

# Erratic Dancing

Joseph Wun-Tat Chan<sup>1,2</sup>, Costas S. Iliopoulos<sup>1,3</sup>, Spiros Michalakopoulos<sup>1,3</sup>,  
and M. Sohel Rahman<sup>1,3</sup>

<sup>1</sup> Algorithm Design Group  
Department of Computer Science, King's College London  
<http://www.dcs.kcl.ac.uk/adg>

<sup>2</sup> [joseph.chan@kcl.ac.uk](mailto:joseph.chan@kcl.ac.uk)

<sup>3</sup> [{csi,spiros,sohel}@dcs.kcl.ac.uk](mailto:{csi,spiros,sohel}@dcs.kcl.ac.uk)

**Abstract.** The problem of classifying dance songs according to rhythms has recently been introduced in [4] and [3]; in this paper, we present new efficient algorithms that take into account temporal errors, which could not be handled by the algorithms presented in [4] and [3]. We analyze and compare the running times of two algorithmic variants and furthermore show examples of their implementation.

## 1 Introduction

With the continuing advancement in database and archiving technology in recent years, there comes a need for fast and efficient algorithms for storing, indexing and retrieving this information. Music kept in digital music libraries, collections or databases, and its representation for use in computer applications is the subject of extensive studies in computer science literature [2, 9, 14, 20, 18].

Computer assisted music analysis [19, 15] and music information retrieval [10, 13, 12, 11] have 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 [6]. A useful functionality of multimedia databases is the automatic classification of songs by one or more of their characteristics, like genre, melody, rhythm etc. For human beings, the process of identifying these characteristics seems natural, though sometimes ambiguous. However, computerized classification is hard to achieve, given that there does not exist a complete agreement on the definition of these features. Very recently, Christodoulakis et al. [4] studied the problem of classification of a music text by rhythms. We extend the definitions used in their work, allowing temporal errors and continue to focus on music of the ballroom dance genre.

Musical sequences can be thought of as having 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's rhythm. In particular, there are two types of intervals in the rhythm of a song: *quick* ( $Q$ ) and *slow* ( $S$ ), [21, 1, 16]. *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  $SSQQSSSSQQS$ , while a foxtrot is given as  $SSQQSSQQ$ , and a jive as  $SSQQSQQS$ .

In [4], the authors presented an efficient algorithm for locating the maximum-length substring of a music text  $t$  that can be “exactly” (no temporal errors) covered by a given rhythm  $r$ . The goal of this paper is to extend their work in a more practical setting by considering temporal errors and defining an approximate paradigm on top of their model. The motivation of our work follows from the fact that temporal errors often occur, either in the playing of the rhythm by a drummer or percussionist for example, or in the extraction of the data, particularly when converting from analog to digital. The problem we wish to handle in this paper can be seen as a  $\mu$ -approximation version of the original problem, where  $\mu$  is the degree of error allowed.

## 2 Definitions

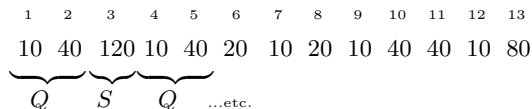
In this paper, we follow the notations and definitions of [4]. For the sake of completeness, we discuss the notations and definitions before introducing the error model.

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 milliseconds. Then, the sequence  $S_1 = [0, 50, 100, 200, 240]$  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,  $S_2 = [50, 50, 100, 40]$  represents the same music signal. It is clear that the two definitions are equivalent. We use the latter definition throughout the rest of this paper. Formally, a *musical sequence*  $t$  is a string  $t = t[1]t[2] \dots t[n]$ , where  $t[i] \in \mathbb{N}^+$ , for all  $1 \leq i \leq n$ .

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,  $r = QSS$ . Here  $Q$  and  $S$  correspond to durations of activities (intervals between the start of consecutive events), such that the duration represented by an  $S$  is double that represented by a  $Q$ . However, the exact value of a  $Q$  or an  $S$  is not a priori known. Note that the *alphabets* for the musical text and that of the rhythm differ. The alphabet for the musical text is  $\Sigma = \{t[i] \mid 1 \leq i \leq n\}$ , whereas the alphabet for the rhythm is  $\Sigma_r = \{Q, S\}$ .

Into this paradigm we introduce error  $\mu$ . We claim that allowing this is significant because of *human* errors, e.g. a percussionist is slightly off in her drumming, intentional or otherwise, because of *technological* errors, e.g. conversion of analog to digital data using an appropriate algorithm [5, 7, 8, 17] can have rounding and other types of errors. We would expect the value of  $\mu$  to be small. For the purposes of the algorithm, we restrict  $\mu$  to a relatively large value and we justify this value for a very simple reason. First we define a  $\lambda$ -match and then we discuss the possible range of values for  $\mu$ .

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



**Fig. 1.**  $Q$ - and  $S$ -matching in musical sequences.

$t$ , if and only if,

$$q = t[i] + t[i + 1] + \dots + t[i'] + \lambda_q,$$

where  $1 \leq i \leq i' \leq n$  and  $\lambda_q$  is an integer such that  $|\lambda_q| \leq \mu$ . If  $i = i'$ , then the match is said to be *solid*. Similarly,  $S$  is said to  $\lambda$ -match with  $t[i..i']$ , if and only if,

$$\sigma = t[i] + t[i + 1] + \dots + t[i'] + \lambda_\sigma,$$

where  $\lambda_\sigma$  is an integer such that  $|\lambda_\sigma| \leq 2\mu$ . As with  $Q$ , the match of  $S$  is said to be *solid* if  $i = i'$ .

Here,  $\lambda$  represents the *local* error. The *global* error is defined later on in this section. In what follows, we use the term  $\lambda_q$ -match when we refer to the matching of a  $Q$  and  $\lambda_\sigma$ -match when matching an  $S$ . However, when it is clear from the context or we don't need to differentiate between the two types, we simply use the term “match” or “ $\lambda$ -match”. Next, we discuss the permissible values for  $\mu$ .

We allow errors both for quick ( $Q$ ) and slow ( $S$ ) beats, so we need to allow an error margin which makes it unambiguous whether a  $\lambda$ -match is a  $\lambda_q$ -match or a  $\lambda_\sigma$ -match. In particular, possible values for a  $q$  should not overlap with possible values for a  $\sigma$ , i.e. the upper bound of a  $q$  should not be larger than or equal to the lower bound of a  $\sigma$ . We solve the equation:

$$q + \mu \leq \sigma - 2\mu \Rightarrow q + \mu \leq 2q - 2\mu \Rightarrow \mu \leq q/3 \quad (1)$$

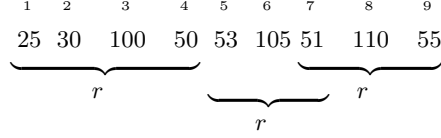
And hence the maximum error  $\mu$  is equal to  $q/3$ .

In the case that this maximum value for  $\mu$  is used, we would have *valid values* for  $q \in [2q/3..4q/3]$  and valid values for  $\sigma \in [4q/3+1..8q/3]$ . Notice that we “give the advantage” to a  $q$  for the “border” condition of  $4q/3$ . This is justified by the fact that, according to the definition, two  $q$ 's make a  $\sigma$ , but a  $\sigma$  cannot be split into two  $q$ 's. For example, consider the musical sequence shown in Figure 1. For  $q = 60$  and  $\mu = 10$ ,  $Q$   $\lambda$ -matches with  $t[1..2]$ ,  $t[4..5]$ ,  $t[4..6]$  etc. and  $S$   $\lambda$ -matches with  $t[3]$  (solid),  $t[4..8]$  with  $\lambda_\sigma = 20$  as well as with  $t[4..9]$  with  $\lambda_\sigma = 10$ .

A rhythm  $r = r[1] \dots r[m]$  is said to  $(\kappa\text{-}\lambda)$ -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  $i_1 < i_2 < \dots < i_m < i_{m+1}$ , such that,

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

For instance, the rhythm  $r = QSQ$ ,  $(\kappa\text{-}\lambda)$ -matches with  $t[1..4]$  as well as with  $t[5..7]$  and  $t[7..9]$ , in Figure 2, for  $q = 50$  and  $\mu = 5$ .



**Fig. 2.**  $q$ -matches of  $r = QSQ$  in  $t$ , for  $q = 50$  and  $\mu = 5$ .

Note the different total number of elements of  $t$  that  $(\kappa-\lambda)$ -matched with the  $r$  in the above two instances. This means that, reporting only the start (or the end) position may not convey the complete information. Therefore, we report both the start and end positions to denote the  $\mu$ -approximate  $q$ -occurrences against the  $(\kappa-\lambda)$ -matches. Therefore, the  $\mu$ -approximate  $q$ -occurrence list for the above case is  $Occ_{\mu q} = \{(1, 4), (5, 7), (7, 9)\}$ .

A rhythm  $r$  is said to  $(\kappa-\lambda)$ -cover the substring  $t[i..i']$  of the musical sequence  $t$ , if and only if, there exist integers  $i_1, i'_1, i_2, i'_2, \dots, i_k, i'_k$ , for some  $k \geq 1$ , such that:

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

In our example, in Figure 2, the rhythm  $r = QSQ$   $q$ -covers  $t[1..9]$  for  $q = 50$  and  $\mu = 5$ .

We now define a *constant*  $c$  to be the maximum number of consecutive elements in  $t$  that we can *merge* to give us a  $Q$  or an  $S$ . We define a *valid sequence* to be one in which each element of the sequence does not have an error greater than  $\mu$ . We define  $\kappa$  to be the sum of all the local errors in a valid sequence and name it the *global error* of this sequence. Finally, we define the *best sequence*  $R_\sigma$  over  $t$  for a given  $\sigma$  to be a valid sequence which minimizes the overall error, i.e. the global error  $\kappa$ .

### 3 Maximal Coverability Algorithm

The *maximal coverability* problem is formally defined as follows:

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

The following restriction is applied on the above problem.

**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  $r[j] = t[k] = \sigma + \lambda_\sigma$ ,  $i \leq k \leq i'$  and  $|\lambda_\sigma| \leq 2\mu$ , for some value of  $\sigma$ .

The justification of the above restriction follows from the following pathological example: consider a musical sequence consisting of a single tone repeated every 1 ms,  $t = 111\dots 1$ . 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 sequence of  $q$  1's, and any  $S$  in  $r$  will match with a sequence of  $2q$  1's. Therefore, as it was argued in [4], from a musical point of view, it is meaningful to have at least one event that is solid.

The algorithm presented in [4] to solve the exact version of Problem 1 consisted of 4 stages. We focus on Stage 2, because this is the only stage that is affected by the introduction of errors. We however briefly review Stages 1, 3 and 4 for the sake of completeness, as we proceed.

### 3.1 Stage 1 – Finding all occurrences of $S$

In this stage, we need to find all occurrences of  $S = \sigma$ , for each possible value of  $\sigma$ . This is done in  $O(n \log |\Sigma|)$  time and  $O(n + |\Sigma|)$  space in the following manner. Consider balanced binary search tree *first*, of size  $|\Sigma|$  and height  $\log |\Sigma|$ , and vector *next*, of size  $n$ , such that

- $(\sigma, i)$  is an item in tree *first*, with *key* =  $\sigma$  and *data* =  $i$ , iff the leftmost occurrence of the symbol  $\sigma$  appears at position  $i$  of  $t[1..n]$ .
- $\text{next}[i] = j$  iff  $t[i] = t[j]$  and for all  $k$ ,  $i < k < j$ ,  $t[k] \neq t[i]$ ; if no such  $j$  exists, then  $\text{next}[i] = 0$ .

A single scan through  $t$  suffices to compute *first* and *next*. Insertions into *first* require  $O(\log |\Sigma|)$ , and hence the total runtime of this stage is  $O(n \log |\Sigma|)$ .

### 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  $\sigma$ . Each sequence belonging to  $\mathcal{R}_\sigma$  is a *best*  $\mu$ -approximate  $q$ -sequence over  $\{Q, S\}$  for the chosen  $q = \sigma/2$ . Our aim is to identify all the  $\mu$ -approximate  $q$ -matches of  $r$  in  $t' \in \mathcal{R}_\sigma$ .

This follows the same objectives of stage 2 in [4]. The inputs and outputs of this stage are the same, but what differs is that the error needs to be dealt with. We solve this subproblem using two different approaches: a) Depth First Search (DFS), b) Dynamic Programming (DP).

We first present the  $\lambda()$  subroutine, which is used by both algorithms for finding the local error and then we discuss the two algorithmic approaches.

**$\lambda()$  subroutine** The local error  $\lambda$  simply gives us the error from the exact  $q$  or  $\sigma$  of a value  $r$ , regardless of how many merges of elements of  $t$  are performed to obtain this  $r$ :

$$\lambda(r) = \begin{cases} |r - q|, & \text{if } r \in [q - \mu..q + \mu], & \text{a } \lambda_q\text{-match} \\ |r - 2q|, & \text{if } r \in (2q - 2\mu..2q + 2\mu], & \text{a } \lambda_\sigma\text{-match} \\ +\infty, & \text{otherwise} \end{cases} \quad (2)$$

In Algorithm 1, the call to subroutine  $\lambda()$  also returns the appropriate character (a  $Q$  or an  $S$ ), from which the sequence is built.

---

**Algorithm 1** Calculate  $\lambda$ 


---

```

1: function  $\lambda(r, q, \mu)$ 
2:    $local\_error \leftarrow \infty$ 
3:    $beat\_char \leftarrow \text{"-"}$ 
4:   if  $r < q - \mu$  then
5:      $local\_error \leftarrow \infty$ 
6:   else if  $r \leq q + \mu$  then
7:      $local\_error \leftarrow |r - q|$ 
8:      $beat\_char \leftarrow \text{"Q"}$ 
9:   else if  $r < 2q - 2\mu$  then
10:     $local\_error \leftarrow \infty$ 
11:  else if  $r \leq 2q + 2\mu$  then
12:     $local\_error \leftarrow |r - 2q|$ 
13:     $beat\_char \leftarrow \text{"S"}$ 
14:  return  $local\_error, beat\_char$ 

```

---

In the algorithms presented in this paper, we produce only the *best* sequences and not all possible sequences. So we need appropriate tie-breaking rules to deal with two different merges with the same local error  $\lambda$ .

1. In the case of a tie between two  $Q$ 's or two  $S$ 's, we choose the one with less merges. This is justified in Section 2 and may be better understood by the simple fact that the *ideal* case is for solid  $Q$ 's and  $S$ 's, the  $2^{nd}$  best case is for merges of only 2 intervals, and so on.
2. In the case of a tie between a  $Q$  and an  $S$ , we choose the  $Q$ , i.e. if we have two merges, one that gives us a  $Q$  and one an  $S$ , and they have the same local error  $\lambda$ , we choose the  $Q$ . This is justified by the fact that, according to the definitions, two  $Q$ 's make an  $S$ , but an  $S$  cannot be split into two  $Q$ 's.

**DFS** Algorithm 2 takes a Greedy approach. For each value of  $\sigma$  identified in Stage 1, we produce a set  $R_\sigma$  of sequences. Starting from the position of  $\sigma$ , we first go to the left as far as we can, producing valid  $Q$ 's and  $S$ 's and then do the same to it's right. Given constant  $c$ , we attempt to merge up to  $c$  intervals and choose the best fit, i.e. the merged intervals that minimize the local error  $\lambda$ . If neither a  $Q$  nor an  $S$  can be produced within  $c$  merges, this part of the algorithm terminates.

Note that,  $\mu$ , by definition, cannot exceed  $q/3$ ; so the algorithm reduces  $\mu$  when necessary to adhere to this. Because of this, in these  $\mu$ -approximate versions of the algorithm, we use floating point arithmetic instead of integers.

The execution of the DFS algorithm is shown in the partial example of Figure 3. Note that, sequences which are less than the length of the rhythm can

---

**Algorithm 2** Depth First Search (DFS)
 

---

```

1: function TRANSFORM( $t[1..n], \sigma, \mu, c$ )
2:    $\mathcal{R}_\sigma \leftarrow \{\}$ 
3:    $i \leftarrow \text{data in } \text{first}(\sigma, i) \text{ at } \text{key} = \sigma$ 
4:    $q \leftarrow \sigma/2$ 
5:   if  $\mu > q/3$  then
6:      $\mu \leftarrow q/3$ 
7:   while  $i \geq 0$  do
8:      $x \leftarrow "S"$ 
9:      $iLeftPos \leftarrow i$ 
10:     $pos \leftarrow n$ 
11:     $r \leftarrow 0$ 
12:     $local\_error \leftarrow best\_local\_error \leftarrow \infty$ 
13:     $beat\_char \leftarrow "-"$ 
14:    while  $i > 1$  do ▷ to the left
15:      for  $k \leftarrow 1$  to  $c$  do
16:         $r \leftarrow r + t[i - k + 1]$ 
17:         $(local\_error, local\_char) \leftarrow \lambda(r, q, \mu)$ 
18:        if  $local\_error < best\_local\_error$  then ▷ we've found a new best
19:           $pos \leftarrow i - k$ 
20:           $best\_local\_error \leftarrow local\_error$ 
21:           $beat\_char \leftarrow local\_char$ 
22:          if  $local\_error = 0$  then ▷ we've found THE best
23:            break
24:          if  $best\_local\_error = \infty$  then ▷ no more to the left
25:             $i \leftarrow 0$ 
26:          else
27:            push  $beat\_char$  to the front of  $x$ 
28:             $i \leftarrow iLeftPos \leftarrow pos$ 
29:             $i \leftarrow i + 1$ 
30:             $iRightPos \leftarrow i$ 
31:            while  $i < n$  do ▷ to the right
32:              for  $k \leftarrow 1$  to  $c$  do
33:                 $r \leftarrow r + t[i + k - 1]$ 
34:                 $(local\_error, local\_char) \leftarrow \lambda(r, q, \mu)$ 
35:                if  $local\_error < best\_local\_error$  then ▷ we've found a new best
36:                   $pos \leftarrow i + k$ 
37:                   $best\_local\_error \leftarrow local\_error$ 
38:                   $beat\_char \leftarrow local\_char$ 
39:                  if  $local\_error = 0$  then ▷ we've found THE best
40:                    break
41:                  if  $best\_local\_error = \infty$  then ▷ no more to the right
42:                     $i \leftarrow n$ 
43:                  else
44:                    push  $beat\_char$  to the back of  $x$ 
45:                     $iRightPos \leftarrow i - 1$ 
46:                 $\mathcal{R}_\sigma \leftarrow \mathcal{R}_\sigma \cup \{x\}$  ▷ add sequence to set
47:                 $i \leftarrow next[i]$ 
48:    return  $\mathcal{R}_\sigma$ 

```

---



**Algorithm 3** Dynamic Programming

---

```

1: function TRANSFORM( $t[1..n], i, \sigma, \mu, c$ )
2:    $\mathcal{R}_\sigma \leftarrow \{\}$ 
3:    $q \leftarrow \sigma/2$ 
4:   if  $\mu > q/3$  then
5:      $\mu \leftarrow q/3$ 
6:    $i \leftarrow$  data in first at key =  $\sigma$ 
7:   while  $i > 0$  do
8:      $x \leftarrow$  "S"
9:      $j \leftarrow i$ 
10:     $iLeftPos \leftarrow j$ 
11:    if  $j > 1$  then
12:       $x \leftarrow TransformToLeft(t, j, q, \mu, c)$ 
13:     $j \leftarrow i + 1$ 
14:     $iRightPos \leftarrow j$ 
15:    if  $j < n - 1$  then
16:       $x \leftarrow TransformToRight(t, j, q, \mu, c)$ 
17:     $\mathcal{R}_\sigma \leftarrow \mathcal{R}_\sigma \cup \{x\}$ 
18:     $i \leftarrow next[i]$ 
19:  return  $\mathcal{R}_\sigma$ 

```

---

right (resp. left) search, when  $c$  consecutive  $\kappa(i)$ 's are  $\infty$ . After termination, the sequence is reproduced from the optimal solution back to the starting point, as is common in Dynamic Programming. This is achieved by retrieving the information stored in the three vectors:  $\kappa$ ,  $index$  and  $beat\_char$ .

We apply the recursion on the same example as was done for the DFS approach and show  $\kappa$ ,  $index$  and  $beat\_char$  in Table 1. From these vectors we produce the sequence "SQQQQSQSS" as indicated by the asterisks.

**Table 1.** The DP tables computed by TransformToRight() for  $\sigma = 12$

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
$\kappa$	0	$\infty$	2	1	2	3	3	4	5	4	5	7	4	8	4	6	8	5	5
$index$	-	1	2	3	2	2	2	2	2	2	1	1	3	2	2	1	1	3	1
$beat\_char$	S	-	Q	Q	Q	Q	Q	Q	Q	Q	Q	Q	S	Q	Q	Q	Q	S	S
sequence	*		*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*

It should be clear that, to find each entry in  $\kappa$ ,  $index$  and  $beat\_char$  takes  $c$  time and we have  $n$  such entries, which gives us the same running time for Transform() as for the DFS approach, i.e.  $O(cn)$ . And again, Transform() is called for each element, thus giving us a total runtime of  $O(n^2c)$ .

**Algorithm 4** DP subroutine: to the left

---

```

1: function TRANSFORMTOLEFT( $t[1..n]$ ,  $i$ ,  $q$ ,  $\mu$ ,  $c$ )
2:    $pos \leftarrow n$ ,  $r \leftarrow 0$ 
3:    $\kappa[1..i-1] \leftarrow \{\infty, \infty, \dots, \infty\}$ 
4:    $\kappa[i] \leftarrow 0.0$ 
5:    $index[1..i] \leftarrow \{\infty, \infty, \dots, \infty\}$ 
6:    $beat\_char[1..i] \leftarrow \{‘-’, ‘-’, \dots, ‘-’\}$ 
7:   for  $j \leftarrow i-1$  downto 0 do
8:     for  $k \leftarrow 1$  to  $c$  do
9:       if  $j+k \leq i$  then
10:         $r \leftarrow t[j+k-1] - t[j-1]$ 
11:         $(\kappa', local\_char) \leftarrow \kappa[j+k-1] + \lambda(r, q, \mu)$ 
12:        if  $\kappa' < \kappa[j-1]$  then
13:           $\kappa[j-1] \leftarrow \kappa'$ 
14:           $index[j-1] \leftarrow k$ 
15:           $beat\_char[j-1] \leftarrow local\_char$ 
16:           $pos \leftarrow j-1$ 
17:        if  $\kappa[j..j-c] = \infty$  then
18:          break ▷ no more  $Q$ 's to the left
19:        for  $m \leftarrow pos$  to  $i-1$  do
20:           $m \leftarrow m + index[m]$ 
21:          push  $beat\_char$  to back of  $y$ 
22:          push  $y$  to front of  $x$ 
23:        return  $x$ 

```

---

The problematic example in the DFS section is solved using DP and the results are shown in Table 2. The resultant sequence, i.e. “SSS”, is the correct one.

**Table 2.** The problematic example, correctly solved by DP

	0	1	2	3	4	5	6	7	8
$\kappa$	0	0	2	0	2	0	1	$\infty$	1
$index$	-	1	1	2	2	3	5	-	6
$beat\_char$	-	S	S	Q	S	S	Q	-	S
sequence		*			*			*	

The two different approaches have the same theoretical running times, but the DP’s hidden constant is larger than that of the DFS’s. So, in practice, the implementation of the DFS algorithm is faster, and it is believed that, despite the problems at boundaries as discussed above, such problems are rarely encountered in real pieces of music as we intend to show in experimental results in future works.

**Algorithm 5** DP subroutine: to the right

---

```

1: function TRANSFORMTORIGHT( $t[1..n], i, q, \mu, c$ )
2:    $pos \leftarrow i, r \leftarrow 0$ 
3:    $\kappa[i..n] \leftarrow \{\infty, \infty, \dots, \infty\}$ 
4:    $\kappa[i] \leftarrow 0.0$ 
5:    $index[i..n] \leftarrow \{\infty, \infty, \dots, \infty\}$ 
6:    $beat\_char[i..n] \leftarrow \{‘-’, ‘-’, \dots, ‘-’\}$ 
7:   for  $j \leftarrow i$  to  $n$  do
8:     for  $k \leftarrow 1$  to  $c$  do
9:       if  $j - k \leq i$  then
10:         $r \leftarrow t[j] - t[j - k]$ 
11:         $(\kappa', local\_char) \leftarrow \kappa[j - k - 1] + \lambda(r, q, \mu)$ 
12:        if  $\kappa' < \kappa[j - 1]$  then
13:           $\kappa[j - 1] \leftarrow \kappa'$ 
14:           $index[j - 1] \leftarrow k$ 
15:           $beat\_char[j - 1] \leftarrow local\_char$ 
16:           $pos \leftarrow j - 1$ 
17:        if  $\kappa[j..j - c] = \infty$  then
18:          break ▷ no more  $Q$ 's to the right
19:        for  $m \leftarrow pos$  downto  $i$  do
20:           $m \leftarrow m - index[m]$ 
21:          push  $beat\_char$  to front of  $y$ 
22:          push  $y$  to back of  $x$ 
23:        return  $x$ 

```

---

**3.3 Stage 3 – Find the Matchings**

In this stage we consider each  $t' \in \mathcal{R}_\sigma$ , for all valid values of  $\sigma$  and identify all the  $q$ -matches of  $r$  in  $t'$ . To do that efficiently we exploit a bit-masking technique. The details are given in [4]. The runtime of this stage is  $O(n \log H \times m/w)$ , where  $w$  is the size of the word of the target machine,  $H$  is the size of the largest value for  $\sigma$  and  $m$  is the size of the rhythm  $r$ .

**3.4 Stage 4 – Find the Cover**

In this stage, we have sets of  $q$ -occurrence lists corresponding to the  $q$ -matches for the  $r$  in  $t$ . From these, we find the corresponding  $q$ -covers and identify the best such cover. This can be done in  $O(n \log H)$ .

The total running time of the algorithm is thus  $O(n^2c)$ , because of Stage 2.

**4 Conclusions**

The task of identifying the rhythmic structure of a piece of music and categorizing it according to a predefined set of rhythms is, within the realm of music information retrieval, a non-trivial task. One of the inherent difficulties in extracting rhythmic structure from music is the presence of errors, intentional or

otherwise. In this paper, we have presented two efficient algorithms that can be used to categorize songs of the ballroom dance genre, taking temporal errors in the transcription into account.

One assumption we have made is that, the rhythm remains constant throughout the duration of the piece of music. This could be relaxed to allow rhythm changes in songs. The new algorithm could have applications on a wider spectrum of musical forms. Furthermore, we plan to implement the algorithms and extend them for a wider type of music and rhythms.

## References

1. Jeff Allen, *The Complete Idiot's Guide to Ballroom Dancing*. Alpha Books, 2002.
2. D. Byrd and E. Isaacson, A Music Representation Requirement Specification for Academia. *The Computer Music Journal*, vol. 27, 2003, pp.43–57.
3. A. L. P. Chen and C. S. Iliopoulos and S. Michalakopoulos and M. S. Rahman. Implementation of Algorithms to Classify Musical Texts According to Rhythms. In *Proceedings of the 4th Sound and Music Computing Conference*, Lefkada, Greece, July 2007.
4. Manolis Christodoulakis, Costas S. Iliopoulos, M. Sohel Rahman and William F. Smyth. Identifying Rhythms in Musical Texts, *International Journal of Foundations of Computer Science*, in press.
5. Nick Collins. Beat Induction and Rhythm Analysis for Live Audio Processing: 1st Year PhD Report.
6. T. Crawford and C.S. Iliopoulos and R. Raman, String Matching Techniques for Musical Similarity and Melody Recognition. *Computing in Musicology*, vol. 11, 1998, pp.227–236.
7. Stephen W. Hainsworth and Malcolm D. Macleod. The Automated Music Transcription Problem.
8. S. Hainsworth and M. Macleod. Automatic bass line transcription from polyphonic music, *In Proc. International Computer Music Conference*, Havana, Cuba, 2001.
9. Peter Howell and Robert West and Ian Cross, *Representing Musical Structure*. Academic Press London, 1991.
10. C. S. Iliopoulos and K. Lemstrom and 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*, 2002, pp.67–76.
11. K. Lemstrom, String Matching Techniques for Music Retrieval. *PhD Thesis, University of Helsinki, Department of Computer Science*, 1998.
12. K. Lemstrom and P. Laine, Musical Information Retrieval Using Musical Parameters, *International Computer Music Conference*, 1998, pp.341–348.
13. K. Lemstrom and J. Tarhio, Detecting Monophonic Patterns Within Polyphonic Sources, *Multimedia Information Access Conference*, vol. 2, 2000, pp.1261–1279.
14. Alan Marsden and Anthony Pople, *Computer Representations and Models in Music*. Academic Press London, 1992.
15. M. Mongeau and D. Sankoff, Comparison of Musical Sequences, *Computers and the Humanities*, vol. 24, 1990, pp.161–175.
16. Alex Moore, *Ballroom Dancing*. Taylor & Francis, 2002.
17. Iroro Orife. Riddim: A Rhythm Analysis And Decomposition Tool Based On Independent Subspace Analysis, 2001. Masters thesis.

18. Eleanor Selfridge-Field, *Beyond MIDI: The Handbook of Musical Codes*. The MIT Press, 1997.
19. D.A. Stech, A Computer Assisted Approach to Micro Analysis of Melodic Lines, *Computers and the Humanities*, vol. 15, 1981, pp.211–221.
20. G. A. Wiggins and E. Miranda and A. Smaill and M. Harris, A Framework for the Evaluation of Music Representation Systems, *The Computer Music Journal*, vol. 17, no. 3, 1993, pp.31–42.
21. Judy Patterson Wright, *Social Dance: Steps to Success*. Human Kinetics, 2002.