A QUANTITATIVE EVALUATION OF A TWO STAGE RETRIEVAL APPROACH FOR A MELODIC QUERY BY EXAMPLE SYSTEM

Justin Salamon
Music Technology Group
Universitat Pompeu Fabra, Barcelona, Spain
justin.salamon@upf.edu

Martin Rohrmeier
Centre for Music & Science, Faculty of Music,
University of Cambridge, United Kingdom
mrohrmeier@cantab.net

ABSTRACT

We present a two-stage approach for retrieval in a melodic Query by Example system inspired by the BLAST algorithm used in bioinformatics for DNA matching. The first stage involves an indexing method using n-grams and reduces the number of targets to consider in the second stage. In the second stage we use a matching algorithm based on local alignment with modified cost functions which take into account musical considerations.

We evaluate our system using queries made by real users utilising both short-term and long-term memory, and present a detailed study of the system's parameters and how they affect retrieval performance and efficiency. We show that whilst similar approaches were shown to be unsuccessful for Query by Humming (where singing and transcription errors result in queries with higher error rates), in the case of our system the approach is successful in reducing the database size without decreasing retrieval performance.

1. INTRODUCTION

The transition to digital media and the growing popularity of portable media devices over the past decade has resulted in much research into new ways of organising and searching for music. Of note are Content Based Music Retrieval (CBMR) systems [21] which search the musical content directly as opposed to using song meta-data for retrieval.

A specific case of CBMR is that of performing a melodic search in a collection of music, where the input query can be made either symbolically (e.g. text, score, MIDI controller) [8, 9, 19, 22] or by the user singing/humming the query, called Query by Humming (QBH) [4, 15]. For convenience we will refer to the symbolic input case as Query by Symbolic Example (QBSE). Both QBSE and QBH rely on an underlying model of melodic similarity [6]. In [17], a detailed review of algorithms for computing symbolic melodic similarity is provided. In recent years QBH systems have become increasingly popular, as they do not require musical knowledge such as playing an instrument or understanding musical notation. On the other hand, QBSE can be advantageous over QBH in certain cases – firstly, it affords more elaborate query specification, which might be preferred by advanced users and music researchers. Secondly, it does not require the automatic transcription of audio queries, which introduces additional errors into the queries.

In [4] a detailed comparative evaluation of different algorithms for QBH was carried out, comparing approaches based on note intervals, n-grams, melodic contour, HMMs and the Qubyhum system. The authors noted that the most successful approaches lacked a fast indexing algorithm, which is necessary in order to apply them to large databases. They studied a potential solution to the problem using a two-stage approach – an n-gram algorithm is used as a first stage for filtering targets in the database. The remaining targets are then passed to the second stage which uses a slower note interval matching algorithm which has better retrieval performance. The authors concluded that the approach was unsuccessful, as any significant improvement in search time resulted in a dramatic degradation in retrieval performance. They attributed this degradation to the errors introduced into the queries due to singing and transcription errors, which inhibited successful exact matching in the n-gram stage.

Nonetheless, other approaches for searching symbolic data exist for which efficient indexing is possible. Set-based methods (which also support polyphonic queries) have been shown to be effective for both matching and indexing – Clausen et al. use inverted files [3], Romming and Selfridge-Field use geometric hashing [16], Lemström et al. use index based filters [10] and Typke et al. use vantage objects [18]. For string based approaches (such as ours) many efficient indexing algorithms exist for metric spaces [2]. However, due to the melodic similarity measure used in our system (section 2.1.3), we can not use these algorithms and require an alternative solution (a recent solution to indexing non-metrics is also proposed in [20]).

In the following sections we present a two-stage indexing and matching approach for a QBSE system, inspired by the BLAST algorithm used in bioinformatics for DNA matching [5]. The first stage involves an indexing method (section 2.1.2) similar to the n-gram approach studied and evaluated in [4]. As our system avoids the need to transcribe user queries, the degree of errors in the queries depends only on the user, and is lower as a result. Conse-

quently, it allows us to considerably reduce the database size for the second stage without degrading retrieval performance. In section 2.1.3 we present the second stage in which we perform matching using local alignment. We then detail the evaluation methodology used to evaluate our system, using real user queries utilising both short-term and long-term memory. Finally in the results section we show that this two-stage approach can be applied successfully in the case of QBSE, and study how the parameters of the indexing and matching algorithms affect retrieval performance and efficiency.

2. THE SONGSEER QBSE SYSTEM

2.1 System overview

SongSeer is a complete query by symbolic example system. The user interacts with the system through a graphical user interface implemented as a Java applet, allowing access from any web-browser [1] . The interface is further described in section 2.2. User queries are sent to a server application which contains a database of songs and performs the matching and returns the results to the client applet.

2.1.1 Query and target representation

The internal representation of user queries and database targets is based on the one proposed in [15] – pitch is represented as pitch intervals and rhythm as LogIOI Ratios (LogIOIR) [14]. This representation is independent of key and tempo, as well as concise, allowing us to match queries against targets even if they are played in another key or at a different tempo.

The targets are extracted from polyphonic MIDI files – every track in the MIDI file results in a single monophonic target. Tracks that are too short, as well as the drum track which is easily detectable are filtered out, but otherwise all tracks are considered. This allows the user to search for melodic lines other than the melody, though at the cost of considerably increasing the database size and adding targets which could possibly interfere with the search.

2.1.2 Indexing

The first stage in our two-stage approach is the indexing algorithm. Indexing is required in order to avoid comparing the query against every target in the database, thus improving system scalability an efficiency. As previously noted, our melodic matching is non-metric meaning we can not use existing indexing approaches, leading us to propose an alternative solution based on the BLAST algorithm.

The BLAST algorithm [5] was designed for efficiently searching for DNA and protein sequences in large databases. The steps of the original BLAST algorithm are the following: first, low-complexity regions or sequence repeats are removed from the query. Then, the query is cut into "seeds" – smaller subsequences of length n which are evaluated for an exact match against all words (of the same length) in the database using a scoring matrix. High scoring words (above a threshold T) are collected and the database is then scanned for exact matches with these words. The exact matches are then extended into high-scoring pairs (HSP) by extending the alignment on both sides of a hit till the score starts to decrease (in a later version gaps in the alignment are allowed). HSPs with scores above a cutoff score S are kept, and their score is checked for statistical significance. Significant HSPs are locally aligned, and an expectation value E is calculated for the alignment score. Matches with E smaller than a set threshold are reported as the final output.

Our indexing algorithm is based on this concept of preceding the slower local alignment stage with a fast exact matching stage. Given a query, we cut it into "seeds" as in BLAST, and search for exact matches in the database. This can be efficiently implemented by storing the targets of the database in a hash table where every key is a seed which hashes to a collection of all targets containing that seed. We also implement the idea of filtering less informative parts of the query as detailed in section 4.5. This first stage allows us to return a much reduced set of targets, which we then compare to the query using our matching algorithm. A crucial parameter of this approach is the seed size n – a longer seed will return less targets making the retrieval faster, but requires a longer exact match between query and target potentially reducing performance for queries which contain errors.

2.1.3 Matching

For determining the similarity of a query to a target, we use the dynamic programming approach for local alignment [13], similar to the one proposed in [15] with one significant difference – in an attempt to make the matching procedure more musically meaningful, we replace the *skip* and *replace* costs in the local alignment algorithm with *cost functions*. These functions determine the skip and replace cost based on the specific pitch intervals and LogIOIRs being compared. The underlying assumption is that some errors should be penalised less heavily than others, based on the errors we can expect users to make when making a query:

- Repeated notes – the user might repeat a note more or less times than in the original melody (for example when translating a sung melody into piano strokes). Thus, the penalty for skipping a repeated note should be reduced.
- Pitch contour – the user might not remember the exact pitch interval, but remember the pitch contour correctly (big/small jump up/down or same). Thus, the penalty for replacing an incorrect interval which has the correct contour should be reduced.
- Rhythm contour – the user might not remember the exact rhythm ratio between two notes, but remember the "rhythmic contour" correctly (slower, faster or same). Thus, the penalty for replacing an incorrect LogIOIR which has the correct contour should be reduced.
- Octave errors – the user might play the correct pitch class but in the wrong octave relative to the previous

[1] `http://www.online-experiments.com/SongSeer`

note. Thus, the penalty for replacing a note which is off by an integer number of octaves should be reduced.

Following this rationale, we define two cost parameters – a *full cost* and a *reduced cost*. When one of the aforementioned cases is detected the cost functions return the reduced cost, and in all other cases the full cost. Another issue is the relative importance we give to the pitch and rhythm match scores. As the pitch and rhythm of a query might be represented with different degrees of accuracy, the match scores should be weighted differently when combining them to obtain the final match score. To do this we introduce a *pitch factor* and *rhythm factor* which weight the pitch and rhythm scores when combining them.

By default, the *full cost* is set to 2 and the *reduced cost* to 1, and both pitch and rhythm factors are set to 1 (so that the pitch and rhythm scores are weighted equally and summed into the final score). In the results section we explain how these parameters are optimised based on real user queries.

2.2 The SongSeer GUI

The SongSeer GUI is displayed in Figure 1. Two query input methods are provided – a text search allowing to make textual queries similar to the ones supported by the Themefinder [8] system by Huron et al. (including pitch and rhythm contour), and a virtual keyboard that can be played using the mouse, the computer keyboard or a connected MIDI controller. Once a query is made the top 10 results are displayed back to the user with a percentage indicating the matching degree, and the user can select a song and play back the corresponding MIDI file stored in the database.

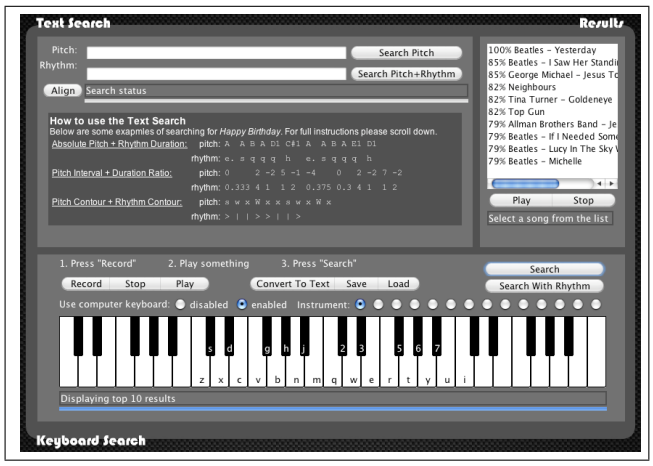

Figure 1. The SongSeer user interface.

3. EVALUATION METHODOLOGY

3.1 Test collections and machines used

For the evaluation, we compiled a corpus of 1,076 polyphonic MIDI files of pop and rock music, including 200 songs by the Beatles. After ignoring short tracks and drum tracks, this translates into 6,541 targets in the database. For scalability tests we have also compiled several more corpora of increasing size, the largest containing 18,017 songs which translates into 88,034 targets.

All user experiments were run on standard pentium IV PCs running windows XP. The quantitative evaluation was run on a server machine with two Intel® Dual Core Xeon® 5130 @ 2GHz with 4MB Cache and 4GB RAM, running Linux 2.6.17-10 and Java HotSpot™ 64-Bit Server VM.

3.2 Collecting user queries

We conducted a user experiment with 13 participants of varying musical experience, ranging from amateur guitar players to music graduates. The first part of the experiment involved a usability test in which the subjects were asked to complete a set of tasks using the SongSeer interface, which also allowed them to familiarise themselves with the system. The second part involved the subjects making queries which would then be used to perform a quantitative evaluation of the system. For the purpose of the quantitative evaluation, all subjects were asked to play on the virtual keyboard, using either the mouse or computer keyboard.

To collect queries, subjects were presented with a list of 200 Beatles songs, and asked to record queries of songs they can remember from the list. This stage simulates the event where a user remembers part of a song they have not heard recently, and resulted in 63 "long term memory" queries. Next, subjects were asked to listen to 10 audio recordings of Beatles songs and then record a query, simulating the event where the user has recently heard the song, resulting in 123 "short term memory" queries. This gives us a total of 186 real user queries for system evaluation.

3.3 Evaluation metrics

As there is always only a single correct result (the database contained no cover versions), we use a metric based on the rank of the correct song based on match score, the *Mean Reciprocal Rank* (MRR) which is given by:

$$MRR = \frac{1}{K}\sum_{i=1}^{K}\frac{1}{rank_i} \quad (1)$$

where K is the numbers of queries evaluated and $rank_i$ is the rank of the correct song based on the match score for query i. This is similar to taking the average rank of the correct song over all queries but is less sensitive to poor ranking outliers, and returns a value between $1/M$ and 1 (where M is the number of songs in the database) with higher values indicating better retrieval performance.

It is important to note however that as it is possible for several songs to have the same match score, they may share the same rank (in which case they are returned by alphabetical order in the results list). In order to evaluate performance from a user perspective (where the position of a song in the result list is significant), we introduce a second metric – the *ordered MRR* (oMRR) which is computed in the same way as the MRR but where the rank is based not on the match score but on the actual position of the song in the final results list.

4. RESULTS

4.1 Initial results

In order to asses the performance of our approach, we start by estimating a baseline performance for the problem. The baseline is estimated using the following procedure: Given a query, we randomly generate the rank of the correct song in the result list (between 1 and 1076). We repeat this process 99 times for the same query, saving the best randomly generated rank out of the 99 repetitions. We perform this procedure for all 186 queries, and use the saved ranks to compute an overall oMRR. This gives us an oMRR of 0.211 (with a variance of 0.051).

Next we turn to evaluate our algorithm. As a first step, we compute the MRR and oMRR taking only the pitch information into account, using a seed size $n = 3$ and the matching parameters set to their default values (full cost = 2, reduced cost = 1). The results are presented in Table 1.

Query Group	#Queries	MRR	oMRR
All queries	186	0.800	0.659
Long term memory	63	0.765	0.627
Short term memory	123	0.818	0.675

Table 1. Initial results.

The table shows that our oMRR values are significantly ($P < 10^{-10}$, Wilcoxon rank-sum test) higher than the baseline. Though results for the short term memory queries are slightly higher than for the long term memory queries, the difference is not statistically significant, and for the rest of the evaluation we use all the queries together. Finally, we observe that, as expected, the MRR values are higher than the oMRR values. This suggests that songs are not sufficiently distinguished using the default parameters.

4.2 The effect of rhythm on performance

We now include the rhythm information in the matching procedure, setting the pitch and rhythm factors to 1:1. The MRR and oMRR go down from 0.800 and 0.659 (pitch only) to 0.677 and 0.594 (pitch+rhythm) respectively. This indicates that giving rhythm equal importance as pitch degrades performance. We could argue that as the rhythm information is less detailed compared to the pitch information, it will match a greater set of songs, so when given equal importance as the pitch information it ends up degrading the results. Another possibility is that due to the use of a virtual keyboard rather than a real one users found it harder to accurately play the rhythm of a query.

4.3 Choice of seed size

As previously mentioned, in [4] it was shown that a two stage retrieval process for QBH was unsuccessful in reducing search time without considerably degrading retrieval performance. In Figure 2 we evaluate the effect of the seed size n in our indexing algorithm (the first stage of our two-stage approach) on retrieval performance and search time.

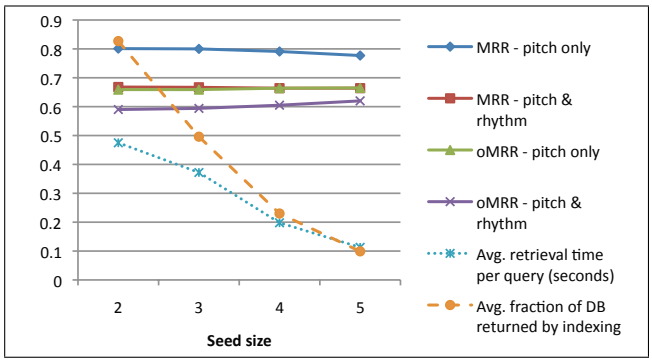

Figure 2. MRR, oMRR, retrieval time and DB reduction vs seed size.

Figure 2 shows that the number of targets to search returned by the indexing algorithm is almost halved every time we increase the seed size, and consequently the search time goes down. Interestingly, the MRR and oMRR values remain stable as we increase the seed size (only the MRR for pitch only shows slight signs of decline). Increasing the seed size n does however require the user query to contain at least n sequential correct pitch intervals, in addition to increasing memory and storage requirements for the database. All in all it is a trade-off between retrieval performance, efficiency and resource usage. For the rest of the evaluation we choose a seed size of 4, providing a significant reduction in database size (reduced to 23%) with practically no degradation of retrieval performance.

4.4 Parameter optimisation

In section 2.1.3 we introduced the notion of having a *full cost* and a *reduced cost* in the matching algorithm, and in section 4.2 we saw that giving rhythm equal importance as pitch in the matching is detrimental to retrieval performance. In this section we optimise these parameters, namely the ratio between the full and reduced costs, and the ratio between the pitch and rhythm factors.

To do so we divided the queries into two groups of roughly equal size – the optimisation is performed using the queries of group 1, and then validated on the queries of group 2. For the optimisation we use the *Simulated Annealing* approach for global optimisation [7]. The performance for the two query groups before optimisation is given in Table 2.

Query Group	Pitch:Rhythm Ratio	Full:Reduced Cost	MRR	oMRR
1	1:1	2:1	0.704	0.646
2	1:1	2:1	0.634	0.574

Table 2. Results for the groups before optimisation.

We start by optimising the pitch and rhythm weighting factors. The effect of these parameters on performance is visualised in Figure 3. The optimal pitch to rhythm ratio was found to be 3:1 (Table 3), and is used in all fur-

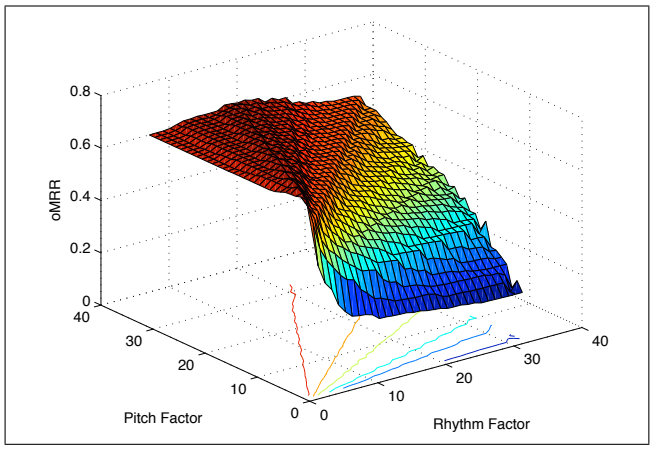

Figure 3. oMRR as a function of the pitch and rhythm factors.

ther evaluations. A slightly higher MRR value could be achieved by ignoring rhythm altogether, but at the cost of significantly reducing the oMRR indicating that the rhythm information is useful for distinguishing between targets which have the same pitch match score.

We next perform the optimisation for the cost parameters in the matching algorithm. The optimal values were found to be 3 for $full\ cost$ and 1 for $reduced\ cost$ (Table 3 last row). This suggests that the modified cost functions provide an improvement to performance, as otherwise the optimal full and reduced costs would have been equal to each other (Table 3 penultimate row).

Pitch:Rhythm Ratio	Full:Reduced Cost	MRR	oMRR
1:1	2:1	0.704	0.646
3:1	2:1	0.784	0.745
3:1	1:1	0.759	0.717
3:1	3:1	0.787	0.761

Table 3. Results for group 1 before and after pitch:rhythm optimisation and full:reduced optimisation.

Finally we compute the MRR and oMRR for query group 2 and for all queries together using the optimised parameters. Figure 4 shows that in all cases performance is improved, though only in the case of oMRR for all queries (Group 1&2) is the improvement statistically significant (p=0.018, Wilcoxon rank-sum test).

4.5 Seed filtering

Next we examine the distribution of seeds in the queries and the database, displayed in Figure 5.

Both distributions constitute a power law probability distribution, obeying a kind of Zipf's law for musical interval sequences [23]. Accordingly, the most frequent seeds comprise the largest proportion of the database but convey the least amount of information useful for distinguishing between songs. By filtering from the query the seeds which are most common in the database we can further reduce the

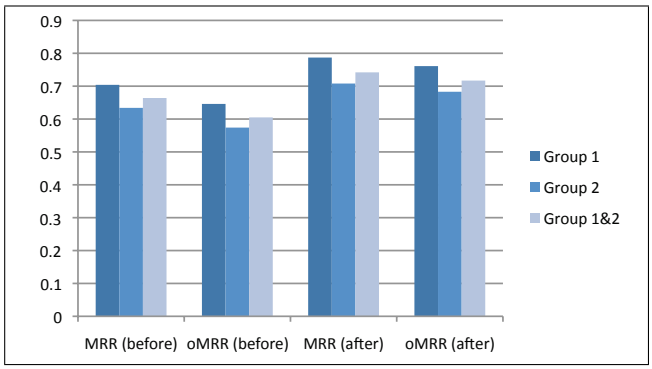

Figure 4. MRR and oMRR results, before and after optimisation.

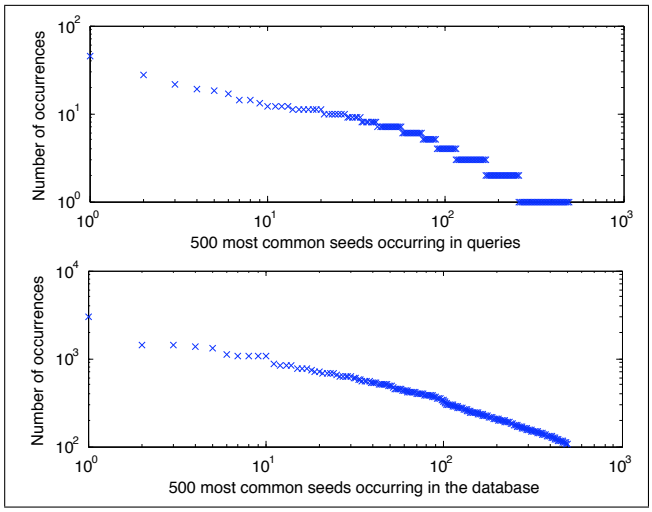

Figure 5. Seed distributions for queries and songs in the database.

fraction of the database returned by the indexing algorithm while maintaining retrieval performance.

When filtering just the three most common seeds in the database, we reduce the database size by a further 40% (from 23% to 14% of the original size) while the MRR and oMRR values go down by less than 1.3%. This concept could be further extended by introducing a seed weighting function, for example using $tf * idf$ [1] like weighting based on seed distributions. This could also help in ranking songs which have the same match score after the second stage, however we have not explored this option and leave it for future work.

4.6 Scalability

Finally, we evaluate how retrieval performance and efficiency are affected as we scale the database size up to 18,017 songs which translates into 88,034 targets. The results are presented in Figures 6 and 7.

First we note that our indexing algorithm provides a considerable reduction in the fraction of the database returned by the first stage, reducing it to 15% of its original size. Nonetheless, further work would be required for our approach to be applicable to collections of millions of

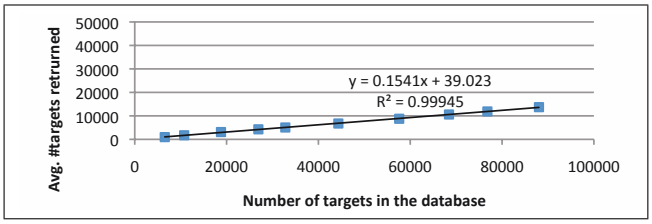

Figure 6. Avg. #targets returned by the first stage vs #targets in the database.

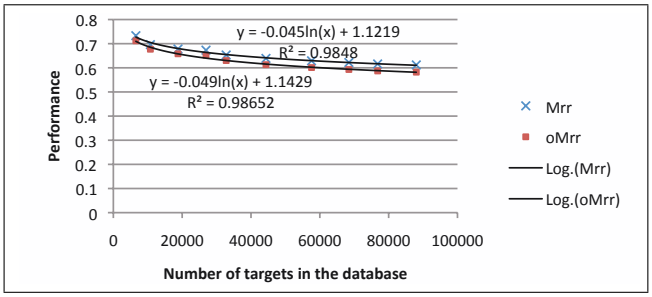

Figure 7. MRR and oMRR vs numbers of targets in the database.

songs, ideally obtaining a sublinear relation between the number of targets in the database and the number of targets returned by the indexing stage.

Next we note that both the MRR and oMRR decrease as $O(log(n))$ as the database size n is increased ($R^2 > 0.98$), indicating that performance scales well with database size.

5. CONCLUSIONS

In this paper we introduced a two-stage retrieval approach for a melodic QBSE system. We demonstrated that whilst for QBH systems similar approaches were unsuccessful, for QBSE this approach can successfully reduce the database size while maintaining high MRR values. We provided a detailed study of the effect of different parameters of the system, namely the seed size, the relative weighting of pitch and rhythm and the full and reduced costs in the matching algorithm.

Finally we consider some ideas for future work. Instead of considering almost every track in a MIDI file as a target, we could aim to extract only the most relevant melodic parts of the piece, as done in [11, 12]. Next, it would be interesting to further study the seed distributions in the queries and the database, which could help develop more elaborate seed filtering and/or a seed weighting scheme. [2]

6. REFERENCES

[1] R. Baeza-Yates and B. Riberto-Neto. *Modern Information Retrieval*. Addison Wesley and ACM Press, 1999.

[2] T. Bozkaya and M. Ozsoyoglu. Indexing Large Metric Spaces for Similarity Search Queries. *ACM Transactions on Database Systems*, 24(3):361–404, September 1999.

[2] This work was supported in part by Microsoft Research through the European PhD Scholarship Programme, and the Programa de Formación del Profesorado Universitario (FPU) of the Ministerio de Educación de España.

[3] M. Clausen, R. Engelbrecht, D. Meyer, and J. Schmitz. PROMS: A Web-based Tool for Searching in Polyphonic Music. In *ISMIR Conference Proceedings*, 2000.

[4] 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. *Journal of the American Society for Information Science and Technology*, February 2007.

[5] W.J. Ewens and G. Grant. *Statistical Methods in Bioinformatics: An Introduction (Statistics for Biology and Health)*. Springer, 2nd edition, 2005.

[6] W. Hewlett and E. Selfridge-Field, editors. *Melodic Similarity: Concepts, Procedures and Applications*. MIT Press, Cambridge, 1998.

[7] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598), 1983.

[8] A. Kornstädt. Themefinder: A Web-based Melodic Search Tool. *Computing in Musicology*, 11:231–236, 1998.

[9] K. Lemström, V. Mäkinen, A. Pienimäki, M. Turkia, and E. Ukkonen. The C-BRAHMS Project. In *ISMIR Conference Proceedings*, pages 237–238, 2003.

[10] K. Lemström, N. Mikkilä, and V. Mäkinen. Fast Index Based Filters for Music Retrieval. In *ISMIR Conference Proceedings*, pages 677–682, 2008.

[11] S. T. Madsen, R. Typke, and G. Widmer. Automatic Reduction of MIDI Files Preserving Relevant Musical Content. In *6th International Workshop on Adaptive Multimedia Retrieval (AMR 2008)*, Berlin, Germany, 26-27 June 2008.

[12] C. Meek and W.P. Birmingham. Automatic Thematic Extractor. *Journal of Intelligent Information Systems*, 2003.

[13] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, Cambridge, UK, 2002.

[14] B. Pardo and W.P. Birmingham. Encoding Timing Information for Musical Query Matching. In *ISMIR Conference Proceedings*, Paris, France, 2002.

[15] B. Pardo, J. Shifrin, and W. Birmingham. Name That Tune: A Pilot Study in Finding a Melody From a Sung Query. *Journal of the American Society for Information Science and Technology*, 2004.

[16] C. A. Romming and E. Selfridge-Field. Algorithms for polyphonic music retrieval: the Hausdorff metric and geometric hashing. In *ISMIR Conference Proceedings*, pages 457–462, Vienna, Austria, 2007.

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

[18] R. Typke, P. Giannopoulos, R. C. Veltkamp, F. Wiering, and R. Van Oostrum. Using Transportation Distances for Measuring Melodic Similarity. In *ISMIR Conference Proceedings*, pages 107–114, 2003.

[19] R. Typke, R. C. Veltkamp, and Wiering F. Searching Notated Polyphonic Music using Transportation Distances. In *Proceedings of the ACM Multimedia Conference*, pages 128–135, New York, 2004.

[20] R. Typke and A. Walczak-Typke. A Tunneling-Vantage Indexing Method for Non-Metrics. In *ISMIR Conference Proceedings*, pages 683–688, 2008.

[21] R. Typke, F. Wiering, and R. C. Veltkamp. A Survey of Music Information Retrieval Systems. In *ISMIR Conference Proceedings*, pages 153–160, 2005.

[22] A. L. Uitdenbogerd. N-gram Pattern Matching and Dynamic Programming for Symbolic Melody Search. In *Proceedings of the Third Annual Music Information Retrieval Evaluation eXchange*, Sept. 2007.

[23] D. H. Zanette. Zipf's Law and the Creation of Musical Context. *Musicae Scientiae*, 10(1):3–18, 2006.