

MidiFind: Fast and Effective Similarity Searching in Large MIDI databases

Tongbo Huang, Guangyu Xia, Yifei Ma
Roger Dannenberg, Christos Faloutsos

School of Computer Science, Carnegie Mellon University
{tongboh,gxia,yifeim,rbd,christos}@andrew.cmu.edu

Abstract. While there are perhaps millions of MIDI files available over the Internet, it is difficult to find performances of a particular piece because well labeled metadata and indexes are unavailable. We address the particular problem of finding performances of compositions for piano, which is different from often-studied problems of Query-by-Humming and Music Fingerprinting. Our *MidiFind* system is designed to search a million MIDI files with high precision and recall. By using a hybrid search strategy, it runs more than 1000 times faster than naive competitors, and by using a combination of bag-of-words and enhanced Levenshtein distance methods for similarity, our system achieves a precision of 99.5% and recall of 89.8%.

1 Introduction

As music computing becomes more advanced, we have the opportunity to incorporate more data from human performances and to apply machine learning and music analysis to make computer music systems more musical and more expressive. Most existing human performances databases, e.g., the CrestMuse dataset [1] and the ones used in Widmer's works [21, 22], are collected manually and take years to build. Moreover, they are either small in scale or not openly available for research. Potentially, an excellent source of music performance information is MIDI files on the Internet. There are at least one million MIDI files online and there are reasons to expect the number to increase. The online MIDI files are often free and they are also very easily distributed since their size is about 1000 times smaller than audio files.

However, these files are disorganized and difficult to search by metadata due to careless or casual labeling. Our goal is to automatically retrieve and organize these files so that comparative studies of different performances of the same pieces can be carried out on a large scale. Hence, we need a method to search on the order of one million MIDI files quickly, in a way that robustly deals with performance variation, and without using metadata, which would be too unreliable. Specifically, we aim to solve the following problem:

- Given: A query MIDI file

- Find: similar pieces, i.e., different performance versions (including pure quantized versions) of the same composition.

The main challenges to solve these problems are the search quality and scalability. I.e., the system should be both accurate and fast enough to deal with a database with a million of MIDI files.

The logical structure of our solution is shown in Figure 1. The first step is to guarantee good search quality by carefully designing different similarity measurements for different representations. We present novel features for MIDI data based on a bag-of-words idea and melodic segments, and introduce a new variation of Levenshtein distance that is especially suitable for music melody. The second step is to dramatically speed up the search process. We present different hybrid indexing strategies that combine different representations and similarity measurements. The final step is to find the ideal thresholds for different similarity measurements.

To evaluate the system, we use a small and labeled MIDI dataset with 325 files. We also use a large unlabeled dataset that is downloaded and combined from several smaller datasets which are all free from the Internet. The large database contains 12,484 MIDI files with around 2,000 similar pieces.

Our *MidiFind* system is now deployed and hosted on <http://midifind.aws.af.cm>. The main contributions of the system are:

- It is *effective*: it achieve **99.5% precision** and **89.8% recall**, compared to pure Levenshtein distance measurement, which achieves 95.6% precision and 56.3% recall.
- It is *scalable*, with sub-linear complexity for queries, and outperforms naive linear scanning competitors by more than **1000 times**.

The following section describes related work. Section 3 describes feature extraction and search quality. Section 4 discusses various strategies to achieve scalability. Section 5 describes the construction of the *MidiFind* system. In Section 6, we present experimental results.

2 Related Work

Music Information Retrieval has emerged as an active research area in the past decade. Much work has been done on music search. Both Music Fingerprint systems [9, 6] and Query-by-Humming systems [7, 14, 20, 19, 13, 4, 10, 23] are related to our work.

For Music Fingerprint systems, users record a short period of audio to query the system and the results are expected to be an *exact* match, i.e., the query audio must be a copy of a fragment of the reference audio. These systems are generally very robust to audio noise but a query of the same song with a slightly different performance will almost always lead to a failure. On the contrary, our *MidiFind* system deals with *similar* match, i.e., given a query, we aim to find

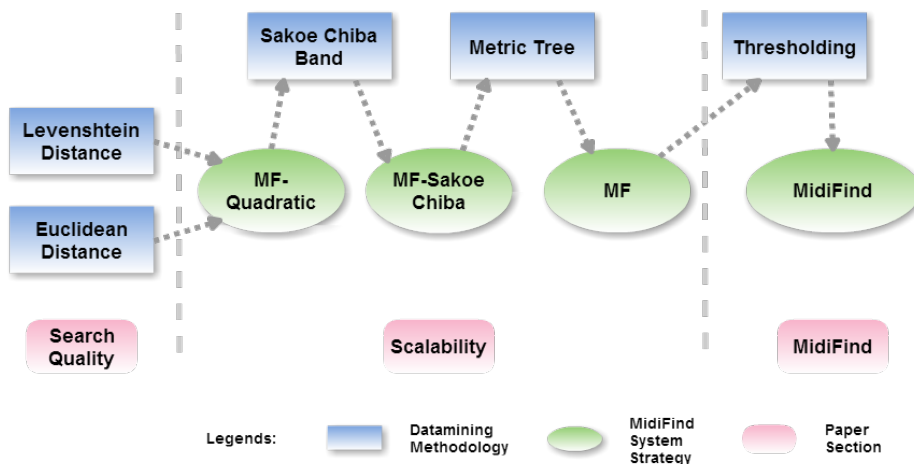


Fig. 1. The logical structure of Section 3,4,5 of the paper.

different performance versions. Audio noise is out of our consideration since our query inputs are pure MIDI files.

Query-by-Humming systems share a similar architecture with *MidiFind* system. Most of them store MIDI files as references and they also implement approximate matching since human performances are not exact. The differences lie in the query part and the goal of the system. The queries of Query-by-Humming systems are usually very short audio snippets, while the queries for our *MidiFind* system are much longer MIDI files. Therefore, we can take advantage of the discrete nature of MIDI data and the full information contained in the full-length MIDI query, but at the same time have to deal with larger variations and a potentially longer matching process for longer sequences. The goals of Query-by-Humming systems are usually Nearest-Neighbor search, while our *MidiFind* system deals with range query, which aims to find out all different performance versions of the same composition.

Early Query-by-Humming systems [7, 14, 20] used melodic contour (defined as a sequence of up, down, and same pitch intervals) and string matching to match similar melodies. Later on, melodic contour was proved unable to distinguish melodies in large datasets [19] and researchers started to resort to dynamic time warping on melody notes [13, 4, 10, 23]. One method studied is a brute-force fashion of dynamic time warping [13] which is certainly slow due to the $O(mn)$ complexity (m is the length of query and n is the total length of references) but serves as a baseline for future research. Different methods have been tested to speed up the searching process. Two of them [23, 4] are closely related to our work in that they both use a 2-step pipeline approach to first shrink the target of candidates and then use dynamic time warping to test the surviving candidates. However, the first method relies only on dynamic time warping and

has a limitation on the length of music. It cannot handle long queries and also requires segmentation labels on the reference music. The method of [4] has an innovative idea to combine N-grams with dynamic time warping but the search performance was poor due to random errors in the queries. Compared to them, the query of our *MidiFind* system is longer with few errors, at least at the beginning and ending. This enables us to use bag-of-words and novel clipped melody features to dramatically shrink the target of candidates and speed up the string comparison process, respectively.

3 Search Quality

We begin by parsing MIDI files into music note strings. After that, we design two different representations for each piece of music: the bag-of-words and clipped melody representation. For the bag-of-words representation, we adopt Euclidean distance; while for the clipped melody representation, we use enhanced Levenshtein distance.

3.1 Euclidean Distance for Bag-of-Words Representation

Inspired by the bag-of-words idea, we create a bag-of-words feature for music. Every piece of music is treated as a sequence of words, where each note is considered as a word by ignoring its length and octave. We consider each word as one of the 12 pitch classes within an octave (in other words, we use the MIDI key number modulo 12. We can also use modulo 24 and so forth) and consider the word count as the total number of times that each pitch occurs within the piece of music. (We actually first tried to incorporate the timing information in the feature vector but the performance was much worse.) Finally, the word count is normalized by the total number of pitch occurrences, resulting in a probability mass table. In the case of 12 pitch classes, this is equivalent to pitch class histograms often used in key finding [11].

The similarity of two pieces of music is measured by the Euclidean distance, as shown in Definition 1, between the corresponding bag-of-words feature vectors. This method works well, or at least is capable of filtering out most of the different pieces, since different pieces of music usually have different distributions over the pitch classes.

Definition 1. *The Euclidean distance (ED) between S and T , where $|S| = |T|$, is defined as:*

$$ED(S, T) = \sqrt{\sum_{i=1}^n (S_i - T_i)^2}$$

3.2 Enhanced Levenshtein Distance and Melody Representation

Besides the bag-of-words representation, we extract the melody from each piece of music as in most Query-by-Humming systems [7, 14, 20]. As suggested by G.Widmer [22], we can simply use the highest pitch at any given time as an

estimate of the melody for each piece of music. The detailed extraction algorithm is described in Algorithm 1. We then use Levenshtein distance measurement with different enhancements on the extracted melodies.

Algorithm 1: Melody Extraction Algorithm

Data: Note Strings
Result: Melody Strings
 sortedNotes = sort(all notes, prioritize higher pitches);
 melodyNotes = empty list;
while sortedNotes is not empty **do**
 note = the note with highest pitch in sortedNotes;
 remove note from sortedNotes;
 if the period of note is not entirely covered by notes in melodyNotes **then**
 split note into one or more notes of the same pitch named splitNotes,
 where each note corresponds to time period that has not been covered;
 insert every note in splitNotes into melodyNotes;
 end
end
 return melodyNotes;

Standard Levenshtein Distance Levenshtein distance (a kind of Dynamic Time Warping) has been shown empirically to be the best distance measure for string editing [5], and this is the reason that it is also named string editing distance as shown in Definition 2. To calculate Levenshtein distance of two melody strings S and T of length m and n , we construct an m -by- n *Levenshtein matrix* where the (i^{th}, j^{th}) element of the matrix is the Levenshtein distance between the prefix of S of length i and the prefix of T of length j . However, it suffers high computational complexity, $O(mn)$, which we will discuss in Section 4. For our melody string distance, we set insertion, deletion, and substitution costs to be 1. (We actually tried to incorporate the note durations in the melody representation and weight the costs by the durations, but the performance turned out to be much worse.)

Definition 2. *The Levenshtein (string editing) Distance [2] between two sequences is the minimal number of substitutions, insertions, and deletions needed to transform from one to the other. Formally, the Levenshtein distance between the prefixes of length i and j of sequences S and T , respectively, is:*

$$lev_{S,T}(i, j) = \begin{cases} \max(i, j), & \text{if } \min(i, j) = 0 \\ \min \begin{cases} lev_{S,T}(i-1, j) + 1 \\ lev_{S,T}(i, j-1) + 1 \\ lev_{S,T}(i-1, j-1) \\ +(S_i \neq T_j) \end{cases} & \text{otherwise} \end{cases}$$

Enhancement 1: Lev-400 As previously discussed, standard Levenshtein distance is a good metric for measuring difference between strings. However, it does have one drawback in that the distance is strongly correlated to the string length. Unfortunately, melody string lengths vary significantly within our database. Figure 2 shows the histogram of melody string lengths.

Observation 1 *The distribution over the length of melody strings follows a power law.*

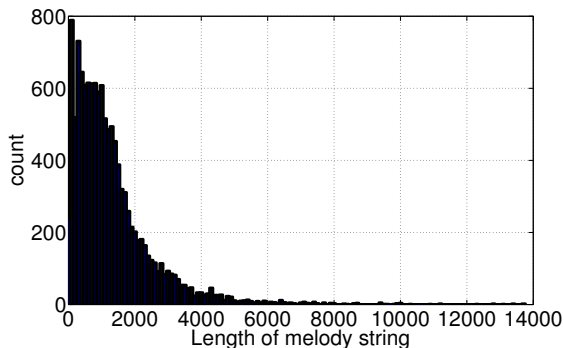


Fig. 2. Melody string length histogram on target dataset. Mean: 1303. Standard Deviation: 1240. This follows a power-law pattern.

Such a large variance on the length will cause problems in matching. For instance, two melody strings S_1 and T_1 both have length 500, and the other two melody strings S_2 and T_2 both have length 1000. If we get a Levenshtein distance of 100 from both pairs, the first pair is trivially more different from each other compared to the second pair. This inspires us to find a way to turn melody strings into equal length and we find a nice property that chopping and concatenating the first 200 and last 200 notes of long melody strings actually increases Levenshtein distance accuracy in a large-scale dataset, as in Observation 2. For melody strings shorter than 400 notes, we do not modify them but scale up the distances. The reason that this manipulation works is that (1) a unified length leads to a unified threshold for Levenshtein distance, (2) similar melodies tend to share more common notes at the beginning and the ending of the music piece, while performers tend to introduce larger variation in the body part. We call this enhanced Levenshtein distance *Lev-400*.

Observation 2 *Chopping and concatenating the first 200 and last 200 notes of long melody strings increases Levenshtein distance accuracy in a large-scale dataset.*

Enhancement 2: Lev-400SC Lev-400 gives us melody strings with $l \leq 400$, where l is length of any string. A by-product of the Levenshtein distance computation is the sequence of notes that is shared by both strings, which can also be considered the *alignment* of the strings. By checking how strings align, we find another property of similar MIDI files: The optimal melody alignment path stays close to the diagonal in the Levenshtein matrix for similar MIDI files, as described in Observation 3. The reason for this observation is that we expect the entire pieces to match without any major insertions or deletions on the notes, so that the best alignment for similar strings should fall along the diagonal in the Levenshtein matrix. This property suggests using the Sakoe-Chiba Band, which constrains the string alignment path by limiting how far it may divert from the diagonal [16]. An illustration of the Sakoe-Chiba Band is shown in Figure 3. We propose using a Sakoe-Chiba Band and finding a reasonable band width to balance the trade off between speed and accuracy. The speed factor will be discussed in Section 4. We call this enhanced distance metric *Lev-400SC*.

Observation 3 *The melody string alignment path corresponding to the smallest Levenshtein distance stays close to the diagonal for similar MIDI files in large-scale datasets.*

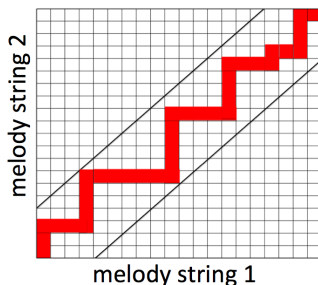


Fig. 3. The illustration of Sakoe-Chiba Band (between thin diagonal black lines) that acts as a global constraint on the Levenshtein alignment path of a Levenshtein matrix.

4 Search Scalability

The similarity measurements mentioned in Section 3 lay the ground for accurate matching between MIDI files. However, since there are at least one million MIDI files on the internet, to search through all those files and find similar ones for any query can be very time consuming. That is why we design a set of hybrid methods (MF-Q, MF-SC, MF) that combine advantages from both similarity measurements and provide a way to search through the database that is both fast and accurate.

4.1 MF-Q: Combine Euclidean and Lev-400 Distance

We have discussed in Section 3 that using Euclidean distance on the bag-of-words representation can differentiate MIDI files that are dramatically different. However, we also need to consider the fact that some MIDI files might share the same notes but have entirely different orderings. Bag-of-words will not differentiate such MIDI files since mapping them to a low dimension (multiples of 12 depending on number of octaves involved), we lose a big chunk of information. The parsing step for the Euclidean distance will convert note sequences to low dimensional vectors. The complexity is linear to the size of MIDI files, and it only needs to be performed once. However, after finishing the parsing step, the calculation of Euclidean distance between two files is very fast: proportional to d , where d is the dimension of the word space.

Levenshtein distance is generally considered to be highly accurate but time consuming. All calculations are performed on melody strings extracted from the note strings, as introduced in section 3.2. For two melody strings S and T with length m and n , the runtime is proportional to $m \cdot n$. By clipping and concatenating melody strings to 400 notes, we effectively set an upper bound on the runtime of Lev-400: $\min\{m, 400\} \cdot \min\{n, 400\} < 400 \cdot 400$. As shown in Figure 2, the average length of melody strings is 1303; therefore, the clipped melody representation will lead to a speed-up of about 10.

Building on the two representations and similarity measurements, we design a hybrid method that runs bag-of-word first and then further filters the result by using Levenshtein distance. This is named *MF-Q* (short for MidiFind-Quadratic). The idea is that we want to shrink down the number of possible similar MIDI candidates by thresholding Euclidean distance. Although the candidate set from this step contains high probability of false-positives, they will be identified and removed by the Levenshtein distance step. The MIDI files returned in the final result has high probability to be either the query itself or some variation of that same music piece. Assume we retain only a percentage of p out of total melody strings through bag-of-words thresholding, then the total runtime needed to find similar pieces (excluding one-time parsing time) will be proportional to $(d + (400 \cdot 400)p)N$, where d is the bag-of-words dimension and N is the total number of MIDI files. We finally achieve a p as small as 0.025 which leads to a further speed-up of about 40. Therefore, the *MF-Q* speeds up the system about 400 times. We will discuss how to choose the p in Section 5 and give detailed experimental results in Section 6.

4.2 MF-SC: Sub-Quadratic Levenshtein Distance with Sakoe-Chiba Band

MF-Q combines two distance metrics, but the Lev-400 step is still time-consuming. As mentioned in the Lev-400SC distance metric, we can limit the string editing path in the Levenshtein matrix. Consider our MIDI dataset and take melody string S and T with length m and n as an example, we limit the bandwidth to be

$$b = \max\{0.1 \cdot \min\{m, n, 400\}, 20\}$$

which is at least 20 notes and increases with the actual length. After using Sakoe-Chiba Band, the complexity of string comparison is sub-quadratic: $\min\{m, n, 400\}$.
 b. We call this method *MF-SC* (short for MidiFind-Sakoe-Chiba). MF-SC can achieve an accuracy performance that is close to MF-Q with a speed-up of about 10. We show the experimental results in Section 6.

4.3 MF: Search Using Metric Tree

MF-SC speeds up the Levenshtein distance step. We propose a further speed-up for the Euclidean distances by adopting the Metric Tree (M-tree), and call this method *MF*. An M-tree is constructed with a distance metric and relies on the triangle inequality for efficient range and k-NN queries. It is very effective when there is a clear threshold to differentiate close nodes and distant nodes [3]. However, it is not very effective when overlaps are big among similar and distant nodes and there is no clear strategy to avoid them. The M-tree has a hierarchical structure just like other common tree structures (R-tree, B-tree), and it tries to balance its nodes according to the given metric. Each node has a maximum and minimum capacity c . When exceeding the maximum capacity, the node will be split into two nodes according to a given splitting policy. For MF, we tried using two splitting policies: maximum lower bound on distance and minimum sum of radii, as in Definition 3 and Definition 4, we also set the the maximum and minimum capacity of nodes to be 8 and 4.

Definition 3. Let N be the current node and \mathcal{S} be the set of N and its children, then the maximum lower bound on distance is achieved by promoting S_i and S_j to be new center nodes, in which $S_j \equiv N$, and S_i s.t. $d(S_i, N) = \max_j\{d(S_j, N)\}$.

Definition 4. Let N be the current node and \mathcal{S} be the set of N and its children, then the minimum sum of radii is achieved by promoting S_i and S_j to be new center nodes, and assign all nodes in \mathcal{S} to S_i or S_j , which gives the smallest sum of radii.

The trade-off is that Minimum Sum of Radii needs to calculate every possible distance pair in \mathcal{S} , but is a better split spatially and ensures minimum overlap. It is faster while performing range queries but the performance decays as the threshold increases. The actual data entries in M-trees are all stored in leaf nodes while non-leaf nodes are duplicates of the leaf nodes. Optimally, M-trees can achieve $O(\log_c |D|)$, where c is the maximum capacity of nodes and D is the dataset. However, the M-tree performance degrades rapidly when there are overlaps between nodes. By testing different thresholds, we finally achieve a speed-up of a factor of 2 to compute the Euclidean distances. More detailed experimental results will be given in Section 6.

5 MidiFind: A Music Query System

In this section, we describe how to build the *MidiFind* system by taking both searching quality in Section 3 and searching scalability in Section 4 into consideration. We start by finding ideal thresholds for different similarity measurements,

and then formally present the pipeline searching strategy which achieves both effectiveness and efficiency in similarity search.

5.1 Find Similarity Measurement Thresholds

The goal of threshold setting is to maximize the benefits from both similarity measurements. We first compute the precisions, recalls, and F-measures as functions of different thresholds. Then, we choose the Lev-400SC distance threshold (ϵ_{Lev}) that leads to the largest F-measure, and choose the Euclidean distance threshold (ϵ_{ED}) that leads to a large recall and a reasonable time cost.

It is important to notice the different roles between ϵ_{ED} and ϵ_{Lev} . The role of ϵ_{ED} is to not only dramatically shrink the number of target candidates, but also retain a high recall. In other words, the candidates returned by using ϵ_{ED} should balance the number of false negatives and retained candidates. The role of ϵ_{Lev} is to identify similar MIDI performances accurately. Therefore, we choose ϵ_{Lev} that leads to the highest F-measure. Our final *MidiFind* system uses $\epsilon_{ED} = 0.1$ and $\epsilon_{Lev} = 306$.

5.2 MidiFind System Pipeline

Here we formally present the pipeline strategy to find similar MIDI pieces based on a user-submitted MIDI file query Q to the *MidiFind* system, as shown in Algorithm 2.

Algorithm 2: MidiFind System Algorithm

Data: The query melody string Q , and reference melody strings

$$\mathcal{R} = \{R_1, R_2, \dots, R_{|R|}\}$$

Result: The set of similar melody string \mathcal{M}

Step1: Within \mathcal{R} , do range query on Euclidean distance (M-tree) based on bag-of-words representation and get a set of candidates $\mathcal{S}_{B|W}$, where the distance between each element of $\mathcal{S}_{B|W}$ and Q is less than ϵ_{ED} ;

Step2: Within $\mathcal{S}_{B|W}$, do range query on melody Lev-400SC distance (Sequential Scan) and get \mathcal{M} , where the distance between each element of \mathcal{M} and Q is less than ϵ_{Lev} ;

return \mathcal{M} ;

6 Experiments

6.1 Quality Experiments

In these experiments, we examine how well our proposed similarity measurements can find pairs of MIDI performances of the same music composition on real datasets. In essence, we claim a discovery of a pair if their distance is smaller than a given threshold. Since truly different performances of a same music composition should indeed be very similar at some threshold, our algorithms can discover these pairs with high precision and recall.

The MIDI files in these experiments come from the Music Performance Expression Database, which belongs to the CrestMuse Project [1]. There are 325 different MIDI files consisting of 79 unique compositions and 2,289 pairs of MIDI files sharing the same composition. Our goal is to discover all these 2,289 pairs.

We compared four discovery methods based on the following three feature sets and their corresponding similarity measurements:

- ED (Section 3.1): Each MIDI file is represented by a 12-dimensional vector where every element is the proportion of melody notes that is played on this key at any octave. The ED similarity of two MIDI files corresponds to the Euclidean distance of their two 12-dim vectors.
- Standard-Lev (Section 3.2): Each MIDI file is represented by a string of melody pitches without any truncation. The Standard-Lev similarity of two MIDI files corresponds to the Standard Levenshtein distance.
- Lev-400SC (Section 3.2): Each MIDI file is represented by a string of melody pitches. The string is then truncated to have the first 200 and the last 200 notes only. The Lev-400SC similarity of two MIDI files corresponds to the Levenshtein distance with Sakoe-Chiba band of their two length 400 strings. In the case that a melody string has length smaller than 400, the distance is scaled up.

The four discovery methods we compare are:

- ED-thresholding: Claiming two MIDI files to be different performances of the same music composition if their ED distance is below some threshold.
- Lev-400SC-thresholding: Claiming two MIDI files to be different performances of the same music composition if their Lev-400SC distance is below some threshold.
- Standard-Lev-thresholding: Claiming two MIDI files to be different performances of the same music composition if their standard Levenshtein distance is below some threshold.
- MF-thresholding: Claiming two MIDI files to be different performances of the same music composition if both their ED distance and their Lev-400SC distance are below some thresholds.

We first consider the precisions, recalls, and F-measures of all methods with different threshold parameters. The true set of MIDI file pairs is hand labeled. As can be seen in Figure 4(a),(b),(c)&(d), better precision appears when the thresholds ϵ are set smaller, because this eliminates many false positives. On the other hand, better recall appears when ϵ is set larger. We can clearly see that the accuracy of Lev-400SC thresholding dramatically outperforms Standard-Lev thresholding. The fact that both precision and recall become high at some $\hat{\epsilon}$, (the choices and their qualities are in Table 1), and remain high in its neighborhood indicates that there is a big overlap between the true similar set and the similar set we found. This fact also give us some flexibility to tune the parameters.

Finally, the best parameter set that optimizes the F-measure for the MF-thresholding method is (0.18, 306) with F-measure 96.6% whereas our choice of

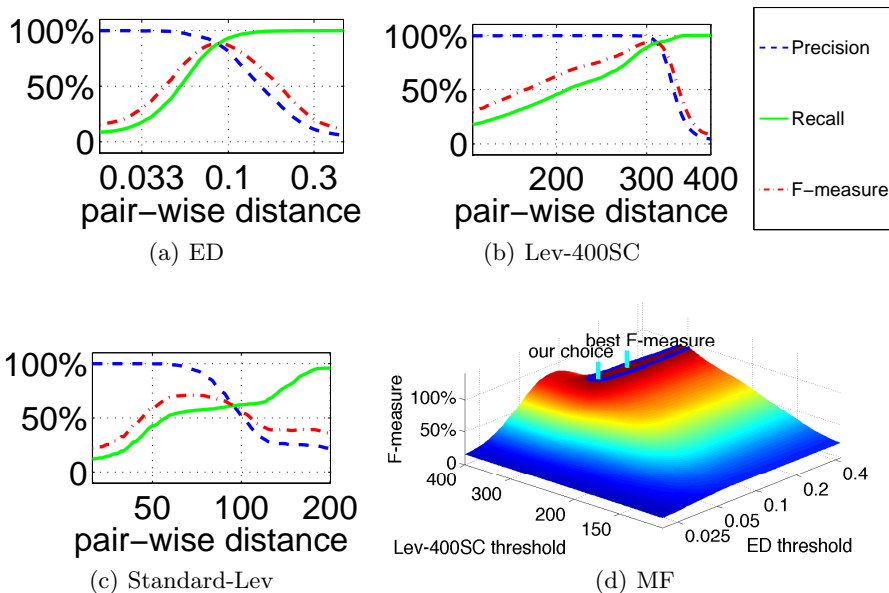


Fig. 4. (a)(b)(c): Precision, recall, and F-measure of the four methods against various distance threshold parameters. (d): F-measure of the MF method against different threshold parameters.

(0.1, 306) which balances quality and scalability achieves an F-measure of 94.4% (Figure 4(d)).

Table 1. Best thresholds and their qualities

Method	Threshold	Precision	Recall	F-measure
ED	0.087	88.6%	88.3%	88.4%
Lev-400SC	302	98.5%	94.3%	96.4%
Standard-Lev	66	95.6%	56.3%	70.8%
MF (our choice)	(0.1, 306)	99.5%	89.8%	94.4%
MF (optimal)	(0.18, 306)	98.6%	94.7%	96.6%

6.2 Scalability Experiments

The scalability experiments are conducted by using the large dataset which contains 12484 MIDI files. The experiments all run on a 3.06 GHz, 2-core (Intel Core i3) machine with 4GB Memory, so that users of *MidiFind* system could achieve similar performance by using personal computers.

We begin the scalability experiments by testing how much speed we can gain by using a hybrid searching strategy. Intuitively, more candidates will be filtered

out if a smaller threshold for Euclidean distance (ϵ_{ED} in Algorithm 2) is adopted for bag-of-words features, and vice versa. Figure 5 shows the relationship between the Euclidean threshold and the fraction of remaining candidates. It is clearly shown that we can filter out about 97.5% if we adopted a threshold $\epsilon_{ED} = 0.1$.

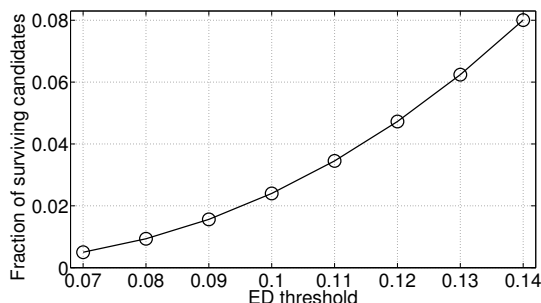


Fig. 5. The relationship between ϵ_{ED} and the fraction of remained candidates.

We then test how much speed we can gain by using different M-tree algorithms mentioned in 4.3. Figure 6 shows the relationship between the Euclidean threshold ϵ_{ED} and the fraction of candidates whose Euclidean distances need to be checked. It can be seen that the maximum lower bound approach works better, and with $\epsilon_{ED} = 0.1$, we can skip 55% of the candidates when we compute the Euclidean distance.

Finally, we compare the speed of all mentioned searching strategies based on how many MIDI files can be searched within in one second. As shown in Figure 7, the fastest method is MF which only takes less than 0.1 second even if the dataset size is more than 10,000. The MF-SC is slightly slower than MF since MF only speed up the procedure of computing Euclidean distances, which is less costly than computing Levenshtein distances. MF-Q is about 10 times slower than MF, while the linear scanning on Lev-400 distances is about 400 times slower. Compared with the naive linear scan competitor (standard-Lev), our MF method is more than 1000 times faster.

7 Conclusions

We present *MidiFind*, a MIDI query system for effective and fast searching of MIDI file databases. The system has the properties we mentioned earlier:

- *Effectiveness*: it achieves high precision and recall by using novel similarity measurements based on bag-of-words and melody segments, which outperforms standard Levenshtein distance.
- *Scalability*: our *MidiFind* system is dramatically faster than the naive standard Levenshtein distance linear scanning, which is $O(mnN)$, where m and

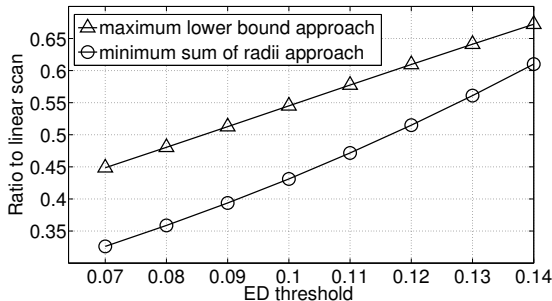


Fig. 6. Search time comparison between M-tree split policies. The y-axis is the fraction of Euclidean distance calculations compared to linear scan. Minimum sum of radii has fewer calculations than maximum lower bound on distance, but it takes longer to build the M-tree. The advantage on search time decreases as threshold increases.

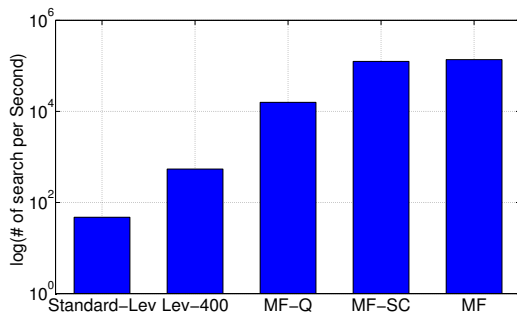


Fig. 7. A comparison of the speed of all searching strategies.

n are lengths of two compared strings and N is the size of the database. By using melody segments representation, bag-of-words filtering, Sakoe-Chiba Band, and M-tree, we achieve speed-ups of 10, 40, 10, and 1.05, respectively, which finally leads to a speed-up of more than 1000 times. Since the methods scales linearly, we are able to achieve one search within 10 seconds even if the size of the database is 1 million.

8 Future Work

Potentially, we can improve the *MidiFind* system by substituting existing rule-based methods by more machine-learning based approaches. Here, we discuss the possibilities in terms of both effectiveness and scalability.

Effectiveness: We see a small gap of recall between the optimal threshold choice and our choice in Table 1. The optimal parameters are not chosen since it will lead to a very low precision for Euclidean distance, which will create a

very large overhead for the next string matching step. It is possible to learn a representation from data which could achieve higher precision than the current bag-of-words representation.

One possibility is to design more “words” based on musical knowledge, and then use Principle Component Analysis (PCA) [18] to reduce the dimensionality. The advantage of PCA is that it automatically “groups” the related information, so that the final representation contains richer information and pays less attention to uninformative details. Another possibility is to use Kernel PCA [17] to directly learn a representation from the strings of various lengths. By using a string kernel [12, 15], we can also take the structure of the string into account rather than just counting the times of the words.

We also see that though the melody string matching process is very accurate, it may rely on the fact that highest pitches are representative enough for piano pieces. For non-piano pieces, we may need more advanced melody extraction algorithms.

Scalability. We see a speed-up factor of 2 to compute the Euclidean distance by using M-tree indexing. It might be possible to increase the speed-up factor by using locality-sensitive hashing (LSH) [8]. Someone may argue that this step is not very critical since that the overhead of Euclidean distance computation is just about 10% of the one of whole computation. However, it is possible that the fraction of Euclidean distance computation will increase as the data size increases to 1 million, in which case the Euclidean distance computation step will become more significant.

We could adopt a k-bit (e.g., 32-bit or 64-bit based on the CPU architecture) LSH function which could basically perform a query in a constant time. There is certainly a trade-off between accuracy and speed. As for precision, the LSH can at least return a rough set of candidates very quickly. After performing LSH, we can check the true Euclidean distance between the set of candidates and the query by linear scanning. In other words, LSH will serve as another filter, so that we end up using a pipeline approach to sequentially filter the candidates by using LSH, Euclidean distance, and finally the actual string matching. As for recall, our pipeline approach will unavoidably create some false negatives, though it has been shown that the false negative probability can be driven very low by tuning the parameters. However, considering our goal of searching 1 million files, a small trade-off on recall, we would argue, will not be a big issue.

References

1. Crestmuse. <http://www.crestmuse.jp/pedb/>
2. Levenshtein distance. http://en.wikipedia.org/wiki/Levenshtein_distance
3. Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. In: Proceedings of the 23rd Athens Intern. Conf. on VLDB. pp. 426–425 (1997)
4. Dannenberg, R.B., Birmingham, W.P., Pardo, B., Hu, N., Meek, C., Tzanetakis, G.: A comparative evaluation of search techniques for query-by-humming using the musart testbed. *J. Am. Soc. Inf. Sci. Technol.* 58(5), 687–701 (Mar 2007)

5. Ding, H., Trajcevski, G., Scheuermann, P., Wang, X., Keogh, E.: Querying and mining of time series data: experimental comparison of representations and distance measures. *Proceedings of the VLDB Endowment* 1(2), 1542–1552 (2008)
6. Ellis, D., Whitman, B., Jehan, T., Lamere, P.: The Echo Nest Musical Fingerprint (ENMFP). In: *International Symposium on Music Information Retrieval* (2010)
7. Ghias, A., Logan, J., Chamberlin, D., Smith, B.C.: Query by humming: musical information retrieval in an audio database. In: *ACM Multimedia*. pp. 231–236 (1995)
8. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: *Proceedings of the 25th International Conference on Very Large Data Bases*. pp. 518–529. VLDB '99, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (1999)
9. Haitsma, J., Kalker, T.: A highly robust audio fingerprinting system. In: *International Symposium on Music Information Retrieval* (2002)
10. Jang, R., Lee, H.: Hierarchical filtering method for content-based music retrieval via acoustic input. In: *Proc. ACM Multimedia*. pp. 401–410. ACM Press (2001)
11. Krumhansl, C.L.: *Cognitive Foundations of Musical Pitch*. Oxford University Press, New York (2001)
12. Lodhi, H., Saunders, C., Shawe-Taylor, J., Cristianini, N., Watkins, C.: Text classification using string kernels. *J. Mach. Learn. Res.* 2, 419–444 (Mar 2002)
13. Mazzone, D., Dannenberg, R.B.: Melody matching directly from audio. In: *International Symposium on Music Information Retrieval*. pp. 17–18 (2001)
14. Mcnab, R.J., Smith, L.A., Bainbridge, D., Witten, I.H.: The New Zealand Digital Library MELody inDEX. *D-Lib Magazine* 3(5), 4–15 (1997)
15. Paass, G., Leopold, E., Larson, M., Kindermann, J., Eickeler, S.: Svm classification using sequences of phonemes and syllables. In: *Proceedings of the 6th European Conference on Principles of Data Mining and Knowledge Discovery*. pp. 373–384. PKDD '02, Springer-Verlag, London, UK, UK (2002)
16. Papapetrou, P., Athitsos, V., Potamias, M., Kollios, G., Gunopulos, D.: Embedding-based subsequence matching in time-series databases. *ACM Transactions on Database Systems (TODS)* 36(3), 17 (2011)
17. Scholkopf, B., Smola, A., Müller, K.R.: Kernel principal component analysis. In: *Advances in Kernel Methods - Support Vector Learning*. pp. 327–352. MIT Press (1999)
18. Shlens, J.: A tutorial on principal component analysis. In: *Systems Neurobiology Laboratory, Salk Institute for Biological Studies* (2005)
19. Uitdenbogerd, A., Zobel, J.: Manipulation of Music for Melody Matching. In: *ACM Multimedia*. pp. 235–240 (1998)
20. Uitdenbogerd, A., Zobel, J.: Melodic matching techniques for large music databases. In: *Proceedings of the seventh ACM international conference on Multimedia (Part 1)*. pp. 57–66. Multimedia '99, ACM, New York, NY, USA (1999)
21. Widmer, G., Flossmann, S., Grachten, M.: YQX plays Chopin. *AI Magazine* 30(3), 35–48 (2009)
22. Widmer, G., Tobudic, A.: Playing Mozart by analogy: Learning multi-level timing and dynamics strategies. *Journal of New Music Research* 32(3), 259–268 (2003)
23. Zhu, Y., Shasha, D.: Warping indexes with envelope transforms for query by humming. In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. pp. 181–192. SIGMOD '03, ACM, New York, NY, USA (2003)