FAST INDEX BASED FILTERS FOR MUSIC RETRIEVAL

ABSTRACT

We consider two content-based music retrieval problems where the music is modeled as sets of points in the Euclidean plane, formed by the (on-set time, pitch) pairs. We introduce fast filtering methods based on indexing the underlying database. The filters run in a sublinear time in the length of the database, and they are lossless if a quadratic space may be used. By taking into account the application, the search space can be narrowed down, obtaining practically lossless filters using linear size index structures. For the checking phase, which dominates the overall running time, we exploit previously designed algorithms suitable for local checking. In our experiments on a music database, our best filter-based methods performed several orders of a magnitude faster than previous solutions.

1 INTRODUCTION

In this paper we are interested in content-based music retrieval (CBMR) of symbolically encoded music. Such setting enables searching for excerpts of music, or *query patterns*, that constitute only a subset of instruments appearing in the full orchestration of a musical work. Instances of the setting include the well-known query-by-humming application, but our framework can also be used for more complex applications where both the query pattern searched for and the music database to be searched may be polyphonic.

The design of a suitable CBMR algorithm is always a compromise between robustness and efficiency. Moreover, as robustness means high precision and recall, the similarity/distance measure used by the algorithm should not be too permissive to detect false matches (giving low precision) and not too restrictive to omit true matches (giving low recall). In this paper, we concentrate on a modeling of music that we believe is robust in this sense, and at the same time provides computationally feasible retrieval performance.

As symbolically encoded monophonic music can easily be represented as a linear string, in literature several solutions for monophonic CBMR problems are based on an appropriate method from the string matching framework (see e.g. [4, 6]). Polyphony, however, imposes a true challenge, especially when no voicing information is available or the occurrence is allowed to be distributed across the voices. In some cases it may suffice to use some heuristic, as for an example the SKYLINE algorithm [8], to achieve a monophonic reduction out of the polyphonic work. This, however, does not often provide musically meaningful results.

In order to be able to deal with polyphonic music, geometric-based modeling has been suggested [1, 7, 9, 10]. Most of these provide also another useful feature, i.e., extra intervening elements in the musical work, such as grace notes, that do not appear in the query pattern can be ignored in the matching process. The downside is that the geometric online algorithms [2, 5, 9, 10] are not computationally as efficient as their counterparts in the string matching framework. Moreover, the known offline (indexing) methods [1, 7] compromise on crucial matters.

These downsides are not surprising: the methods look at all the subsequences and the number of them is exponential in the length of the database. Thus, a total index would also require exponential space.

We deal with symbolically encoded, polyphonic music for which we use the pitch-against-time representation of note-on information, as suggested in [10] (see Figs. 1 and 2). The musical works in a database are concatenated in a single geometrically represented file, denoted by T. In a typical case the query pattern P to be searched for is often monophonic and much shorter than the database T to be searched. If P and T are readily not given in the lexicographic order, the sets can be sorted in $|P| \log |P|$ and $|T| \log |T|$ times, respectively. The problems of interest are the following:

(P1) Find translations of P such that each point in P match with a point in T.

(P2) Find translations of P that give a partial match of the points in P with the points in T.

Notice that the partial matches of interest in P2 need to be defined properly, e.g. one can use a threshold k to limit the minimum size of partial matches of interest.

Ukkonen et al. [9] presented algorithms PI and PII solving problems P1 and P2 in worst case times $O(mn)$ and $O(mn \log m)$, respectively, where m is the length of the query pattern and n the length of the database. Their algorithms require $O(m)$ space. Noteworthy, the algorithm solving P1 has an $O(n)$ expected time complexity. Clifford et al. [2] showed that problem P2 is 3SUM-hard, i.e., it is unlikely that an exact solution could run faster than in quadratic time $O(mn)$, and give an approximation algorithm, called MSM, for P2, that runs in time $O(n \log n)$.

In this paper we introduce index-based filtering algorithms for the problems presented above. Our contribution is twofold. Firstly, our methods outperform its competitors;

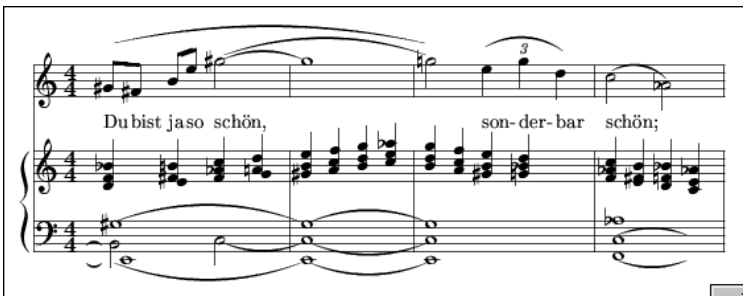

Figure 1. A musical excerpt.

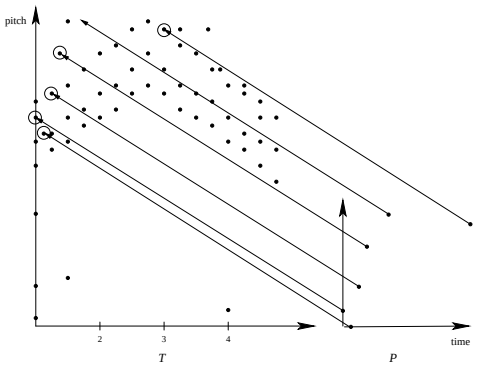

Figure 2. Pointset T represents Fig. 1 in the geometric representation. Pointset P corresponds to the first two and half bars of the melody line with a delayed fifth point. The depicted vectors correspond to translation f that gives a partial match of P within T.

in particular, the algorithms are *output sensitive*, i.e., the running time depends more on the output than on the input. This is achieved by exploiting a simple indexing structure that is not a total index. Secondly, we show how to keep the index of a practical, linear size. The index enables fast filtering; best results are obtained with filters running in $O(f(m)\log n + s)$ time. The found s ($s \leq n$) candidate positions are subsequently checked using Ukkonen et al's PI and PII algorithms. Thus, executing checking take time $O(sm)$ and $O(sm \log m)$, in the worst case, respectively.

2 INDEX BASED FILTERS

We will denote by $P + f$ a translation of P by vector f, i.e., vector f is added to each component of P separately: $P + f = (p_1 + f)(p_2 + f) \cdots (p_m + f)$. Problem P1 can then be expressed as the search for a subset I of T such that $P + f = I$ for some f. Please note that a translation corresponds to two musically distinct phenomena: a vertical move corresponds to transposition while a horisontal move corresponds to aligning the pattern time-wise (see Fig. 2).

The idea used in [9, 10] is to work on trans-set vectors. Let $p \in P$ be a point in the query pattern. A translation vector f is a *trans-set vector*, if there is a point $t \in T$, such that $p + f = t$. Without loss of generality, let us assume all the points both in the pattern and database to be unique. So, the number of trans-set vectors is $O(n^2)$ in the worst case.

For the indexing purposes we consider translation vectors that appear within the pattern and the database. We call translation vector f *intra-pattern vector*, if there are two points p and p', $p, p' \in P$, such that $p + f = p'$. The *intra-database vector* is defined in the obvious way. The number of intra-pattern and intra-database vectors are $O(m^2)$ and $O(n^2)$, respectively. A nice property of Ukkonen et al's PI and PII algorithms is that they are capable of starting the matching process anywhere in the database. Should there be a total of s occurrences of the pattern within the database and an oracle telling where they are, we could check the occurrences in $O(sm)$ and $O(sm \log m)$ time, in the worst case, by executing locally PI and PII, respectively.

We will exploit this property by first running a filter whose output is subsequently checked by PI or PII. If a quadratic size for the index structure is allowed, we have a lossless filter: all intra-database vectors are stored in a balanced search tree in which translations can be retrieved in $O(\log n)$ time; Let $C(f)$ be the list of starting positions i of vector $f = t_j - t_i$, for some j, in the database, then the list is stored in the leaf of a binary search tree so that a search from root with key f leads this leaf. Let us denote by $|C(f)|$ the number of elements in the list. Since the points are readily in the lexicographic order, building such a structure takes a linear time in the number of elements to be stored.

However, for large databases, a quadratic space is infeasible. To avoid that, we store only a subset of the intra-database vectors. In CBMR, an occurrence is a compact subpart of the database typically not including too many intervening elements. Now we make a full use of this locality and that the points are readily sorted: for each point i in the database, $1 \leq i \leq n-1$, we store intra-database vectors to points $i+1, \ldots, i+c+1$ ($c = \min(c, n-i-1)$), where c is a constant, independent of n and m. Constant c sets the 'reach' for the vectors. Thus, the index becomes of linear, $O(n)$ size. Naturally such filters are no more totally lossless, but by choosing a large c and by careful thinking in the filtering algorithms, losses are truly minimal.

2.1 Solving P1

To solve the problem P1 we consider four straightforward filters. The first one works in $O(\log n + s)$ time, where s is the number of candidate position found, but we consider also two $O(m^2 \log n + s)$ time filters with a better filtration power. The simplicity of these filters are due to the fact that all the points need to find its counterpart within the database. Thus, to find candidate occurrences, we may consider occurrences of any of the intra-pattern vectors. The filters are

represented in an increasing order in their complexities; the order also reflects their filtration power.

In FILTER0 we choose a random intra-pattern vector $f = p_j - p_i$. The candidate list $C(f)$ to be checked contains thus $s = |C(f)|$ candidates. For FILTER1 and FILTER2 we calculate frequencies of the distinct intra-database vectors. FILTER1 chooses the intra-pattern vector $f^* = p_j - p_i$ that occurs the least in the database, i.e., for which the $s = |C(f^*)|$ is smallest. In FILTER2, we consider the two intra-pattern vectors $f^* = p_j - p_i$ and $f^{**} = p_l - p_k$ that have the least occurrences within the database, i.e., for which $s' = |C(f^*)| + |C(f^{**})|$ is smallest. Then the set $S = \{i'' \mid t_{i'} - t_{i''} = p_i - p_1, i' \in C(f^*), t_{k'} - t_{i''} = p_k - p_1, k' \in C(f^{**})\}$ contains the candidates for starting positions of the pattern, such that both f^* and f^{**} are included in each such occurrence.

For the running time, FILTER0 uses $O(\log n)$ time to locate the candidate list. FILTER1 and FILTER2 execute at most $O(m^2)$ additional inquiries each taking $O(\log n)$ time. FILTER2 needs also $O(s')$ time for intersecting the two occurrence lists into the candidate list S; notice that values i'' can be scanned from left to right simultaneously to the scanning of lists $C(f^*)$ and $C(f^{**})$ from left to right, taking amortized constant time at each step of the intersection.

With all the filters we may consider only translations between consecutive points of the pattern. Thus, we would somewhat compromise on the potential filtration power, but the 'reach constant' c above would get an intuitive interpretation: it tells how many intervening points are allowed to be in the database between any two points that match with consecutive pattern points. For long patterns, the search for the intra-pattern vector that occurs the least in T may dominate the running time. Hence, we have FILTER3 that is FILTER2 with a random set of intra-pattern vectors as the input.

2.2 Solving P2

The same preprocessing as above is used for solving P2, but the search phase is modified in order to find partial matches. We will concentrate on the case where a threshold k is set for the minimum size of a partial match. Since any pattern point can be outside the partial match of interest, one should in principle check the existence of all the $O(m^2)$ vectors among the intra-database vectors, merge the candidate lists into multiset S' and accept any candidate position i'' into set S that occurs at least k times in S', and run the checking on each candidate position in S with algorithm PII. More formally, the multiset S' contains position i'' for each intra-pattern vector $f = p_j - p_i$ such that $i' \in C(f)$ and $p_i - p_1 = t_{i'} - t_{i''}$. We call this basic lossless filter FILTER4. We will also consider a lossy variant FILTER5, where for each pattern point p only one half of the intra-pattern vectors (the least frequent ones) having p as an endpoint is chosen.

The *pigeon hole principle* can be used to reduce the amount of intra-pattern vectors to check: If the pattern is split into $(m-k+1)$ distinct subsets, then at least one subset must be present in any partial occurrence of the complete pattern. Therefore, it is enough to run the filters derived for P1 on each subset independently and then check the complete set of candidates. The total amount of intra-subset vectors is bound by $O((m-k+1)(\frac{m}{m-k+1})^2) = O(\frac{m^2}{m-k+1})$. This is $O(m)$ whenever k is chosen as a fraction of m. FILTER0 and FILTER2 both select constant number of vectors among each subset, so the total number of candidate lists produced by each filter is $O(m-k+1)$. Hence, this way the filtration efficiency (number of candidates produced) can be expected to depend linearly on the number of errors $m-k$ allowed in the pattern. This is an improvement to the trivial approach of checking all $O(m^2)$ intra-pattern vectors.

Notice that these pigeon hole filters are lossless if all the intra-database vectors are stored. However, the approach works only for relatively small error-levels, as each subset needs to contain at least two points in order the filtering to be possible. Let us focus on how to optimally use FILTER1 for the subsets in the partition, as FILTER0 and FILTER2 are less relevant for this case. The splitting approach gives the freedom to partition the pattern into subsets in an arbitrary way. For optimal filtering efficiency, one should partition the pattern so that the sum of least frequent intra-subset vectors is minimized. This sub-problem can be solved by finding the *minimum weight maximum matching* in the graph whose nodes are the points of P, edges are the intra-pattern vectors, and edge weights are the frequencies of the intra-pattern vectors in the database. In addition, a set of dummy nodes are added each having an edge of weight zero to all pattern points. These edges are present in any minimum weight matching, so their amount can be chosen so that the rest of the matched edges define the $m-k$ non-intersecting intra-pattern vectors whose frequency sum is minimum.

We use an $O(m^3)$ time minimum weight maximum matching algorithm to select the $m-k+1$ intra-pattern vectors in our filter. Some experiments were also done with an $O(m^2)$ time greedy selection. We call this algorithm FILTER6 in the experiments.

3 EXPERIMENTS

We set the new algorithms against the original Ukkonen et. al.'s PI and PII [9]. We set the window length within the database (the reach constant c) to 50, the window length within the pattern in FILTERS 0–3 and 6 to 5, and in FILTERS 4–5 to 12. In FILTER 6, we experimented with different values of k in range $\lceil m/2 \rceil$ to $\lceil \frac{15}{16} m \rceil$. Overall, these settings are a compromise between search time, index memory usage and search accuracy for difficult queries. Larger window lengths may be required for good accuracy depending on the level of polyphony and the type of expected queries.

We also compare with Clifford et. al.'s MSM [2] and

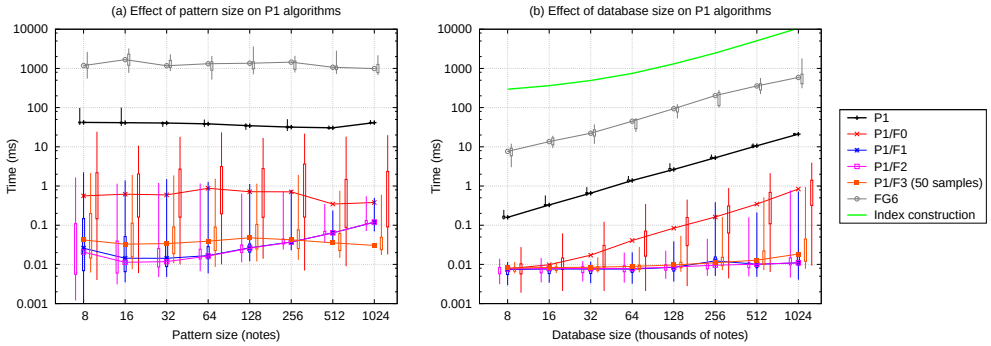

Figure 3. Solving P1. Search time as function of pattern and database sizes in the MUTOPIA collection of 1.9 million notes. Database size experiments were done with a pattern size of 64 notes. Note the log scales.

Fredriksson and Grabowski's FG6 algorithm [3, Alg. 6.]. The latter solves a slightly different problem; the time information is ignored, and the pitch values can differ by $\delta \geq 0$ after transposition. We set $\delta = 0$ to make the setting as close to ours as possible. We tested also other algorithms in [3] but Alg. 6 was constantly fastest among them on our setting.

To measure running times we experimented both on the length of the pattern and the database. Reported times are median values of more than 20 trials using randomly picked patterns. As substantial variations in the running times are characteristic to the filtering methods, we have depicted this phenomenon by using box-whiskers: The whisker below a box represents the first quartile, while the second begins at the lower edge and ends at the median; the borders of third and fourth quartiles are given by the median, the upper edge of the box and the upper whisker, respectively.

We also measured how robust the PII-based methods (FILTERS 4-6) are against noise, and calculated a precision-against-recall plot. Here we reported the mean value of a 100-trial experiment in which the set of occurrences found by PII was considered as the ground truth. All the experiments were carried out with the MUTOPIA database; at the time it consisted of 1.9 million notes in 2270 MIDI files.

3.1 Experimenting on P1

In experimenting on the pattern size, the variations in the running times of the PI-based filters are clearly depicted in Fig. 3a. Out of the four filters, FILTER2 performed most stably while FILTER0 had the greatest variation. The figure shows also that all our filtering methods constantly outperform both the original PI and the FG6 algorithm. With pattern sizes $|P| \lesssim 400$, FILTERS 1 and 2 are the fastest ones, but after that FILTER3 starts to dominate the results.

The evident variation in our filters is caused by difficult patterns that only contain relatively frequently appearing vectors. In our experiments, FILTERS 1–3 had search times of at most 2 ms. It is possible to generate patterns that have much longer search times, especially if the database is very monotonic or short windows are used. However, in practice these filters are at least 10 times faster than PI and 200 times faster than FG6 for every pattern of less than 1000 notes.

When experimenting on the size of the database as shown in Fig. 3b, execution times of online algorithms PI and FG6 increases linearly in the database size. Also FILTER0's search time increases at the same rate due to the poor filtering efficiency. FILTERS 1–3 have much lower slope because only few potential matches need to be verified after filtering for more information on execution time allocation within FILTER2). Fig. 3b also depicts the construction time of the index structure for the filters. Remember that this costly operation needs to be executed only once for any database.

3.2 Experimenting on P2

Fig. 4 shows the experimentally measured search times for P2 filters, PII and MSM. When varying the size of the pattern, our PII-based filters clearly outperformed MSM in all practical cases (Fig. 4a): MSM becomes faster than PII only when $|P| > 400$ and faster than FILTER4 when $|P| > 1000$. In this experiment FILTER6 performs the best until $|P| \gtrsim 250$ after which FILTER5 starts to dominate. However, a greedy version of FILTER6 outperforms both FILTER5 and FILTER6 by an order of magnitude.

Results of the experiments on the length of the database are rather similar (Fig. 4b), exceptions being that MSM is constantly outperformed by the others and that FILTER6 performs the best throughout the experiment. Again, the greedy FILTER6 is the fastest: it is nearly 100 times faster than FILTER6 and over million times faster than MSM.

3.2.1 Comparing whole musical works.

In [2], Clifford et al. carried out experiments for partial music matching and concluded that their algorithm is faster than PII when two whole documents are to be matched against each other. Fig. 5 depicts results of our experiment using their setting but including also FILTERS 4–6.

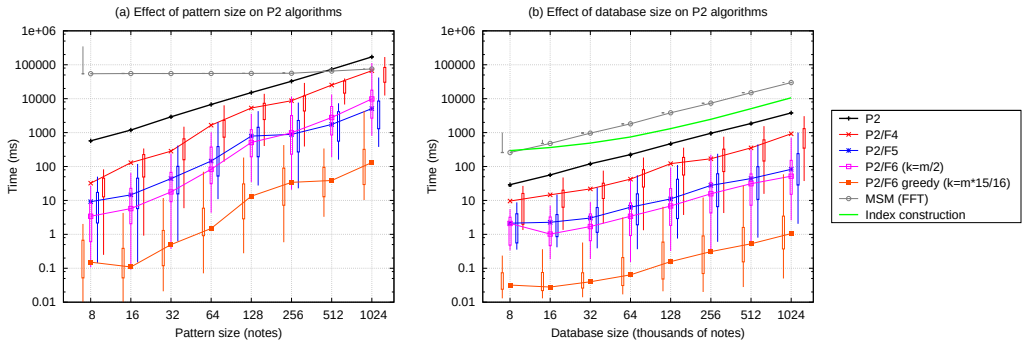

Figure 4. Solving P2. Search times as function of pattern and database sizes. Database size in Fig. a was 1.9 million and pattern size in Fig. b 64 notes.

In our experiment, MSM becomes faster than PII when $|P| = |T| \gtrsim 600$ and dominates FILTERS 4 and 5 when the size of the matched documents exceeds 1000 and 5000 notes, respectively. Depending on the value of k, MSM becomes faster than FILTER6 at document sizes larger than 1500–20,000 notes. However, in this specific task algorithms would be expected to return matches that are relatively poor if measured as a ratio between the matched notes and pattern size. Solving the task by using FILTER6 with $k = m/16$ would not give good results, but FILTER4, 5 and FILTER6 with $k = m/2$ are comparable with MSM.

3.2.2 Precision and recall.

For experimenting on the robustness against noise we selected a random pattern of length 100 from the database and introduced mutations (substitutions) to it; the number of mutations is given by the parameter *error rate*. Then we retrieved all approximate occurrences of the pattern from the database by using PII to form the ground truth for the experiment. To keep the ground truth at a manageable size, we used the parameter *ground truth similarity*, gts, as a similarity cutoff point for the retrieved matches. For example, ground truth similarity value 0.1 defines that each item in the ground truth must match with at least one tenth of the pattern notes. In the end, we measured the precision of the algorithms as the function of recall with different settings of ground truth similarity and error rate. This was repeated 100 times with different randomly selected patterns and the resulting graphs were averaged into one result.

Basic evaluation of the measurements is doable by comparing areas: the larger the area below a curve the better the accuracy and robustness of the corresponding algorithm. Both increasing error rate and lower ground truth similarity values make the task harder: higher error rate makes the original patterns, with possibly several occurrences, be less pronounced in the retrieved result lists. When good matches have been toned down like this, algorithms need to find more difficult matches to score well. Also correct ordering of the

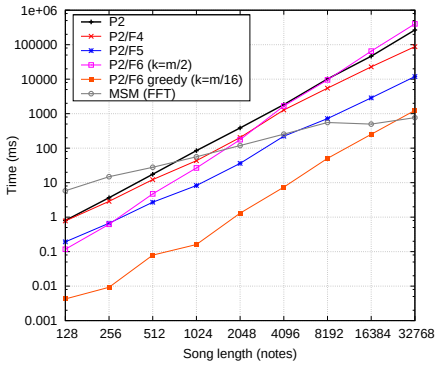

Figure 5. Matching whole music works against each other.

results by decreasing similarity with the pattern is necessary.

We bound ground truth similarity to 0.10 and let the error rate vary between 0, 0.5 and 0.70; this setting requires the algorithms to find potentially very sparse matches. Fig. 6 depicts that, at low error rates, there is not much difference between the index filters, and MSM performs similarly. However, at error rate 0.75, only FILTER4 achieves good scores. The results suggest that all the filters are well balanced when comparing their robustness to execution speed: FILTERS 5 and 6 are fast but somewhat inaccurate, while FILTER 4 has its place between them and PII.

4 CONCLUSIONS

We considered point pattern matching problems applicable to CBMR and showed how they can be solved using index-based filters. Given a point pattern P of m points, the problems are to find complete and partial matches of P within a database T of n points. The presented filters are lossless if $O(n^2)$ space is available for indexing. We also introduced a more practical, linear size structure and sketched how the filters based on this structure become virtually lossless.

After the preprocessing of the database, the proposed fil-

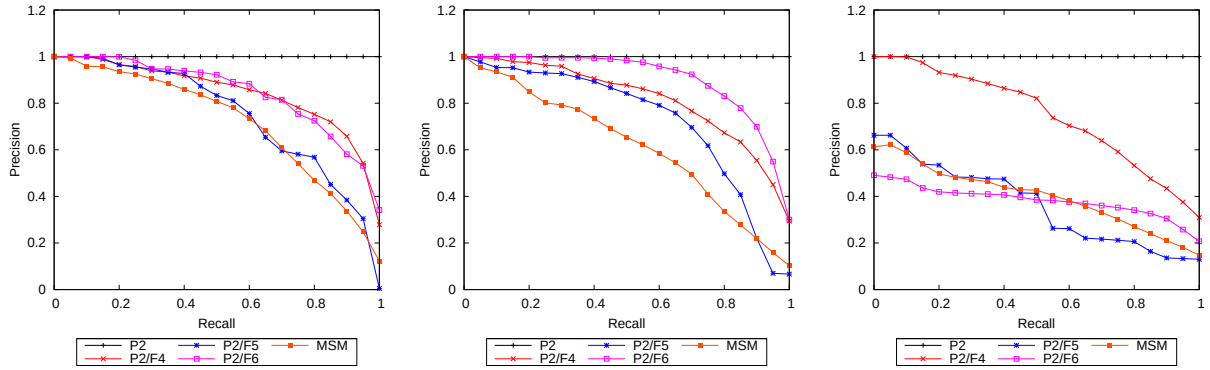

Figure 6. Precision-Recall at error rates 0, 0.5 and 0.75 (left to right), with $gts = 0.1$.

ters use $O(f(m) \log n + s)$ time to produce s candidate positions that are consequently checked for real occurrences with the existing checking algorithms. The filters vary on the complexity of $f(m)$ and on the output size s. Since the filtering power of the proposed filters is hard to characterize theoretically, we ran several experiments to study the practical performance on typical inputs.

The experiments showed that our filters perform much faster than the existing algorithms on typical application scenarios. Only when comparing large musical works in their entirety, the MSM algorithm [2] is faster than our new filters.

Since the guarantee of losslessness in the filters is only valid on limited search settings (number of mismatches allowed, constants limiting maximal reach, etc.), it was important to study also the precision and recall. This comparison was especially fruitful against MSM, that is an approximation algorithm, and can hence be considered as a lossy filter as well. The experiments showed that our filters typically performs at the same level that MSM in this respect.

As a future work, we plan to study the extensions of the filters to approximate point pattern matching; in addition to allowing partial matching, we could allow matching a point to some ϵ-distance from its target. Such setting gives a more robust way to model the query-by-humming application. Although it is straightforward to extend the filters to consider the candidate lists of all intra-database vectors within the given ϵ-threshold from any intra-pattern vector, the overall amount of candidate positions to check grows fast as the threshold is loosen. Therefore, finding better strategies for filtering in this scenario is an important future challenge.

5 REFERENCES

[1] M. Clausen, R. Engelbrecht, D. Meyer, and J. Schmitz. Proms: A web-based tool for searching in polyphonic music. In *Proc. ISMIR'00*, Plymouth, 2000.

[2] R. Clifford, M. Christodoulakis, T. Crawford, D. Meredith, and G. Wiggins. A fast, randomised, maximal subset matching algorithm for document-level music retrieval. In *Proc. ISMIR'06*, pages 150–155, Victoria, 2006.

[3] K. Fredriksson and Sz. Grabowski. Efficient algorithms for pattern matching with general gaps, character classes and transposition invariance. In *Proc. SPIRE'2006*, pages 267–278, Berlin, 2006.

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

[5] A. Lubiw and L. Tanur. Pattern matching in polyphonic music as a weighted geometric translation problem. In *Proc. ISMIR'04*, pages 289–296, Barcelona, 2004.

[6] M. Mongeau and D. Sankoff. Comparison of musical sequences. *Comp. and the Humanities*, 24:161–175, 1990.

[7] R. Typke. *Music Retrieval based on Melodic Similarity*. PhD thesis, Utrecht University, 2007.

[8] A. Uitdenbogerd and J. Zobel. Manipulation of music for melody matching. In *Proc. ACM Multimedia*, pages 235–240, Bristol, 1998.

[9] E. Ukkonen, K. Lemström, and V. Mäkinen. Geometric Algorithms for Transposition Invariant Content-Based Music Retrieval. In *Proc. ISMIR'03*, pages 193–199, Baltimore, 2003.

[10] G. Wiggins, K. Lemström, and D. Meredith. SIA(M)ESE: An algorithm for transposition invariant, polyphonic content-based music retrieval. In *Proc. ISMIR'02*, pages 283–284, Paris, 2002.