Implementation of Algorithms to Classify Musical Texts According to Rhythms

Arbee L. P. Chen*, C.S. Iliopoulos†, S. Michalakopoulos†, M. Sohel Rahman†
*Department of Computer Science, National Chengchi University, Taiwan
†Algorithm Design Group, Department of Computer Science, King's College London, {csi, spiros, sohel}@kcl.ac.uk

***Abstract*—An interesting problem in musicology is to classify songs according to rhythms. A rhythm is represented by a sequence of "Quick" (Q) and "Slow" (S) symbols, which correspond to the (relative) duration of notes, such that $S = 2Q$. Recently, Christodoulakis et al. [16] presented an efficient algorithm that can be used to classify musical texts according to rhythms. In this paper, we implement the above algorithm along with the naive brute force algorithm to solve the same problem. We then analyze the theoretical time complexity bounds with the actual running times achieved by the experiments and compare the results of the two algorithms.**

***Keywords*— algorithms, music information retrieval, pattern matching, quick-slow, rhythm.**

I. 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]. Automatic music classification 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.

Very recently, Christodoulakis et al. [16] considered the problem of classification of a music text by rhythms. In [16], the authors proposed a new framework for the identification of rhythms in a musical text and devised an efficient algorithm for that task. In the sequel, this algorithm can be used for the automatic music classification based on rhythms. In this paper, we are interested in the practical performance of the above algorithm. In particular, we implement the above algorithm along with the naive brute force algorithm to solve the same problem. We then analyze the theoretical time complexity bounds with the actual running times achieved by the experiments and compare the results of the two algorithms.

The paper is organized as follows. In Section II, we briefly review the framework presented in [16] and in Section III, we briefly describe their algorithm. We also suggest some changes in their algorithm in order to avoid some data structural overhead and to simplify coding. We present our experimental setting along with the results in Section IV. Finally, we briefly conclude in Section V.

II. Background

In [16], a new model for song classification based on dancing rhythms were presented. For the sake of completeness, in this section, we briefly review the notations and definitions used in [16].

The musical sequences (e.g. a song) can be considered to 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. The formal definition of a musical sequence is as follows:

Definition 1: A *musical sequence* t, is a string $t = t[1]t[2]\ldots t[n]$, where $t[i] \in \mathbb{N}^+$, for all $1 \leq i \leq n$. Here each $t[i], 1 \leq i \leq n$ represents the duration of the consecutive musical events.

Example 1: Consider a music signal having 5 musical events occurring at 0th, 50th, 100th, 200th and 240th milliseconds. Then $t = [50, 50, 100, 40]$ is the musical sequence representing the above musical signal.

In this particular setting, rhythms are assumed to consist of a number of intervals. In particular, there are two types of intervals in a 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 rhythms, cha-cha, foxtrot and jive are represented as shown in table I.

The formal definition of a rhythm is as follows:

Definition 2: A *rhythm* r is a string $r = r[1]r[2]\ldots r[m]$, 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 length of the interval represented by an S is double the length of the interval

TABLE I
A Few Dancing Rhythms

cha-cha	$SSQQSSSQQS$
foxtrot	$SSQQSSQQ$
jive	$SSQQSQQS$

represented by Q. However, the exact length of Q or S is not a priori known. Note that, the *alphabet* for the musical text and that of the rhythm differs from each other: the *alphabet* for the musical text is $\Sigma = \{t[i] \mid 1 \leq i \leq n\}$, whereas the alphabet for the rhythm is $\Sigma_r = \{Q, S\}$. The notion of match and cover in this framework is extended from the notion of classical string matching in the following way.

Definition 3: Let Q represents intervals of size $q \in \mathbb{N}^+$ milliseconds, and S represent intervals of size $2q$. Then Q is said to *q-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 q-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 $i = i'$.

Example 2: Consider the musical sequence shown in Fig. 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 $\sum_{i=7}^{9} t[i] = 2q$.

Definition 4: 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 $1 \leq j \leq m$

Example 3: For instance, the rhythm $r = QSS$, q-matches with $t[1..5]$ as well as with $t[5..8]$, in Fig. 2, for $q = 50$.

One very important fact is that reporting only the start (or end) position of a q-matches of a rhythm may not convey the complete information. This can be easily realized from the difference in length of the portion of t that q-matched with the r in the above two instances. Therefore we have to 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 $Occ_q = \{(1, 5), (5, 8)\}$.

Definition 5: 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 $2 \leq \ell \leq k$

Example 4: In our example, Fig. 2, the rhythm $r = QSS$ q-covers $t[1..8]$ for $q = 50$.

III. Maximal Coverability Algorithm

The *maximal coverability* problem, can be formally defined as follows [16]:

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.

The following restriction was 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 $r[j] = t[k] = 2q$, $i \leq k \leq i'$, for some value of q.

The justification of the above restriction follows from the following pathological example: consider a musical sequence consisting of a single tone repeated every 1ms, $t = 111 \ldots 1$. 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 [16], from musical point of view, it is meaningful to have at least one event that is solid.

The algorithm presented in [16] works in the 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.

In the rest of this section we discuss these stages along with implementation details of the algorithm.

A. 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

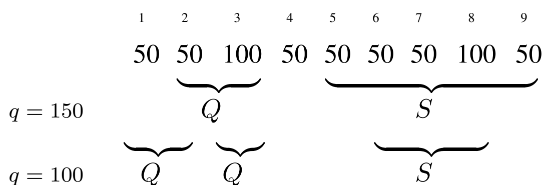


Fig. 1. Q- and S-matching in musical sequences.

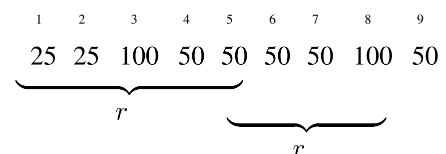


Fig. 2. q-matches of $r = QSS$ in t, for $q = 50$.

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 occurrences of all the symbols and storing them in appropriate data structures. This can be 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

- (σ, i) is an item in tree $first$, with $key = \sigma$ and $data = i$, if and only if the leftmost occurrence of the symbol σ appears at position i of $t[1..n]$
- $next[i] = j$ 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 $first$ and $next$. Insertions into $first$ require $O(\log |\Sigma|)$, hence the total runtime of this stage is $O(n \log |\Sigma|)$.

The data structures $first$ and $last$ were implemented using the STL associative container *map*, which in turn is normally implemented as a balanced search tree. This ensures a $O(\log |\Sigma|)$ running time for lookups of particular σ's and has the added advantage that it keeps the elements sorted, a fact which is useful in the next stage. In particular, this helps us to avoid using the complex range maxima query data structure and associated overhead.

Note that the size of the alphabet Σ is much smaller on average than the size of the piece of music, i.e. $|\Sigma| \ll n$. As our experimental results show (see section IV), in the actual social dance music pieces we used, $|\Sigma| \approx 10$, while a 5 minute song can have thousands of rhythmic events. So, $\log |\Sigma|$ can be assumed to be constant with respect to n.

B. 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 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).

For each occurrence of the current symbol $\sigma = 2q = S$, we try to convert the area surrounding that S into sequences or a *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. Fig. 3 shows an example of the transformation process.

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.

The running time of Stage 2 depends on the total length of all the sequences produced in this stage. The following lemmas from [16] give us the bound on number of sequences produced in this stage and the resulting running time of Stage 2.

Lemma 1: 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.

Lemma 2: The running time of Stage 2 is $O(n \log H)$, where H is the maximum value in t.

To ensure the theoretical running time of $O(n \log H)$ the algorithm in [16] uses the complex range maxima query data structure of [14], [15]. However, we prefer not to use this data structure to avoid the associated data structural overhead and complex coding. What we do is as follows. We use the STL map once again, which is a balanced search tree. The tree is of the form $(key, data) = (\langle startPos, endPos\rangle, sequence)$, e.g. $(\langle 1,5\rangle, \text{“}QQSQQ\text{”})$,
$(\langle 7,9\rangle, \text{“}QSQ\text{”})$ ...etc. This ensures that we have the sequences in sorted by start position order. But this adds to the asymptotic run time by $O(\log |\mathcal{R}_\Sigma|)$. However, as it turns out from our experiments, the resulting increase in running time is almost negligible.

C. 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.

To implement the bitwise operations we have used the C++ *bitset* data structure. This ensures minimal storage (each bit is stored as one bit) and speed since all operations used in this algorithm are constant (AND, OR, and left and right bit shifts). The total running time of this stage as deduced in [16] is $O(|t''| \times |r'|/w) = O(n \log H\ m/w)$ where w is the size of the word of the

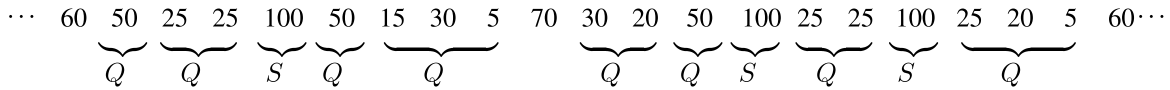


Fig. 3. Transforming the area around $t[5] = S = 100$ and then around $t[14] = S = 100$.

target machine. This follows from the fact that we have to do the above procedure for every sequence produced in Stage 2. One final remark is that, as our experiments suggest, the actual number of sequences used in this stage is far less than $O(n \log H)$. This is because we can ignore all the sequences that are less than m in length which is done in the implementation to achieve speedup.

D. 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 with $q = \sigma/2$. 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. This can easily be done by checking, respectively, the end and start positions of consecutive occurrences. Also, we maintain a global variable to keep track of the longest cover so far. It is easy to observe that the running time of this stage can't exceed $O(n \log H)$.

IV. Performance of the Algorithm

We have implemented and tested the algorithm of [16] (referred to as CIRS Algorithm henceforth) and the naive brute-force algorithm using pieces of music converted from MIDI to plain text. The tests were run in a Windows environment with an Intel Celeron M processor of 1.60GHz and 512MB RAM. The implementation was done in C++ using the STL.

The tests were carried out on dance rhythms that can be categorized as belonging to the ballroom dance variety. The definitive list of rhythms that can be said to belong to this category is still under debate, with various schools and regions disagreeing. We have chosen 9 of the most popular rhythms, listed in table II and have performed our program runs using each rhythm separately. The results are plotted in Fig. 4 to 6.

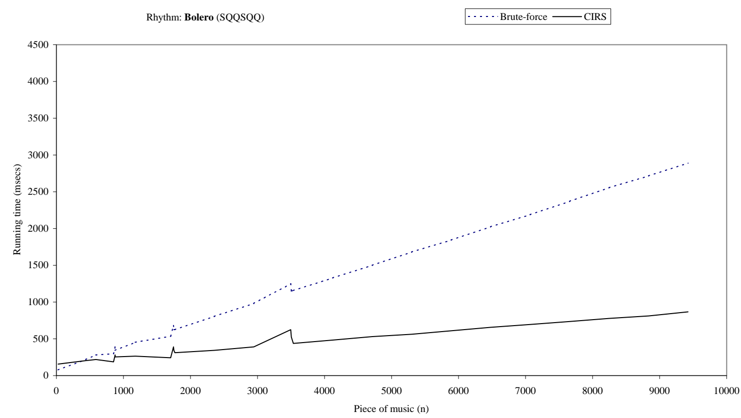
We have presented separate graphs for each rhythm. In the figures, the pieces of music are represented on the X axis. Where $n = 1000$ means there were 1000 rhythmic events in that song and this corresponds roughly to a 4 minute song. The longest songs we ran the algorithms on are around 17 minutes long. The Y axis shows the running time, in milliseconds. Based on the graphs presented we have the following observations.

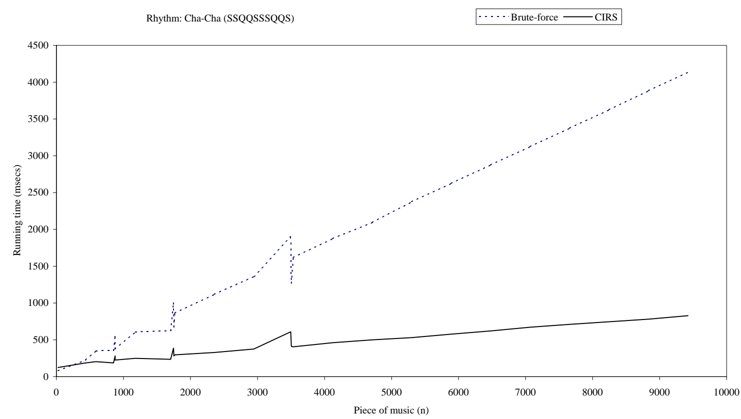
TABLE II
Dance Rhythms and their Representations

1	Bolero	$SQQSQQ$
2	Cha-Cha	$SSQQSSSQQS$
3	Foxtrot	$SSQQSSQQ$
4	Jive	$SSQQSQQS$
5	Mambo/Salsa	$QQSQQS$
6	Quickstep	$SQQSSQQS$
7	Rumba	$SQSSQ$
8	Tango	$SSQQS$
9	Waltz	SSS

1) It is easy to observe that the behaviour of CIRS algorithm is almost linear. This observation is backed by the theoretical running time along with the facts that in practical cases the size of the dance rhythms are quite small.
2) We observe that, for very small values of n, i.e. music size, the brute force algorithm outperforms the CIRS algorithm. This behavior is also expected and easy to understand as follows. Recall that the Brute-force algorithm basically compares the rhythm against the music sequence at every position. There are no other significant overheads. Whereas, CIRS algorithm sets up the data structures and has four stages regardless of the size and nature of the music sequence and rhythms. To give more detail, the brute-force algorithm starts with only one call to a function called NaiveSearch(), which doesn't need any sophisticated data structures. The CIRS, on the other hand, calls FindOccurrences() (stage 1), Transform() (stage 2) and FindMatch() (stage 3). Both algorithms, finally, call the same function UpdateBestCover(). Therefore, these functional and data structural overheads of CIRS makes it inferior to the brute force algorithm in cases where the music sequence is quite small. To get a better understanding of this particular situation we also performed experiments with short and long made up rhythms (the graphs are not shown here for space constraints) and we tried to pinpoint the value up to which the brute force algorithm outperforms CIRS algorithm. We have found that for larger rhythms, this value is around 300 whereas for shorter rhythms this is near 500.
3) A final observation on the graphs concerns the near vertical lines we get on all the graphs at certain point, e.g. for n around 3500. This is because we are testing on various pieces of music of similar length and although the rhythm and thus the length of the string m remains the same, the nature of the pieces of music is such that the algorithms speed will vary according to the number of direct comparisons and occurrences of the rhythm string in the music string.

Fig. 4. Performance of Dance Rhythms: Bolero, ChaCha and foxtrot





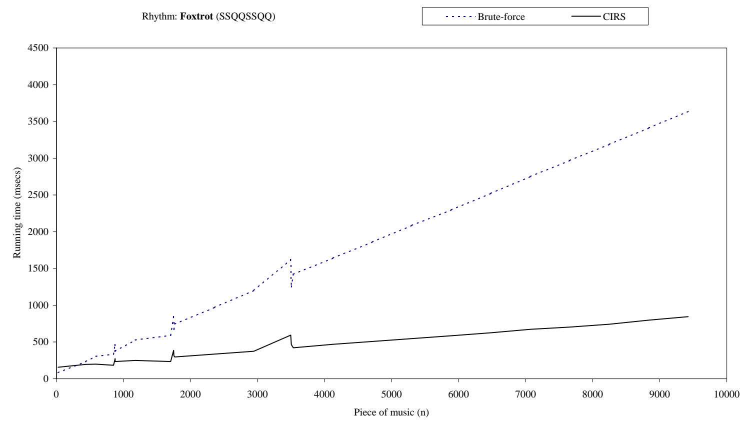
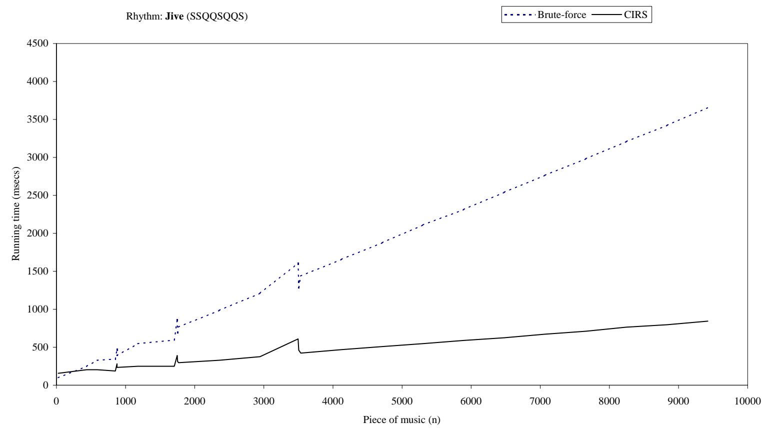
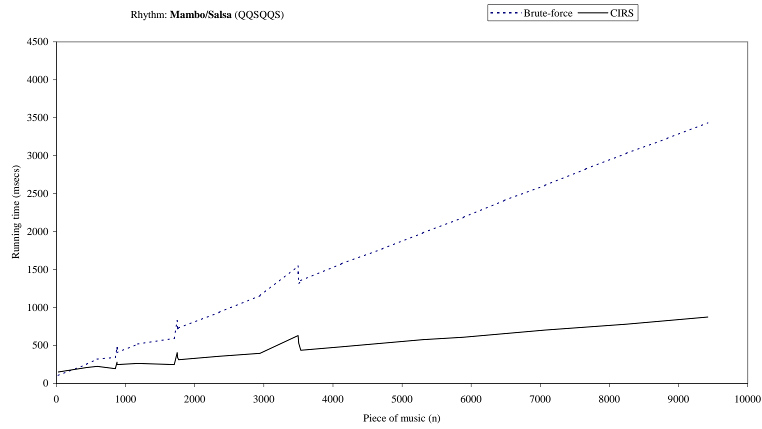


Fig. 5. Performance of Dance Rhythms: Jive, Mambo and Quickstep





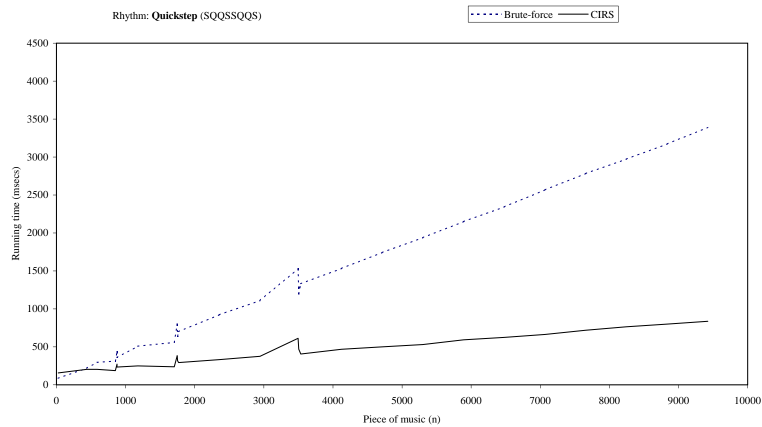
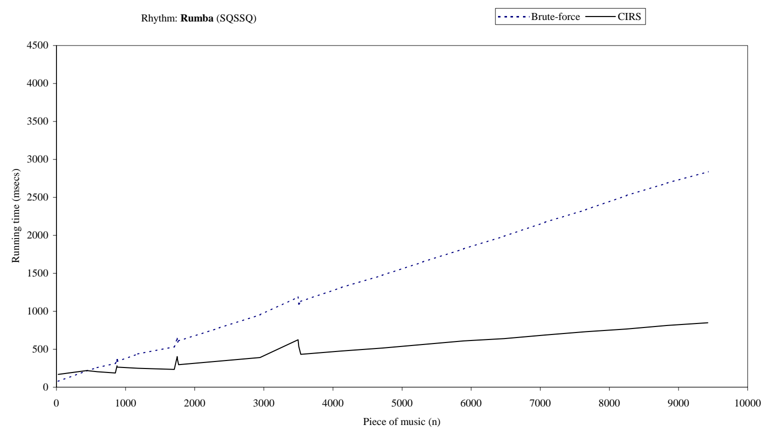
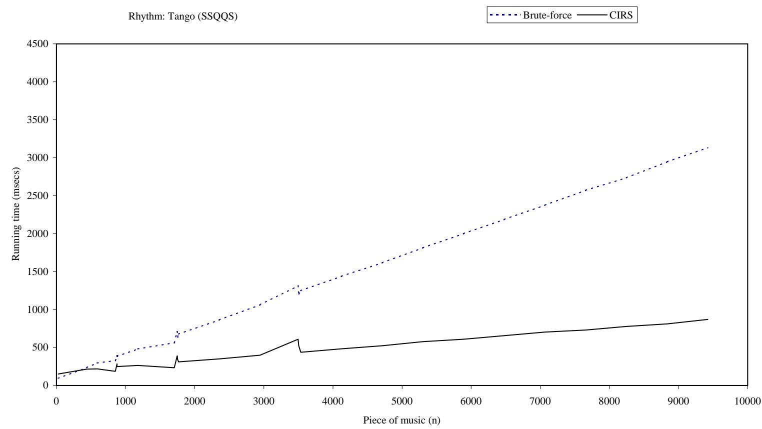
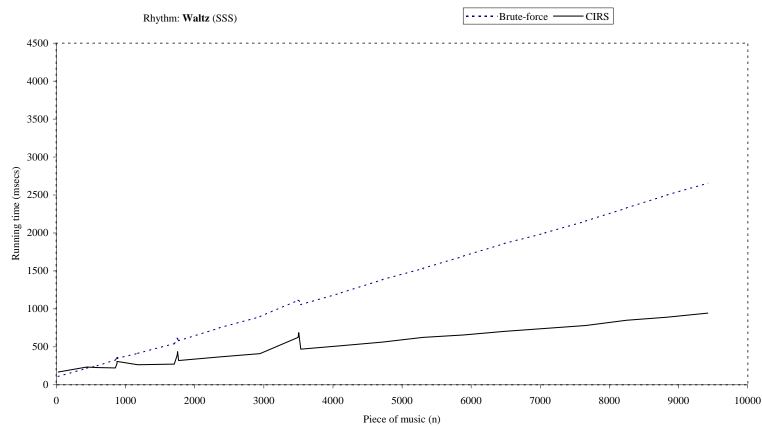


Fig. 6. Performance of Dance Rhythms: Rumba, Tango and Waltz







V. Conclusions

In this paper we have considered the problem of the automated classification of songs according to rhythms from practical point of view. We have implemented the CIRS algorithm [16] for identifying musical texts according to rhythms along with the naive brute force algorithm to solve the same problem. We have then analyzed the theoretical time complexity bounds with the actual running times achieved by the experiments and compare the results of the two algorithms.

To the best of our knowledge the CIRS algorithm [16] is the first attempt to identify musical texts according to rhythms and in this paper we have made the effort to analyze the algorithm experimentally. Due to the absence of any other algorithms with a similar goal in the literature, we have only compared CIRS with a naive brute force algorithm. We have also suggested some changes in the original CIRS algorithm and discussed the justifications of those changes. It would be interesting to see whether the CIRS algorithm could be improved both theoretically and experimentally. Another interesting research direction could be to investigate the case where the assumption of S being double the duration of Q is relaxed.

Acknowledgement

C.S. Iliopoulos is supported by EPSRC and Royal Society grants. S. Michalakopoulos is supported by an EPSRC studentship. M. Sohel Rahman is supported by the Commonwealth Scholarship Commission in the UK under the Commonwealth Scholarship and Fellowship Plan (CSFP) and he is on leave from Department of CSE, BUET, Dhaka-1000, Bangladesh.

References

[1] Alexander 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*, vol. 27, 2003, pp.43–57.

[3] 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.

[4] Peter Howell and Robert West and Ian Cross, *Representing Musical Structure*. Academic Press London, 1991.

[5] 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.

[6] K. Lemstrom, String Matching Techniques for Music Retrieval. *PhD Thesis, University of Helsinki, Department of Computer Science*, 1998.

[7] K. Lemstrom and P. Laine, Musical Information Retrieval Using Musical Parameters, *International Computer Music Conference*, 1998, pp.341–348.

[8] K. Lemstrom and J. Tarhio, Detecting Monophonic Patterns Within Polyphonic Sources, *Multimedia Information Access Conference*, vol. 2, 2000, pp.1261–1279.

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

[10] M. Mongeau and D. Sankoff, Comparison of Musical Sequences, *Computers and the Humanities*, vol. 24, 1990, pp.161–175.

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

[12] D.A. Stech, A Computer Assisted Approach to Micro Analysis of Melodic Lines, *Computers and the Humanities*, vol. 15, 1981, pp.211–221.

[13] 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.

[14] H. Gabow and J. Bentley and R. Tarjan, Scaling and Related Techniques for Geometry Problems, *Symposium on the Theory of Computing (STOC)*, 1984, pp.135–143.

[15] Michael A. Bender and Martin Farach-Colton. The LCA Problem Revisited, *Latin American Theoretical INformatics (LATIN)*, 2000, pp.88–94.

[16] 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.