Retrieving Music — To Index or not to Index *

Kjell Lemström, Atso Haapaniemi, Esko Ukkonen
Department of Computer Science, University of Helsinki
P.O. Box 26 (Teollisuuskatu 23), FIN-00014 University of Helsinki, Finland

Abstract Pattern matching methods are used as an essential component in a large variation of multimedia applications. The search agents for Internet as well as various systems for retrieving images from image databases by image content are good examples. However, not too much research effort has been directed yet for information retrieval in music databases. We present a coding of music (melodies) that is invariant under different musical keys and tempos. It can applied on melodies written in traditional notation using notes and staves. The conversion e.g. from MIDI files is straightforward. As our codes are linear discrete sequences, general string matching algorithms can be used for finding the occurrences of a melody sample in a music database. We report some experimental results on the performance of two alternative ways to organize such a search. One uses the suffix-trie as an index for the database, the second makes an on–line search with the Boyer–Moore algorithm.

Introduction In this paper we discuss some issues and ideas that we are experimenting with in our project which develops methods and software for music retrieval by music content.

We use a simple *two–dimensional relative coding* to represent the melodies in the database. The coding of a note consists of two elements, the "vertical distance" or an *interval* between the note and its predecessor, and the "horizontal distance" or the *relative duration* of the note as compared to a designated base duration. The coding is invariant in two important respects, namely under transpositions and changes of the tempo. This considerably simplifies the music retrieval task.

There are rather small number of intervals and relative durations of notes, that traditionally are used in (western) music. Therefore our coded presentation can be seen as a discrete linear string over a finite alphabet. Then the matching of a given melody sample, encoded in the same way, against the melodies in the database can be done by using the string matching algorithms originally designed for key–word matching in texts [3, 7].

*Work supported by the Academy of Finland under grant 8745.

ACM Multimedia '98, -Art Demos - Technical Demos - Poster Papers -, September 1998, Bristol, UK

In this paper we compare two basic approaches of string matching in the context of music retrieval. In the *indexing approach* the encoded music database is first preprocessed into a suitable index structure. We use the *suffix–trie* as the index. The occurrences of the music sample can then retrieved fast (in fact, in time $O(m)$ where m is the length of the sample) from the trie. In the *on–line approach* the entire database is scanned from the beginning to the end to find the occurrences. The fastest on–line algorithm is the well–known *Boyer–Moore algorithm* [2] whose running time is $O(n/m)$ where n is the length is the entire database; we used it in our experiments.

We concentrate on exact matching in which the exact occurrences of the sample (up to transposition) are found. Approximate matching is possible with appropriate string matching algorithms as well [7, 10].

The construction time and storage space of a suffix–trie is quadratic in the length of the database. For large databases this is too much. Therefore we used an abridged version that consists of the top part of the trie up to certain depth. Even this can be too large. Then the *suffix–tree* should be used. It is of size $O(n)$.

Related work The first attempt of music retrieval was done by Ghias et. al. [5]. They used a reduced alphabet of intervals; they had only 3 different possibilities for intervals: upwards, downwards and the same as previous. Thus, the alphabet contained the letters U, D and S, respectively. They claimed that using such an alphabet and a k-mismatches string matching algorithm by Baeza-Yates and Perleberg [1] they could discriminate 90 % of the 183 melody database by using a sample of 10–12 intervals.

In [4] Foote gives a method for retrieving audio and music. The retrieving of the music is based on classification of the music in the database. For each piece of music in the database the similarities to a set of templates and the piece of music is measured. The classification of the piece of music is based on the distribution of the similarities.

To our knowledge, the only working system for retrieving music by melody sample is the MELDEX system [8]. They have a database of 9 400 melodies of mostly ballads and folksongs. For the matching they use *dynamic programming* or its variant by Wu and Manber [12].

Coding a melody The two-dimensional relative code, *tdr-code*, is a sequence S consisting of *elements* s_i, $S = s_1, s_2, \ldots, s_n$, where each element s_i consists of a pair of components: $\langle x_i, y_i \rangle$. The pair $\langle x_i, y_i \rangle$ corresponds to the

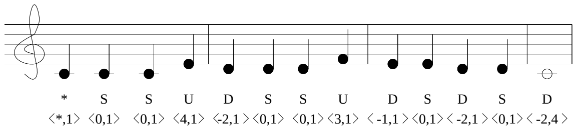

Figure 1: *The traditional notation and corresponding sequences of [5] and our tdr-sequence.*

ith note of the melody that is to be coded. The first component x_i is the difference between the two notes using a chromatic scale. We call a sequence $X_j = x_j, \ldots, x_n, j = \{1, \ldots, n\}$ an *interval sequence*. The first element x_j of an interval sequence is always assigned with a "don't care" value '$*$'. If the interval goes downwards (the tune corresponding to the successing note is lower than the tune of previous one), x_i is negatively valued.

The second component y_i of a tdr-code sequence is the duration of the element s_i. The scaling of the components $y_i, i = \{2, \ldots, n\}$ is determined by the first component y_1. The values of the remaining components are settled according to the value of the y_1. The ratio $\frac{y_i}{y_1}$ gives the duration of the element s_i with respect to the duration of the element s_1.

In accordance with the previous definition, the sequence $y_j, \ldots, y_n, j = \{1, \ldots, n\}$, is called a *duration sequence*. The negative values can be used with pauses.

An example of this coding can be found in the Fig. 1.

Suffix-trie and Boyer-Moore Algorithm

A *trie* is a rooted tree with two main properties, i.e. (1) each node, except the root, is associated with a symbol of a alphabet, (2) any two descendants of the same node cannot be associated with the same symbol. A *suffix* trie contains all the suffices of the given string or sequence S.

Boyer-Moore is a fast on-line string searching algorithm. Its basic and original idea is to move the search pattern P (length m) from left to right according to the similarities found in the text S (length n). In each step, after moving P to the right, the comparison is started from the end of P moving backwards in P. In the case where the compared characters do not match, a special function is used to move the P to the right by $1 \ldots m$ steps.

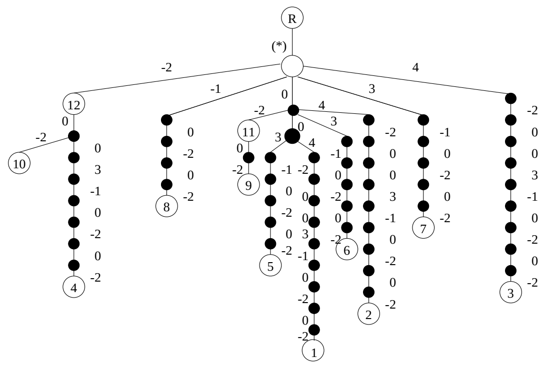

Figure 2: *The suffix-trie containing every suffix of the interval sequence of the Fig. 1. The nodes indicated by white circles contain pointers to the corresponding positions of the melody.*

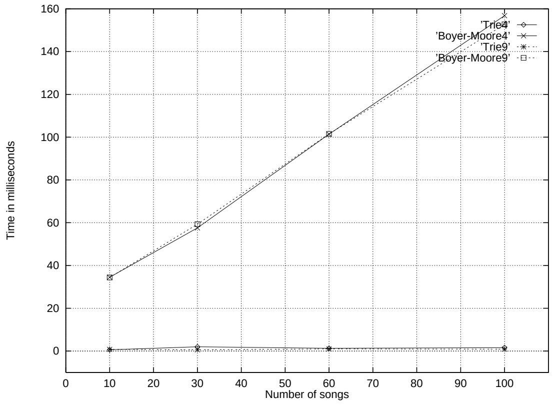

Figure 3: *Query times for Boyer-Moore and suffix-trie. The lowermost curves (Trie4 and Trie9) represent the query times for suffix-trie (when $|p| = 4$ and $|p| = 9$, respectively). The two uppermost curves represent the query times for Boyer-Moore algorithm.*

Experimental Results Our melody database consists of 100 different MIDI-files that are automatic compositions of Pauli Laine [1]. One MIDI-file contains approximately 150 notes (varying from 100 to 200). Experiments were made with preprocessed MIDI-files, which contained the relative interval and duration for each note. Only the relative intervals were considered in our experiments. The duration queries can be implemented almost similarly. Only the k-mismatches case is somewhat more difficult; one has to be careful in choosing the correct scaling constant in order to get the minimal number of mismatches between the two sequences. A query in which both the intervals and durations are considered is somewhat more complex but can also be implemented without excessive difficulty.

In our case, the trie is formed by the interval sequences $X_j, j = \{1, \ldots, n\}$. Fig. 2 presents a suffix-trie that corresponds to the interval sequences of the melody presented in Fig. 1. The leaves (and possibly some internal nodes) of the tree contain indices to the corresponding positions of the melody. Using such a structure, one can find the occurrences of a given sample pattern in the structure, and therefore in the sequence, very fast. If the depth (q) of the trie is restricted, only all the q-grams, i.e. the sequences exactly of the length q are stored in the trie.

When constructing the suffix-trie structure we had to limit the depth of the path to maintain the structure in the main memory. In our experiments we limited the depth to $q = 10$, which makes about 31 000 nodes in trie when using the whole database. Depth 14 uses already about 56 000 nodes.

We made two series of experiments; in the first we compared the performances when the length of pattern p was $|p| = 4$ and in the second experiment we had $|p| = 9$. In both cases the depth of the tree was $q = 10$. We measured the average performances of 10 trials for both techniques in both series. In each trial the pattern was randomly chosen among the melodies in the database at a random position. The time spent on the file handling was excluded in all the cases, in order to get as comparable results as possible.

As shown in Fig. 3, the size of the used database varies between 10 and 100 files. The first observation is that there is no noticeable difference between the performances in the

[1] University of Helsinki, Department of Musicology

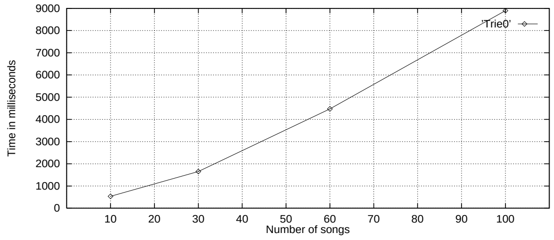

Figure 4: *The construction time of suffix-trie.*

cases when $|p| = 4$ and when $|p| = 9$, neither with Boyer-Moore nor with Suffix-trie. It can be seen that the response time for The Boyer-Moore algorithm is linear to the size of the database. Queries with the suffix-trie algorithm maintain the same response time when the size of the database increases. Suffix-trie algorithm shows better response time already, when the database consists of 10 melodies only. However, the time for constructing the suffix-trie structure (see Fig. 4) is linear to the size of the database. Although this operation is rather expensive, the overhead it causes is almost nonexistent because it has to be constructed only once.

The experiments were made on a fast microcomputer under Linux. $Java^{TM}$ programming language was used to achieve maximum portability, but due to the lack of fast performance another programming language is considered (C/C++).

Discussion The used tdr-code gives two important invariances, i.e. the transposition invariance and the note duration invariance. This means that two melodies can be interpreted as the same melody. Though, they might be written in different modes or by note durations, where corresponding durations are only multiples of a same constant c of each other.

Using preprocessing and suffix-trie, a query, whether the query pattern occurs in a melody, can be applied in linear time. The presented method allows also efficient solution for the k-mismatches case with certain restrictions; If we are dealing with a search pattern p of the length $|p| \leq q$, in order to have a very efficient performance, the two sequences to be matched against each other have to be of the same length. Thus, insertion and deletion operations are not allowed. The case when $|p| > q$ is considered by Salosaari and Järvelin [9]. In this case a standard *q-gram* search can be used with k-mismatches.

As it was presumed, the bigger the database the more the indexing is needed. However, the efficiency of the indexing is in strong relation to the fitting of the suffix-trie in the main memory. Due to our experiments on the necessity of indexing and the fact that the suffix-trie structure grows relatively rapidly, we are encouraged to implement a compacted, but a more complex, form of the trie, i.e. the *suffix tree* [11]. This structure grows only linearly with respect to the length of the melodies. Moreover, we can still restrict the size of the tree by reducing the value of the q. In [6] the usage of suffix-trees with only a portion of the tree in the main memory at a time is studied. The applications are usually noticeably slowed down.

The method described in this paper can be used with minor changes to melodies with multiple staffs. The staffs are just concatenated and some extra information, which tells the starting position index for each staff, is maintained.

References

[1] R. Baeza-Yates and C. H. Perleberg. Fast and practical approximate string matching. In *Combinatorial Pattern Matching, Third Annual Symposium*, pages 185–192, 1992.

[2] R. Boyer and S. Moore. A fast string searching algorithm. *Communications of the ACM*, (20):762–772, 1977.

[3] M. Crochemore and W. Rytter. *Text Algorithms*. Oxford University Press, 1994.

[4] J.T. Foote. Content-based retrieval of music and audio. In *Multimedia Storage and Archiving Systems II, Proceedings of SPIE*, pages 138–147, 1997.

[5] A. Ghias, J. Logan, D. Chamberlin, and B. C. Smith. Query by humming - musical information retrieval in an audio database. In *ACM Multimedia 95*, 1995.

[6] G. H. Gonnet, R. A. Baeza-Yates, and T. Snider. Lexicographical indices for text: Inverted files vs. PAT trees. Technical Report OED-91-01, Centre for the New OED, University of Waterloo, 1991.

[7] D. Gusfield. *Algorithms on strings, trees, and sequences*. Cambridge University Press, 1997.

[8] R.J. McNab, L.A. Smith, D. Bainbridge, and I.H. Witten. The New Zealand digital library MELody inDEX. *D-Lib Magazine*, 1997.

[9] P. Salosaari and K. Järvelin. MUSIR – a retrieval model for music. Technical Report RN-1998-1, University of Tampere, Department of Information Studies, July 1998.

[10] J. Tarhio and E. Ukkonen. Approximate Boyer-Moore string matching. *SIAM Journal of Computing*, 22(5):243–260, 1993.

[11] P. Weiner. Linear pattern matching algorithms. In *Proceedings of IEEE 14th Annual Symposium on Switching and Automata Theory*, pages 1–11, 1973.

[12] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91, 1992.