Technische Universiteit Eindhoven University of Technology

Department of Mathematics and Computer Science
Architecture of Information Systems Research Group

Discovering Audibly Recognizable Patterns within Tune Families

Master Thesis

L.W.P.H. van Gansewinkel

Supervisors:
dr. K. A. Buchin
dr. A. Volk

Committee:
dr. K. A. Buchin
dr. A. Volk
external member

Partial first draft version

Eindhoven, June 2017

Abstract

THIS IS MY ABSTRACT

Contents

Chapter 1

Introduction

Chapter 2

Musical Background

2.1 Music Representation in MIR

2.2 Music Terminology

2.3 Symbolic Music Representation Structures

2.4 Conclusion

Chapter 3

Pattern Discovery and Similarity in Music

3.1 Sequence Based Pattern Discovery

3.2 Geometry Based Pattern Discovery

3.3 Evaluation using Ground Truth

3.4 Melodic Similarity using Sequence Alignment

3.5 Conclusion

Chapter 4

Problem Analysis

4.1 Fundamental Problems with Pattern Discovery

4.2 Problems with Tune Families

4.3 Existing Partial Solutions

Chapter 5

Solution Pipeline

As previously discussed, the set of properties we deem desirable in a set of discovered patterns is quite varied. If we were to approach all desired properties in a single-step algorithm, the algorithm would likely require complexity that is hard to oversee. In order to address this issue, we opt to utilize a pipeline format, to break up the problem into smaller chunks. Each node in the pipeline takes care of a particular problem in flexible way, while keeping both its own resource requirements and output size to a minimum.

The pipeline consists of four major steps. These steps are normalization, pattern discovery, pattern clustering and pattern selection. These translate essentially to pre-processing, computing, post-processing and output filtering.

In this chapter we describe the goals and requirements of each node in our pipeline. We also include the input and output format requirements. The chapters that follow will expand on each node, its challenges and their solutions.

5.1 Input

The input accepted by the pipeline comes in the form of a set of songs of size n, called the tune family $F = \{S_1, S_2, \ldots S_n\}$. Each song S is a sequence of points of size k, $S = \{p_1, p_2, \ldots p_k\}$. Finally, each point d consists of integers c indicating chromatic pitch number (MIDI numbers), m morphetic pitch number, v indicating voice number, and a real numbers o indicating onset, d indicating duration, so $p = (o, c, m, d, v)$. For most, if not all, purposes, we ignore m, d and v.

We abstract away from the input format that is directly read into the pipeline. This is an implementation detail of no immediate importance to this thesis. For completeness: The input format the implementation we use supports is Collins Lisp format. This format is a sequence of tuples, indicating notes, of the form (A, B, C, D, E). Here A and D, the onset and duration respectively, are rational numbers, $A, D \in \mathbb{Q}$. The tokens B, C and E, the chromatic pitch number, morphetic pitch number and voice number, respectively, are integers, $B, C, E \in \mathbb{Z}$.

In order for the pipeline to generally run more smoothly, we start out by ordering the points within each song. The order we use for this is lexicographic order, such that onset o matters most, then chromatic pitch c, then voice v and finally duration d. That is:

$$\forall_{p_i, p_j}[i < j | o_i < o_j \vee (o_i = o_j \wedge (c_i < c_j \vee (c_i = c_j \wedge (v_i < v_j \vee (v_i = v_j \wedge d_i \leq d_j)))))]$$

5.2 Normalization

The first node manipulating the data in the pipeline is normalization. In this node, we take the tune family F and transform all points p in each song S, such that any pair of songs (S_i, S_j) can be processed better by the discovery algorithms. Here we use the fact that pairs of songs may become a lot more directly similar if their time units are scaled and their pitch units are translated. This makes discovery algorithms a lot more likely to pick up on similar patterns. The primary purpose is to synchronize the time signatures and keys of the songs, for a level playing field.

Since later, when we undo the normalization on the results of the pipeline, we only have song, chromatic pitch and onset for each point, the normalization must form a two-way function that only relies on this information. This function takes the following form:

$$Normalize(S, o, c) = (S, o', c')$$

However, some discovery algorithms might make use of the other information as well. As such, we need an extended function, to convert all information correctly:

$$NormalizeEx(S, o, c, m, d, v) = (m', d', v')$$

This node has no immediate influence on the performance of other nodes. The reversion node, which builds on how the transformation information from this node is stored, might, however, perform worse. This is discussed in section 5.2.1.

Since this is a transformation step, output of the normalization step is the same form as that of the input, but with some points being transformed.

5.2.1 Reversion

In order to still correctly output the resulting discovered patterns at the end of the pipeline, we still need to undo the transformation made during normalization. This is done as the last node in the pipeline, before output is written. Our goal here is to transform each point in each pattern discovered back to how it was in the original song, before normalization.

The input we get in this node is the output of the pattern selection node, described in section 5.5. In short, we receive a set of patterns P, of which each pattern has occurrences O, which have a set of points p' and a song S they belong to. The points at this point in the pipeline are simplified to only containing their their onset o' and chromatic pitch number c', so $p' = (o', c')$.

In order to do reversion correctly, we need to apply the inverse function created by the normalization step, creating the original point $p = (o, c)$:

$$Normalize^{-1}(S, o', c') = Revert(S, o', c') = (S, o, c)$$

This reversion step may become a bottleneck for the pipeline. Since we have to consider each note in each discovered pattern, the performance becomes at least linear in the output size l. In the absolute worst case, we must maintain a mapping from normalized notes to original notes, ready to be queried using, for instance, binary search. This mapping would be linear in the input size n, and using it to convert back would likely be logarithmic in the input size, per note. As such, for input size n and output size l, we can expect an upper bound to performance of $O(l \log n)$. In most cases, however, it should be sufficient to just map the songs and the modulo 12 of the pitches to a reversion transformation. In this case, this step would be linear in the output size l times the number of songs k, so $O(lk)$

The output of the reversion step is the same form as that of the pattern selection step, explained in section 5.5. In short, the the set of pairs of

5.3 Pattern Discovery

5.4 Pattern Clustering

5.4.1 Pattern Cropping

5.5 Pattern Selection

Chapter 6

Family Normalization

6.1 Pitch Normalization

6.2 Tempo Normalization

6.3 Compound Normalization

6.4 Problem Range

How many ways can this problem be approached?

6.5 Solution 1

Chapter 7

Family Pattern Discovery

7.1 Discovery in a Normalized Family

7.2 Pitch Variance

7.3 Embellishments

7.4 Rhythmic Variation

7.5 Position of Occurrence

7.6 Output Size

7.7 Solution 1

Chapter 8

Family Pattern Clustering

8.1 Subsets and Overlap

8.2 Inter-Pattern Distance

8.3 Importance of Strict Subsets

8.4 Solution 1

Chapter 9

Family Pattern Selection

9.1 Determining Importance

9.2 Limiting Output Size

9.3 Problem Range

How many ways can this problem be approached? How to find the right importance protocol?

9.4 Solution 1

Chapter 10

Results

10.1 Comparing through P. Boot's Pipeline

10.2 Discovered Pattern Properties

10.3 Exploration of Parameters

10.4 Evaluation of Problem Solving

10.5 Scalability

10.6 Conclusion

Chapter 11

Improvements

Chapter 12

Conclusions

Write your conclusions here.

List of Figures

List of Tables

Listings

Appendix A

Analysis Tool

A.1 Program Requirements

A.2 Program Architecture

A.3 Visualizations