Efficient and Simple Algorithms for Time-Scaled and Time-Warped Music Search

Antti Laaksonen*

Department of Computer Science
University of Helsinki
ahslaaks@cs.helsinki.fi

Abstract. We present new algorithms for time-scaled and time-warped search from symbolic polyphonic music database. Our algorithm for time-scaled search works in $O(n^2m)$ time and $O(m)$ space, where n denotes the number of notes in the database and m denotes the number of notes in the pattern. Our algorithm for time-warped search works in $O(n(m+\log n))$ time and $O(n)$ space. If the set of possible pitch values is constant, the time complexity of the latter algorithm is only $O(nm)$. The new algorithms are more efficient than the earlier algorithms, both in theory and practice. Moreover, the new algorithms are conceptually simple.

Keywords: symbolic music retrieval, time-scaling, time-warping

1 Introduction

In this paper, we consider algorithms for finding occurrences of a given pattern in a polyphonic music database. Both the database and the pattern are given in symbolic form as a sequence of notes. Applications for polyphonic music search include music analysis [1] and query by humming [2].

Let $D_1, \ldots, D_n$ denote the database notes and $P_1, \ldots, P_m$ denote the pattern notes. Each note is a pair (t, p) that consists of two real numbers: onset time t in seconds and pitch value p in semitones. We use the notation $S_{k,a}$ to refer to the ath member in a pair S_k; for example, $D_{3,2}$ refers to the pitch of the third note in the database. We assume that the pairs in D and P are in lexicographic order and that every note in P has a distinct onset time.

The goal is to find all of the note indices in database D from which an occurrence of pattern P begins. An occurrence is a sequence $D_{k_1}, \ldots, D_{k_m}$ for which $1 \leq k_1 < k_2 < \ldots < k_m \leq n$ and there is a constant s such that $D_{k_i,2} = P_{i,2}+s$ for every $i = 1, \ldots, m$. In other words, s denotes the transposition of the pattern in the occurrence. Let α_i denote the scaling factor between pattern notes i and $i+1$ i.e. $\alpha_i = (D_{k_{i+1},1} - D_{k_i,1})/(P_{i+1,1} - P_{i,1})$. In *exact search* [6] $\alpha_i = 1$ for every $i = 1, \ldots, m$, in *time-scaled search* [4] $\alpha_i = \alpha$ where α is a constant, and in *time-warped search* [5] $\alpha_i \geq 0$.

* This work has been supported by the Helsinki Doctoral Programme in Computer Science and the Academy of Finland (grant number 118653).

Fig. 1. (a) A theme from Bach's fugue (BWV 871) in *Das Wohltemperierte Klavier*. (b) An exact occurrence of the theme in another key. (c) A time-scaled occurrence of the theme. (d) A time-warped occurrence of the theme.

For example, consider the opening theme from Bach's fugue (BWV 871) in *Das Wohltemperierte Klavier* in Figure 1(a). The theme appears more than twenty times throughout the fugue in exact, time-scaled and time-warped form [1]. Figure 1(b) shows an exact occurrence where the theme is transposed to a different key. Figure 1(c) shows a time-scaled occurrence where every note has twice the length. Finally, Figure 1(d) shows a time-warped occurrence where the rhythm of the theme is a little bit different.

Search	Algorithm	Time	Space
Exact	[6]	$O(nm)$	$O(m)$
Time-scaled	[4]	$O(n^2 m \log n)$	$O(n^2)^*$
	this paper	$O(n^2 m \log n)$	$O(1)$
	this paper	$O(n^2 m)$	$O(m)$
Time-warped	[5]	$O(n^2 m \log n)$	$O(n^2)^*$
	[3]	$O(n^2 m)$	$O(n)^*$
	this paper	$O(n^2)$	$O(1)$
	this paper	$O(nm \log n)$	$O(n)$
	this paper	$O(n(m + \log n))$	$O(n)$

Table 1. Summary of the algorithms for exact, time-scaled and time-warped search. Space complexities marked with * are lower than those in the original publications, and they can be obtained through a careful implementation of the algorithm.

Several algorithms have been proposed for polyphonic music search. Exact search is the most straight-forward type of search: it can be solved in $O(nm)$ time

with a simple algorithm [6]. Both time-scaled and time-warped search can be performed in $O(n^2 m \log n)$ time using database preprocessing and priority queues [4, 5]. Another approach to time-warped search has been dynamic programming [3] which leads to an algorithm that works in $O(n^2 m)$ time. Sometimes a window w is used to limit the number of database notes between consecutive notes in the occurrence. When a window is used, the time complexities for time-scaled and time-warped algorithms are $O(nmw \log n)$ and $O(nmw)$, and the algorithms can be applied to searches from large databases.

In this paper we present new algorithms for time-scaled and time-warped search. We focus on complete search without a window. For time-scaled search, we present an algorithm that can be implemented in $O(n^2 m \log n)$ or $O(n^2 m)$ time. Our algorithms for time-warped search work in $O(n^2)$, $O(nm \log n)$ and $O(n(m + \log n))$ time. Moreover, if the set of possible pitches is constant, we can perform time-warped search in $O(nm)$ time. The new algorithms have a simple structure and small constant factors, and they are also efficient in practice. Table 1 presents a summary of both previous and new algorithms.

The organisation of the rest of the paper is as follows. In Sections 2 and 3 we present our new algorithms for time-scaled and time-warped search. Then, in Section 4 we compare the practical performance of previous search algorithms to the new algorithms. Finally, in Section 5 we present our conclusions.

2 Time-Scaled Search

In this section, we present a new algorithm for time-scaled search. To start with, the algorithm fixes the first and last database note of a potential occurrence and then checks to see if an occurrence actually exists between these notes. The efficiency of the algorithm depends on how the existence of an occurrence is checked for. We use two methods for this: the first works in $O(n^2 m \log n)$ time and $O(1)$ space, and the second works in $O(n^2 m)$ time and $O(m)$ space.

The previous time-scaled search algorithm [4] differs considerably from our approach. For each pattern note pair $(k, k + 1)$, the previous algorithm first precomputes a sorted list of all database note pairs that have the same interval as pattern notes k and $k + 1$. After this, the algorithm uses priority queues to track pattern occurrences that have a constant scaling factor. The algorithm works in $O(n^2 m \log n)$ time and $O(n^2)$ space.

Listing 1 shows the structure of our algorithm. Variables i and j point to the first and last database note of a potential occurrence. After fixing these variables, the algorithm calculates the scaling factor α. Then the algorithm goes through all pattern notes and checks to see if each note actually appears in the database. Variables β and p contain the onset time and pitch of the required pattern note, and they can be directly calculated when the first note and scaling factor of the occurrence are known. Variable c maintains the number of matched pattern notes, and if all notes are matched, the algorithm reports an occurrence.

The time complexity of the algorithm depends on how the check at line 8 is implemented. One solution is to search for (β, p) directly from D using

```
 1: for i ← 1, . . . , n do
 2:     for j ← i + 1, . . . , n do
 3:         α ← (D_{j,1} − D_{i,1})/(P_{m,1} − P_{1,1})
 4:         c ← 0
 5:         for k ← 1, . . . , m do
 6:             β ← D_{i,1} + α(P_{k,1} − P_{1,1})
 7:             p ← D_{i,2} + P_{k,2} − P_{1,2}
 8:             if D contains (β, p) then
 9:                 c ← c + 1
10:             end if
11:         end for
12:         if c = m then
13:             Report an occurrence at i.
14:         end if
15:     end for
16: end for
```

Listing 1. Time-scaled search

binary search in $O(\log n)$ time. This is possible because the notes in D are lexicographically sorted. The use of binary search leads to an algorithm that works in $O(n^2 m \log n)$ time and $O(1)$ space. However, we achieve a lower time complexity $O(n^2 m)$ by using the fact that j and α only increase for a fixed i.

To make the algorithm more efficient, we use a technique similar to the exact search algorithm in [6]. For each pattern note k, we maintain a pointer Q_k to the database. After fixing i, we set $Q_k = i$ for each k. Then, when j increases, we always increase each Q_k until we reach the last database note that is not larger than (β, p). The pair (β, p) is calculated separately for each k, as in Listing 1. Using this technique, the amortized time complexity of the algorithm is $O(n^2 m)$. This is true because each pointer Q_k is increased at most n times for a fixed i. The space complexity is $O(m)$, because there are m pointers.

3 Time-Warped Search

In this section, we present new algorithms for time-warped search. The first method resembles our time-scaled algorithm: the algorithm initially fixes the first database note of the occurrence and then checks to see if all subsequent pattern notes exist in the database. A simple implementation works in $O(n^2)$ time and $O(1)$ space. However, we can use an auxiliary array and binary search to create an algorithm that works in $O(nm \log n)$ time and $O(n)$ space. Finally we present another more sophisticated algorithm that works in $O(n(m + \log n))$ time and $O(n)$ space. Its time complexity is further reduced to $O(nm)$ if the set of pitches is constant.

Two earlier algorithms have been proposed for time-warped search: The algorithm presented in [5] is a modification of the time-scaled search algorithm [4] that uses database preprocessing and priority queues. As with its predecessor,

the algorithm works in $O(n^2 m \log n)$ time and $O(n^2)$ space. Another approach [3] uses dynamic programming to calculate partial occurrences of the pattern. This algorithm is easy to implement and works in $O(n^2 m)$ time and $O(n)$ space.

```
 1: for i ← 1, ..., n do
 2:     j ← i, c ← 0
 3:     for k ← 1, ..., m do
 4:         p ← D_{i,2} + P_{k,2} − P_{1,2}
 5:         if next(j, p) < ∞ then
 6:             c ← c + 1
 7:         end if
 8:         j ← next(j, p) + 1
 9:     end for
10:     if c = m then
11:         Report an occurrence at i.
12:     end if
13: end for
```

Listing 2. Time-warped search, first approach

Listing 2 shows the structure of our first algorithm. The variable i fixes the first database note of the occurrence. After this, the algorithm checks to see if all pattern notes can be found in the remaining part of the database in the correct order. The variable j maintains the current database note, the variable c counts the number of matched pattern notes, and the function *next* is defined as follows:

$$next(j,\, p) = \min\{a : a \geq j,\ D_{a,2} = p\} \cup \{\infty\}$$

The most simple way to calculate $next(j,\, p)$ is to increase j until a database note with pitch p is found, or when all of the remaining notes have been checked. This yields an algorithm that works in $O(n^2)$ time and $O(1)$ space. However, once again, a more efficient implementation is possible using binary search. We build an array E that allows us to efficiently search for database notes with a given pitch. The array E contains a pair $(D_{i,2},\, i)$ for each $i = 1, \ldots, n$ and is sorted lexicographically. After this, we can find $next(j,\, p)$ in $O(\log n)$ time using binary search in E. The time complexity for constructing E is $O(n \log n)$ and $O(nm \log n)$ for the rest of the algorithm, so the algorithm works in $O(nm \log n)$ time. The space complexity is $O(n)$ because of E.

In fact, we can use E to construct an even more efficient algorithm, shown in Listing 3. The idea is to track the pattern occurrences simultaneously. First, all database notes are potential beginning notes for an occurrence. Then, we extend each occurrence one note at a time as long as it is possible. If an occurrence cannot be extended, it is removed from the set of possible occurrences. It turns out that we can implement the extension of all occurrences efficiently when the database notes are sorted by their pitches.

```
 1: Set E_i ← (D_{i,2}, i) for each i = 1, 2, ..., n.
 2: Sort E lexicographically.
 3: Set F_i ← (E_{i,2}, E_{i,2}) for each i = 1, 2, ..., n.
 4: u ← n
 5: for k ← 2, ..., m do
 6:     j ← a ← 1
 7:     for i ← 1, ..., u do
 8:         p ← D_{F_{i,1},2} + P_{k,2} − P_{1,2}
 9:         Increase a while E_a ≤ (p, F_{i,2}).
10:         if E_{a,1} = p and E_{a,2} > F_{i,2} then
11:             if k = m then
12:                 Report an occurrence at F_{i,1}.
13:             end if
14:             F_j ← (F_{i,1}, E_{a,2})
15:             j ← j + 1
16:         end if
17:     end for
18:     u ← j
19: end for
```

Listing 3. Time-warped search, second approach

The algorithm in Listing 3 uses two arrays: E is initialized like in the previous algorithm, and initially $F_i = (E_{i,2}, E_{i,2})$ for each $i = 1, \ldots, n$. The algorithm maintains information about partial pattern occurrences in F. Each pair in F consists of two indices for D: the first and the current note in an occurrence. For each pattern note k we use a merge-like technique to extend the occurrences in F. This is possible because the pairs in F are sorted according to the pitch of the first note in the occurrence. The variable u contains the number of active occurrences, and the variable j is used to determine new indices for occurrences that can be extended. Note that the indices of pairs inside F change during the algorithm and the last $n - u$ indices are not used anymore.

For example, consider a situation where $D = \{(1, 3), (2, 2), (3, 7), (4, 6)\}$ and $P = \{(1, 1), (2, 5), (3, 4)\}$. At the beginning of the algorithm, all the database notes are potential beginning notes for an occurrence. Therefore, $F = \{(2, 2), (1, 1), (4, 4), (3, 3)\}$. When $k = 2$, the algorithm tries to extend each occurrence by the second note. The pairs (2, 2) and (1, 1) can be extended and they become (2, 4) and (1, 3). After this, $F = \{(2, 4), (1, 3), -, -\}$ where $-$ denotes indices that are not used anymore. Finally, when $k = 3$, the algorithm tries to extend each occurrence by the third note. Now (1, 3) becomes (1, 4) and $F = \{(1, 4), -, -, -\}$. This corresponds to the only time-warped occurrence of the pattern that consists of database notes D_1, D_3 and D_4.

The time complexity of the algorithm consists of two phases. First, the arrays E and F are constructed in $O(n \log n)$ time. After this, the search is performed in $O(nm)$ time. This results in a time complexity of $O(n(m+\log n))$. However, if the set of possible pitches is constant, we can use a linear-time sorting algorithm

such as counting sort to construct E, which leads to an overall time complexity of $O(nm)$. The space complexity of the algorithm is $O(n)$.

4 Experiment

In this section, we present some results concerning the practical performance of the algorithms. We compared our new algorithms with the previous ones by using random note databases of different sizes. The performance of our algorithms seems to be superior to the previous algorithms, which is as expected because our algorithms have lower complexities and constant factors.

We implemented all the algorithms using C++ and STL. We call the time-scaled search algorithms S_1–S_3 and the time-warped search algorithms W_1–W_5. The algorithms for time-scaled search were:

- S_1: the algorithm described in [4]
- S_2: our $O(n^2 m \log n)$ time algorithm
- S_3: our $O(n^2 m)$ time algorithm

The algorithms for time-warped search were:

- W_1: the algorithm described in [5]
- W_2: the algorithm described in [3]
- W_3: our $O(n^2)$ time algorithm
- W_4: our $O(nm \log n)$ time algorithm
- W_5: our $O(n(m + \log n))$ time algorithm

We implemented algorithms S_1, W_1 and W_2 as described in the original publications, yet we made the following modifications. We implemented S_1 and W_1 in $O(n^2)$ space instead of $O(n^2 m)$ by using the fact that we can calculate the K tables for each pattern note k separately. Furthermore, we only stored the information needed for retrieving the first notes of the occurrences. Finally, we implemented the dynamic programming in W_2 iteratively in $O(n)$ space, while the original implementation uses a recursive function and memoization, i.e. the calculated values of the function are stored in a look-up array.

For experiment material, we used three collections (C_1, C_2 and C_3), each consisting of twenty databases and patterns. We constructed all databases and patterns randomly. All onset times and pitches were integers. We chose onset times for the database and pattern notes randomly from the ranges $[1, n/10]$ and $[1, 10m]$, respectively. Finally, we chose pitch values randomly from the range $[1, 50]$. We required that all notes in each database and all onset times in each pattern were distinct. Thus, the databases are highly polyphonic, and this kind of material could originate from modern orchestral music.

The parameters for the collections were:

- C_1: $n = 5000, 10000, \ldots, 50000$; $m = 5$
- C_2: $n = 5000, 10000, \ldots, 50000$; $m = 10$

– C_3: $n = 200000, 400000, \ldots, 2000000$; $m = 100$

We used C_1 in time-scaled search and C_2 and C_3 in time-warped search. In time-scaled search, we only used small databases because all the algorithms are quadratic with respect to n. In time-warped search, we used both small and large databases; algorithms W_4 and W_5 were efficient enough to process large databases. We used shorter patterns in time-scaled search because time-scaled occurrences are rare in random material.

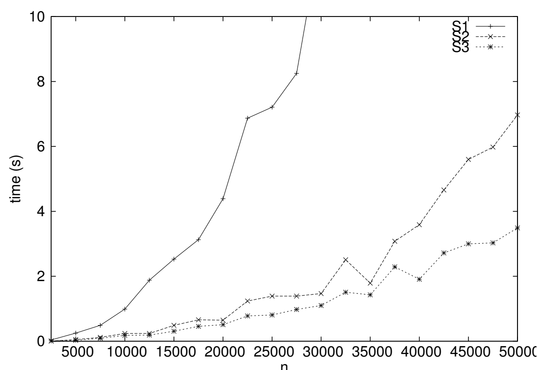

Fig. 2. Running times of time-scaled search in C_1.

Figure 2 shows the running times of time-scaled search using C_1. Interestingly, while algorithms S_1 and S_2 have the same time complexity $O(n^2 m \log n)$, S_2 seems to be much faster in practice. The probable reason for this is that the structure of S_2 is simple and thus it has lower constant factors. When n was small, the performance of S_2 and S_3 was about equal, but for the largest databases, S_3 used about the half the time compared to S_2.

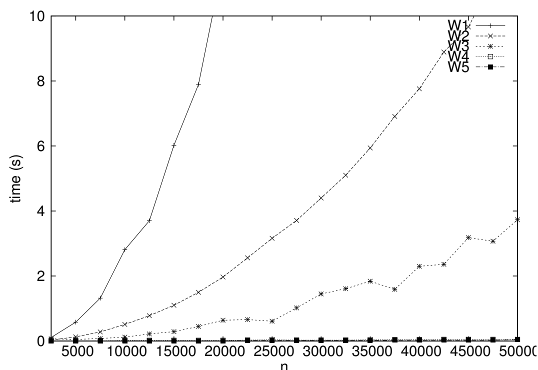

Fig. 3. Running times of time-warped search in C_2.

Figure 3 shows the running times of time-warped search using C_2. In an earlier experiment [3], W_1 performed better than W_2, but this time our results were the opposite. A possible reason for this is that we implemented W_2 iteratively which is more efficient than using memoization. W_3 was relatively efficient despite the quadratic time complexity. This is because of its simple structure and the fact that usually not all database note pairs need to be examined. W_4 and W_5 were superior to the other algorithms and we further compared them using larger databases.

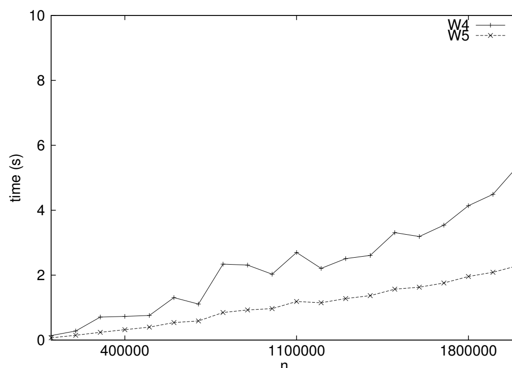

Fig. 4. Running times of time-warped search in C_3.

Figure 4 shows the running times of time-warped search using C_3. We only compared algorithms W_4 and W_5 because they were efficient enough to process databases of this size. The algorithms were usable even when there were millions of notes in the database. For the largest databases, the running time of W_5 was about the half compared to that of W_4.

5 Conclusions

We presented new algorithms for time-scaled and time-warped search from symbolic polyphonic music. Our algorithms are efficient both in theory and practice: in comparison with earlier algorithms, they have lower complexities and constant factors. Our best algorithm for time-scaled search works in $O(n^2m)$ time and $O(m)$ space, and our best algorithm for time-warped search works in $O(n(m + \log n))$ time and $O(n)$ space. If the set of pitches is constant, which is usually the case, the latter algorithm works in $O(nm)$ time.

In this paper we concentrated on the general situation where the onset time difference between two consecutive database notes in an occurrence is unlimited. However, in practice, there should be a limit: for example, if the onset time difference of two database notes is 30 seconds, they cannot be two consecutive notes in a real-world melody occurrence. On the other hand, it is difficult to

determine how long the windows should be. Furthermore, algorithms that are efficient in the general situation are also efficient in a more restricted situation.

Using the algorithms presented in this paper, we can perform efficient time-warped search even if the database consists of millions of notes. However, time-scaled search seems to be a more difficult type of search because partial pattern occurrences with different scaling factors cannot be combined. The current algorithms for time-scaled search cannot be used if the database is large. Another problem arises when both the database and the pattern are long. In this case, even an algorithm with a time complexity of $O(nm)$ may be inefficient. However, even with exact search, no algorithm that works in $o(nm)$ time is known. Thus, there are still several interesting unsolved problems with polyphonic symbolic music search.

References

1. Bruhn, S: J. S. Bach's Well-Tempered Clavier: In-Depth Analysis and Interpretation. Mainer Int Ltd (1993)
2. Ghias, A., Logan, J., Chamberlin, D., Smith, B.: Query by Humming – Musical Information Retrieval in an Audio Database. In: ACM Multimedia 1995, pp. 231–236 (1995)
3. Laitinen, M., Lemström, K.: Dynamic Programming in Transposition and Time-Warp Invariant Polyphonic Content-Based Music Retrieval. In: 12th International Society for Music Information Retrieval Conference, pp. 369–374 (2011)
4. Lemström, K.: Towards More Robust Geometric Content-Based Music Retrieval. In: 11th International Society for Music Information Retrieval Conference, pp. 577–582 (2010)
5. Lemström, K., Laitinen, M.: Transposition and Time-Warp Invariant Geometric Music Retrieval Algorithms. In: 3rd International Workshop on Advances in Music Information Research, pp. 369–374 (2011)
6. Ukkonen, E., Lemström, K., Mäkinen, V.: Geometric Algorithms for Transposition Invariant Content-Based Music Retrieval. In: 4th International Symposium on Music Information Retrieval, pp. 193–199 (2003)