Song Classifications for Dancing

Manolis Christodoulakis[1,4], Costas S. Iliopoulos[1,3,⋆], M. Sohel Rahman[1,3,⋆⋆,⋆⋆⋆], and William F. Smyth[4,5,†]

[1] Algorithm Design Group
Department of Computer Science, King's College London
Strand, London WC2R 2LS, England
http://www.dcs.kcl.ac.uk/adg
[2] Algorithms Research Group,
Department of Computing and Software,
McMaster University, Canada
[3] {csi, sohel}@dcs.kcl.ac.uk
[4] manolis.christodoulakis@kcl.ac.uk
[5] smyth@mcmster.ca

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 $S = QQ$. In this paper we present a linear algorithm for locating the maximum-length substring of a music text t that can be covered by a given rhythm r. An efficient algorithm to solve this problem, can then be used to find which rhythm, from a given set of such rhythms, covers the largest part of the music sequence under question, and thus best describes that sequence.

Keywords: algorithms, music sequence

1 Introduction

The subject of musical representation for use in computer application has been studied extensively in computer science literature [2, 1, 4, 9, 13, 11]. Computer assisted music analysis [12, 10] and music information retrieval [5, 8, 7, 6] has 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 [3]. 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 by dancing rhythm. We will define what a dancing rhythm is, and how it can be identified in a musical sequence, a song. 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 characterize how the song is danced.

⋆ 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.

In particular, there are two types of intervals in the dancing 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, a 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 describe the notation that is used throughout the paper, and we define the terms matching and covering in musical sequences. In Section 3 we describe in detail our algorithm for finding the largest area in a musical sequence that is covered by a given rhythm. As will be seen, under the restrictions we impose on our problem, the algorithms we devise run in linear time. Finally, Section 4 contains our concluding remarks.

2 Definitions

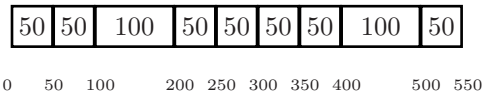
A *musical sequence* t is a string $t = t[1]t[2]\dots t[n]$, where $t[i] \in \mathbb{N}^+$, for all $1 \leq i \leq n$. For example the sequence

$$[0, 50, 100, 200, 250, 300, 350, 400, 500, 550]$$

represents a sequence of events occurring at 0 milliseconds, 50 milliseconds, 100 milliseconds, and so on, in the original music signal. Alternatively, we can represent musical sequences by the duration of the events, as follows

$$[50, 50, 100, 50, 50, 50, 50, 100, 50]$$

The two definitions above are equivalent. We prefer the latter here for the sake of clarity. The above musical sequence can then be represented graphically as shown in the following figure.



A *rhythm* r is a string $r = r[1]r[2]\dots r[m]$, where $r[j] \in \{Q, S\}$, for all $1 \leq j \leq m$. For example, $r = QSS$. Q and S correspond to intervals between 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. The length m of the rhythm, in practical cases, is usually 10-13 characters and thus we can consider it to be constant.

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

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

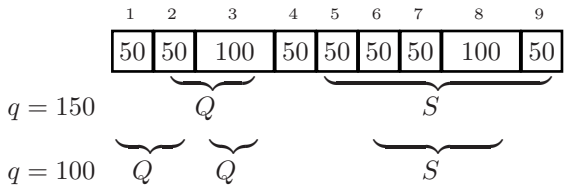


Figure 1. Q- and S-matching in musical sequences

As with Q, the match of S is said to be solid if $i = i'$.

For example, consider the musical sequence shown in Figure 1. 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 $r = r[1] \dots r[m]$ is said to *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. $r[j]$ matches $t[i_j..i_{j+1} - 1]$, for all $1 \leq j \leq m$

For instance, the rhythm $r = QSS$ matches with $t[2..5]$ as well as with $t[5..8]$, in Figure 2, for $q = 50$.

Finally, a rhythm r is said to *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 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 2, $r = QSS$ covers $t[2..8]$ for $q = 50$.

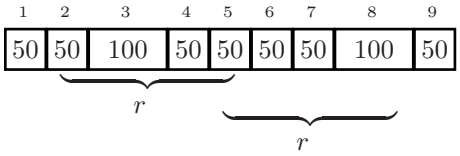


Figure 2. Matches of $r = QSS$ in t, for $q = 50$

3 Maximal Coverability Algorithm

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

Problem 1. Given a musical sequence $t = t[1]t[2] \dots t[n]$, $t[i] \in \mathbb{N}^+$, and a rhythm $r = r[1]r[2] \dots r[m]$, $r[j] \in \{Q, S\}$, find the largest (longest) substring $t[i..i']$ of t that is covered by r.

Note that the definition above is very general, allowing extreme cases like the following: consider a musical sequence consisting of a single tone repeated every 1ms, $t = 111 \dots 1$. 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

Algorithm 6 Stage 1: Computing vectors $first$ and $next$

```
1: function FindOccurrences(t[1..n])
2:     first[1..|Σ|] ← 00...0
3:     next[1..n] ← 00...0
4:     last[1..|Σ|] ← 00...0        ▷ Keeps track of the last occurrence of a particular σ ∈ Σ so far
5:     for i ← 1 to n do
6:         if last[t[i]] = 0 then
7:             first[t[i]] ← i
8:         else
9:             next[last[t[i]]] ← i
10:        last[t[i]] ← i
11:    return first, next
```

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 q, provided of course that $2\sigma \in \Sigma$, and for that particular q all the occurrences of the rhythm r must be identified. Equivalently, we can consider each σ to be equal to $S = 2q$, provided that $\sigma/2 \in \Sigma$. In our algorithm, we will be using the latter form. Then, for each such $\sigma \in \Sigma$, the algorithm sets $S = 2q = \sigma$ and proceeds in three stages:

- *Stage 1*: Find all occurrences of S in t.
- *Stage 2*: Transform the areas around all the S's into a sequences of Q's.
- *Stage 3*: Find the maximal area covered by r, for the current q.

We next explain each of these stages in detail.

3.1 Stage 1 – Finding all occurrences

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. 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. Given that the alphabet Σ is indexed and its size is bounded, this can be done in $O(n)$ time and $O(n + |\Sigma|)$ space in the following manner. Consider vectors $first$, of size $|\Sigma|$, and $next$, of size n, such that

- $first[\sigma] = i$ if and only if the first 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 $next$. Algorithm 6 shows how this is done in detail.

3.2 Stage 2 – Transformation

The task of this stage is to transform t, which is a sequence of integers, into a sequence t', over $\{Q, S\}$ for the chosen $q = \sigma/2$, so that all the matches of r into t' (and consequently, into t) are identified. However, this transformation is ambiguous, in several ways, as the following example demonstrates.

Consider the musical sequence shown in Figure 3(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 3(b) the transformation that is generated while processing t from left to right is different from that generated while moving from right to left.

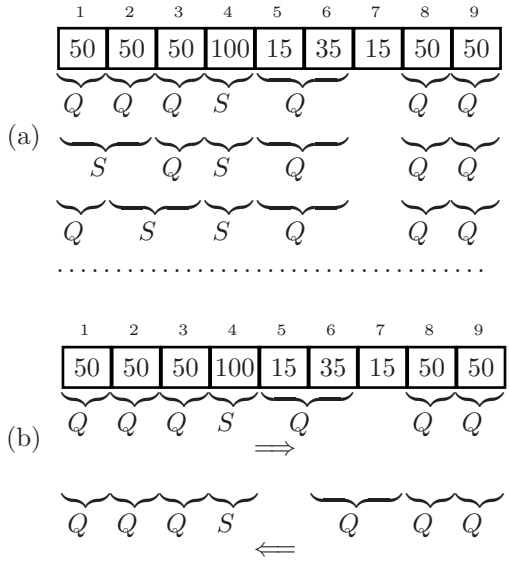


Figure 3. Ambiguities in transformation

For each occurrence of the current symbol $\sigma = 2q = S$, we convert the area surrounding that S into sequences of Q's. Algorithm 7 gives the details.

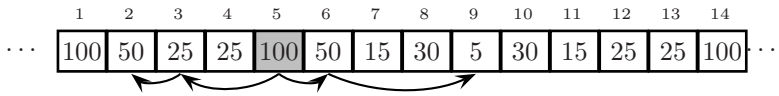


Figure 4. Transforming the area around $t[5] = S = 100$

3.3 Stage 3 – Maximal Covered Area

As soon as we get t', a sequence over $\{Q, S\}$, transformed from t in Stage 2, our job is to first identify all the occurrences 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''.

Algorithm 7 Stage 2: Transformation

```
 1: function TRANSFORM(t[1..n],σ)
 2:     q ← σ/2
 3:     R_σ ← {}
 4:     i ← first[σ]
 5:     while i ≠ 0 do
 6:         x ← "S''
 7:         r ← 0
 8:         j ← i
 9:         while r < q and j < n do
10:             j ← j + 1
11:             r ← r + t[j]
12:             if r = q then
13:                 Push Q at the back of x
14:                 r ← 0
15:         r ← 0
16:         j ← i
17:         while r < q and j > 1 do
18:             j ← j − 1
19:             r ← r + t[j]
20:             if r = q then
21:                 Push Q at the front of x
22:                 r ← 0
23:         R_σ ← R_σ ∪ {x}
24:         i ← next[i]
25:     return R_σ
```

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 then we report an occurrence at position i of t''. The details are formally given in the form of Algorithm 8.

We now discuss the correctness of Algorithm 8. 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 'miss-aligned'. 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 $S_r(say\ S_r^k)$ miss-aligned with an $S_{t'}(say\ S_{t'}^l)$.

Algorithm 8 Reporting Occurrences of r in t'

```
1: function FindMatch(t',r)
2:     Occ[1..|t'|] ← 0 0 ... 0                          ▷ Preprocessing Step
3:     I[1..|t''|] ← 0 0 ... 0
4:     j = 1
5:     for i = 1 to t' do
6:         track[j] = i
7:         if t'[i] = "S" then
8:             t''[j] = "01"
9:             I[j + 1] = 1                              ▷ Position j + 1 is invalid
10:            j = j + 2
11:        else
12:            t''[j] = "1"
13:            j = j + 1
14:    j = 1
15:    for i = 1 to r do
16:        if r[i] = "S" then
17:            r'[j] = "10"
18:            j = j + 2
19:        else
20:            r'[j] = "0"
21:            j = j + 1
                                                         ▷ Matching Step
22:    for i = 1 to t'' do
23:        if I[i] ≠ 1 then
24:            if t'[i..i + m1 − 1] or p' = "11 ... 1" then
25:                Occ[track[i]] = 1
26:    return Occ
```

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 its a 'false positive'.

The above discussion establishes the correctness of Algorithm 8. Since the size of the rhythm is considered constant, Algorithm 8 runs in $O(|t''|/w)$ time where w is

the size of the word of the target machine. Finally, once we get the occurrences of the rhythm r in t' considering every choice of σ it is easy to report the maximal covered area in linear time. In fact we can compute this area on the fly while computing all the occurrences of r in t' by slightly modifying Algorithm 8.

4 Open Problems

In this paper we have presented algorithms for computerized song classifications under some specific constraints. A number of issues remain unsolved as follows:

1. Designing an algorithm that avoids the restriction that one symbol has to be *solid*.
2. Applying a limit on the number of "additions" in the numeric text to match a Q and/or S.
3. Removing the dependency on m from the algorithm.

References

[1] A. R. Brinkman: *PASCAL Programming for Music Research*, The University of Chicago Press, Chicago and London, 1990.

[2] D. Byrd and E. Isaacson: *A music representation requirement specification for academia*. The Computer Music Journal, 27(4) 2003, pp. 43–57.

[3] T. Crawford, C. Iliopoulos, and R. Raman: *String matching techniques for musical similarity and melody recognition*. Computing in Musicology, 11 1998, pp. 227–236.

[4] P. Howell, R. West, and I. Cross, eds., *Representing Musical Structure*, Academic Press London, 1991.

[5] C. S. Iliopoulos, K. Lemstrom, M. Niyad, and Y. J. Pinzon: *Evolution of musical motifs in polyphonic passages*, in Symposium on AI and Creativity in Arts and Science, Proceedings of AISB'02, G. Wiggins, ed., 2002, pp. 67–76.

[6] K. Lemstrom: *String matching techniques for music re trieval*. PhD Thesis, University of Helsinki, Department of Computer Science, 2000.

[7] K. Lemstrom and P. Laine: *Musical information retrieval using musical parameters*, in International Computer Music Conference, 1998, pp. 341–348.

[8] K. Lemstrom and J. Tarhio: *Detecting monophonic patterns within polyphonic sources*, in Multimedia Information Access Conference, vol. 2, 2000, pp. 1261–1279.

[9] A. Marsden and A. Pople, eds., *Computer Representations and Models in Music*, Academic Press London, 1992.

[10] M. Mongeau and D. Sankoff: *Comparison of musical sequences*. Computers and the Humanities, 24 1990, pp. 161–175.

[11] E. Selfridge-Field, ed., *Beyond MIDI: The Handbook of Musical Codes*, The MIT Press, 1997.

[12] D. Stech: *A computerassisted approach to micro analysis of melodic lines*. Computers and the Humanities, 15 1981, pp. 211–221.

[13] 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) 1993, pp. 31–42.