International Journal of Foundations of Computer Science
© World Scientific Publishing Company

Identifying Rhythms in Musical Texts

Manolis Christodoulakis
Costas S. Iliopoulos[*]
M. Sohel Rahman[†,‡]
Algorithm Design Group
Department of Computer Science, King's College London
Strand, London WC2R 2LS, England
http://www.dcs.kcl.ac.uk/adg

and

William F. Smyth[§]
Algorithms Research Group
Department of Computing and Software
McMaster University, Canada
http://www.cas.mcmaster.ca/cas/research/algorithms.htm
Department of Computing
Curtin University
Perth, Western Australia

Received (received date)
Revised (revised date)
Communicated by Editor's name

ABSTRACT

A fundamental problem in music is to classify songs according to their rhythm. A rhythm is represented by a sequence of "Quick" (Q) and "Slow" (S) symbols, which correspond to the (relative) duration of notes, such that $S = 2Q$. In this paper, we present an efficient algorithm for locating the maximum-length substring of a music text t that can be covered by a given rhythm r.

Keywords: algorithms, musical sequence, rhythm.

1. Introduction

The subject of musical representation for use in computer application has been studied extensively in computer science literature [3, 2, 6, 11, 15, 13]. Computer assisted music analysis [14, 12] and music information retrieval [7, 10, 9, 8] has

[*]Supported by EPSRC and Royal Society grants.
[†]Supported by the Commonwealth Scholarship Commission in the UK under the Commonwealth Scholarship and Fellowship Plan (CSFP).
[‡]On Leave from Department of CSE, BUET, Dhaka-1000, Bangladesh.
[§]Supported in part by an NSERC grant.

a number of tasks that can be related to fundamental combinatorial problems in computer science and in particular to stringology. A survey of computational tasks arising in music information retrieval can be found in [4]. We, in this paper, are interested in automatic music classification, which is one of the fundamental tasks in the area of computational musicology. Songs need to be classified by one or more of their characteristics, like genre, melody, rhythm, etc. For human beings, the process of identifying those characteristics seems natural. Computerized classification though is hard to achieve, given that there does not exist a complete agreement on the definition of those features.

In this work, we will be concerned with classification of a music text by rhythms. We will define what a rhythm is, and how it can be identified in a musical sequence, a song for example. The musical sequences we will be considering consist of a series of onsets (or events) that correspond to music signals, such as drum beats, guitar picks, horn hits, etc. It is the intervals between those events, that characterizes the song.

In particular, there are two types of intervals in the rhythm of a song: *quick* (Q) and *slow* (S). *Quick* means that the duration between two (not necessarily successive) onsets is q milliseconds, while the *slow* interval is equal to $2q$. For example, the dancing rhythm, cha-cha is given as the sequence $SSQQSSSQQS$ while a foxtrot is given as $SSQQSSQQ$, and a jive is given as $SSQQSQQS$.

The paper is organized as follows. In Section 2, we present the notations that we use throughout the paper, and we define the notion of '*match*' and '*cover*' in musical sequences. In Section 3, we describe in detail our algorithm for finding the longest area in a musical sequence that is covered by a given rhythm. Finally, Section 4 contains our concluding remarks.

2. Definitions

A musical sequence can be thought of as a sequence of occurrences (in time axis) of events. Consider a music signal having 5 musical events occurring at 0th, 50th, 100th, 200th and 240th miliseconds. Then, the sequence $S1 = [0, 50, 100, 200, 240]$ can be regarded as the corresponding sequence representing the music signal under consideration. Alternatively, we can represent the same music signal by stating the duration of the consecutive musical events, instead of stating their start times. In this scheme, $S2 = [50, 50, 100, 40]$ represent the same music signal. It is clear that the two definitions are equivalent as it is evident from Figure 1. We use the latter definition throughout the rest of this paper because the duration of the activity can be directly compared with the Q and S parameters (to be formally defined shortly) that also represent duration. So, formally, a *musical sequence* t is a string $t = t[1]t[2]\dots t[n]$, where $t[i] \in \mathbb{N}^+$, for all $1 \le i \le n$.

A *rhythm* r is a string $r = r[1]r[2]\dots r[m]$, where $r[j] \in \{Q, S\}$, for all $1 \le j \le m$. For example, $r = QSS$. Here Q and S correspond to durations of activities (intervals between the start of consecutive events), such that the length of an interval represented by an S is double the length of an interval represented by Q. However, the exact length of Q or S is not a priori known. One interesting fact is that, in our

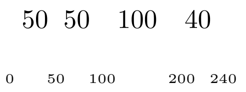

Figure 1: Two equivalent definitions of musical sequence

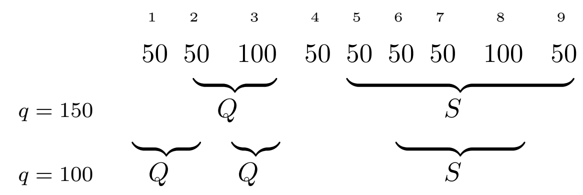

Figure 2: Q- and S-matching in musical sequences.

problem, the alphabet for the musical text and that of the rhythm differs from each other. It should be clear that the alphabet for the musical text is $\Sigma = \{t[i] \mid 1 \leq i \leq n\}$, whereas the alphabet for the rhythm is $\Sigma_r = \{Q, S\}$. As will be seen, this difference in corresponding alphabets, along with the new notion of match and cover (to be defined below), makes our problem combinatorially much more difficult to solve than the traditional string matching problems.

Let Q represent intervals of size $q \in \mathbb{N}^+$ milliseconds, and S represent intervals of size $2q$. Then Q is said to *match* with the substring $t[i..i']$ of the musical sequence t, if and only if

$$q = t[i] + t[i+1] + \ldots + t[i']$$

where $1 \leq i \leq i' \leq n$. If $i = i'$ then the match is said to be *solid*. Similarly, S is said to match with $t[i..i']$, if and only if either of the following is true

- $i = i'$ and $t[i] = 2q$, or
- $i \neq i'$ and there exists $i \leq i_1 < i'$ such that

$$q = t[i] + t[i+1] + \ldots + t[i_1] = t[i_1+1] + t[i_1+2] + \ldots + t[i']$$

As with Q, the match of S is said to be *solid* if $i = i'$. Note that, the notion of match of a Q and an S with the text t is specific to a particular value, namely, q. For the sake of clarity, in what follows, we use the term q-match, when we refer to matching of a Q and/or an S. As it will be seen, this notion of q-match will be extended for the match and cover of rhythms as well. However, if it is clear from the context, we prefer to use the term "match" instead of "q-match".

For example, consider the musical sequence shown in Figure 2. For $q = 150$, Q matches with $t[2..3]$ and S matches with $t[5..9]$. For $q = 100$, Q matches with $t[1..2]$, $t[3]$ etc. and S matches with $t[6..8]$. However, note that for $q = 100$, S does not match with $t[7..9]$ despite the fact that $\sum_{i=7}^{9} t[i] = 2q$.

Consequently, a rhythm $r = r[1] \ldots r[m]$ is said to *q-match* with the substring $t[i..i']$ of the musical sequence t, if and only if there exists an integer $q \in \mathbb{N}^+$, and integers $i_1 < i_2 < \ldots < i_m < i_{m+1}$ such that

1. $i_1 = i$, $i_{m+1} = i' + 1$, and
2. $r[j]$ q-matches $t[i_j..i_{j+1} - 1]$, for all $1 \leq j \leq m$

For instance, the rhythm $r = QSS$, q-matches with $t[1..5]$ as well as with $t[5..8]$, in Figure 3, for $q = 50$. Note the difference in length of the portion of t that q-matched with the r in the above two instances. This means that reporting only the start (or end) position may not convey the complete information. Therefore we report both the start and end positions to denote the *q-occurrences* against the q-matches. Therefore, the q-occurrence list for the above case is $Occ_q = \{(1,5),(5,8)\}$.

Finally, a rhythm r is said to *q-cover* the substring $t[i..i']$ of the musical sequence t, if and only if there exist integers $i_1, i'_1, i_2, i'_2, \ldots, i_k, i'_k$, for some $k \geq 1$, such that

- r q-matches $t[i_\ell..i'_\ell]$, for all $1 \leq \ell \leq k$, and
- $i'_{\ell-1} \geq i_\ell - 1$, for all $2 \leq \ell \leq k$

In our example, Figure 3, the rhythm $r = QSS$ q-covers $t[1..8]$ for $q = 50$.

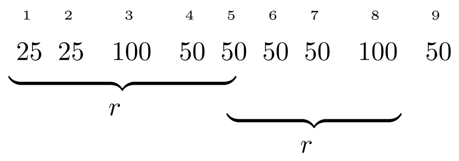

Figure 3: q-matches of $r = QSS$ in t, for $q = 50$.

3. Maximal Coverability Algorithm

In this section, we focus on the *maximal coverability* problem, which is formally defined as follows:

Problem 1 *Given a musical sequence $t = t[1]t[2]\ldots t[n]$, $t[i] \in \mathbb{N}^+$, and a rhythm $r = r[1]r[2]\ldots r[m]$, $r[j] \in \{Q, S\}$, find the longest substring $t[i..i']$ of t that is q-covered by r among all possible values of q.*

Note that the definition above is very general, allowing pathological cases like the following: consider a musical sequence consisting of a single tone repeated every 1ms, $t = 111\ldots1$. Consider also a rhythm r consisting of Q's and S's. Then r will match t in every position i regardless of the value of q, since any Q in r will match with a sequence of q 1's, and any S in r will match with a sequence of $2q$ 1's. To avoid such cases, we introduce the following restriction for the matching of a rhythm r with a substring $t[i..i']$ of t:

Restriction 1 *For each match of r with a substring $t[i..i']$, there must exist at least one S in r whose match in $t[i..i']$ is solid; that is, there exists at least one $1 \leq j \leq m$ such that $r[j] = t[k] = 2q$, $i \leq k \leq i'$, for some value of q.*

As explained before, the value of q is not *a priori* given. Therefore each $\sigma \in \Sigma$ should be considered as a candidate for q, provided of course that $2\sigma \in \Sigma$ (because we need at least one solid S), and for that particular q all the occurrences of the rhythm r must be identified. On the contrary, we can consider each σ to be equal to $S = 2q$. In our algorithm, for the sake of efficiency, we will be using the latter form. Our algorithm works in following main stages:

- *Stage 1*: Find all occurrences of (solid) $S = \sigma$ in t for each possible value of σ.
- *Stage 2*: Transform the areas around all the S's found in Stage 1 into sequences of Q's and S's. A sequence in this stage is identified by $\sigma = S$ as follows: A sequence is said to be a q-sequence, if the solid S is assumed to be of value $2q$, i.e. $\sigma = 2q$.
- *Stage 3*: Find the q-matches of r in corresponding q-sequences from Stage 2.
- *Stage 4*: Find the maximal area q-covered by r for all possible values of q and then report a maximum one.

We next explain each of these stages in detail.

3.1. Stage 1 – Finding all occurrences of S

In this stage, we need to find all occurrences of $S = \sigma$, for the chosen σ, so that we can (in Stage 2) transform the areas around each of those occurrences to sequences of Q's and (possibly) S's. And we have to repeat the above for every possible values of σ. A single scan through the input string suffices to find all occurrences of σ. Since the stage is repeated for every distinct $\sigma \in \Sigma$, overall the algorithm would need $O(|\Sigma|n)$ time on this stage alone.

However, it is easy to speedup this stage, by collectively computing linked lists of the occurrences of all the symbols. This can be done in $O(n)$ time and $O(n+|\Sigma|)$ space in the following manner. Consider vectors $first$, i.e. of size $|\Sigma|$, and $next$, i.e. of size n, such that

- $first[\sigma] = i$ if and only if the leftmost occurrence of the symbol σ appears at position i
- $next[i] = j$ if and only if $t[i] = t[j]$ and for all k, $i < k < j$, $t[k] \neq t[i]$; if no such j exists, then $next[i] = 0$

A single scan through t suffices to compute vectors $first$ and $next$ provided that we can index the alphabet Σ. In order to do that we first compute Σ from t. Let $\Sigma' = \{1, 2, 3, ..., |\Sigma|\}$. We construct a function $f : \Sigma \to \Sigma'$ such that

$$f(\Sigma[i]) = \Sigma'[i], 1 \leq i \leq |\Sigma|. \tag{1}$$

Then we compute the two vectors using the function f to index the alphabet. The details are given in Algorithm 1

3.2. Stage 2 – Transformation

The task of this stage is to transform t, which is a sequence of integers, into a number of sets $\mathcal{R}_\sigma$ of sequences for all possible values of σ. Each sequence belonging

Algorithm 1 Stage 1: Computing vectors $first$ and $next$

```
1: function FindOccurrences($t[1..n]$)
2:     Construct the function $f$ (see Equation 1)
3:     $first[1..|\Sigma|] \leftarrow 00\dots0$
4:     $next[1..n] \leftarrow 00\dots0$
5:     $last[1..|\Sigma|] \leftarrow 00\dots0$    ▷ Keeps track of the last occurrence of a particular
   $\sigma \in \Sigma$ so far
6:     for $i \leftarrow 1$ to $n$ do
7:         if $last[f(t[i])] = 0$ then
8:             $first[f(t[i])] \leftarrow i$
9:         else
10:            $next[last[f(t[i])]] \leftarrow i$
11:        $last[f(t[i])] \leftarrow i$
12:    return $first$, $next$
```

to $\mathcal{R}_\sigma$ is a q-sequence over $\{Q, S\}$ for the chosen $q = \sigma/2$. Our aim is to identify all the q-matches of r in $t' \in \mathcal{R}_\sigma$ (and consequently, into t).

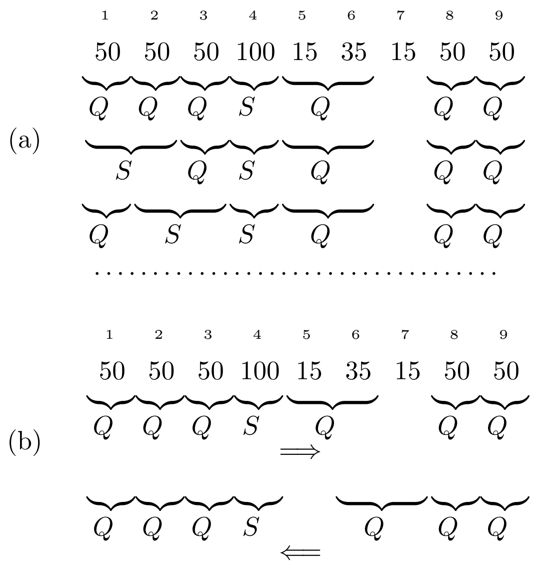

Figure 4: Ambiguities in transformation.

However, this transformation is ambiguous, in several ways, as the following example demonstrates. Consider the musical sequence shown in Figure 4(a), and let $q = 50$. One does not know whether two consecutive Q's should be transformed as QQ or S, and creating all the possible combinations is too time consuming. Moreover, as shown in Figure 4(b) the transformation that is generated while processing t from left to right is different from that generated while moving from right to left. So, what we do is as follows. For each occurrence of the current symbol $\sigma = 2q = S$, we try to convert the area surrounding that S into sequences or *tile* of Q's. When we can't continue to make Q's, we check whether we can make S's instead. Note that we first try to make Q and in case of a failure we try for an S. It is easy to observe that in this way, we can only find S, if S is solid, because, by definition, we cannot have S which can't be divided into two consecutive Q's. If we can't make either of them then we mark the end of the sequence. So each sequence $t' \in \mathcal{R}_\sigma$ consists of one or possibly more solid S's, surrounded by and separated from each other by zero or more Q's. Algorithm 2 gives the details.

What should be the total running time of Stage 2? It should be clear that the function Transform($t[1..n]$,σ,f) in Algorithm 2, has to be called for all possible values of σ. Another question is what will be the length in total of all the sequences generated in this stage. The latter question is important as well because this will affect the running time of the subsequent stages. In the rest of this subsection we answer the above two important questions.

We first define the relational operator "$\succeq$" on t as follows. We say that $t[i] \succeq t[j]$, if and only if, we have either $t[i] > t[j]$ or $t[i] = t[j]$ and $i \leq j$. Now let $\mathcal{H}$ is a multi-set such that $\mathcal{H} = \{t[i] \mid t[i] \geq t[j], 1 \leq j \leq n\}$. Now if $|\mathcal{H}| = \ell > 1$, then let $\mathcal{H} = \{H_1, H_2, ..., H_\ell\}$, where $H_j \succeq H_i, j < i$. Note carefully that $H_1 = H_2 = \ldots = H_\ell$ ($= H$ let). In this case, for all $1 \leq i \leq \ell$, we denote by $index(H_i)$, the index of H_i in t. For example if $t = 5\ 10\ 5\ 15\ 5\ 15\ 15\ 10$, then we have $H = 15$, $\mathcal{H} = H_1 = 15, H_2 = 15, H_3 = 15$, where $index(H_1) = 4$, $index(H_2) = 6$ and $index(H_3) = 7$.

It is easy to see that the vector $first$ gives us the location of H_1 and along with the vector *next* (Algorithm 1) implicitly provides us with the full set $\mathcal{H}$. So, in this stage we first call the function Transform($t[1..n]$,H, f) of Algorithm 2. It is easy to see that, Algorithm 2 will give us all the sequences ($H/2$-sequences to be precise) for $\mathcal{R}_H$. Now we have the following Lemmas.

Lemma 1 $\sum_{t' \in \mathcal{R}_H} |t'| \leq 2n$.

Proof. If $|\mathcal{H}| = 1$ then we are done, so assume otherwise. If $|\mathcal{H}| = \ell > 1$, then we have $\mathcal{H} = \{H_1, H_2, ..., H_\ell\}$. Now assume that we have two separate sequences t_{k_i} and $t_{k_{i+1}}$ each having only one S, due to H_i and H_{i+1}, respectively. Also assume that H_i and H_{i+1} is due to $t[m]$ and $t[n]$, i.e. $index(H_i) = m$ and $index(H_{i+1}) = n$. It is clear that $n > m$. Now, in t_{k_i} the tile of Q's to the right of H_i can not continue after $t[n-1]$ because $t[n] = H$ and $Q = H/2$. Similarly, in $t_{k_{i+1}}$ the tile of Q's to the left of H_{i+1} can not continue before $t[m+1]$. So in the worst case we have two tiles of Q's in the same area namely $t[m+1..n-1]$. From this it is easy to realize

$$\cdots\ \overset{1}{60}\ \underbrace{\overset{2}{50}}_{Q}\ \underbrace{\overset{3}{25}\ \overset{4}{25}}_{Q}\ \underbrace{\overset{5}{100}}_{S}\ \underbrace{\overset{6}{50}}_{Q}\ \underbrace{\overset{7}{15}\ \overset{8}{30}\ \overset{9}{5}}_{Q}\ \overset{10}{70}\ \underbrace{\overset{11}{30}\ \overset{12}{20}}_{Q}\ \underbrace{\overset{13}{50}}_{Q}\ \underbrace{\overset{14}{100}}_{S}\ \underbrace{\overset{15}{25}\ \overset{16}{25}}_{Q}\ \underbrace{\overset{17}{100}}_{S}\ \underbrace{\overset{18}{25}\ \overset{19}{20}\ \overset{20}{5}}_{Q}\ \overset{21}{60}\cdots$$

Figure 5: Transforming the area around $t[5] = S = 100$ and then around $t[14] = S = 100$.

Algorithm 2 Stage 2: Transformation

```
1: function TRANSFORM($t[1..n]$,$\sigma$, $f$)
2:     $q \leftarrow \sigma/2$
3:     $\mathcal{R}_\sigma \leftarrow \{\}$
4:     $i \leftarrow first[f(\sigma)]$
5:     while $i \neq 0$ do
6:         $x \leftarrow$ “$S''$
7:         $r \leftarrow 0$
8:         $j \leftarrow i$
9:         while $r < q$ and $j > 1$ do  ▷ Converting to the left. Here we don’t need
   to check for any solid $S$.
10:            $j \leftarrow j - 1$
11:            $r \leftarrow r + t[j]$
12:            if $r = q$ then
13:                Push $Q$ at the front of $x$
14:                $r \leftarrow 0$
15:        $r \leftarrow 0$
16:        $j \leftarrow i$
17:        $solid \leftarrow false$
18:        while $r < 2q$ and $j < n$ do                    ▷ Converting to the right
19:            $j \leftarrow j + 1$
20:            $r \leftarrow r + t[j]$
21:            if $r = q$ then
22:                Push $Q$ at the back of $x$
23:                if $t[j + 1] = 2q$ then▷ This means the next position has a solid $S$
24:                    $solid \leftarrow true$
25:                $r \leftarrow 0$
26:            if $r = 2q$ and $solid = true$ then
27:                Push $S$ at the front of $x$
28:                $solid \leftarrow false$
29:                $i \leftarrow j$             ▷ We should not consider this $S$ as next candidate
30:                if $t[j + 1] = 2q$ then▷ This means the next position has a solid $S$
31:                    $solid \leftarrow true$
32:                $r \leftarrow 0$
33:        $\mathcal{R}_\sigma \leftarrow \mathcal{R}_\sigma \cup \{x\}$
34:        $i \leftarrow next[i]$
35:    return $\mathcal{R}_\sigma$
```

that, in the worst case, a particular region can be covered by 2 tiles of Q and this worst case can only occur when each sequence have exactly one S. This proves the Lemma. □

Lemma 2 *The length of the total number of sequences generated in Stage 2 is $O(n \log H)$, where H is the maximum value in t.*

Proof. Assume that we have computed $\mathcal{R}_H$ and each part of t is covered by at least one sequence $t' \in \mathcal{R}_H$. Now, consider, again, two separate sequences t_{k_i} and $t_{k_{i+1}}$ each having only one S, due to H_i and H_{i+1}, respectively. Also assume that H_i and H_{i+1} is due to $t[m]$ and $t[n]$ i.e. $index(H_i) = m$ and $index(H_{i+1}) = n$. Now we need to compute the next value, let H', to consider as S. We claim that $H' = H/2$ as follows. We show it considering the area of t namely, $t[m+1..n-1]$. Note that according to our assumption $t[m+1..n-1]$ is covered by either t_{k_i} or $t_{k_{i+1}}$ or both. It is easy to observe that there can not exists a $t[i] > H/2, m+1 \leq i \leq n-1$ because otherwise any tile of $Q(= H/2)$ would have to stop before that, contradicting our assumption. Therefore, we can see that each value to be considered as S, is half of the previous value considered as S on so on. Therefor, from Lemma 1, it follows, in this case, that total number of sequences in Stage 2 is $O(n \log H)$. Finally, it is easy to realize that it would be lesser in the case when not every part of t is covered by $\mathcal{R}_H$ and also in the case when a sequence may have more than one S. □

In the rest of this subsection, we show that the total running time of Stage 2 would be $O(n \log H)$ as well. This is so as follows. Recall that we first compute $\mathcal{R}_H$ which takes linear time. Now we have to find the next value to consider as S. Consider the situation of Lemma 2 once again. It should be clear that the next value for S, in this case, is the highest value in the range $t[m+1..n-1]$. It is not very difficult to observe that, for each pair of sequences in $\mathcal{R}_H$ we get a range, the maximum value in which gives us the next possible values for S. With a linear preprocessing on t, this can be done in constant time per query by using Range Maxima Query technique [1, 5]. So it should be clear that the total running time is bounded by $O(n \log H)$ as is the total length of all the sequences generated in this stage.

One essential detail is that in this stage we may have to work with more then one values for S at the same time. Also note that, if there exists more than one instances for a particular value, we first consider the one having lowest index and so on. This ensures that in every $\mathcal{R}_\sigma$, we have the sequences in sorted order according their start position in t. And when we check for the next value we consider each sequence (for deducing the range to check) in that order. This preserves the order in each $\mathcal{R}_\sigma$ and ensures that the occurrences we find in later stages are in sorted order as well. This fact is used by us in Stage 4 to gain efficiency.

3.3. Stage 3 – Find the Matchings

In this stage we consider each $t' \in \mathcal{R}_\sigma$, for all valid values of σ and identify all the q-matches of r in t'. To do that efficiently we exploit a bit-masking technique as described below. We first define some notations that we use for sake of convenience. We define $S_{t'}$ and S_r to indicate an S in t' and r respectively. $Q_{t'}$ and Q_r are defined analogously. We first perform a preprocessing as follows. We construct t'' from t' where each $S_{t'}$ is replaced by 01 and each $Q_{t'}$ is replaced by 1. Note that we have to keep track of the corresponding positions of t' in t''. We then construct the 'Invalid' set I for t'' where I includes each position of '1' of $S_{t'}$ in t''. For example, if $t' = QQSQS$ then $t'' = 1101101$ and I = {4,7}. It is easy to see that no occurrence of r can start at $i \in I$. We also construct r' from r where each S_r is replaced by 10 and each Q_r is replaced by 0. This completes the preprocessing. After the preprocessing is done, at each position $i \notin I$ of t'' we perform a bitwise 'OR' operation between $t''[i..i + |r'| - 1]$ and r'. If the result of the 'OR' operation is all 1's, i.e. $1^{r'}$ then we have found a match at position i of t''. However, we need to ensure that there is a solid S in the match. To achieve that we simply perform a bitwise 'XOR' operation between $t''[i..i + |r'| - 1]$ and $1^{r'}$ and only if the result of this 'XOR' returns a nonzero value, we go on with the 'OR' operation stated above. The details are formally given in the form of Algorithm 3.

Algorithm 3 Reporting Occurrences of r in t'

```
1:  function FindMatch($t'$,$r$)
2:      $Occ[1..|t'|] \leftarrow 0\,0\ldots 0$                        ▷ Preprocessing Step
3:      $\mathcal{I}[1..|t''|] \leftarrow 0\,0\ldots 0$
4:      $j \leftarrow 1$
5:      for $i = 1$ to $|t'|$ do
6:          $track[j] \leftarrow i$
7:          if $t'[i] = S$ then
8:              $t''[j] \leftarrow 0$
9:              $t''[j+1] \leftarrow 1$
10:             $\mathcal{I}[j+1] \leftarrow 1$                       ▷ Position $j+1$ is invalid
11:             $j \leftarrow j + 2$
12:         else
13:             $t''[j] \leftarrow 1$
14:             $j \leftarrow j + 1$
15:     $j \leftarrow 1$
16:     for $i = 1$ to $|r|$ do
17:         if $r[i] = S$ then
18:             $r'[j] \leftarrow 1$
19:             $r'[j+1] \leftarrow 0$
20:             $j \leftarrow j + 2$
21:         else
22:             $r'[j] \leftarrow 0$
23:             $j \leftarrow j + 1$
                                                                  ▷ Matching Step
24:     for $i = 1$ to $|t''|$ do
25:         if $(\mathcal{I}[i] \neq 1)$ and $(t'[i..i + |r'| - 1] \oplus 1^{r'} > 0)$ then    ▷ '$\oplus$' is the bitwise
    xor operator
26:             if $t''[i..i + |r'| - 1] \,||\, r' = 1^{r'}$ then     ▷ '||' is the bitwise or operator
27:                 $Occ[track[i]] \leftarrow 1$
28:     return Occ
```

We now discuss the correctness of Algorithm 3. We use the symbol $\sim$ and $\nsim$ to denote, respectively "*matches*" and "*doesn't match*". It is easy to see that for the problem in hand we must meet the following conditions.

1. $Q_{t'} \sim Q_r$
2. $Q_{t'}Q_{t'} \sim S_r$
3. $S_{t'} \sim S_r$
4. $S_{t'} \nsim Q_rQ_r$

All the conditions stated above are obeyed by the encoding we use as shown below. Recall that we do bitwise or operation and that we report a match when the result of the operation is all 1's.

1. $Q_{t'}(=1)$ and $Q_r(=0)$ always matches: $(1\ or\ 0 = 1)$.
2. $Q_{t'}Q_{t'}(=11)$ always matches with $S_r(=10)$: $(11\ or\ 10 = 11)$.
3. $S_{t'}(=01)$ can only match with $S_r(=10)$: $(01\ or\ 10 = 11)$.
4. Since $S_{t'}(=01)$ can't give a match with $Q_rQ_r(=00)$: $(01\ or\ 00 = 01)$.

However we have a problem when the S_r and $S_{t'}$ are 'misaligned'. We define $start(S_r) = 1$ and $end(S_r) = 0$. Similarly, we have, $start(S_{t'}) = 0$ and $end(S_{t'}) = 1$. Assume that we have an $S_r(say\ S_r^k)$ misaligned with an $S_{t'}(say\ S_{t'}^l)$.

Case 1- $end(S_r^k)$ **is aligned with** $start(S_{t'}^l)$**:** We have $end(S_r^k)$ or $start(S_{t'}^l)$ $0\ or\ 0 = 0$. So we have no match as required.

Case 2- $start(S_r^k)$ **is aligned with** $end(S_{t'}^l)$**:** Unfortunately here we have $start(S_r^k)$ or $end(S_{t'}^l) = 1\ or\ 1 = 1$ which may create problems. We distinguish between two subcases. We say an S_r is 'inside' r (or equivalently r') if this S_r is not the start of r.

Case2.a- S_r^k **is inside** r**:** There must be either a Q_r or another S_r (say S_r^j) just before this S_r^k. In any case we will have either $Q_r(=0)$ or $end(S_r^j)(= 0)$ to align with $start(S_{t'}^l)(= 0)$ which will give 0 after the or operation and hence we have no problem.

Case2.b- S_r^k **is the start of** r**:** In this case, we have $start(S_r^k)\ or\ end(S_{t'}^j) = 1$ which may give us a 'false positive' starting at this position. To exclude these false positives we have the 'Invalid' set $\mathcal{I}$. The main idea is that no occurrence of the rhythm can start at $end(S_{t'})$. So each $end(S_{t'})$ is included in $\mathcal{I}$. And we check whether the position we are checking is in $\mathcal{I}$ or not.

Here we give an example of a 'false positive' as discussed above. Suppose $t' = QQSQQ$ and $r = SQ$. Then we have $t'' = 110111$ and $r' = 100$. It is easy to see that if we perform the bitwise or operation at each position of t'' we get two matches starting at $t''[3]$ and also at $t''[4]$. But it is easy to verify that position 4 of t'' doesn't really exist in t'. So it is a 'false positive'. It remains to show how we ensure that there exists a solid S in the match. Recall that we perform an 'xor' operation between $t''[i..i+|r'|-1]$ and $1^{r'}$ before we perform the 'or' operation between $t''[i..i+|r'|-1]$ and r'. Now if there exists a solid S in $t''[i..i+|r'|-1]$, assuming $i \notin I$, we must have $t''[j] = 0, j \in [i..i+|r'|-1]$ and therefore, the 'xor' operation above returns a nonzero value only when there exists a solid S in $t''[i..i+|r'|-1]$.

The above discussion establishes the correctness of Algorithm 3. Now we discuss the the total running time of this stage. It is easy to see that Algorithm 3 runs in $O(|t''| \times |r'|/w)$ time where w is the size of the word of the target machine. Note that the function FindMatch(t',r) of Algorithm 3 is called for every sequence generated in Stage 3. So it follows from Lemma 2 that the total running time in Stage 3 is $O(n \log H \times m/w)$.

3.4. Stage 4 – Find the Cover

In this stage we can assume that we have sets of q-occurrence lists corresponding to the q-matches for the r in t. Let us assume that we have $\mathcal{O} = \{Occ_\sigma\}$ where Occ_σ is the set of occurrences corresponding to q-matches assuming $q = \sigma/2$. Recall that we have the occurrences in sorted order. Now what we do is as follows. For each $Occ_\sigma \in \mathcal{O}$, we try to find the corresponding q-covers ($q = \sigma/2$). This can be easily done by checking, respectively, the end and start positions of consecutive occurrences. Also, we maintain global variable to keep track of the longest cover so far. Algorithm 4 gives the details. It is easy to observe that the running time of stage can't exceed $O(n \log H)$ because the total number of occurrences in the worst case can't exceed the total length of all the sequences.

So in total, the algorithm takes $O(n \log H m/w)$ time. Hence, if $m \sim w$, the running time reduces to $O(n \log H)$. It may further be noted here that the length m of the rhythm, in practical cases, is usually 10-13 characters and thus we can consider it to be constant. Therefore, for all practical purposes our algorithm runs in $O(n \log H)$ time.

4. Conclusions

In this paper we have presented efficient algorithms that can be used to classify musical texts according to given rhythms. We have defined the musical text, rhythm and the notion of match and cover in this regard and formulated a new problem (Problem 1). As a by product we end up defining a yet another new pattern matching paradigm where addition of consecutive entries in the text are allowed to find a match with the pattern. Note also that, in our model, the pattern and the text are from different alphabets. The algorithm we have presented can be used for a number of different variants of our problem with slight or no modification as follows:

Cover versus Tile: In some cases, instead of covers, finding a tile, i.e. consecutive

Algorithm 4 Finding Covers

```
1: function FindCover($\mathcal{O}$)
2:     $globalCover\{start, end, value\} \leftarrow \{0, 0, 0\}$
3:     for each $Occ_\sigma \in \mathcal{O}$ do
4:         $Cover_\sigma\{start, end\} \leftarrow \{0, 0\}$
5:         $start \leftarrow true$
6:         for $i = 1$ to $|Occ_\sigma|$ do
7:             if $start = true$ then
8:                 $Cover.start \leftarrow Occ_\sigma[i].start$
9:                 $Cover.end \leftarrow Occ_\sigma[i].end$
10:                $start \leftarrow false$
11:            else if $start = false$ then
12:                if $Cover.end \geq Occ_\sigma[i].start$ then        ▷ A continuing cover
13:                    $Cover.end \leftarrow Occ_\sigma[i].end$
14:                else if $Cover.end < Occ_\sigma[i].start$ then    ▷ A discontinued cover
15:                    $start \leftarrow true$
16:                    if $Cover_\sigma.end - Cover_\sigma.start < Cover.end - Cover.start$ then
                       ▷ current cover is longer than the global one for this $\sigma$
17:                        $Cover_\sigma.start \leftarrow Cover.start$
18:                        $Cover_\sigma.end \leftarrow Cover.end$
                                                                   ▷ End of one $Occ_\sigma$
19:        if $globalCover.end - globalCover.start < Cover_\sigma.end - Cover_\sigma.start$
           then                          ▷ Cover for this $\sigma$ is longer than the global one
20:            $globalCover.start \leftarrow Cover_\sigma.start$
21:            $globalCover.end \leftarrow Cover_\sigma.end$
22:            $globalCover.value \leftarrow \sigma/2$
    return $globalCover$
```

repetition may turn out to be more meaningful. Our algorithm can be easily adapted to solve this variant. It is easy to see that the only change that need to be done is in Stage 4 of our algorithm.

Rhythms of Different Tempo: In our problem we look for the longest cover for a for a specific value of q. However, if the same rhythm exists in the same music at more than one tempo, then it would definitely be interesting to find the best cover considering different values of q. This variant, as well, can be solved using our algorithm with slight modification.

Despite that there does not exist a complete agreement on different definitions in Computational Musicology, we believe that our model and the problem we have handled would be of both practical and theoretical interest in both Computing and music research. A number of interesting issues remain open as follows:

1. Designing an algorithm that avoids the restriction that one symbol has to be 'solid'. In this cases it may be meaningful to apply restriction on the number of consecutive entries to be added to match a Q or S.

2. We have assumed that the duration of S is double the duration of Q. It may be interesting to relax this assumption and allow other possibilities simultaneously.

3. As is the case in all pattern matching algorithms, it would be interesting to introduce different kinds errors and define an approximate paradigm on top of this new model.

References

1. Michael A. Bender and Martin Farach-Colton. The lca problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *Latin American Theoretical INformatics (LATIN)*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000.
2. Alexander R. Brinkman. *PASCAL Programming for Music Research.* The University of Chicago Press, Chicago and London, 1990.
3. D. Byrd and E. Isaacson. A music representation requirement specification for academia. *The Computer Music Journal*, 27(4):43–57, 2003.
4. T. Crawford, C.S. Iliopoulos, and R. Raman. String matching techniques for musical similarity and melody recognition. *Computing in Musicology*, 11:227–236, 1998.
5. H. Gabow, J. Bentley, and R. Tarjan. Scaling and related techniques for geometry problems. In *Symposium on the Theory of Computing (STOC)*, pages 135–143, New York, NY, USA, 1984. ACM Press. Chairman-Richard DeMillo.
6. Peter Howell, Robert West, and Ian Cross, editors. *Representing Musical Structure.* Academic Press London, 1991.
7. C. S. Iliopoulos, K. Lemstrom, M. Niyad, and Y. J. Pinzon. Evolution of musical motifs in polyphonic passages. In G. Wiggins, editor, *Symposium on AI and Creativity in Arts and Science, Proceedings of AISB'02*, pages 67–76, 2002.

8. K. Lemstrom. String matching techniques for music re trieval. *PhD Thesis, University of Helsinki, Department of Computer Science*, 2000.
9. K. Lemstrom and P. Laine. Musical information retrieval using musical parameters. In *International Computer Music Conference*, pages 341–348, 1998.
10. K. Lemstrom and J. Tarhio. Detecting monophonic patterns within polyphonic sources. In *Multimedia Information Access Conference*, volume 2, pages 1261–1279, 2000.
11. Alan Marsden and Anthony Pople, editors. *Computer Representations and Models in Music*. Academic Press London, 1992.
12. M. Mongeau and D. Sankoff. Comparison of musical sequences. *Computers and the Humanities*, 24:161–175, 1990.
13. Eleanor Selfridge-Field, editor. *Beyond MIDI: The Handbook of Musical Codes*. The MIT Press, 1997.
14. D.A. Stech. A computerassisted approach to micro analysis of melodic lines. *Computers and the Humanities*, 15:211–221, 1981.
15. G. A. Wiggins, E. Miranda, A. Smaill, and M. Harris. A framework for the evaluation of music representation systems. *The Computer Music Journal*, 17(3):31–42, 1993.