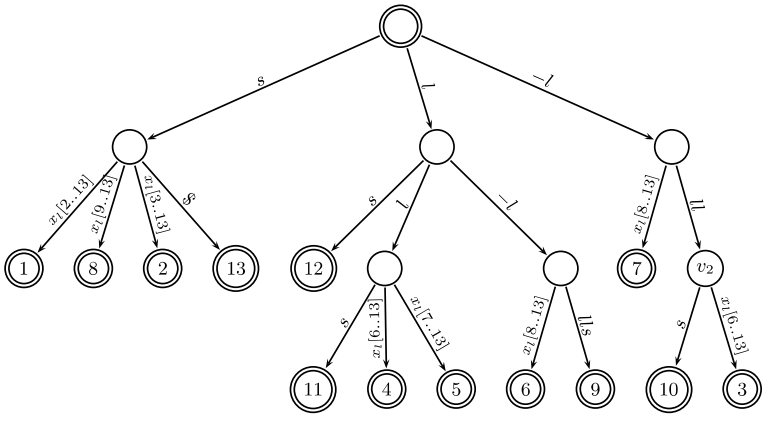


Fig. 5. The suffix tree of $x_l = s\ s\ -l\ l\ l\ l\ -l\ s\ l\ -l\ l\ l\ s\ \$$.

```
Algorithm All-Maximal-Pairs(x)
Input: A string x of length n
Output: All Maximal-Pairs
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 T(x_s) and T(x_l)
14. Find-Maximal-Pairs(x, T(x_s))
15. Find-Maximal-Pairs(x, T(x_l))
```

Fig. 6. *All-Maximal-Pairs* algorithm.

```
Algorithm Jump&Report(x, i, j, c, d)
1.  length ←d
2.  while x[i + length] = x[j + length]
3.          if c = 's' then c ←'l'
4.            else  c ←'s'
5.          v ←T(x_c).LCA(i + length, j + length)
6.          length ←length + depth(v)
7.  Report(length; i, j)

Algorithm Find-Maximal-Pairs(x, T(x_c))
1.  for each leaf node u ∈ T(x_c)
2.          if u represents the ith suffix of x_c
3.             then LL_{x[i−1]}(u) ←{i}
4.          for each α ∈ Σ ∪ {*, #} and α ≠ x[i − 1]
5.                  LL_α(u) ←∅
6.  for each internal node u ∈ (x_c) in bottom-up manner
7.          u_1, u_2 ←the left and the right children of u
8.          for each (i ∈ LL_{α_1}(u_1) and j ∈ LL_{α_2}(u_2)) where α_1 ≠ α_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 α ∈ Σ ∪ {*, #}
13.                 LL_α(u) ←LL_α(u_1) ∪ LL_α(u_2)
```

Fig. 7. *Jump&Report* and *Find-Maximal-Pairs* subroutines.

5. Running time

In this section, we analyse the running time of *All-Maximal-Pairs* algorithm. Recall that, for constant size alphabet, a suffix tree can be built in O(n)-time. Thus, building 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 maximal-pairs 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 constant size alphabet.

Table 2
The values calculated and reported by *All-Maximal-Pairs* algorithm

		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*													–

The bold values represent the periods of the reported maximal-pairs.

For each reported maximal-pair the algorithm performs a series of jumps from one tree to another. Each jump costs constant time which is the cost of the lowest common ancestor (LCA) query. In the following we will estimate an upper bound for the number of jumps performed by the algorithm.

Observe that, we jump from $\mathscr{T}(x_s)$ to $\mathscr{T}(x_l)$ to extend the current pair to the right by a right-maximal pair in x_l. This is only possible, if and only if, the first two characters of both two copies of this new right-maximal pair are either ∗ and l or # and $-l$. Similarly, we jump back from $\mathscr{T}(x_l)$ to $\mathscr{T}(x_s)$, if and only if, the current pair can be extended to the right by a right-maximal pair in x_s, where the first two characters of both copies of this pair are either ∗ and s or # and $-s$. Thus, the total number of jumps is $O(d(n-d))$, where d is the number of binary don't cares occurring in x.

Summing the above gives that the total running time as follows:

Theorem 1. *Given string* $x[1..n] \in \{\Sigma \cup \{*, \#\}\}^*$, *algorithm All-Maximal-Pairs calculates all maximal-pairs in* x *in space* $O(n)$ *and time* $O(n + d(n-d) + z)$, *where* d *is the total number of binary don't cares in* x *and* z *is the number reported maximal-pairs.*

Clearly, our algorithm might have a quadratic running time if x has $n/2$ binary don't care symbols. For example, finding all maximal-pairs 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 our algorithm to compute all maximal-pairs in string $x = s\,s\,\#\,l\,l\,*\,-l\,s\,*\,-l\,l\,l\,s$. Note that, in addition to the 22 maximal-pairs, only four intermediate values have been calculated by our algorithm.

6. Conclusion

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.

References

[1] A. Amir, O. Lipsky, E. Porat, J. Umanski, Approximate matching in the L_1 metric, in: Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM'05), 2005, pp. 91–103.

[2] A. Amir, E. Porat, M. Lewenstein, Approximate subset matching with don't cares, in: Proceedings of the 12th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'01), 2001, pp. 305–306.

[3] A. Apostolico, F.P. Preparata, Optimal off-line detection of repetitions in a string, Theoretical Computer Science 22 (1983) 297–315.

[4] Y.J. Pinzon Ardila, M. Christodoulakis, C.S. Iliopoulos, M. Mohamed, Efficient (δ, γ)-pattern-matching with don't cares, in: Proceedings of the 16th Australasian Workshop on Combinatorial Algorithms (AWOCA'05), Ballarat, Australia, 2005, pp. 27–38.

[5] M. Bender, M. Farach-Colton, The LCA problem revisited, in: Proceedings of the 4th Latin American Symposium on Theoretical Informatics (LATIN'00), 2000, pp. 88–94.

[6] O. Berkman, U. Vishkin, Recursive star-tree parallel data-structure, SIAM Journal on Computing 22 (2) (1993) 221–242.

[7] E. Cambouropoulos, M. Crochemore, C.S. Iliopoulos, M. Mohamed, M.-F. Sagot, A pattern extraction algorithm for abstract melodic representations that allow partial overlapping of intervallic categories, in: Proceedings of the 6th International Conference on Music Information Retrieval (ISMIR'05), 2005, pp. 167–174.

[8] E. Cambouropoulos, M. Crochemore, C.S. Iliopoulos, L. Mouchard, Y.J. Pinzon, Algorithms for computing approximate repetitions in musical sequences International, Journal of Computer Mathematics 79 (11) (2002) 1135–1148.

[9] P. Clifford, R. Clifford, C.S. Iliopoulos. Faster algorithms for δ, γ-matching and related problems, in: Proceedings of the 16th Annual Symposium on Combinatorial Pattern Matching (CPM'05), 2005, pp. 68–78.

[10] R. Clifford, C.S. Iliopoulos, Approximate string matching for music analysis, Soft Computing – A Fusion of Foundations. Methodologies and Applications 8 (9) (2004) 597–603.

[11] T. Crawford, C.S. Iliopoulos, R. Raman, String matching techniques for musical similarity and melodic recognition, Computing in Musicology 11 (1998) 73–100.

[12] M. Crochemore, W. Rytter, Jewels of Stringology, World Scientific, 2002.

[13] M. Farach, Optimal suffix tree construction with large alphabets, in: Proceedings of the 38th Annual Symposium on Foundations of Computer Science (FOCS'97), IEEE Computer Society, 1997, pp. 137–143.

[14] M. Fischer, M. Paterson, String matching and other products, in: R.M. Karp (Ed.), Proceedings of the 7th SIAM-AMS Complexity of Computation, 1974, pp. 113–125.

[15] A. Ghias, J. Logan, D. Chamberlin, B.C. Smith, Query by humming musical information retrieval in an audio database, in: Proceedings of ACM Conference on Multimedia, 1995, pp. 231–236.

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

[17] M.J. Hawley, Structure of Sound, Ph.D. thesis, MIT, 1993.

[18] C.S. Iliopoulos, T. Lecroq, L. Mouchard, Y. Pinzon, Computing approximate repetitions in musical sequences, in: Proceedings of Prague Stringology Club Workshop (PSCW'00), 2000, pp. 49–59.

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

[20] K. Lemström, P. Laine, Musical information retrieval using musical parameters, in: Proceedings of the International Computer Music Conference (ICMC'98), 1998, pp. 341–348.

[21] E.M. McCreight, A space-economical suffix tree construction algorithm, Journal ACM 23 (2) (1976) 262–272.

[22] B. Schieber, U. Vishkin, On finding lowest common ancestors: simplifications and parallization, SIAM Journal of Computation 17 (1988) 1253–1262.

[23] E. Ukkonen, Constructing suffix trees on-line in linear time, in: Proceedings of the IFIP 12th World Computer Congress on Algorithms, Software, Architecture – Information Processing'92, vol. 1, North-Holland Publisher, 1992, pp. 484–492.

[24] P. Weiner, Linear pattern matching algorithms, in: V. Ronald (Ed.), in: Proceedings of the 14th Annual Symposium on Switching and Automata Theory, University of Iowa, IEEE Computer Society Press, 1973, pp. 1–11.