Melodic String Matching Via Interval Consolidation And Fragmentation

Carl Barton[1], Emilios Cambouropoulos[2], Costas S. Iliopoulos[1,3], Zsuzsanna Lipták[4]

[1] King's College London, Dept. of Computer Science, London WC2R 2LS, UK
{carl.barton,csi}@kcl.ac.uk

[2] Aristotle University of Thessaloniki, Dept. of Music Studies, Thessaloniki, Greece
emilios@mus.auth.gr

[3] Curtin University, GPO Box U1987 Perth WA 6845, Australia

[4] University of Verona, Department of Computer Science, Verona, Italy
zsuzsanna.liptak@univr.it

Abstract. In this paper, we address the problem of melodic string matching that enables identification of varied (ornamented) instances of a given melodic pattern. To this aim, a new set of edit distance operations adequate for *pitch interval* strings is introduced. Insertion, deletion and replacement operations are abolished as irrelevant. Consolidation and fragmentation are retained, but adapted to the pitch interval domain, i.e., two or more intervals of one string may be matched to an interval from a second string through consolidation or fragmentation. The melodic interval string matching problem consists of finding all occurrences of a given pattern in a melodic sequence that takes into account exact matches, consolidations and fragmentations of intervals in both the sequence and the pattern. We show some properties of the problem and an algorithm that solves this problem is proposed.

Keywords: melodic pattern matching, string matching, pitch intervals.

1 Introduction

As vast amounts of audio recordings, MIDI files and sheet music become available on the web, efficient Music Information Retrieval (MIR) methods are indispensable for organising and accessing this data, not only in terms of metadata but primarily in terms of musical content (content- based MIR). Most current MIR applications are still in early stages of development and are, usually, not robust, general, or efficient enough. One key problem that hampers attempts to build reliable and robust systems is the lack of explicit structural information in musical data (lack of the equivalent of words or phrases in language). Content-based MIR systems commonly operate on primitive

descriptors extracted from audio or on the elementary musical surface (i.e. sequences of note symbols), and do not have access to the kind of rich higher-level musical information that humans use when storing and accessing musical data (in a sense, it is like having text IR systems operating on mere strings of letters without spaces). Extracting musically pertinent features, especially significant repeating melodic patterns as discussed in this paper, can enormously increase effectiveness and efficiency of music information indexing and retrieval systems. Extracting melodies and rhythms from sequences is a difficult problem which has been researched in the past but generally these algorithms work just on notes in a MIDI like representation and not on pitch intervals. This leads to the problem that the methods used are not transposition invariant and can often struggle to take into account some of the natural variations that occur in music sequences.

Ornamentation, embellishment, elaboration, filling in, thinning out, and reduction are common strategies employed by composers in order to generate new musical material that is recognised as being similar to an initial (or underlying) musical pattern. This way musical unity and homogeneity is retained, whilst at the same time, variation and change occur. This interplay between repetition, variation and change makes music 'meaningful' and interesting. Musical passages are often heard as ornamented or reduced versions of other passages. Listeners are capable of discerning common elements between varied musical material primarily through reduction, i.e. identifying 'essential' common characteristics. The capacity of listeners to 'match' varied musical materials is essential to the process of identifying meaningful musical entities such as interesting motifs, themes, melodic and rhythmic patterns, characteristic harmonic progressions, and other memorable musical entities.

Pattern matching methods are commonly employed to capture musical variation, especially melodic variation[3][6][7]. Dynamic programming techniques, often based on various types of edit distance, are used to find patterns in melodic strings. In this paper, we maintain that techniques using standard edit distance operations (replacement, insertion, deletion, along with consolidation and fragmentation) applied on strings of notes are limited and have inherent shortcomings. Instead, we redefine the problem of matching in a way that is appropriate for strings of melodic intervals (not notes). To this aim, we abolish the replacement, insertion and deletion operations, and retain only consolidation and fragmentation operations which are adapted to the interval domain. It is shown that this new definition of the problem of melodic matching enables more reliable matches and is also transposition invariant.

In this paper we consider the 'Melodic String Matching Via Interval Consolidation and Fragmentation' problem and give optimal algorithms that will find all occurrences of a pattern p in a text t allowing for consolidations and fragmentations.

The paper is structured as follows: In Section 1 we introduce the problem, in Section 2 we describe the problem and give preliminaries and present some important properties used in our analysis, in Sections 3-5 we give our algorithms, in Section 6 experimental results and future improvements.

2 The Melodic String Matching Via Interval Consolidation and Fragmentation Problem

Pattern matching methods are commonly employed to capture musical variation (especially melodic variation). Dynamic programming techniques, often based on various types of edit distance, are used to find patterns in melodic strings. The most common edit operations in melodic string matching are insertion (inserting a note), deletion (deleting of a note) and replacement (replacing a note). Mongeau and Sankoff[9] suggest two additional operations: fragmentation (division of a note into multiple notes of the same pitch) and consolidation (combining multiple notes in a single note). The minimum number of operations that are necessary for making two strings identical is called edit distance; this distance is a measure of similarity between the two strings. In Figure 1, the first melody is transformed into the second melody if the notes indicated by an asterisk are deleted/inserted as appropriate.
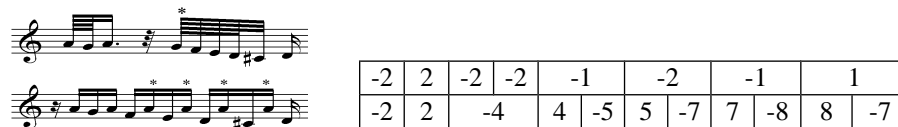

-2	2	-2	-2		-1		-2		-1		1	
-2	2	-4		4	-5	5	-7	7	-8	8	-7	

Figure 1 Beginning of Toccata (a) and theme of Fugue (b) from Bach's *D-minor Toccata and Fugue* BWV 565. Intervals of one excerpt may be fragmented or consolidated in the other excerpt as depicted in the table.

Edit distance is usually applied to strings of pitches. This distance is problematic, however, when applied to strings of pitch intervals. The reason is that the deletion, insertion or replacement of a pitch interval in a melodic sequence changes radically the initial sequence. If for instance, a 2 semitone interval is replaced by a 3 semitone interval, the rest of the melody following this interval is transposed by 1 semitone in a remote key and thus the quality of the melody is altered drastically[3].

The use of edit distance in strings on notes (not intervals) also has some shortcomings. Firstly, in order to account for transpositions it is necessary to transpose the query to 12 different keys and search the target string twelve times. Secondly, it is difficult to define a distance threshold beyond which two strings do not match. If enough edit operations are performed any string can be made identical to any other string – such strings, however, may be very dissimilar. An appropriate distance threshold has to be determined in order for edit distance to account for plausible similarity ratings; determining such a threshold is not straightforward.

In this paper, we address the problem of melodic string matching, introducing a new set of operations that are adequate for pitch interval strings. Insertion, deletion and replacement are abolished as irrelevant. Consolidation and fragmentation are retained but adapted to the pitch interval domain (they can also be applied to durations). That is, two or more intervals of one string may be matched to an interval from a second string through consolidation or fragmentation (see table in Figure 1).

Working with intervals means melodic matching is transposition-invariant. Additionally, matching is confined by the consolidation and fragmentation operations (the only threshold necessary is the maximum number of intervals an interval may be broken down to).

Let us examine one musical example (Figure 2) in more detail. According to our proposed interval consolidation and fragmentation operations (see tables in Figure 2), the top melodic query matches fully with instances (a) and (c) in agreement with standard musical knowledge/intuition (actually the query and instance c are elaborations of instance a). The query matches partially with instance b (except for the last interval). Finally, only the first two intervals of instance d match with the first interval of the query; this is the weakest match. These match results are rather plausible according to our general musical understanding.

Figure 2 The top query matches fully with instances (a) and (c), matches partially with instance (b) except for the last interval, and, in case (d), only the first two intervals match with the first interval of the query (solid line tables indicate matches).

In this same example we get quite different results in terms of the standard edit distance operations (for pitch only – query transposed in 12 different keys). The query is measured to be most similar to instance b (only one replacement for the last note), and the other three instances are equidistant to the query (four operations in each case). These results are counter-intuitive (a and c are most similar to the query, then b, and lastly d). Although it is simple to construct a dynamic programming matrix to find an alignment of the pattern with the melody, this won't find all the occurrences of the melody; an operation which is useful for musical analysis. Additionally by not using the dynamic programming approach we can avoid the problem of determining a good threshold and can reduce the high memory usage needed by such dynamic program techniques. This is our motivation for considering our problem.

3 Formal Definition

Let $\Sigma \subset Z$ be a finite alphabet of integers. A string is a sequence of symbols from Σ and the set of all strings over the alphabet is denoted by Σ^*. Melodic string matching is a pattern matching problem where we wish to find all occurrences of a pattern within a

text. Throughout the paper the use the following notation: p represents the pattern of length m, t is the text of length n. The i-th element of the pattern (text) is denoted by p_i (t_i). A factor of a word t starting at i and finishing at j is represented by $t[i...j]$.

Melodic string matching via consolidations and fragmentation is a string matching problem where we are given a text $t = t_1...t_n$ and pattern $p = p_1, ..., p_m$ over an alphabet Σ. The problem is to find all occurrences of p in t, where an occurrence of p can consist of three operations

– equal (t_i matches p_j)
– consolidation ($t_i + t_{i+1} + ..t_{i+j}$ matches p_k)
– fragmentation (t_k matches $p_i + p_{i+1} + ..p_{i+j}$)
The total number of summations allowed for a single character is
bounded by a parameter 3.
Now we present some important properties of the problem which will be used in our analysis of the algorithm.

Lemma 1 For a pattern of size m there are $O(2^{m-1})$ valid combinations of the pattern, where $3 >= m$.

Proof . We can encode pattern combinations as a bit mask, Where a 1 at position i represents a summation between i and $i+1$ and 0 otherwise. Where summations are bounded by 3 this means that a valid pattern is a binary string of length m which avoid factors 1^{m-1} . So where $3 >= m$ this consists of all the binary words of length m-1, the total combinations is bounded by $O(2^{m-1})$.

Lemma 2 For a valid pattern combination there are $O(2^{n-1})$ valid occurrences at a position i in the text, where $3 = n$.

Proof. We can make a similar argument as Lemma 1 for Lemma 2. If we encode the valid summations in the text as a bit mask. In the worst case a text and pattern of all 0s, where $3 = n$ we have $O(2^{n-1})$.

Lemma 3 There are at most $O(n2^{n+m-2})$ occurrences of a pattern of size m in a text of size n where $3 = n$.

Proof. By Lemma 1 and 2 we have at most $O(2^{n+m-2})$ occurrences at each position in the worst case and at n positions. Therefore $O(n2^{n+m-2})$ occurrences overall.

Theorem 1 It will take at least $\Omega(n2^{n+m-2}m)$ in the worst case to solve the Melodic string matching problem via interval consolidation and fragmentation.

Proof. By Lemma 3 there are at most $O(n2^{n+m-2})$ occurrences in a string, therefore, any algorithm solving this problem will have a worst case of at least $\Omega(n2^{n+m-2}m)$ if we wish to individually report every occurrence.

4 Algorithms

We present a simple binary search based algorithm. The main idea behind this algorithm is to use a binary search to identify only those sections of the text where a valid occurrence of the pattern could possibly occur. This filtering step based on the following 2 observations.

Observation 1 For the pattern to occur it must occur in a section of the text that sums up to M, where M is the sum of the entire pattern. We call an interval of the text which sums to M, a *submass*[1] of size M.

Observation 2 P is a submass of s with occurrence at position (i, j) if and only if $s = s_1 + \ldots + s_j - s_1 + \ldots + s_{i-1}$. [2,5]

Within these valid sections of the text we then need to try pattern combinations of the same length as the section. Although simple this technique means, on average, we will drastically reduce the number of pattern combinations we need to check. This filtering technique is similar to that used in the Karp Rabin[8] string matching algorithm where we are using a very simple hashing function. An outline of the algorithm is given below.

Step 1 Calculate the sum of every prefix of t and store them as an array called *PSA[i]*. Such that $PSA[i] = s_1 + \ldots + s_i$. We store these values as pairs *(PSA[i], i)* to make identifying candidate segments easier later on.

Step 2 Sort the *PSA* array by the first value of each pair using a stable sorting algorithm, we call the resulting array the *SPSA* array.

Step 3 Identify all candidate segments by performing a binary search for all submasses M in the original string, we do this for each position i in the text. For example, if *PSA[i] = 15* and *M = 9* we would search for *15 − 9 = 14*. Where there are multiple occurrences of the same number we can also do a binary search for the end position of this match. For each candidate segment we must do a further check to ensure that all reported candidates are valid, as it is possible that some impossible candidate segments are reported e.g. those segments *(i, j)* with $j < i$.

Step 4 For each candidate segment of the text we must check pattern combinations within this fixed length. Checking the pattern combinations can be done using a restricted version of the brute force method. The restricted brute force method is similar to the standard brute force technique, however, only combinations which fit exactly in the candidate section will be checked. Time taken for this depends on the number of candidate intervals identified.

5 Analysis and Runtime

Step 1 requires us to compute the prefix sum for every index which will take $\Theta(n)$ as each sum can be computed in constant time based on the sum for the previous index. Sorting the *PSA* array is simple and we can use any algorithm such as merge sort and this will take $O(n \log n)$. Step 3 requires us to make n binary searches to find all of the candidate areas where an occurrence of the pattern could occur. As each binary search takes $O(\log n)$ and we need to do n binary searches in total we will take $O(n \log n)$ for every binary search. Step 4 is the most time consuming step in the algorithm and in the worst case it could be up to $O(2^{n+m-2})$ due to the maximum number of possible matches within a candidate segment, although in practice it will be much lower as this can only occur if the pattern and text are all 0 which wouldn't happen in practice. So the total runtime of this algorithm will be dominated by the final step of the algorithm, however, as previously mentioned, in practice we wouldn't ever realise this worst case as such strings wouldn't be interesting pieces of music or interesting melodies.

6 Experimental Results and Discussion

We have implemented and tested our solution along with the naive brute force algorithm. These were implemented in C++ on a computer with an Intel Core 2 Duo T7500 2.20Ghz with 2 GB of RAM. The tests were carried out using MIDI files converted into pitch interval representation. Initially we tested the speed of execution on random sequences with a variety of input sizes; the performance of the brute force algorithm was almost identical to ours for small inputs. As the size of the pattern, text, and the number of summations was increased, the difference became apparent. Due to the nature of the brute force algorithm it always performs an exponential number of comparisons. To give an idea of the difference in execution time, for a pattern of size 20 with up to 4 summations, the algorithm was stopped after 48h of execution time and was not finished. Where as a pattern of size 11 with 4 summations took only a couple of minutes. To further illustrate this difference we ran the algorithms on some patterns with no matches. For a pattern of size 20 and a sumsize of 4 the brute force algorithm takes the same time stated above whereas the binary search algorithm takes 4 seconds.

A focused musical experiment was performed in order to evaluate performance of the algorithm in more detail. More specifically, we used the melody from the first part of Mozart's Sonata in A major, KV331. In this part an initial theme appears in six different variations illustrating various degrees and types of melodic ornamentation and transformation. As queries we use the theme itself and various reduced versions of it (the first two measures of two such reduced versions are depicted in Figure 3).

Figure 3 Beginning of Theme and 6 Variations from Mozart's Sonata in A major KV331. The two queries at the top match most of the variations (see text).

The algorithm correctly identifies most of the variations of the theme in its various guises. Especially, the reduced versions of the theme are successfully recognised. The original theme is recognised directly only in Variation II (match of other variations is unsuccessful as many-to-many interval interval matching is required – see discussion below). A slightly reduced version of the theme (Q1 in Figure 3) is matched to Variations I, II, and V, whereas a further reduced version (Q2 in Figure 3) is matched to Variations I, II, V and VI. The reduced queries would be matched to Variation IV if a mod12 matching of intervals was allowed (or if the first note of each measure was transposed upwards by an octave); they would also be matched to Variation III if a tolerance of +/-1 semitone is introduced (Var III is in C minor). The algorithm is quite successful is capturing quite severe alteration of the melodic material.

Results for some queries can be seen in Table 1. The table shows the number of occurrences of each pattern as identified by our algorithm for three queries (ThemeStart, Reduction1, Reduction2) and for three summation thresholds (7, 8 & 9). The query ThemeStart corresponds to the first two measures of the original theme (Orig. in Figure 3). The queries Reduction1 and Reduction2 correspond to queries Q1 and Q2 of Figure 3 respectively (but Q2 is four measures long). The spurious large numbers for larger summations depicted in the table may occur due to the presence of 0s or intervals that sum to 0 which can lead to reporting many false positives as explained in section 3. The large number of combinations found for ThemeStart may be reduced if, for instance, durations are taken into account (see below).

	Original			VarI			VarII			VarIII			VarIV			VarV			VarVI		
Summations	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9	7	8	9
ThemeStart	61	460		21	26	41	4	6	35	0	0	0	0	0	0	2	2	2	0	0	0
Reduction1	5	11	14	10	14	15	11	13	14	0	0	0	0	0	0	0	0	4	20	25	31
Reduction 2	5	12	16	10	12	16	11	13	14	0	0	0	0	0	0	0	0	4	20	26	31

Table 1 Number of occurrences found in each text, for each query and summation size.

There are, various shortcomings in the current preliminary attempt to solve this relatively difficult melodic matching problem. Firstly, the algorithm identifies the correct instances but additionally finds in some occasions many false positives that are not significant (see table 1); this is particularly strong when the consolidation/fragmentation limit (number of summations) is large. There are various ways to deal with this issue. An obvious way is to extend matching to include duration consolidation/fragmentation (when a pitch interval is fragmented/consolidated so are the rhythmic durations fragmented/consolidated). Another way is to add extra constraints such as an overall number of fragmentations/consolidations allowed per query (similar to γ- approximate matching where a threshold for a valid match is defined as the sum of the differences over the entire match [4]).

Secondly, the current implementation allows only one-to-many and many-to-one matches (that is one interval consolidated/fragmented to many intervals). This way it is unlikely that two ornamented versions of the same underlying melody can be matched (e.g. the original theme of Figure 3 with variations). The current algorithm would be more successful if ornamentations were stripped away from a query melody before applying pattern matching.

Thirdly, in the current version of the algorithm interval matching (of consolidated/fragmented intervals) is exact (i.e. intervals add up exactly to the matched interval). This may be unnecessary. It may be useful to allow some tolerance, e.g., ± 1 semitone, to account for matches of patterns, for instance, in relative or parallel keys. This would be a kind of δ-approximate matching [4].

Overall, the algorithm is performing as expected and is successful in capturing melodic variation. Further testing, however, is necessary. In particular we plan to make use of precision and recall type analysis to determine the accuracy of our algorithm. As our algorithm will identifies all occurrences of a pattern it is clear that in certain situations we will identify many occurrences that are not true occurrences of the melody, so testing the precision of our algorithm will be a very important metric in determining it's effectiveness. Additionally, larger groundtruth testdata are necessary. Apart from testing, further research is required to improve it and make it more robust and reliable.

5 Conclusions

In this paper we have defined the problem of matching melodic patterns in a novel way, such that ornamentation and variation can be naturally accommodated. This new approach allows the development of new flexible transposition-invariant melodic matching techniques that can identify melodic patterns exhibiting various degrees of variation/transformation.

We have proposed one algorithmic solution to this problem and tested it on artificial and actual melodic data. We have shown that the proposed technique yields musically meaningful results. At the same time a number of potential shortcomings have been identified and discussed in the previous section. Although we have shown that the algorithm is quicker in a number of situations, it is clear that we need to perform a more thorough and rigorous experimental analysis of the algorithm. Further research is required to improve the current version and to show its full potential. The improvements proposed in the previous section are expected to increase the performance and effectiveness of the algorithm.

References

1. Bansal, N., Cieliebak, M., and Lipták, Z. Finding submasses in weighted strings with fast fourier transform. *Discrete Applied Mathematics,* 155(67):707-718, 2007. Computational Molecular Biology Series, Issue V.
2. Becker, S. Sequencing from compomers: Using mass spectrometry for DNA de-novo sequencing of 200+ nt. In Gary Benson and Roderic D. M. Page, editors, WABI, volume 2812 of Lecture Notes in Computer Science, pp. 476-497, Springer, 2003.
3. Cambouropoulos E. Crawford T. and Iliopoulos C.S. Pattern processing in melodic sequences: Challenges, caveats and prospects. *Computers and the Humanities,* 35(1):9-21, 2001.
4. Cambouropoulos E. Crochemore M. Iliopoulos C.S. Mouchard L. and Pinzon Y.J. Algorithms for computing approximate repetitions in musical sequences. *International Journal of Computer Mathematics,* 79(11):1135-1148, 2002.
5. Cieliebak, M., Erlebach, T., Lipták, Z, Stoye, J., and Welzl, E. Algorithmic complexity of protein identification: combinatorics of weighted strings. *Discrete Applied Mathematics*, 137(1):27-46, 2004.
6. Ferraro, P., Hanna, P. and Robine M. On optimising the editing algorithms for evaluating similarity between monophonic musical sequences. *Journal of New Music Research,* 36(4):267-279, 2007.
7. Hewlett, W. and Selfridge-Field, E. *Melodic Similarity: Concepts, Procedures, and Applications.* MIT Pess, Cambridge (Ma), 1988.
8. Karp, R and Rabin, M. *Efficient Randomized Pattern-Matching Algorithms.* IBM J. Res. Dev. 249-260 1987
9. Mongeau M. and Sankoff D. Comparison of musical sequences. *Computers and the Humanities*, 24:161-175, 1990.