MidiFind: Fast and Effective Similarity Searching in Large MIDI databases

Guangyu Xia, Tongbo Huang, Yifei Ma
Roger Dannenberg, Christos Faloutsos
School of Computer Science
Carnegie Mellon University

Abstract

How can we quickly search millions of MIDI files for similar pieces of music? Our *MidiFind* system focuses on exactly this problem. It has the following desirable properties: (a) it is *effective*: thanks to our novel features and corresponding similarity measurements, it outperforms traditional competitors, in terms of precision and recall; (b) it is *scalable*: thanks to our hybrid (MF) indexing strategy, it scales more than **1000** time faster than naive competitors.

1 Introduction

MIDI files are widely used by millions of musicians and music amateurs. People generate, upload, and distribute their MIDI files online. Some current large online collections include Free-Midi Zone [3], Classical Archives [1], and Musipedia [4]. Nowadays, with high quality synthesizers, MIDI files can generate various and high quality sounds which compete with recordings. This makes MIDI even more popular. There are at least one million MIDI files online and there are reasons to expect the number to increase. Unlike audio files, MIDI files are often free, and their size is about 1000 times smaller than audio files. Furthermore, MIDI files can be generated easily by exporting them from composition software or playing a MIDI instrument. Finally, since MIDI files capture performance information and every performance is different, the same music composition can have many different versions.

However, online MIDI files are messy and difficult to search by meta data due to careless or casual labeling. We are interested in finding MIDI files through content-based retrieval. Specifically, we aim to solve the following problem:

- Given: A query MIDI file
- Find: similar pieces. I.e., different performance versions (including pure quantized version) of the same composition.

The main challenges to solve these problems are the search quality and scalability. I.e., the *MidiFind* system should be both accurate and fast enough to deal with a database with millions 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, in which 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, in which we present different hybrid indexing strategies (MF-Q, MF-SC, MF) that combine different representations and similarity measurements. The final step is to find the ideal thresholds for different similarity measurements and then construct the *MidiFind* system.

We have a small and labeled MIDI dataset with 325 files. We also use a large non-labeled dataset for the *MidiFind* system that is downloaded and combined from several smaller datasets which are all free from the Internet. It contains 12,484 MIDI files with around 2,000 similar pieces. Our *MidiFind* system is now deployed and hosted on Cloudbees with no maintenance fee, and we plan to upgrade it to a commercial server when daily users increase to thousands.

The main contributions of our *MidiFind* system are:

- It is *effective*: it achieve **99.5% precision** and **89.8% recall**, compared to pure Levinshtein distance measurement, which achieves 95.6% precisionand 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 Sections 6, we present experimental results.

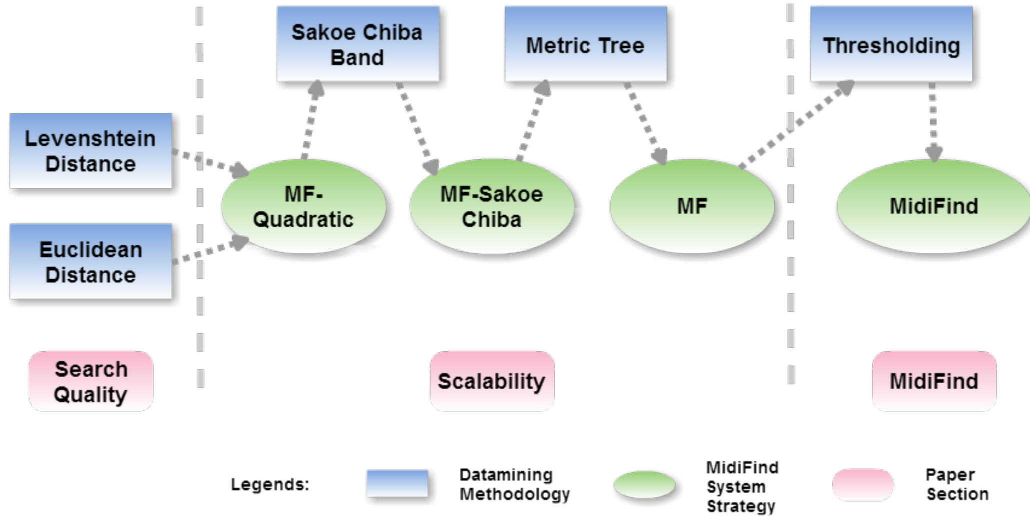


Figure 1: The logical structure of Section 3,4,5 of the paper.

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. And Music Fingerprint systems [10, 7] and Query-by-Humming systems [8, 13, 20, 21, 12, 5, 18, 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 supposed 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 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 deal with

similar match since human queries cannot be that accurate. 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 property of MIDI data and the full information contained by the full-length MIDI query, but at the same time have to deal with a larger variations and potentially longer matching process of 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.

Earliest Query-by-Humming systems [8, 13, 20] could trace back to the 1990s, in which melodic contour (defined as a sequence of up, down, and same pitch intervals) is extracted to match similar melodies. Later on, melodic contour proved unable to distinguish melodies in large datasets [21] and people started to resort to dynamic time warping on melody notes [12, 5, 18, 23]. One of the initial attempts is a brute-force fashion of dynamic time warping [12] which is certainly slow due to the $O(mn)$ complexity (m is the length of query and n is the total length of references) and serves as a baseline for future research. Different methods have been tested to speed up the searching process. Two of them [23, 5] are very related to our work in that they both use a 2-step pipeline fashion 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 [5] 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 enable 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 notes and extracting main melodies. 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 novel 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. In other words, 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 normalized total times that each key is played within a piece of music. Therefore, we re-scale the pitches of the notes into fewer octaves (12 or 24 pitch classes) and count the words by ignoring their durations. (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 note counts and hence a feature vector of probability mass table is recorded.

We use Euclidean distance shown in Definition 1 for bag-of-words representations. I.e., the similarity of two pieces of music is measured by the Euclidean distance between the corresponding bag-of-words feature vectors. This method intuitively 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 notes.

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

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

3.2 Levenshtein Distance and proposed enhancements for melody representation

Besides bag-of-words representation, we extract melody feature from each piece of music like in most of the query-by-humming systems [8, 13, 20], since melody is considered as the distinctive element to help people tell music apart. Also, melody extraction helps make the music pieces more distinct to each other since many of them share very similar non-melodic components.

As suggested by G.Widmer [22], we can simply use the highest pitches (apex) at any given time as the melody for each piece of music. The detailed extraction algorithm is described in Algorithm 1. We then use Levenshtein distance measurement 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 splitNotes each corresponds to the time period
        that has not been covered;
        insert every note in splitNotes into melodyNotes;
    end
end
return melodyNotes;
```

3.2.1 Standard Levenshtein Distance

Levenshtein distance (a special kind of Dynamic Time Warping) has been shown empirically to be the best distance measure for string editing [6], 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 matrix where the (i^{th}, j^{th}) element of the matrix is the Levinshtein distance between the substring of S of length i and the substring of T of length j. However, it suffers high computational complexity, 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 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 sequence S and T:*

$$lev_{S,T}(i,j) = \begin{cases} max(i,j), & min(i,j) = 0 \\ \min \begin{cases} lev_{S,T}(i-1,j)+1 \\ lev_{S,T}(i,j-1)+1 \\ lev_{S,T}(i-1,j-1) \\ +cost[S_i \neq T_j] \end{cases} & else \end{cases}$$

3.2.2 Enhancement 1: Lev-400

As previously discussed, standard Levenshtein distance is a good metric for showing difference between strings. However, it does have one drawback that the distance is strongly correlated to the string length. Unfortunately, melody string lengths vary significantly among 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.*

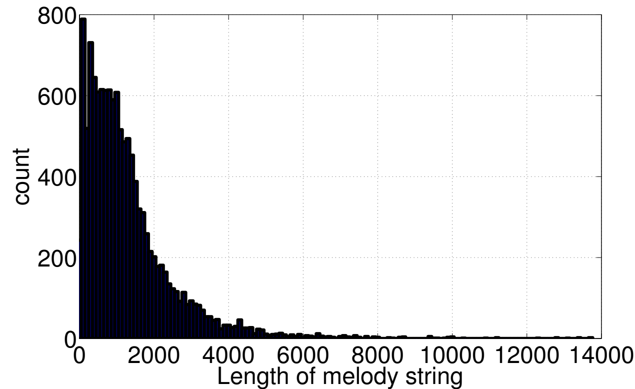


Figure 2: Melody string length histogram on larget dataset. Mean: 1303. Standard Deviation: 1240. This follows a power-law pattern.

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. 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.*

3.2.3 Enhancement 2: Lev-400SC

Lev-400 gives us melody strings with $l \leq 400$, where l is length of any string. By checking the optimal string editing path within Levenshtein matrices, we find another property of MIDI files: The optimal melody editing 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 aligns very well with Sakoe-Chiba Band, which constrains the string editing 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 editing path with smallest Levenshtein distance stays close to the diagonal for similar MIDIs in large-scale datasets.*

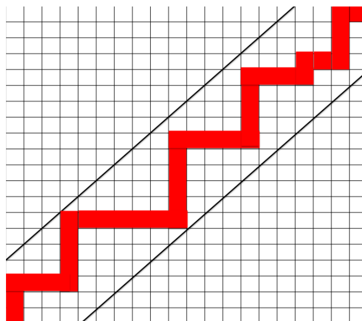


Figure 3: The illustration of Sakoe-Chiba Band that act as a global constraint on the Levenshtein editing path

4 Search Scalability

The similarity measurements mentioned in Section 3 lay the ground for accurate matching between MIDI files, however, since there are about 1 million MIDI files in the world currently, 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 dimensional space (multiples of 12 depending on number of octaves involved), we lose a big chunk of information. However, the calculation of Euclidean distance is very fast: $O(d)$, where d is the dimension of the word space. It is also not related to the length of the MIDI file.

Levenshtein distance is generally considered to be highly accurate but time consuming at the same time. For two melody strings S and T with length m and n, the computational complexity is $m \cdot n$. By clipping and concatenating melody strings to 400 notes, we effectively set an upper bound

on the complexity of Lev-400: $\min\{m, 400\} \cdot \min\{n, 400\} = O(400 \cdot 400)$. As shown in Figure 2, the average length of melody string 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 filter the result by using Lavenshtein distance, and name it *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 Lavenshtein 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 will be $O((d + 400 \cdot 400p)N)$, where d is the bag-of-words dimension and N is the total number of melody strings. We finally achieve a p as small as 0.025 which leads to a further speed-up of 40. Therefore, the *MF-Q* speeds up the system about 400 times. We will discuss how to choose the p in Section 5 and 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 Levenshtein distance step is still time-consuming and take quadratic time. As mentioned in the Lev-400SC distance metric, we can limit the string editing path in 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 is then sub-quadratic: $min\{m, n, 400\} \cdot b$. 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: Further Speeding Up with Metric Tree

MF-SC speeds up the Levenshtein distance step. We propose to a further speed-up for the Euclidean distances by adopting 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 [14]. 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 propose 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 when threshold increase as shown in Figure 6. The actual data entries in M-tree are all stored in leaf nodes while non-leaf nodes are duplicates of the leaf nodes. Optimally, M-tree 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 details 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 benefits from both similarity measurements. We find the ideal thresholds by using the following Algorithm 2 and Algorithm 3 based on the small and labeled dataset. Intuitively, these two algorithms simply compute the F-values for different thresholds and choose the thresholds which lead to large F-values.

Algorithm 2: Estimating ϵ_{ED}

Data: (1) The small labeled set of music MIDI files $D = \{m_1, m_2, \cdots, m_{|D|}\}$, (2) the set of labeled similar pairs S, and (3) the Euclidean distances between every pairs of (m_i, m_j)

Result: Euclidean threshold on bag-of-words representation ϵ_{ED}

for $\epsilon_{ED} = 0$, $\epsilon_{ED} < 1$, $\epsilon_{ED} += 0.01$ **do**

  $\hat{S}$ = the set of pairs whose Euclidean distances are less than ϵ_{ED} ;

  $precision = \frac{|\hat{S} \cap S|}{|\hat{S}|}$;

  $recall = \frac{|\hat{S} \cap S|}{|S|}$;

  $F\text{-}val = \frac{1}{\frac{1}{precision} + \frac{1}{recall}}$;

  record the ϵ_{ED} which lead to biggest F-val so far ;

end

return ϵ_{ED} which lead to highest F-val, and then add a small number $t < 0.05$ to ϵ_{ED} ;

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 high recall. In other words, the candidates returned by using ϵ_{ED} should balance the number of false negatives and retained candidates. This trade-off leads to a small number t, which is usually less than 0.05 added at

Algorithm 3: Estimating ϵ_{Lev}

Data: (1) The small labeled set of music MIDI files $D = \{m_1, m_2, \cdots, m_{|D|}\}$, (2) the set of labeled similar pairs S, and (3) the Euclidean distances between every pairs of (m_i, m_j)

Result: Lev-400sc threshold on clipped melody representation ϵ_{Lev}

for $\epsilon_{Lev} = 0$, $\epsilon_{Lev} < 400$, $\epsilon_{Lev} + +$ **do**

- $\hat{S}$ = the set of pairs whose Lev-400sc distances are less than ϵ_{Lev} ;
- $precision = \frac{|\hat{S} \cap S|}{|\hat{S}|}$;
- $recall = \frac{|\hat{S} \cap S|}{|S|}$;
- $F\text{-}val = \frac{1}{\frac{1}{precision} + \frac{1}{recall}}$;
- record the ϵ_{Lev} which lead to highest F-val so far ;

end

return ϵ_{Lev} which lead to biggest F-va1 ;

the end of Algorithm 2. The role of ϵ_{Lev} is to get final accurate similar pairs. Therefore, we choose ϵ_{Lev} that leads to the highest *F-val*. Our *MidiFind* system final use $\epsilon_{ED} = 0.1$ and $\epsilon_{Lev} = 306$

5.2 MidiFind System Pipeline

Here we formally present the pipeline strategy to find the similar MIDI pieces based on a user submitted MIDI query Q to the *MidiFind* system, as shown in Algorithm 4.

6 Experiments

6.1 Quality Experiments

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

Algorithm 4: MidiFind System Algorithm

Data: The query melody string Q, and reference melody strings $\mathcal{R} = \{R_1, R_2, \cdots, R_{|R|}\}$

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 $\mathcal{S}_{\mathcal{B}l\mathcal{W}}$, where the distance between each element of $\mathcal{S}_{\mathcal{B}l\mathcal{W}}$ and Q is less than ϵ_{ED};

Step2: Within $\mathcal{S}_{\mathcal{B}l\mathcal{W}}$, 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 $\mathcal{M}$;

Expression Database, which belongs to the CrestMuse Project [2]. 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 related similarity metrics:

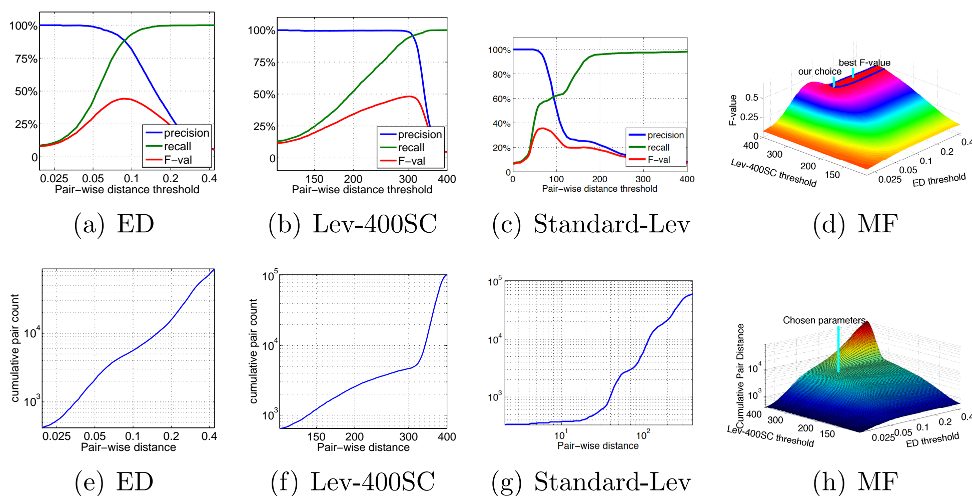
- ED (Section 3.1): Each MIDI file is represented by a 12-dimensional vector where every element is the proportion of melody notes that is played on this key at any octave. The ED similarity of two MIDI files corresponds to the Euclidean distance of their two 12-dim vectors.

- Standard-Lev (Section 3.2.1): 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 3.2.3): 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 400-dim 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.



(a) ED (b) Lev-400SC (c) Standard-Lev (d) MF

(e) ED (f) Lev-400SC (g) Standard-Lev (h) MF

Figure 4: (a)(b)(c)(d): Qualities (Precision, Recall, and F-value) of the four methods. For every method, the "threshold" parameters are plotted as variables. In (d), the F-value of our chosen parameter balancing quality and speed achieves is very close to the very best. (e)(f)(g)(h): The number of MIDI file pairs with distances $\leq d$ as d increases. Notice the plateaus especially in (f)&(h) indicate that certain distances are scarce, i.e. natural clusters exist.

We first consider the precisions, recalls, and F-values of all three methods with different "threshold" parameters. The true set of MIDI file pairs is hand

Method	Threshold	Precision	Recall	F-value
ED	0.087	88.6%	88.3%	0.442
Lev-400SC	302	98.5%	94.3%	0.482
Standard-Lev	66	95.6%	56.3%	0.354
MF (our choice)	$(0.1, 306)$	99.5%	89.8%	0.472
MF (optimal)	$(0.18, 306)$	98.6%	94.7%	0.483

Table 1: Best thresholds and their qualities

labeled. As can be seen in Figure 4(a),(b),(c)&(d), better precision appears when the thresholds ϵ are set smaller, because by claiming discoveries very carefully, few goes wrong. 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.

Next, in Figure 4(e),(f),(g)&(h), we plot the cumulative pair count under various distances, that is, the number of pairs with distances no larger than ϵ as ϵ varies. It can be noted especially in Figure 4(f)&(h) that plateaus occur for a range of ϵ's. This means that few pairs have distances in this range—the distances are either smaller, i.e. the pairs are rather local, or larger, i.e. pairs rather global. These plots show another evidence that MIDI files form natural clusters under our proposed distances. Finally, the best parameter set that optimizes the F-value for the MF-thresholding method is $(0.18, 306)$ with F-value 0.483 whereas our choice of $(0.1, 306)$ which balances quality and scalability achieves a F-value of 0.472 (Figure 4(d)). In Figure 4(h), there is a plateau near the parameters of our choice.

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 smaller threshold for Euclidean distance (ϵ_{ED} in Algorithm 4) 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 $\epsilon_{ED} = 0.1$.

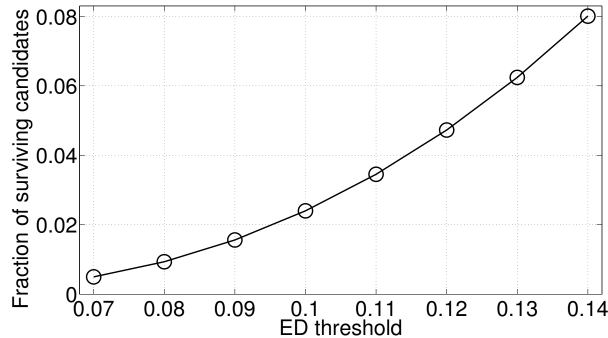


Figure 5: The relationship between ϵ_{ED} and the fraction of remained candidates.

We then test how much speed we can gain by using different M-tree algorithms mentioned in 4.2. 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 we can skip 55% of the candidates when we compute the Euclidean distance.

Finally, we see the speed as a function over dataset size, and compare the speed of all mentioned searching strategies based on their average searching time, which is computed by recording the timing of 300 queries and then take the average. As shown in Figure 7, the fastest method is MF (blue line) which only takes less than 0.1 second even if the dataset size is more than $10,000$. The MF-SC (green line) 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 (red line) is about 10 times slower than MF, while the linear scanning on Lev-400 distances (yellow line) is about 400 times slower. Compared with naive linear scan competitor (black line), our MF method is more than *1000* times faster.

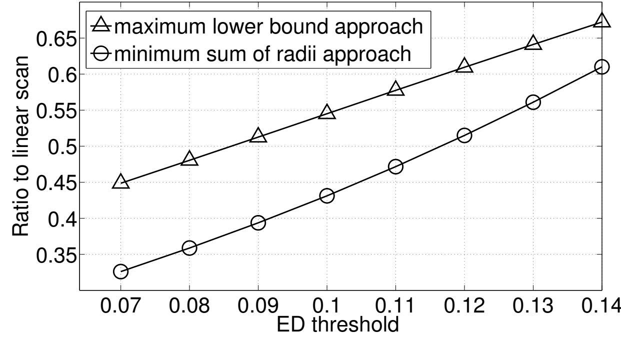


Figure 6: Search time comparison between M-tree split policies. The y-axis is the number of Euclidean distance calculations compared to linear scan. Minimum sum of radii has fewer calculations than maximum lower bound on distance, but it takes longer to build the M-tree. The advantage on search time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

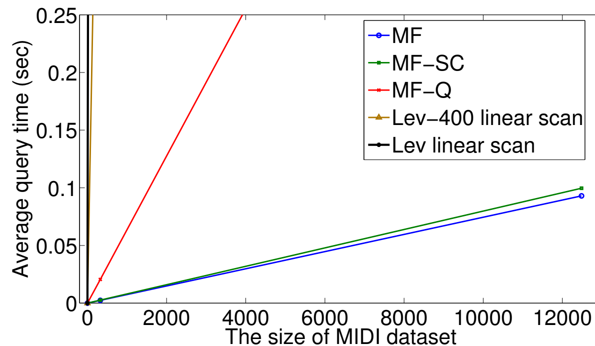


Figure 7: A comparison of the speed of all searching strategies.

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 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

There are many possibilities to improve the *MidiFind* system by substituting existing rule-based methods, especially the bag-of-words representation, 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. We see that 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) [19] to reduce the dimensionality. For example, we can consider pitch classes of more octaves and also incorporate the timing information. If we consider 36 pitch classes (three octaves) weighted by accumulated note duration, the revised bag-of-words feature vectors will have 72 dimensions. After that, we can use PCA to project the vectors into a subspace with a much lower dimensionality, say 10. 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. The string kernels [11] [15] generally consider the similarity between two strings based on the common subsequences or substrings. By using a string kernel, we can also take the structure of the string into account rather than just counting the times of the words.

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) [9]. 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 other words, as the data size

increases, it is possible that the amount of surviving candidates for string matching process will increase slower than the total data size, 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 positives, though it has been shown that the false positive probability can be controlled to be really 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] Classical archives. `http://www.classicalarchives.com/`.

[2] Crestmuse. `http://www.crestmuse.jp/pedb/`.

[3] Free midi zone. `http://www.free-midi.org/`.

[4] Musipedia. `http://www.musipedia.org/`.

[5] R. B. Dannenberg, W. P. Birmingham, B. Pardo, N. Hu, C. Meek, and G. Tzanetakis. 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.

[6] H. Ding, G. Trajcevski, P. Scheuermann, X. Wang, and E. Keogh. Querying and mining of time series data: experimental comparison of representations and distance measures. *Proceedings of the VLDB Endowment*, 1(2):1542–1552, 2008.

[7] D. P. W. Ellis, B. Whitman, T. Jehan, and P. Lamere. The echo nest musical fingerprint.

[8] A. Ghias, J. Logan, D. Chamberlin, and B. C. Smith. Query by humming: musical information retrieval in an audio database. In *In ACM Multimedia*, pages 231–236, 1995.

[9] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[10] J. Haitsma. A highly robust audio fingerprinting system. pages 107–115, 2002.

[11] H. Lodhi, C. Saunders, J. Shawe-Taylor, N. Cristianini, and C. Watkins. Text classification using string kernels. *J. Mach. Learn. Res.*, 2:419–444, Mar. 2002.

[12] D. Mazzoni. Melody matching directly from audio. In *Indiana University*, pages 17–18, 2001.

[13] R. J. Mcnab, L. A. Smith, D. Bainbridge, and I. H. Witten. The New Zealand Digital Library MELody inDEX. *D-Lib Magazine*, 3(5):4–15, 1997.

[14] M. P. Ciaccia and P.Zezula. M-tree: An effcient access method for similarity search in metric spaces. In *In Proceedings of the 23rd Athens Intern. Conf.on VLDB*, pages 426–425, 1997.

[15] G. Paass, E. Leopold, M. Larson, J. Kindermann, and S. Eickeler. Svm classification using sequences of phonemes and syllables. In *Proceedings of the 6th European Conference on Principles of Data Mining and Knowledge Discovery*, PKDD '02, pages 373–384, London, UK, UK, 2002. Springer-Verlag.

[16] P. Papapetrou, V. Athitsos, M. Potamias, G. Kollios, and D. Gunopulos. Embedding-based subsequence matching in time-series databases. *ACM Transactions on Database Systems (TODS)*, 36(3):17, 2011.

[17] B. Scholkopf, A. Smola, and K.-R. Mller. Kernel principal component analysis. In *ADVANCES IN KERNEL METHODS - SUPPORT VECTOR LEARNING*, pages 327–352. MIT Press, 1999.

[18] J. shing Roger Jang and H. ru Lee. Hierarchical filtering method for content-based music retrieval via acoustic input. In *Proc. ACM Multimedia*, pages 401–410. ACM Press, 2001.

[19] J. Shlens. A tutorial on principal component analysis. In *Systems Neurobiology Laboratory, Salk Institute for Biological Studies*, 2005.

[20] A. Uitdenbogerd and J. Zobel. Matching techniques for large music databases. pages 57–66. ACM, ACM Press, 1999.

[21] A. L. Uitdenbogerd and J. Zobel. Manipulation of music for melody matching, 1998.

[22] G. Widmer. Playing mozart by analogy: Learning multi-level timing and dynamics strategies. *Journal of New Music Research*, 32(3):259–268, 2003.

[23] Y. Zhu and D. Shasha. Warping indexes with envelope transforms for query by humming. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, SIGMOD '03, pages 181–192, New York, NY, USA, 2003. ACM.