MUSIC RETRIEVAL ALGORITHMS FOR THE LEAD SHEET PROBLEM

R. Clifford
Algorithm Group
Dept. Computer Science
King's College, London
WC2R 2LS, UK

R. Groult
LIFAR - ABISS
Faculté des Sciences
76821 M. S. Aignan
France

C. S. Iliopoulos
Algorithm Group
Dept. Computer Science
King's College, London
WC2R 2LS, UK

D. Byrd
School of Music
Indiana University
Bloomington
IN 47405, USA

ABSTRACT

Methods from string and pattern matching have recently been applied to many problems in music retrieval. We consider the so-called *lead sheet problem*, where the harmony, melody, and, usually, bass line are presented separately. This is a common situation in some forms of popular music but is also significant for "classical" music in many cases. A number of different but related musical situations are analysed and efficient algorithms are presented for music retrieval in each one.

1. INTRODUCTION

Recently a new approach to music analysis has been taken that attempts to take advantage of the tools of string and pattern matching from computer science. This has required new formalisations of many problems in music retrieval and in some cases a loss in the richness of the original musical source. As a result, arguably the most important innovations have been in increasing the sophistication of the way computational music analysis problems are represented. For example, where previously music was largely assumed to be monophonic, data structures and algorithms for polyphonic music are now commonplace. Furthermore, distinct representations for *voiced* and *unvoiced* polyphonic music allow more appropriate methods to be applied to each. For situations where pitch and rhythm alone are not sufficient, new methods for searching in sets of high dimensional data straddle the border between computational geometry and stringology. Perhaps even more importantly, formal translations of the concept of *musical similarity* have been extended away from their previous form which was largely inspired by text editing and bioinformatics.

We focus here on music that can be considered in two separate but related parts. First, the melody is given as a monophonic sequence of notes. Accompanying it is the harmony presented as a sequence of chords; usually which note of each chord is the lowest (the bass line) is specified as well. The sequence of chords can either be given explicitly, as is usually the case in lead sheets for popular music, or it can be implicit, the usual case in a "classical" piece, for example. The task of music retrieval using data of the form described is termed the *lead sheet problem*. We present four different variants on the lead sheet problem and give efficient algorithmic solutions for each.

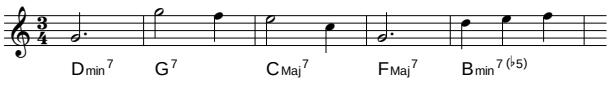

Figure 1. Chord symbols in an excerpt from Alice in Wonderland

Readers who are not familiar with basic concepts of music theory should consult a standard text such as [4].

2. DEFINITIONS

A *lead sheet* generally accompanies the melody only with chord symbols that do not explicitly indicate the bass line; but, in a lead sheet (as distinct from traditional music notation), a chord symbol is usually taken as implying root position by default. In Figure 1–the first few measures of "Alice in Wonderland"–the labels written under the staff such as "Dmin7" and "G7" are chord symbols. If the chords are in root position, the bass (lowest) note of the first chord is D and of the second chord, G.

The subject of musical representation for use in computer applications has been discussed and debated extensively over the past 30 years, if not longer (see for example, [10, 2, 7, 9, 11, 3]). For the rest of the paper, we consider only traditional classical music notation; but all of the ideas apply to lead sheets, and many can readily be extended to other music representations.

We use a simple representation system for harmonic progression and encode all notes in terms of pitch classes. Each chord can be represented using the following notation:

- `PC = pitch class` ($C = 0$, $D\flat = 1$, $D = 2, \ldots$)
- Write $1; 5, 8$, for example, for a chord in which the lowest note is `PC 1`; the chord also includes `PC 5` and `PC 8` in any order.

By using pitch classes we represent notes that that are separated by an exact number of octaves in the same way. Therefore, there are a total of 12 possible pitch classes corresponding to the notes in the chromatic scale. Each

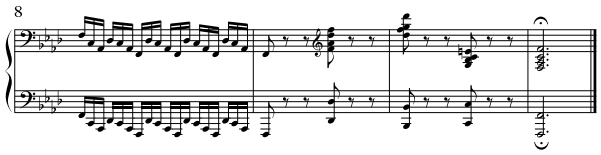

Figure 2. Ending of Chopin's Ballade in F minor, including the final four chords

Melody	PC	Chord Name	Representation
F	5	$D\flat$ major	$\{1;5,8\}$
$D\flat$	1	G half-dim $6-5$	$\{10;1,5,7\}$
E	4	maj-min 7th	$\{0;4,7,10\}$
F	5	F minor	$\{5;0,8\}$

Table 1. A representation of the last four chords of Figure 2

pitch class is also only represented once per chord whether or not the some note occurs in different octaves. Figure 2 shows the final cadence of Chopin's Ballade no. 4 in F minor, Op. 52. The cadence ends with a sequence of four chords, each accompanying one note of the melody. Our representation separates the melody from the harmony and is given in Table 1.

Some basic definitions are required to allow us to formalise the musical representation.

Definition 1. *We call a sequence of sets of characters a* set string. *Throughout we say that the number of sets in a pattern set string p equals m and the number of sets in a text set string t equals n. As the alphabet size σ, is bounded by a constant then $n = O(|t|)$ and $m = O(|p|)$.*

Definition 2. *We say that a set string p* occurs *in set string t if $\exists j : 1 \leq j \leq n - m + 1$ such that $\forall i : 1 \leq i \leq m$ $p_i \subseteq t_{i+j-1}$. In other words, all the notes in chords in p occur in the corresponding chords of t but there may be chords in t that have "extra" notes which do not occur in p.*

Definition 3. *Consider a set string t with a subset of each set t_j identified as the* lead *set. We call such a sequence an* LN string.

Definition 4. *Consider an LN string t. Assume there is some (possibly different) total ordering applied to the characters of each of the lead sets of t. We call such a sequence an* OLN string.

Definition 5. *For any LN or OLN string t, consider the unique set string l with the property that $\forall j : 1 \leq j \leq n$ $l_j =$ the lead set of t_j. We call such a set string the* (ordered) lead set string *of t. Consider also the unique set string nl with the property that $\forall j : 1 \leq j \leq n$ $nl_j = t_j \setminus l_j$. We call such a set string the* (ordered) non-lead set string *of t. We write $t = (l, nl)$ when we need to refer to the lead and non-lead set strings of t.*

Definition 6. *Consider a pattern p and a text t and an order preserving function $f : \{1, \ldots, m\} \rightarrow \{j, \ldots, n\}$. If $p_i = t_{f\{i\}}$ for all $1 \leq i \leq m$ then we call f an* alignment between p and $t[j \ldots n]$. *If furthermore $max_i(f(i+1) - f(i)) \leq \alpha$ then we say that f is an α-alignment. Where p_i and $t_{f(i)}$ are ordered sets then we require both that $p_i = t_{f\{i\}}$ and that the ordering of their elements is the same.*

3. PROBLEMS AND SOLUTIONS

Given a musical pattern and text expressed in terms of a melody and corresponding harmony, the general task is to find all positions in the text where there is a *match*. The exact definition of a match and the type of data that is used as input will determine the algorithm that we propose. We describe four main problems.

3.1. LN string matching

The input pattern and text are split into lead set strings, which correspond to the melody, and non-lead set strings, which correspond to the harmony. The task is to find all positions in the text where both of the following conditions are satisfied:

1. There is an exact match of the melody of the pattern and the melody of text and
2. The harmony of the pattern is *included* in the harmony of the text.

For our purposes we define harmonic inclusion to require that the pitch classes at each position of the pattern are a subset of those in the corresponding position of the text. The problem is expressed more formally as follows:

Problem 3.1. *Consider an LN pattern $p = (pl, pnl)$ and an LN text $t = (tl, tnl)$. Find all positions j that satisfy the following conditions:*

1. p occurs at position j of t (see Definition 2)
2. $pl = tl[j \ldots j + m - 1]$

The special case where the lead sets only have one member each is an instance of this problem. This corresponds to the situation where either the pattern or text (or both) is monophonic.

Solution

We first encode pl, pnl, tl and tnl as strings of bit-vectors called pl', pnl', tl' and tnl' respectively. Each bit-vector represents a set of pitch classes and is defined as follows:

- Let the nth bit of the bit-vector be set to 1 if the pitch class n is in the set. Set all remaining bits to 0.

It is clear that each bit-vector has length σ bits. The following steps are sufficient to solve Problem 3.1:

1. Construct an array A with the property that $A[j] = 1$ if $pl' = tl'[j \ldots j+m-1]$. Let $A[j] = 0$ otherwise. Let a be the number of 1s in the array A.

2. Construct an array B with the property that

$$B[j] = \begin{cases} 1 & \text{if } pnl' \subseteq tnl'[j \ldots j+m-1] \text{ and} \\ & A[j] = 1 \\ 0 & \text{otherwise} \end{cases}$$

Step 1 takes $O(\sigma n)$ time using standard exact string matching techniques (for example [1, 8, 5]). The time required in Step 2 for each j is $O(\sigma m)$, as we can check each set in pnl' in turn ($pnl'[i]$ has at most σ elements). The total running time of Step 2 is therefore $O(\sigma am)$. This is because we can restrict ourselves to only checking every position j of tnl' for which $A[j] = 1$. The overall method is shown in Algorithm 3.2.

The position of the 1s in array B give the solution that is required. The overall time complexity is $O(\sigma nm)$[1] as in the worst case $a = n$. However, in our case σ is a constant and in real musical data a is likely to be very small. This means that the running time will likely be closer to linear time in practice (see Section 4).

Algorithm 3.2. *LN matching* (p, t)
▷***Input:*** *pattern* and *text* both LN strings
▷***Output:*** All locations where *pattern* occurs in *text*
Begin
lmatch ←*Boyer-Moore*(pl, tl)
for i **in** *lmatch* **do**
match$[i]$ =IsSubset(pnl, tnl, i)
End

3.2. OLN string matching

The input pattern and text are split into lead and non-lead sets as before. However, in this case we consider that the order of the notes in the lead sets has to be preserved for there to be a match of the melody.

Problem 3.3. *Consider an OLN pattern* $p = (pl, pnl)$ *and an OLN text* $t = (tl, tnl)$. *Find all positions* j *that satisfy the following conditions:*

1. p occurs at position j of t

2. $pl = tl[j \ldots j+m-1]$

3. $\forall i : 1 \leq i \leq m$ the ordering imposed on the sets $pl[i]$ and $tl[j+i-1]$ is the same.

[1]Recall that the total size of the input data is in fact $O(\sigma(n+m))$ not $O(n+m)$. This is because n and m are the number of sets in the input and each set can have σ elements. Therefore, the complexity $O(\sigma nm)$ is in fact better than the $O(\sigma^2 nm)$ one would have expected if the sizes of the input set strings had simply been multiplied.

Solution

As the lead sets are ordered they must be encoded differently to the non-lead sets. We encode the data as follows:

- Encode pnl and tnl as strings of bit-vectors called pnl' and tnl' respectively, using the same encoding as in Problem 3.1.

- As the lead sets are ordered, each element in a particular set will have a *rank* that corresponds to its position in the ordering. For example, in the ordered set $7, 1, 4$, the rank of 7 is 1, the rank of 1 is 2 and the rank of 4 is 3. Let $rank(e)$ be the rank of element e (with respect to a particular ordered set). We encode each element e in each ordered set as an integer $e' = \sigma(rank(e) - 1) + e$. For example, the ordered set $7, 1, 4$ would be transformed to $\{7, 13, 28\}$ if $\sigma = 12$. This encoding is always unique as $e \in 1 \ldots \sigma$. Let pl' be a string of integers consisting of the encoded elements of each set $pl[i]$ in order. We define tl' in the same way as a string of encoded elements from tl.

Steps 2 combined with Step 3 can be solved by applying a linear time pattern matching algorithm to the expanded arrays pl' and tl'. The total length of tl' is σn and so the total time required for these two steps is $O(\sigma n)$. The running time of Step 1 is $O(\sigma am)$, if we restrict ourselves to checking every position j of tnl' for which there is match in Step 2. This is $O(\sigma nm)$ in the worst case but will be closer to linear time in practice (see Section 4).

3.3. LN matching with α- bounded gaps

Matches of the melody and harmony in the pattern and text may be obscured by the presence of extra notes or chords in the text. In this formulation we allow *gaps* to be inserted into the pattern when attempting to find a match. In order to ensure that the match still has musical relevance the size of the gap is bounded by an integer α.

Problem 3.4. *Consider an LN pattern* $p = (pl, pnl)$, *an LN text* $t = (tl, tnl)$ *and an integer bound* α. *Find all positions* j *that satisfy the following conditions:*

1. There is an α-alignment f between pl and $tl[j \ldots n]$.

2. $\forall i : 1 \leq i \leq m\ p[i] \subseteq t[f(i)]$

In other words, we want to find matches between p and t allowing gaps of size up to α in the alignment.

Solution

The basic idea of the algorithm is to compute α-alignments of increasing prefixes of pattern p in text t. This is achieved by dynamic programming using a table D with m rows and n columns. The value at $D_{i,j}$ contains the last index in t that $p_1 \ldots p_i$ has successfully been aligned with or 0

if the gap to the last successful alignment is larger than α. We define $p_i \cong t_j$ to mean that p_i matches t_j.

$$D_{i,j} = \begin{cases} j & \text{if } p_i \cong t_j, j - D_{i-1,j-1} \leq \alpha + 1, \\ & D_{i-1,j-1} > 0 \\ D_{i,j-1} & \text{if } p_i \ncong t_j \; j - D_{i-1,j-1} < \alpha + 1 \\ D_{i,j-1} & \text{if } p_i \cong t_j, j - D_{i-1,j-1} > \alpha + 1, \\ & j - D_{i,j-1} < \alpha + 1 \\ 0 & \text{otherwise} \end{cases}$$

Boundary conditions for the matrix D are as follows:

$$D_{0,0} = 1, D_{i,0} = 0 \text{ and } D_{0,j} = j$$

The second and third boundary conditions reflect the notion that nothing aligns with the empty string but that the empty string aligns with everything. The first boundary condition is simply by definition. Locations in t for which there is an LN match with p with α-bounded gaps will correspond to the entries in matrix D where $D_{m,j} = j$. These can be found by inspecting the final row of D once it is completed[2]. Algorithm 3.5 gives an overview of the method.

At every entry in matrix D it may be necessary to check if p_i matches t_j and also some previous value in the D. The time required for this is $O(\sigma)$ and so the total time required to construct D is $O(\sigma nm)$. As we regard σ to be a constant, the overall computation time is $O(nm)$.

Algorithm 3.5. *LN matching with α-bounded gaps (p,t)*
▷***Input:*** *pattern* and *text* both LN strings
▷***Output:*** All locations where *pattern* occurs in *text* with gaps bounded by α
Begin
Set boundary conditions for matrix D
for j **in** $\{1 \ldots n\}$ **do**
for i **in** $\{1 \ldots m\}$ **do**
Update entry $D_{i,j}$ following rules above
od
od
Find all entries such that $D_{m,j} = j$
End

3.4. OLN matching with α-bounded gaps

If the order of notes in the lead sets has to be preserved for there to be a match then we can formulate our final pattern matching problem. A match is required between the pattern and text with gap size bounded by α. A further requirement is that the order of notes in the corresponding matching lead sets must be the same.

Problem 3.6. *Consider an OLN pattern p, an OLN text t and a bound α. Find all positions j that satisfy the following conditions:*

1. There is an ordered α-alignment f between pl and $tl[j \ldots n]$.
2. $\forall i : 1 \leq i \leq m \; p[i] \subseteq t[f(i)]$

Note that condition 1 requires that the ordering imposed on $pl[i]$ and $tl[i]$ be the same.

Solution

The method of solution is the same as that for Problem 3.4. We only need to modify the definition of a *match* between p_i and t_j so that the order of the elements in the lead sets are taken into account. The recursion for computing matrix D is defined in the same way and so the overall running time is $O(nm)$ as before. The space requirement can also be reduced to $O(n + m)$ if required as explained in Footnote 2.

4. IMPLEMENTATION AND EXPERIMENTAL RESULTS

Each of the four algorithms was implemented in C and compiled using gcc 3.3.2 with the -O2 optimisation flag. The tests were then run on a 2.40 GHz Pentium 4 processor with 512MB of RAM. Random texts and patterns of different lengths were created using the method described in Section 4.1. Each experiment was repeated 10 times and the average of the running times calculated. The timings given are for the search algorithms only and do not include the time required to create the data.

4.1. Creation of test data

In order to test the different algorithms, random input data is used. Although this form of data is not realistic in a musical sense we expect that the running times given are an accurate indication of what can be expected in practice.

Text: LN matching with/without α-bounded gaps

The lead sets are chosen uniformly and independently at random from the space of possible sets with alphabet size 12. Non-lead set creation are also chosen at random but any elements that are in common with its corresponding lead set are then removed. The result is that there are no elements in common in a lead/non-lead set pair.

Text: OLN matching with/without α-bounded gaps

The ordered lead sets are created by the following algorithm:

1. Uniformly sample a random set s from the space of possible sets with alphabet size 12
2. Uniformly sample a random permutation of the set s

[2]To find the full alignments, as opposed to only the locations in the text where alignments finish, it is necessary to perform a trace-back by reversing the direction of the rules for creating D. The space requirement can also be reduced to $O(n + m)$ by the application of a divide-and-conquer method due to Hirschberg [6].

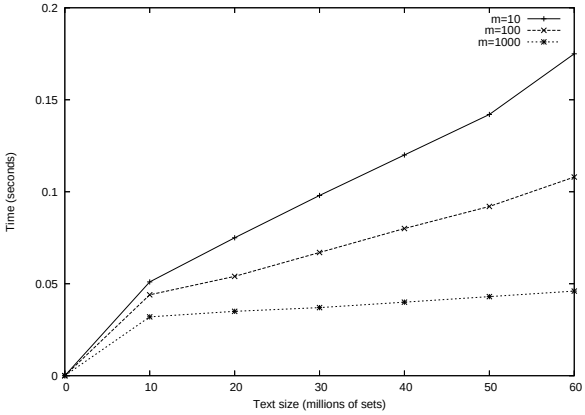

Figure 3. Running times for LN matching using patterns of different lengths

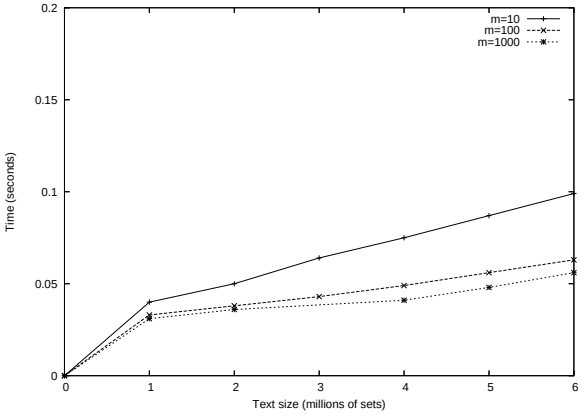

Figure 4. Running times for OLN matching using patterns of different lengths

The result is an ordered set of random size. The lead sets are chosen independently using this scheme. The non-lead sets are chosen in the same way as for LN matching.

Pattern: OLN/LN matching

A random position in the text is chosen and an OLN/LN string of the appropriate length is copied and used as the pattern.

Pattern: OLN/LN matching with α-bounded gaps

A random position in the text is chosen as before. Then the following algorithm is used:

1. Copy current lead and non-lead set from text and add to pattern
2. Skip l positions in the text. l is chosen uniformly at random from the range $[0 \ldots \alpha]$
3. Loop until the pattern has length m

4.2. Running times

Implementations of LN and OLN matching were tested and the results shown in Figures 3 and 4. Pattern lengths of 10, 100 and 1000 sets were used. The running times are practically linear as discussed in Sections 3.1 and 3.2. The Boyer-Moore algorithm, which is typically faster for longer patterns, was implemented for the linear search step. This is the reason for the speedup that can be seen as the pattern length is increased. The limit on the size of the input in each case was the size of available RAM. It is important to note that OLN matching requires considerably more RAM than LN matching as the lead sets must be stored explicitly as arrays of integers rather than as bit-vectors. This is reflected in the maximum input sizes tested for each. The overall running time for both experiments was always less than 0.2 seconds.

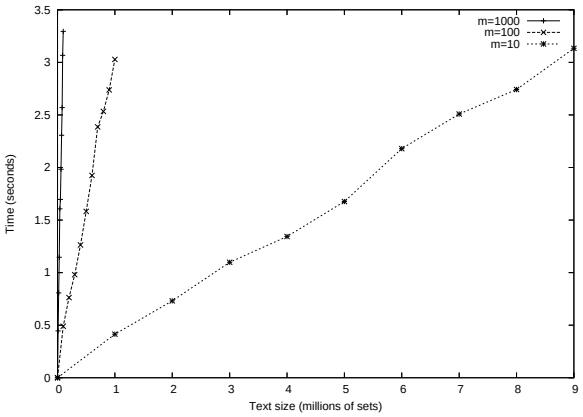

Figure 5. Running times for LN matching with α-bounded gaps

Figures 5 and 6 show the running times for LN and OLN α-bounded matching, respectively. Pattern lengths of 10, 100, 1000 were tested. The running times of these dynamic programming algorithms do not vary with α (this was also confirmed empirically). The results shown are for $\alpha = 2$.

The results show that in practice available RAM and not computational complexity is the limiting factor the size of the input that can be processed. As the dynamic programming method that was implemented has $O(nm)$ space complexity, increasing the pattern length correspondingly decreases the maximum text size that can be searched. An implementation utilising Hirschberg's $O(n+m)$ divide-and-conquer approach would allow much larger databases to be searched at the cost of roughly halving the search speed. As the size of musical databases increases the need for such space saving techniques will undoubtedly become more prominent.

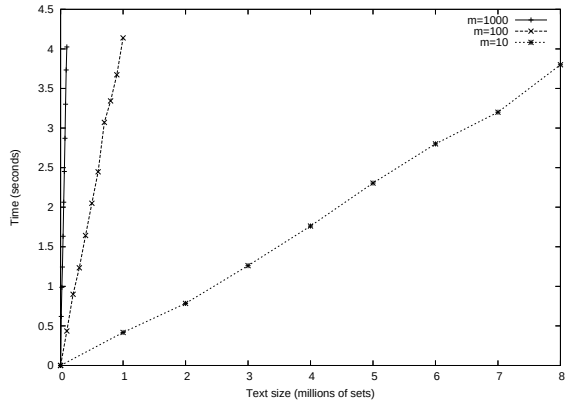

Figure 6. Running times for OLN matching with α-bounded gaps

5. CONCLUSION

Four new algorithms have been given for music retrieval in data where the melody and harmony are presented separately. Each is algorithmically efficient and shown to be very fast in practice, taking at most a few seconds to search the largest dataset. An exciting open problem that would greatly enhance this work is to consider more musically sophisticated concepts of approximation, especially for comparing harmonies. For searching very large databases that will become available in the future, faster algorithms with improved worst case time complexity may also be required.

6. REFERENCES

[1] R. S. Boyer and J. S. Moore. A fast string matching algorithm. *Communications of the ACM*, 20:762–772, 1977.

[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. *Computer Music Journal*, 27(4):43–57, 2003.

[4] J. Clough, J. Conley, and C Boge. *Scales, Intervals, Keys, Triads, Rhythm, and Meter*. W. W. Norton & Company, third edition, 1999.

[5] Z. Galil. On improving the worst case running time of the boyer-moore string matching algorithm. *Communications of the ACM*, 22(9):505–508, 1979.

[6] D. S. Hirschberg. A linear space algorithm for computing longest common subsequences. *Communications of the ACM*, 18:341–343, 1975.

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

[8] D. E. Knuth, J. H. Morris, and V. B. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6:323–350, 1977.

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

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

[11] G. A. Wiggins, E. Miranda, A. Smaill, and M. Harris. A framework for the evaluation of music representation systems. *The Computer Music Journal (CMJ)*, 17(3):31–42, 1993. Machine Tongues series, number XVII; Also from Edinburgh as DAI Research Paper No. 658.