Discovering audibly recognizable patterns within tune families

van Gansewinkel, L.W.P.H.

Award date:
2018

Link to publication
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
prof.dr.ir. R. van Liere

Final Version

Eindhoven, November 2017
Abstract

Repetition is a fundamental concept in music. This fact still applies between derivative works of music. Previous research has shown that folk songs display high similarity in repeating patterns across various versions of a song. For human experts, these repeating patterns are distinguishable, but tedious to discover by hand. Therefore, Pattern Discovery algorithms have been developed to perform this task. Most of these algorithms, however, are not suited for the intricacies of dealing with various works at a time. The algorithms also lack the capability of working with all the differences between what humans would consider similar patterns.

We perform a qualitative analysis of the weaknesses in various pattern discovery algorithms with respect to groups of songs. We relate these weaknesses directly to what we find to be common properties in patterns discovered by a human analyst. These properties and weaknesses are then broken down into a set of requirements, which we try to meet with our own algorithmic approach.

Our solution comes in the form of a pipeline. This pipeline consists of four tasks, each with responsibility for a subset of the established requirements. The first task transforms the set of songs, such that they retain their information, but are similar in the time- and pitch-domains. The second task is a typical pattern discovery algorithm. The third task combines the discovered patterns into clusters. Finally, the fourth task selects a coherent subset of the clustered patterns. We propose a set of solutions for each of these tasks.

We demonstrate our algorithmic solution on the same families of folk songs from our initial quantitative analysis. We then evaluate it in terms of the requirements derived from that analysis. We conclude that our pipeline results in a manageable set of patterns, which display properties typically only handled well by human experts. These properties include rhythmic variation between pattern occurrences, embellishments, and positional context.

Next to our algorithmic solution, we developed a tool for visualizing patterns in songs. We used this tool for the qualitative analysis, both of previous pattern discovery algorithms and our own. This tool has also been evaluated by an expert user study.
Contents

I Introduction and Background  ix

1 Introduction  1
   1.1 Music Information Retrieval  1
   1.2 Repeated Patterns in Music  2
   1.3 The Discovery of Repeated Patterns  2
   1.4 Repeated Patterns in Tune Families  3
   1.5 Research Goal  3
   1.6 Thesis Structure  4

2 Musical Background  5
   2.1 Music Representation in MIR  5
   2.2 Music Terminology  6
      2.2.1 Notes, Pitch and Scales  6
      2.2.2 Duration, Tempo and Time Signature  7
      2.2.3 Monophonic and Polyphonic Music  7
   2.3 Symbolic Music Representation Structures  7
      2.3.1 Sequential Structure  7
      2.3.2 Geometric Structure  8
   2.4 Algorithms  8
      2.4.1 SIA  8
      2.4.2 SIAR  9
      2.4.3 SIARCT-CFP  9
      2.4.4 SIATEC  10
      2.4.5 COSIATEC  10
      2.4.6 MGDP  10
      2.4.7 Other Relevant Algorithms  11

II Problem Analysis  13

3 Visualization  14
   3.1 Requirements  14
      3.1.1 Input  14
      3.1.2 Perspectives  16
      3.1.3 Usability  17
   3.2 Features  17
      3.2.1 Views  17
      3.2.2 Navigation  19

Discovering Audibly Recognizable Patterns within Tune Families  v
CONTENTS

3.3 Evaluation ............................................. 20
  3.3.1 Protocol ........................................ 20
  3.3.2 Results ......................................... 21

4 Problem Analysis ................................. 23
  4.1 Fundamental Problems with Pattern Discovery ................. 23
    4.1.1 General Weaknesses of Pattern Discovery .................. 23
    4.1.2 Weaknesses of Geometric Representation ................ 25
    4.1.3 Weaknesses of Sequential Representation ............. 25
    4.1.4 Fundamental Problems Summed Up .................... 26
  4.2 Problems with Tune Families .......................... 26
    4.2.1 Discovering Patterns in Families .................. 26
    4.2.2 Selecting Appropriate Patterns ................... 28
    4.2.3 Tune Family Problems Summed Up ................... 28
  4.3 Existing Partial Solutions ........................... 28
    4.3.1 SIAR ........................................ 29
    4.3.2 SIARCT-CFP .................................. 29
    4.3.3 COSIATEC ................................... 29
    4.3.4 MGDP ....................................... 29

5 Formal Problem Description ....................... 31
  5.1 Formal Syntax ....................................... 31
  5.2 Unformalized Requirements ........................... 32
  5.3 Interplaying Requirements ............................ 32
  5.4 Independent Requirements ............................ 33
  5.5 Requirements Combined and Compiled ..................... 34

III Methods ........................................... 37

6 Solution Pipeline .................................. 38
  6.1 Input and Output ................................... 38
  6.2 Normalization ...................................... 39
    6.2.1 Reversion .................................... 39
  6.3 Pattern Discovery ................................... 39
  6.4 Pattern Clustering ................................... 40
    6.4.1 Pattern Cropping ................................ 41
  6.5 Pattern Selection ................................... 42
  6.6 Conclusion .......................................... 42

7 Family Normalization .............................. 43
  7.1 Challenges .......................................... 43
    7.1.1 Pitch Normalization ................................ 43
    7.1.2 Tempo Normalization ................................ 45
    7.1.3 Choosing an Anchor Point ......................... 46
  7.2 Solution ............................................. 47
    7.2.1 Algorithm ...................................... 47
    7.2.2 Performance .................................... 50
    7.2.3 Reversion ...................................... 51
8 Family Pattern Discovery
  8.1 Helping Factors .................................................. 53
  8.2 Challenges ....................................................... 53
    8.2.1 Transpositions ........................................... 53
    8.2.2 Embellishments .......................................... 54
    8.2.3 Gap Size ................................................. 54
    8.2.4 Rhythmic Variation ...................................... 54
    8.2.5 Output Size and Performance .......................... 55
  8.3 Solution .......................................................... 55
    8.3.1 Algorithm .................................................. 56
    8.3.2 Performance and Output Size .......................... 59

9 Family Pattern Clustering ....................................... 63
  9.1 Challenges ...................................................... 63
    9.1.1 Pattern Similarity ...................................... 63
    9.1.2 Pattern Spread ........................................... 64
    9.1.3 Pattern Overlap Reduction ............................. 64
    9.1.4 Performance .............................................. 64
  9.2 Solution .......................................................... 65
    9.2.1 Algorithm .................................................. 65
    9.2.2 Performance .............................................. 67
    9.2.3 Cropping .................................................... 69

10 Family Pattern Selection ..................................... 71
  10.1 Challenges ...................................................... 71
    10.1.1 Overlap Prevention ...................................... 71
    10.1.2 Pattern Order and Position ............................ 71
    10.1.3 Pattern Spread ........................................... 72
    10.1.4 Determining Importance ............................... 72
  10.2 Solution .......................................................... 73
    10.2.1 Algorithm .................................................. 73
    10.2.2 Performance .............................................. 76

IV Discussion ......................................................... 81

11 Results .............................................................. 82
  11.1 Discovered Pattern Properties ............................. 82
  11.2 Exploration of Parameters ................................. 83
  11.3 Comparison to Annotated Patterns ....................... 87
  11.4 Evaluation of Problem Solving ............................. 88
  11.5 Conclusion ..................................................... 89

12 Conclusion .......................................................... 91
  12.1 Improvements and Future Work ............................ 92

Bibliography .......................................................... 93

List of Figures ........................................................ 95

List of Tables .......................................................... 97

Listings ................................................................. 99

Appendix ................................................................. 101
CONTENTS

A  Visualizer Instructions 101
   A.1  Main Screen 101
   A.2  Container Filler 102
   A.3  Views 104
      A.3.1  General notes on views 104
      A.3.2  Piano Pattern View 104
      A.3.3  Piano Heat Map View 105
      A.3.4  Family Pattern View 105
      A.3.5  Family Heat Map View 105
      A.3.6  Parameter Pattern View 105
      A.3.7  Parameter Heat Map View 105
      A.3.8  Single Pattern View 106
      A.3.9  Single Pattern Family View 107
      A.3.10  Pattern Strips View 108
      A.3.11  Warning 108

B  Visualizer Evaluation Tasks 111

Discovering Audibly Recognizable Patterns within Tune Families
Part I

Introduction and Background
Chapter 1

Introduction

In this age of information, we are exposed to an enormous amount of data. To handle this, we need proper ways of searching and filtering the right information. Structuring all this data and finding the right ways to refer to, relate, and identify pieces of information is becoming a crucial, and difficult task. This applies to music as well. We often relate music through its genre, or the artists involved in its production. An often overlooked, yet important property of many styles of music is repetition. The presence of certain repeated patterns in music can be a valuable property to identify and relate pieces of music by. These repeated patterns are the subject of this thesis.

This thesis builds upon the work by Boot [2], demonstrating the importance of repeated patterns in music for classification of songs. Although the findings supported the importance of repeated patterns, the patterns discovered by algorithms did not produce results of similar quality as pattern discovery performed by human experts. In fact, the algorithms were often outperformed by simple heuristics. Our goal is to determine an algorithmic approach that works around the weaknesses of existing algorithms, and produce results more closely resembling those of human experts.

In this chapter, we provide an introduction the research field we are working in, and state our goals for the thesis. We start by introducing the overarching field of Music Information Retrieval. We will then gradually go into more detail about the specific problem space we are working with. Finally, we will formulate our research goal and questions, and give an outline of the remainder of the thesis.

1.1 Music Information Retrieval

The field this thesis falls under is Music Information Retrieval (MIR), a branch of Information Retrieval (IR) in Computer Science. IR comes down to what we see search engines such as Google do every day. We communicate what information we need, which IR provides to us from a large, not necessarily structured set of data. This also takes into account that the need may not be communicated in its entirety. An example in musical context would be to look up some song by a segment from its lyrics. In this case, regular information retrieval approaches through text run into the problem of not being able to distinguish song lyrics from some unrelated similar text.

With MIR, we work with musical information, narrowing our search down to the subject of music, and providing possibly more distinctive information to search by. If we were to search by the melody of the piece, for instance, we would likely narrow down the possibility space significantly. Furthermore, we gain the ability to retrieve musical information that does not have any (memorable) textual data associated with it.

Working with musical information does come with its own challenges, as this information is much more complex than text. Musical information consists of factors such as melody, instrumentation, rhythm, harmony, and more. To make matters worse, music is often not recorded in a lossless manner the way text is. Determining what exactly the instrument being played in an
audio recording is already difficult for the average person, much more so for a computer. The concept in MIR we will be working on is the discovery of repeated themes and sections, which falls under the subject of music similarity. This is the process of taking raw musical data, and cataloging important repetitions found in it. We will discuss this further in the following sections.

1.2 Repeated Patterns in Music

Music is an art in which repetition is a fundamental concept. This is easily seen in the structure of songs, such as the repetition of a chorus or refrain. Repetition is also heavily present in the lower level of musical structure. The note-to-note progression within each of the higher level structures is typically repeated several time within a single repetition of that structure.

It is the lower level repetitions that we are particularly interested in, as these constitute concepts such as motifs and themes. Motifs and themes form a core part of the identity of a musical piece, and can even repeat in other pieces relating to it. As such, these concepts give us a basis on which to identify and relate songs to one another. We primarily focus on the repetitions that relate different songs.

Repetition by definition requires that something repeats at least once. We will refer to a repeating structure in a song, be it theme, motif or otherwise, as a pattern. These patterns must then have at least two occurrences. The caveat is that these occurrences, in our case, need not be in the same song.

When considering whether or not something is a repetition, the more general concept of music similarity comes into play. Individual musical segments may be similar to one another in some way, and vary in some other. The qualities that constitute the similarity between these segments determine whether this can be considered a repetition.

1.3 The Discovery of Repeated Patterns

Discovery of repeated patterns can be done in two ways. The way we will be working with is the automated discovery of patterns using pattern discovery algorithms. The alternative is for human experts to annotate these patterns. We will use the latter as a point of reference for what constitutes good pattern discovery.

Contemporary pattern discovery algorithms typically face a number of immediately apparent challenges. These largely come down to the slightly subjective nature of musical patterns. An algorithm can not typically determine the importance of one pattern over another. This can be caused by lacking priority of one aspect of a pattern over the other. Even when we have patterns discovered by a human expert as a reference, different experts will not fully agree on what patterns constitute the ground truth. These factors all result in the challenge of restraining the output size of an algorithm. Since an algorithm cannot distinguish which patterns are of most importance, it cannot output one over the other, so it must output all patterns it discovers.

We aim to analyze the problems of pattern discovery algorithms in further detail, and meet resulting challenges. In doing so, we aim to formulate an approach for handling each challenge we encounter. We will briefly explain the most relevant pattern discovery algorithms in Section 2.4.
CHAPTER 1. INTRODUCTION

Figure 1.1: The first four measures of two songs in the same tune family, shared patterns according to human experts marked by coloured bounding boxes.

1.4 Repeated Patterns in Tune Families

Our work builds upon a problem encountered by Boot [2] in the pursuit of classification of folk songs using repeated patterns. The problem we are looking to solve is that existing pattern discovery algorithms are not sufficiently geared towards handling tune families. A tune family refers to a series of folk songs, which stem from the same original, yet have been altered through the imperfect process of passing the songs on between people. Due to the nature of these tune families, it is natural for prominent musical patterns to remain largely intact across versions of a song.

Here the concept of music similarity comes into play. Whereas typically a repeated pattern is nearly exact, the context of tune families brings more variance with it. As such, we need to identify what makes a pair of musical segments similar within this context, and be able to relay this similarity to algorithms.

Using similarity for pattern discovery in tune families, rather than singular songs, brings its own set of challenges. Different properties of patterns become important, such as their position or role in the songs of the family. A particularly difficult factor here is the slight differences within what a human would consider the same pattern. These differences may prevent algorithms from seeing the similarity between pattern occurrences. Alternately, an algorithm may pick up on the slightly different pattern occurrences, at the expense of considering many more somewhat similar segments as patterns. This then becomes a heuristic problem with trade-off between false-positives and false-negatives.

Figure 1.1 shows the first four measures of two songs in the same tune family. The coloured boxes mark the occurrences of the same pattern with the same colour. This immediately shows how these patterns can vary enough to be difficult for algorithms to discover.

It is specifically these slightly different patterns across multiple songs that we are interested in. In this thesis, our focus is on discovering patterns that occur across multiple songs in a tune family. This combines the problem space of patterns in tune families, with that of pattern discovery in general.

1.5 Research Goal

Our research surrounds laying the ground works for pattern discovery in tune families. We aim to determine what factors constitute a good pattern discovery approach within this context. We then wish to determine a manageable decomposition of the problem space, and provide a way
of approaching the sub-problems independently. Finally, we want to provide and verify a set of solutions for these sub-problems. Exploring the space of possible solutions to each sub-problem is beyond the scope of this thesis.

**Research Goal** Determine a way to perform pattern discovery on tune families, resulting in patterns with qualities comparable to those discovered by human experts.

We split this research goal into three research questions:

**Research Question 1** What are the qualities present in patterns discovered by human experts, that are not necessarily present in those discovered by pattern discovery algorithms?

**Research Question 2** How can we approach discovering patterns algorithmically without having to account for the entire problem space at the same time?

**Research Question 3** Is it feasible to algorithmically discover patterns with the same qualities as patterns discovered by human experts?

### 1.6 Thesis Structure

This thesis will be structured in four major parts. Part I includes this introduction. We also discuss the subject background, with a brief discussion of related work, in Chapter 2.

Part II forms the problem analysis. In Chapter 3 we introduce a visualization tool we developed to analyze qualitative aspects of discovered patterns. Then, in Chapter 4, we use our tool to analyze results from existing pattern discovery algorithms and patterns annotated by human experts, to determine a set of informal requirements. We then build on this in Chapter 5, in which we formalize the informal requirements.

We solve the problems found in the problem analysis in Part III. Here we start by introducing a pipeline in Chapter 6, which delegates the problem space into separate tasks. The following chapters, Chapters 7 through 10, discuss the challenges and our solutions of each task.

Finally, in Part IV, we discuss our results. This contains Chapter 11, in which we reflect on our own solution as we did for others in the problem analysis. And finally, in Chapter 12, we present our conclusions and prospects for future work.
Chapter 2

Musical Background

This thesis centers around the algorithmic branch of computer science, but the domain we apply this on is music. In this chapter, we will make the reader familiar with some basic concepts of music (research). We also make the first steps towards representing the data as we will use it. In Section 2.1, we discuss the different representations of music, and which one we will work with. Then, in Section 2.2 we bring the user up to speed with the music terminology we will use, in order to maintain proper context of the data. Finally, Section 2.3 discusses the two possible interpretations of our chosen music representations, which our algorithms will make use of. This chapter is heavily inspired by the Chapter 2 of Peter Boot’s Master thesis [2], which we are building upon.

2.1 Music Representation in MIR

In the field of Music Information Retrieval, access to digital music is required, as the music is the data. This data can come in different forms. The most prominent forms are of course the sound waves themselves, and the symbolic representation given by sheet music, as seen in Figure 2.1.

(a) Audio representation: the audio signal. Contains only the exact sound information.

(b) Symbolic representation: the music score. Contains various forms of information, as instructions to reproduce the music accurately.

Figure 2.1: The primary two representations of music.

Although the audio representation, in Figure 2.1a, is the most straightforward, its simplicity makes it much harder to examine high-level properties of the music. Audio can be non-musical things as well, it is largely up to the human mind to interpret it as music. This just adds another step of complexity to the pattern discovery problem, so we will be focusing on our other possible representation. Audio still has its advantages for some problems, as it preserves how the music manifests as a whole.

The symbolic representation, in Figure 2.1b, as seen in sheet music has no sound on its own. Instead, it forms more of a recipe to create the sound from. Without something to perform the symbolic music, there is no actual music to hear. This also means that it depends on the level of detail of the symbolic music, as well as the interpretation and skill of the performer, how the audio is going to sound. Working with the symbolic representation directly skips having to interpret the performance.
CHAPTER 2. MUSICAL BACKGROUND

The symbolic representation is what we will be using throughout our research. This representation provides us with very clear cut pieces of information, such as what notes are exactly being played and when. This also allows us to abstract away from different factors, such as tempo and volume, and hone in on what we deem important for our problem, with minimal overhead.

We can also change the symbolic music to different formats we deem convenient. A prominent format is MIDI (Musical Instrument Digital Interface), which follows fairly directly from symbolic music. We can use this format to have the computer be a consistent performer of our music. This will prove highly useful in performing problem analysis and evaluating results.

2.2 Music Terminology

Since this thesis is about finding patterns in music that are interesting to the human listener, we are tied very closely to the music domain. In order to speak about our data in the context of music, we require that the reader knows some basic music terminology.

2.2.1 Notes, Pitch and Scales

Our most elementary data points correspond to one of the most fundamental elements in music: the note. Notes have a pitch and a duration, but in the context of a song, also a time of onset.

![Piano Keyboard](image)

Figure 2.2: One octave worth of keys of the piano keyboard. The same pattern is repeated, with repetition having a number to distinguish them.

The pitches of notes can come in two primary interpretations. One is the chromatic pitch, the other is the morphetic pitch. The chromatic pitches includes all keys on the piano, as seen in Figure 2.2. Each such segment of 12 is known as an octave, and repeats across the piano, so a pitch 12 pitches higher is considered the same pitch in a different octave.

Similarly, the different letters denote the morphetic pitches, which come in 7 per octave. As seen on the figure, every letter appears 3 times, on different keys. Which key actually corresponds to a letter, and thus the morphetic pitch, depends on the scale. A scale is typically a sequence of all 7 letters, with the variants (♯, ♭) determining what type of scale it is. Note that both these forms of pitch stem from western music, not all music fits into these systems.

We will typically make use of the chromatic pitch, since the encoding of MIDI maps each chromatic pitch to a number. Mapping morphetic pitch to numbers requires the context of the scale, which is less convenient.

Finally, it is important to understand the notion of pitch intervals. Intervals are the distance between pitches, which are a very strong part of how a sequence of notes sounds to the human ear. Following a note up with one 7 pitches higher, for example, sounds similar regardless of which these exact notes are. This may become important for cases in which parts of a song, or even songs in their entirety, are transposed. This means that the entire set of notes has its pitch shifted by the same interval. Comparing the transposed version of something to its original can result in a complete mismatch, while there is, in fact, great similarity.
CHAPTER 2. MUSICAL BACKGROUND

2.2.2 Duration, Tempo and Time Signature

As mentioned in the introduction of this section, notes have a duration in addition to their pitch. The duration, in symbolic representation, denotes the number of counts, or beats, that a note persists for. The actual time a note persists is determined by both the duration and the tempo. The tempo is the real-time rate at which the counts in the song transpire.

Another important timing property in music is the time signature. This denotes how many beats each measure of a song has, and what rhythmic roles the notes have, based on their timing. A measure is a stretch of notes of consistent length in time, the first note of which is typically accented to provide a clear rhythmic frame of reference for the listener. An example of a time signature and measures is shown in Figure 2.3.

![Figure 2.3: An example of a 4/4 time signature, and two measures.](image)

Even more important than the duration in this thesis is the onset, the time in beats since the beginning of the song, at which the note starts. This means that the first note in a song will always have an onset of 0. Songs may be changed in tempo and time signature such that they mostly preserve of the original sound of the song. This changes what onset refers to what time, as such, two near identical versions of a song may have completely different onsets for their matching notes.

2.2.3 Monophonic and Polyphonic Music

Some music only features a single note at a time, while others may use multiple notes at the same time for a richer sound. These are referred to as monophonic and polyphonic music, respectively. In this thesis, for simplicity, we only consider monophonic music. With some adjustment, our methods may also be applicable to polyphonic music, however.

2.3 Symbolic Music Representation Structures

For pattern discovery algorithms, there are two ways to treat the symbolic music data. In this section, we discuss the two representations used. First we will discuss treating music as a sequence of notes. Then we will discuss treating music as point on the plane of time and pitch.

2.3.1 Sequential Structure

Sequential structures treat the melody as a sequence of notes. This often ignores timing information, making it consist purely as a sequence of pitches. This approach is convenient, since this essentially transforms the problem into the more generic string or sequence matching problem space. These are used for subjects like text and DNA already. As example, here we have the first couple of notes of Für Elise, also seen in Figure 2.4a:

\[ E_5, D#_5, E_5, D#_5, E_5, B_4, D_5, C_5 \]

or as MIDI note numbers:

\[ 76, 75, 76, 75, 76, 71, 74, 72 \]
CHAPTER 2. MUSICAL BACKGROUND

There are a few limitations to this format. For one, polyphonic music cannot be represented, as notes have to be strictly sequential. We also lose timing information, which may be very important for musical patterns, as rhythm is a prominent factor. This also means that the perceived difference between two near identical patterns, where one has an extra note in the middle, can become blown up.

Sequential structures do have the advantage of being flexible in their usage. Rather than taking the pitch sequence, various aggregate data sequences can be constructed and used instead. For example, the intervals between the sequential notes, or whether the intervals go up or down in pitch. These different forms of the sequence, called viewpoints, were introduced by Conklin and Witten [8]. Bergeron and Conklin [1] even developed a viewpoint to allow polyphonic music to be processed this way.

2.3.2 Geometric Structure

Geometric structures overcome the limitations of sequential structures, at the cost of increased complexity. David Meredith [10] proposed the approach of representing notes as multidimensional vectors. The dimensions coincide with the different viewpoints in sequential structure.

(a) The sheet music of the first four measures of Für Elise.
(b) Für Elise, as mapped onto the plane of MIDI note number and onset time.

Figure 2.4: The first four measures of Für Elise, as sheet music and as geometric representation.

In Figure 2.4b, we demonstrate how a geometric representation might look. This particular representation is very convenient to reason about, as it parallels sheet music, seen in Figure 2.4a, quite closely. There may of course be more dimensions at play, such as duration or voice. The typical dimensions used for geometric computations are the time of onset and the pitch notes can be in. This representation closely mimics conventional sheet music.

2.4 Algorithms

In this section, we will discuss some previous approaches to the pattern discovery problem. We will primarily discuss the details of pattern discovery algorithms from which we borrow heavily. We will also discuss the MGDP algorithm [7], which is the only algorithm already aimed at tune families. Finally, we will briefly mention some further pattern discovery algorithms, for completeness.

2.4.1 SIA

SIA is a geometry-based pattern discovery algorithm by Meredith [10], and forms the foundation for pattern discovery with geometric structures. As explained in Section 2.3.2, SIA uses a the 2D plane of onset time and pitch to give a geometric representation of a song. The algorithm determines all vectors from each note to each note later in the song. It then sorts these vectors.
lexicographically on both pitch difference between the notes and time difference between the onsets of the notes. After this, we have sets of identical vectors. We can simply take the set of all notes at the start of each vector as one pattern occurrence, and the set of all notes and the end of each vector as another. This then constitutes a pattern with two occurrences, since all notes from one occurrence have a single displacement that translates them into the other occurrence. This is performed for every set of identical vectors to get a set of patterns with two occurrences each.

2.4.2 SIAR

The SIAR algorithm by Collins [5] builds on the foundation of SIA. Normally, the SIA algorithm constructs a list of all vectors from one note to a note later in the song. SIAR limits the creation of vectors from a note to only the \( r \) notes directly after it, where \( r \) is a parameter for how many notes to consider. An example of this is shown in Figure 2.5a for \( r = 2 \). Using the normal process of SIA it now finds what notes share the same short vectors. It then runs SIA once more, but only using the vectors between the notes with shared short vectors, an example of these vectors can be seen in Figure 2.5b. The resulting set of patterns contains only the patterns where each note has at least one other note in the pattern nearby. A few of these resulting patterns are shown in Figure 2.5c, note how all these patterns are very compact. The patterns with 3 occurrences are actually only found pairwise by SIAR.

Depending on the value chosen for \( r \), this can be significantly faster than SIA. The real performance gain we get from this, however, is that the set of discovered patterns is smaller. Since the patterns missing from the output are very sparse, these patterns can typically be considered uninteresting.

2.4.3 SIARCT-CFP

The SIARCT-CFP algorithm by Collins et al [4] contributes two useful ideas. The first is the so-called Compactness Thrawler, which removes all patterns from from the results that do not satisfy a parameterized ratio \( c \) between the number of notes in a pattern and the size of the bounding box of that pattern. This makes sure that the next step in the algorithm has less to process, and that uninteresting patterns are removed.

The second contribution of SIARCT-CFP is the fingerprinting and categorization step it takes. The fingerprinting takes triplets of notes and makes a token representing them, which is primarily identified by the ratio between the time from the first to second note and the second to third note. These tokens still match when a set of notes is stretched, and matching tokens can gain some leniency by allowing a factor of variance to the ratio. This gives the algorithm the ability to discover patterns with rhythmic variation.

The categorization step sorts the discovered patterns in order of importance, as determined by a metric of choice. For the most important pattern, it starts adding patterns to it as occurrences,
with the criterion that an added pattern must be similar enough when using the fingerprinted
tokens to compare them. The added patterns are removed from the original pattern set and
the same process repeats until every pattern has been categorized. This essentially clusters the
patterns together into larger sets of occurrences, each containing the rhythmic variants of its
patterns.

2.4.4 SIATEC

SIATEC is an extension of the SIA algorithm, also by Meredith [10], which is built to allow SIA
to discover patterns with multiple occurrences. The change SIATEC makes is that it searches
for translational equivalence classes when determining patterns from vectors. SIATEC first of all
creates all vectors, including the ones that point backwards in the song. Then, when SIATEC
determines a pattern based on notes that have the same vector coming from them, a search is done
to find whether this same set of notes shares a vector with another set of notes. For each note, the
set of vectors coming from it is stored. Since vectors are pointing both backwards and forwards,
we can simply take the sets of vectors from all notes in a pattern according to SIA, and intersect
these sets. The remaining vectors after the intersection are set of displacements the entire pattern
can do to form another occurrence of the same pattern.

2.4.5 COSIATEC

Meredith [10] made another extension to the SIATEC algorithm in the form of COSIATEC.
This extension is intended as a compression algorithm, it finds patterns and stores them as one
occurrence with its possible displacement vectors. It does this with as many patterns as possible
to encode a song in patterns.

Since COSIATEC is a compression algorithm, it takes the discovered patterns from SIATEC
and picks the pattern with the best potential for compression for its final output. The notes from
the chosen pattern are removed and the discovery and selection process is repeated until no more
patterns are found. This way, the discovered set of patterns has no overlapping notes and consists
of patterns important for compression.

The selection of the best pattern for compression is done by a lexicographic ordering on prop-
erties of the discovered patterns. First, the pattern with the best compression ratio is selected.
This is the ratio between the storage cost of one pattern occurrence and its displacement vectors,
over the storage cost of storing all occurrences explicitly. If multiple patterns score best on this
property, the next in line is considered, which is the compactness. This is defined as the number of
points in a pattern, divided by the size of the bounding box around this pattern in the geometric
space. Then, in order of importance, the tie breakers are: the number of notes in all pattern
occurrences, the duration of the pattern, and the size of the bounding box around the pattern.

2.4.6 MGDP

MGDP by Conklin [7] is a sequence-based pattern discovery algorithm, which is the only algorithm
that originally applies to tune families. The MGDP algorithm works by constructing a tree of
possible patterns. Each sub tree is created by adding another note to the sequence from root
to leaf. If this sub tree still satisfies a number of criteria set by MGDP, then the sub tree gets
expanded further. Another set of criteria exists to determine when a pattern is worth selecting as
output of the algorithm. These criteria depend mostly on the pattern itself and its frequency in
the tune family, and its frequency in a data set outside the tune family.

The first criterion is that a pattern should be supportive. The support of a pattern is the
ratio of the number of songs a pattern occurs in, against the number of songs in the tune family.
A pattern is supportive if its support is above a given threshold. This property indicates that a
pattern is representative of the tune family in its entirety.

The second criterion is that a pattern should be distinctive. The distinctiveness of a pattern
is the ration of the support a pattern has in the tune family, to the support a pattern has outside
CHAPTER 2. MUSICAL BACKGROUND

the tune family. A pattern is distinctive if its distinctiveness is above a given threshold. This property indicates that a pattern is more representative of the tune family than anything else.

The third and last criterion is that a pattern should be maximally general. This means that a pattern should be as small as possible, while also satisfying the other criteria. An additional constraint here is that a pattern must have at least a given number of elements in it.

The algorithm runs by constructing a tree of patterns while keeping track of the three criteria. The tree starts with just the root, representing the empty pattern. The tree is grown with possible extensions to patterns represented by paths down the tree. Every time we extend the tree, the newly extended pattern is verified to be supportive. If the pattern is not supportive, then no extensions of it will be supportive, and the sub-tree is closed. If the pattern extension is long enough, supportive and distinctive, the pattern is added to the output and that sub-tree is closed, as this was the most general pattern in that sub-tree. In every other case, the tree continues to grow as normal.

2.4.7 Other Relevant Algorithms

Two more pattern discovery algorithms worth mentioning are PatMinr [9] and MotivesExtractor [11]. These two algorithms, in addition to the ones we have already discussed, make up the set of algorithms we have data on for our analysis.

PatMinr by Lartillot [9] is a sequence-based pattern discovery algorithm similar to MGDP. This algorithm discovers the patterns in a single sequence of some minimum length, which have more occurrences than any more specific version of them. The algorithm functions by making one pass through the sequence, and maintaining a tree of pattern prefixes as it goes. Every path down the tree is a pattern that occurs in the sequence, and every edge in the tree indicates how often the pattern prefix up to that point occurs. If a branch exceeds the desired minimum pattern length, and branches off into more specific patterns that occur fewer times, then the path leading up to that branch becomes a pattern for the output.

MotivesExtractor by Nieto and Farbood [11] is a completely different sequence-based pattern discovery algorithm. This algorithm creates the matrix of a sequence compared to itself. Each cell in the matrix denotes the similarity of two elements in the sequence, with the diagonal of this matrix being completely similar. It then applies image processing techniques to enhance diagonal lines of similarity, and extracts these from the matrix. Every diagonal line aside from the diagonal of the matrix itself indicates a pair of sub-sequences that are highly similar. These pairs are then considered occurrences of the same pattern.
Part II

Problem Analysis
Chapter 3

Visualization

Between ground truth and algorithmic results, there is a lot of data to consider for analysis. If we want to figure out what exactly sets the two apart, we either need very appropriate metrics, or a way to efficiently inspect patterns directly. In order to even find what kind of metrics would be appropriate, we would still need to look into the patterns themselves. In this chapter, we introduce the visualization tool built to support our problem analysis.

We first list what we require of the visualization tool. We then introduce the tool itself. Finally, we document the evaluation used to verify our desired functionality.

3.1 Requirements

This section discusses the requirements of the visualization tool. We discuss what input it should support, what kind of perspectives it should have on the data, and what its usability requirements are.

3.1.1 Input

The input comes in the form of song files and patterns files. The song files contain songs, while the patterns files contain patterns corresponding to those songs. The data to perform analysis on was supplied by Peter Boot [2].

**Songs**

The song files are found in an architecture as seen in Figure 3.1. Each box represents a folder, except the ones named ”Song0”, ”Song1”, which are song files. The dotted boxes are optional, there may be any number of data sets available, and the ”intra-opus” and ”lisp” folders may be skipped in the structure.

The song files each have the song title and some series of non-letters indicating their version as the name. For example, ”Heer_Halewijn_2 + NLB072254_01.txt”, where the song is read as ”Heer_Halewijn_2 + NLB”, indicating its tune family, and the unique ID separating it from its siblings is ”072254_01”. We expect ”.txt” as the primary file type.

The file format that needs to be accepted is **Collins Lisp Format** [6], also described by Peter Boot [2]. This format consists of line-separated tuples such as (3/2, 71, 66, 1/2, 0). These files are to be interpreted according to Definition 3.1.1.

**Definition 3.1.1** (Songs). Songs are represented as a sequence of tuples $S = \{t_0, t_1, \ldots, t_k\}$. Each tuple, representing a note, is of the form $t = (o, c, m, d, v)$, such that $o, d \in \mathbb{Q}$ and $c, m, v \in \mathbb{Z}$. The component $o$ represents onset, $c$ represents chromatic pitch, $m$ represents morphetic pitch, $d$ represents duration and $v$ represents the voice.
The chromatic pitch comes from the chromatic scale, which has 12 values for each cycle, so 71 and 59 are the same notes, one octave apart. Similarly, the morphetic pitch uses only the letter of the note, without accidentals, giving it cycles of size 7. The voice is simply used to group notes together in layers of the song, this is usually a single value throughout the entire song.

Patterns

Patterns files are found in an architecture as seen in Figure 3.2. As with the song file system, each box is a folder, except for "Patterns0" and "Patterns1". In order to match the correct patterns file to the correct song, the file name of "Patterns0" should match the file name of its corresponding song file "Song0", the same goes for the data set folders.

The "AnnotatedMotifs" folder contains patterns discovered by experts, these should take into account information from the entire tune family. The "InterOpusDiscoveryClassTask" (inter-opus) folder contains patterns discovered by algorithms, when taking into account information from the entire tune family. The "IntraOpusDiscoveryTask" (intra-opus) folder has the same architecture as inter-opus does, but contains patterns that do not take the rest of the tune family into account. Two patterns with identical ID, found under the same settings, but for different songs in the same tune family, will not be the same pattern under intra-opus, but will under the other two.

The names of the folders "Algorithm0" and "Algorithm1" indicate what algorithm was used for the pattern discovery task. These algorithm folder names all end with "Algorithm". The names of the folders "Parameters0" and "Parameters1" indicate what parameters were used for their algorithm.

Definition 3.1.2 (Pattern Sets). Pattern sets are represented as a sequence of patterns \( R = \{P_0, P_1, \ldots, P_k\} \). Each pattern consists of an ID \( \text{pid} \in \mathbb{N} \) and a sequence of occurrences \( P = (\text{pid}, \{O_0, O_1, \ldots, O_l\}) \). Each occurrence consists of a n ID \( \text{oid} \in \mathbb{N} \) and a sequence of tuples \( O = (\text{oid}, \{t_0, t_1, \ldots, t_m\}) \). Each tuple, representing a simplified note from the respective song, is of the form \( t = (o, p) \), such that \( o \in \mathbb{R} \) and \( p \in \mathbb{Z} \). The component \( o \) represents the onset of the note, and \( p \) represents the pitch. The pitch we use will always be the chromatic pitch, but morphetic pitch may be used if adjusted for.

Patterns files contain a multitude of patterns, discovered by algorithms or experts, either using...
only the respective or all songs in the respective family. The format is according to the output format of the Discovery of Repeated Themes & Sections MIREX Task [6], giving us data of the form described in Definition 3.1.2. The file format denotes the start of a new pattern, with its pattern ID, then the start of each occurrence within that pattern with its occurrence ID, followed by tuples of onset and pitch. The following is an example of the contents of a pattern sets file:

```plaintext
pattern1
occurrence1
  7.00000, 45.00000
  7.00000, 48.00000
  11.00000, 60.00000
occurrence2
  31.00000, 57.00000
  31.00000, 60.00000
  35.00000, 72.00000
pattern2
occurrence1
  7.00000, 45.00000
  7.00000, 48.00000
```

### 3.1.2 Perspectives

We require a few perspectives on the data in order to gain useful information out of it. First and foremost, we need to be able to easily read the song and listen to it. Mapping vertical screen space to pitch and horizontal screen space to time in the song, and placing tokens representing notes appropriately should be a good basis for this. Having an interface to play the songs should also be present, with an indication of what part of the song is currently being played.

Second, we require a perspective on the patterns within the songs. For this, we need to see
what notes are contained in each pattern, and where the pattern occurs in the song. We also need
to view multiple song-pattern set pairs at the same time, for cross referencing the data. Interesting
sets of these include looking at the entire tune family with the same pattern discovery approach,
and looking at a single song under different algorithm parameters.

Finally, since a lot of algorithms tend to return a lot of patterns, we need a way to look at
more global pattern behaviour. Indicating what kind of patterns are common and what kind are
rare.

3.1.3 Usability

We need this tool exploring a lot of data to be properly usable to a human user. It is important
to note that users have limited memory. We cannot assume that remembering the exact settings
leading to a certain visualization can be reproduced later. Similarly, navigating from one view
to another view of the same data shouldn’t require too much thinking overhead. As such, views
should be easily put away and switched back to, as well as allow moving to a related view in a
simple way.

Since the data is distributed across a specific file system architecture, we should alleviate the
effort of the user with respect to selecting data. Rather than let the user pick files and tag the
exact data in those files, the tool should infer as much as possible from the file system and the file
contents. The data selection from the user perspective should abstract away from the file system
and towards the nature of the data.

3.2 Features

This section discusses the tool developed to meet the requirements described in the last section.
We cover the views realizing the data perspectives, and how navigation of the data is facilitated.
Further illustration of the tool may be found in the instruction manual, found in the appendix,
Chapter A.

3.2.1 Views

In order to accommodate the viewing and listening of songs, we used a simple piano roll-style
illustration, as seen in Figure 3.3. The top bar is reserved for a music player complete with seek
bar, whose position is directly above the notes being played. The notes also show how long their
duration is, as their width corresponds to their duration, so the end corresponds to the end of the
duration as well.

Patterns are highlighted using bounding boxes and colors on the notes belonging to them. We use 12 distinct colors to denote the different patterns, such that the multiple occurrences
of the same pattern can be distinguished from the rest. Clicking a bounding box will highlight
that pattern and fade out the others, for clearer distinction. For even more convenient listening,
clicking a bounding box also plays the pattern.

In the event of patterns overlapping on the same notes, the colors of up to 5 patterns get
incorporated into a flag over the note. More than 5 patterns may occur on one note, in which case
the note is struck through with white to indicate this. For these cases, we also have a heat map
view, indicating what notes are used often in the viewed pattern set, as illustrated in Figure 3.4. Alternatively, the user may select one pattern to be shown in isolation. These provide a more
global and more manageable view, respectively, on the pattern sets.

To accommodate the observation of multiple songs or pattern sets, we also added the family
view, illustrated in Figure 3.5. Here, we use the vertical axis to denote different song-pattern set
pairs, as replacement for pitch information. The bounding box of each pattern is still present in
this view, so it mirrors the piano roll representation. Similarly, patterns can still be clicked, to
the same effects.
CHAPTER 3. VISUALIZATION

Figure 3.3: The piano roll illustration of a song, including marked patterns and integrated song player.

Figure 3.4: The piano roll illustration of a song, as heat map.

This view on the data can be applied to view a tune family using the same source of pattern sets, but also to view one song, processed using different parameters for the same algorithm. If the clutter from having too many patterns in the piano roll representation is too great, this view can even be used to denote one song, with each line showing the positions of a different pattern. Of course, a heat map version of this view also exists.
Figure 3.5: The illustration of a tune family, marking the pattern positioning only, as simplified song representation.

### 3.2.2 Navigation

A good approach to visualizing data for cross referencing, and allowing users to keep track of more information, is using large singles and small multiples. Based on this notion, we made the viewer consist of a few large panels, and a bunch of smaller ones, each allowing a single view to be put on it, illustrated in Figure 3.6. Aside from the interactivity on the views themselves, these panels can be right-clicked to change what data they contain.

Figure 3.6: The main screen of the tool, contains large and small panels to put visualizations in.
When right-clicking a panel, the "Container Filler" opens, as seen in Figure 3.7. This window allows the user to change what panels contain what views, as well as easily fill in panels with new views. Abstracting from the file structure, the user can select the songs or families to inspect, as well as the patterns to inspect them with. After selecting the desired data from the combo boxes, the user can pick any of views by pressing the corresponding button.

![Container Filler](image)

Figure 3.7: The container filler, the window through which all view changes are made.

To make navigation much smoother, the container filler pre-selects the combo boxes to match the data originally in the panel, even taking into account what pattern was last clicked. This allows the user to easily zoom in and out of the same data.

### 3.3 Evaluation

In this section we discuss how we conducted the evaluation of the visualization tool, and the results of this evaluation.

#### 3.3.1 Protocol

As our evaluation protocol, we let a few potential users in the field of pattern discovery use the tool to perform some specific tasks. These tasks are followed up by a brief discussion. The specific
tasks used will be omitted here, instead they are noted in the appendix, in Chapter B. These tasks are geared towards the problem analysis we do for this thesis.

Before starting to perform the task, the user gets to read through the instructions in Chapter A, and play around with the tool for about 5 minutes. During this time, we note how experienced the user is with similar tools, how familiar the person is with recognizing musical patterns, and how familiar the user is with the discovery algorithms used. We also make sure that users who will do the evaluation later are not present in the room.

As the tasks are executed, the steps the user takes to solve the task are written down by the evaluator, who only observes. During this process, the subject is encouraged to speak their thoughts, and may consult the instructions further. When a user gets stuck, the evaluator may nudge them to take a look at the instructions again, the time stuck is written down. The evaluator takes care to prioritize writing down steps that deviate from expectations.

After each task, the time taken is written down, and a few questions are asked to the user. Which tools were particularly useful or useless is asked, as well as whether or not the visualization tool was suitable for the task. At the end, users and evaluators discuss the findings of the evaluation, with open user input.

3.3.2 Results

The navigational functions of the tool, although usually compounded from smaller functions, were deemed highly useful, especially together with the multiple panels. Since all users involved have a good ear for music, playing the songs and patterns was also a very essential feature, which got used very extensively.

The heat map views were not considered very useful. Most users either do not consider using them, or feel there is no intuitive use for them. After explaining where in the tasks these might be applicable, they were reconsidered as a potentially valuable option.

Some tasks were not completed by all users, and some users took longer than expected on the tasks. The causes of this come from various sources. The most consistent problem was that, even with the tool, working with large numbers of patterns is difficult. Sometimes the phrasing of the tasks or the clutter of the instructions caused considerable delay. Sometimes the users went past the scope of the task. Experience with the tool or intuition for it considerably shortened the time taken, while the lack of both caused delays. This indicates that the tool could be made more intuitive for new users, or the instruction manual could be improved. Either case is not urgent, as a bit of experience alleviates the problem.

Background knowledge of the algorithms and data sets inspected sped up task execution considerably. This implies that the tool would be highly useful to further inspect familiar data, or supplement other tools. In addition to the tasks described, which all correspond to the problems approached in this thesis, the tool was also considered to have potential for other analysis tasks. The tool is likely worth expanding to further accommodate other branches of analyzing the data.

Due to the amount of loading the tool has to do on the spot sometimes, the responsiveness is lacking in a few areas. Additionally, allowing the panels to be scaled, shaped and positioned dynamically at run time was deemed a desirable feature. Lastly, some more data previewing would speed up task execution. This can come in the form of pattern shape previews, or noting the number of patterns for given settings.
Chapter 4

Problem Analysis

In this chapter we take a look at the extensive data generated through the evaluation pipeline by Peter Boot [2], through the lens of the visualization tool described in Chapter 3. We aim to isolate problems with the existing family pattern discovery methods. This is done in comparison to the common properties we see in human annotated patterns, as discovered by Volk and Van Kranenburg [12].

We will divide the problems we find in two categories and formulate corresponding informal requirements. First we discuss problems fundamental to pattern discovery as is, in Section 4.1. Second, in Section 4.2, we discuss the problems that specifically appear when performing patterns discovery on a tune family, rather than single songs. Finally, we discuss how these problems are already partially solved by existing algorithms, in Section 4.3.

4.1 Fundamental Problems with Pattern Discovery

Pattern discovery has its own problems without extending it to include an entire tune family. We look at the general issues in this field. We will also look at problems with each representation. Finally, we sum up all the requirements we get from these problems.

4.1.1 General Weaknesses of Pattern Discovery

The first problem that often shows up in discovery algorithms, is the problem of simply finding too many results. Typically, a lot of superfluous patterns are found, either by not being all that interesting, or simply being slight variations of other discovered patterns. In this case, finding the actual meaningful patterns in the results may not even be more efficient than just discovering by hand.

![Figure 4.1: The MGDP algorithm illustrating the problem of finding too many patterns.](image)

(a) The pattern distribution found for a subset of parameterizations by MGDP.

(b) The patterns found for the bottom parameterization in Figure 4.1a.

Figure 4.1: The MGDP algorithm illustrating the problem of finding too many patterns.
CHAPTER 4. PROBLEM ANALYSIS

Figure 4.1 illustrates this problem. The illustrated pattern distributions are, in fact, some of the better ones picked out of the entire parameter space. The topmost parameterization in Figure 4.1a results in 24 patterns, while the bottom one results in 8 patterns. Actually, there are only 2 distinct patterns present, as seen in Figure 4.1b. One of these is ignored by the annotated motifs, while the remaining pattern is also grouped together with the first pattern appearing in the topmost parameterization.

From this we gather the requirement IF01.0 that our discovery approach should not return too many patterns. This means that the discovery results should be in small linear proportion to the song. This is only natural, as notes being present in many patterns makes these patterns less significant among each other, and less oversee-able. We can build on this requirement further by looking at Figure 4.2. In this figure, we see that patterns do not overlap with each other in the time domain. We take this as our requirement IF01.1, as being a more demanding, but possibly easier to work with, requirement.

This also requires, as requirement IF02, that the output should not be bloated with slight variations on the same pattern. If a slight variation is present somewhere the exact pattern is not, then this should be considered the same pattern, as opposed to generating an entirely new set of patterns. If slight variations on a specific occurrence are present, only one of these should be taken as part of the output.

Another general problem is that the usage of chromatic pitch misses out on patterns that are clearly valid for morphetic pitch. In Figure 4.2, we see the light green pattern being considered as valid by human analysts. This pattern can only be seen by algorithms if they are using the morphetic pitch, which provides information on what scale the music is written in. Sometimes, although it is outside our scope, this may not be provided, so a single chromatic pitch shift of lenience can be desirable. Another part of this problem is that the implementations that use morphetic pitch for processing also deliver the results in morphetic pitch. This makes these results lose the exact pitch information that chromatic pitch provides, as a morphetic pitch can be a few different chromatic pitches.

This gets us to requirement IF03, stating that patterns valid in morphetic pitch should be accounted for. This needs to be done without losing the detail that chromatic pitch provides.

Algorithms also have the problem of being too exact. Often the occurrence of a musical pattern is still considered to be valid, although it is not an exact match. This problem occurs mostly between different songs, so we will discuss this in Section 4.2.
4.1.2 Weaknesses of Geometric Representation

Geometric representation is powerful in the sense that it can ignore a lot of embellishment found between patterns. This, however, comes at a price. Geometric algorithms may find patterns that span a large part of the song, yet have only very few notes. This is illustrated in Figure 4.3, where we can see one occurrence of the yellow pattern starting before the other has even ended. In most cases, we should be splitting these up into smaller patterns. In this case we can even see that both the start and end of each pattern are duplicates of each other.

Figure 4.3: The COSIATEC algorithm illustrating the problem of finding wide, sparse patterns.

Another way to look at these problematic patterns is by essentially performing transposition on the pattern. That is, for pattern occurrences $O, O', O''$, we can take their respective first notes $t_1, t'_1, t''_1$ as one transposed occurrence $O_T^1$. We do the same with their respective second notes $t_2, t'_2, t''_2$ to make the transposed occurrence $O_T^2$, and so on for the rest of the notes. In the event of these wide patterns, where the the same notes among all occurrences lie closer together than the notes in a single occurrence do, this might be better.

This gives us the requirement $IF04$, stating that pattern occurrences should not be sparse. More specifically, a pattern occurrence must not contain large gaps of silence. The gap size allowed can either be considered based on the song as a whole, or on the note density of the pattern. The latter would not really solve the width issue, but it may allow for pattern discovery for more high level constructions such as chord progressions. For our purposes, we will only consider the former.

Since the exact geometry of patterns is used in this approach, we get another potential problem. Rhythmic variation may take place, causing the shape to change between two occurrences of, what could be considered, the same pattern. This creates another requirement $IF05$, stating that minor variations in rhythm should not hinder patterns from being discovered.

A variant of this problem occurs between songs, in that they can deform in their entirety. We discuss this further in Section 4.2.

4.1.3 Weaknesses of Sequential Representation

Where the geometric approach finds its strength in ignoring embellishments and weakness in rhythmic variation, sequential representation displays the exact opposite problem. Having an extra note between a pattern occurrence completely changes how the pattern looks, whereas changing the timing of a note does not. This occurs since the order of notes is the primary timing measure, rather than the onsets of notes. The timing of notes can still be taken into account, but since rhythmic variations are welcome, this can largely be ignored. The main problem is that the efficiency of sequential pattern discovery largely stems from considering only the next note in sequence for the pattern.

Sequential representations also have a trade off between handling transposition and properly working around embellishments. Transposition refers to the pitch shift of an entire song or pattern...
CHAPTER 4. PROBLEM ANALYSIS

occurrence. Transposition between pattern occurrences should be allowed, but looking at the song as a sequence of pitches makes two transposed patterns appear completely different to an algorithm. The typical solution is to look at the song as a sequence of pitch transitions, the pitch difference between two sequential notes, but this disallows the algorithm from easily looking ahead in the sequence to circumvent embellishments.

As example, consider the two sub-sequences “ABCD” and “DEG”. We can easily see that “DEG” would be a transposition of “ABD”. We can look at the sequences as is, and see that they are completely different. Alternately, we can look at the pitch transitions [+1, +1, +1] and [+1, +2], and still have the sequence look different.

This problem gives us two requirements. The first is requirement IF06, indicating that transposed variants of patterns should be discovered. The second is requirement IF07, indicating that patterns should still be discovered if there are differing or extra notes in between.

4.1.4 Fundamental Problems Summed Up

To collect the requirements described in this section, we include Table 4.1. This table concisely describes each informal problem and shows by which ID each requirement is identified.

| IF01.0 | The output size must be linear in the input size. |
| IF01.1 | Pattern occurrences may not overlap in the time domain. |
| IF02  | Slight variations on represented patterns should be omitted. |
| IF03  | Patterns should be returned in chromatic pitch, but take into account matches in morphetic pitch. |
| IF04  | Pattern occurrences should not contain large gaps without notes. |
| IF05  | Rhythmic variations of a pattern should be discovered. |
| IF06  | Transposed variations of a pattern should be discovered. |
| IF07  | Embellished variations of a pattern should be discovered. |

Table 4.1: Informal requirements from fundamental problems with pattern discovery.

4.2 Problems with Tune Families

Rather than the traditional approach of simply discovering patterns in a single song, we look at patterns across a family of songs. This setting brings in other requirements to make patterns interesting. It is these requirements that we will discuss in this section, along with any others that might arise from this change.

We will first discuss the problems that arise from discovering patterns in tune families specifically. Next we will discuss how we can use our extra information to better select important patterns. Finally, we again end with a compilation of all requirements of this section.

4.2.1 Discovering Patterns in Families

First and foremost, we have the simple problem of performance. Usually within single songs, there is not all that much data to consider, so our algorithms can get away with doing more resource-intensive computations. Since we are now raising our input size, typically in one order of magnitude, we require that our algorithms run in low polynomial time. This will be our requirement IG01. This problem occurred for Peter Boot [2] when processing using SIARCT-CFP [4].

We mentioned in Section 4.1 that we have a variant of rhythmic variation to deal with. In Figure 4.4, we see two songs around the upper middle of the list display behavior slightly different from the rest. These songs are stretched in the time-domain, which preserves their patterns, but makes them more difficult to pick up on for algorithms. We take as our requirement IG02, that patterns stretched in the time domain should be treated as if they were not stretched.
A similar problem occurs when entire songs are transposed. Since transposition is common within songs, this was already discussed in Section 4.1. We may still choose to address it from this perspective, however.

As described in requirements IF05, IF06 and IF07, variations of patterns should be discovered and grouped together. It is important to note that these variations occur mostly between different songs in a tune family, rather than in the song itself. This is illustrated in Figure 4.5, where we see three songs from the same family displaying different instances of the pattern in blue, yet within each song they remain consistent. With this, we note that pattern variations should primarily be picked up between songs, rather than within songs. The algorithm should be more lenient with inter-song variations, as well as prioritize these over intra-song variations, this is our requirement IG03. Transposed variations are relatively common within songs, however.

Another important thing to note in the annotated patterns of Figure 4.5, is that the variations do not always share the same notes. As can be imagined with gradually evolving songs with many variations along the way, patterns may change in different ways. If we discover the pattern between Figure 4.5a and Figure 4.5c, we likely miss one of the middle notes on the first. This middle note may be important to tie this pattern to the one discovered between 4.5a and 4.5b. In the annotated motifs, we also see that patterns are always taken as all the notes within their time-range. As such, we make requirement IG04, stating that all discovered patterns should contain all notes within their time-range. This may, of course, change when building a solution for polyphonic music.
4.2.2 Selecting Appropriate Patterns

An important thing to note in Figure 4.4 is that all songs are roughly described by the same patterns in the same way. From this we can derive two requirements. Only one of these requirements need be satisfied at a time, as they are variants of each other.

The first requirement we can derive, requirement \textit{IG05.0}, is that the patterns of each song must occur in similar sequence for each song. This simply stems from each song in the family having a similar global structure, which changes very rarely. When there are differences, this is usually by patterns missing from the sequence or being changed enough to not qualify as the same pattern anymore. Each song is different from another both in a slight change in this sequence of patterns, as well as the exact details of each patterns. The patterns themselves typically differ more than the sequence, so we should group and select pattern occurrences accordingly.

The second requirement we derive, requirement \textit{IG05.1} is that the positions of the patterns in the time-domain of the song are consistent. This consistency, of course, is when working around the stretching described for requirement \textit{IG02}. Typically, the songs have equally spaced time between patterns, keeping their structure intact. Since patterns are this consistent in positioning, we could even go as far as including the notes around smaller occurrences, to include the same time-range as typically occupied by that same pattern in other songs.

Another thing we can infer from Figure 4.4 is that only patterns that occur in a lot of the songs are selected. Since we are primarily looking for patterns that identify a tune family, finding the patterns that occur many times across different songs are more important to us than the ones that occur often within a small subset. This will be requirement \textit{IG06}, indicating that selected patterns should occur in the majority of the tune family.

Finally, there is an approach the MGDP algorithm \cite{7} that is likely very good to adopt. This is referred to as the distinctiveness of a pattern, which is essentially the ratio between how often a pattern occurs in the tune family and how often the pattern occurs in a set of other songs. Using this makes the algorithm return patterns that are actually indicative of the tune family, rather than simply using any pattern that occurs everywhere as well. From this, we construct our final requirement \textit{IG07}, indicating that discovered patterns should be over-represented in the tune family with respect to a diverse song selection.

4.2.3 Tune Family Problems Summed Up

Table 4.2 collects all the requirements of this section. This, once again, concisely describes the core requirements that stem from applying pattern discovery on tune families.

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IG01</td>
<td>The algorithm should use a low polynomial amount of time and memory.</td>
</tr>
<tr>
<td>IG02</td>
<td>Patterns stretched in the time-domain should be discovered just as non-stretched patterns.</td>
</tr>
<tr>
<td>IG03</td>
<td>Pattern variations between songs should be prioritized over those within songs.</td>
</tr>
<tr>
<td>IG04</td>
<td>Pattern occurrences should contain all notes in their time-range.</td>
</tr>
<tr>
<td>IG05.0</td>
<td>Returned patterns should occur in similar order for each song.</td>
</tr>
<tr>
<td>IG05.1</td>
<td>Returned patterns should have consistent positioning across songs.</td>
</tr>
<tr>
<td>IG06</td>
<td>Returned patterns should occur in the majority of the tune family.</td>
</tr>
<tr>
<td>IG07</td>
<td>Returned patterns should occur more in the tune family than elsewhere.</td>
</tr>
</tbody>
</table>

Table 4.2: Informal requirements from problems with pattern discovery on tune families.

4.3 Existing Partial Solutions

Not all problems described in this chapter are unsolved, of course. The issue is that the solutions have not been applied all in one algorithm. In this section, we look at some previous work, from which we can draw inspiration for our own approach.
In general, of course, it still holds that requirements IF04 and IF05, as they are specific to geometric approaches, are solved by taking a sequential approach. This goes vice versa for requirements IF06 and IF07, although requirement IF06 is easily worked around for most sequence based algorithms.

4.3.1 SIAR

The improvement made in SIAR [5] makes a step towards satisfying requirement IG01, which is to lower the time and memory required for the algorithm. It then determines which notes are worth comparing, purely based on the notes immediately surrounding it. The resulting set of patterns contains only the patterns where each note has at least one other note in the pattern nearby. This makes the patterns very compact.

Depending on the input parameter, this can be significantly faster than SIA. The real performance gain we get from this, however, is that the set of discovered patterns is smaller. Since the patterns missing from the output are very sparse, these are uninteresting to us, as per requirement IF04.

4.3.2 SIARCT-CFP

The SIARCT-CFP algorithm by Collins et al [4] contributes two useful ideas. The first is the Compactness Thrawler, which serves as a filter to remove uninteresting patterns. This serves both to help with requirement IG01 in lowering the data size for post-processing, and in proposing the idea of a linear filter.

The second contribution of SIARCT-CFP is the fingerprinting and categorization step it takes. The fingerprinting ignores time-stretch entirely and allows lenience for rhythmic variation. This concept itself already provides a foundation for dealing with requirements IF05 and IG02, taking care of stretched and rhythmic variations of patterns.

The categorization step sorts the discovered patterns in order of importance, as determined by a metric of choice. It then takes the most important pattern and adds similar ones to it to form a more frequent pattern. This essentially clusters the patterns together into larger sets of occurrences, each containing the rhythmic variants of its patterns. This could possibly be used to cluster for other types of variants as well. As such, this process already provides a solution for requirement IF05 and potentially also for requirement IF07.

4.3.3 COSIATEC

The COSIATEC algorithm by Meredith [10], despite being primarily a compression algorithm, still provides us with useful solutions to a few requirements. The important matter COSIATEC introduces is the pattern selection process. Since COSIATEC selects a pattern and removes it from the input before rerunning, we prevent patterns from having the same notes. This immediately solves requirement IF01.0, as the lack of duplicate notes in the output means it must be linear in the input size. We can take this selection concept and adjust it to select for other criteria than compression potential. If we solve requirement IG04 so that all pattern occurrences cannot be overlapped in the time domain. Doing so upgrades our solution to requirement IF01.0 to a solution for requirement IF01.1.

4.3.4 MGDP

Finally, the MGDP algorithm by Conklin [7] satisfies requirement IG07, as its contribution gave rise to the idea of that requirement. We explained in Section 2.4.6 that MGDP constructs a tree of patterns, and regulates this construction by a number of criteria.

The criterion we are interested in is the distinctiveness of a pattern, which is the primary determining factor for pattern selection. The ratio between the number of occurrences outside the tune family and inside the tune family is queried and calculated. This ratio is then compared to a
CHAPTER 4. PROBLEM ANALYSIS

threshold in the form of a parameter for the algorithm. If the pattern is sufficiently more frequent in the family than outside of it, the pattern is accepted for the output.

MGDP also covers requirement IG06. It does this through the criterion that patterns should be supportive. This simply states that a pattern must occur in a certain portion of the songs in the tune family.
Chapter 5

Formal Problem Description

In this chapter, we take the findings of Chapter 4, and formalize them into algorithmically solvable problems. We start by introducing the formal syntax that we will be using for our formal definitions. We will then discuss the formalization of each requirement, first considering the ones that are not strictly formalized, then the requirements that have interplay among each other, and finally the requirements that function in isolation. Finally, we combine and compile all formal requirements together.

5.1 Formal Syntax

Before we can formalize any requirements, we need to establish the notation that we will use for the data. For this, we first refer to the definitions already given in Section 3.1.1. The first definition involves a tune family $F$ consisting of songs $S$, which in turn consist of tuples $t$ denoting the notes of the songs, as seen in Definition 3.1.1. Similarly, we have pattern sets $R$ consisting of patterns $P$, which have occurrences $O$, also made up of tuples $t$, as seen in Definition 3.1.2.

Although the tuples previously defined are of a different form, we will use them interchangeably. Our input still has the format given for songs, that is, tuples of the form $(o, c, m, d, v)$. Our output similarly still has the format given for pattern sets, except that we will refer to the tuples in the same form as the input. As per requirement IF03, the actual output we give with our solution should have tuples using the chromatic pitch, hence they would have the form $(o, c)$. All calculations, however, should be done using morphetic pitch $m$, so we will incorporate this in our formalization.

In order to easily reason about all our requirements, we need to establish how we denote tuple properties and introduce a few reference functions. A tuple property $a$ for a tuple $t_i$ will be denoted as $t_i.a$, for example, the chromatic pitch of $t_1$ is denoted as $t_1.c$. We also may need to refer to the origin of a pattern occurrence $O$, for this we use Song($O$) to return the song $O$ comes from, or Pattern($O$) or PatternSet($O$) to return its pattern or pattern set, respectively. Similarly, the song a tuple $t$ belongs to can be returned with Song($t$).

For further convenience, we also introduce a few formal functions, which will occur regularly. First, we denote the number of elements in a set $A$ as $|A|$. The number of tuples in a family $F$ will be defined as $\text{Size}(F) = \sum_{S \in F} |S|$. The same applies to patterns and pattern sets, defined as $\text{Size}(P) = \sum_{O \in P} |O|$ and $\text{Size}(R) = \sum_{P \in R} \text{Size}(P)$. We define the start and end of a song or pattern occurrence $Q$ as $\text{Start}(Q) = \min_{t \in Q} t.o$ and $\text{End}(Q) = \max_{t \in Q} t.o$, respectively, where $\min$ takes the least number in a set, and $\max$ takes the greatest. From this, we also define the length of $Q$ as $\text{Length}(Q) = \text{End}(Q) - \text{Start}(Q)$. Each requirement formalization may still introduce its own functions.
5.2 Unformalized Requirements

In this section we discuss the requirements that are not strictly formalized. We first look at some requirements that are superseded by others, since some other requirements include their properties already. Lastly, we will look at the requirements that do not have a significant formalization component. The requirements stated here are compiled in Table 5.1 in 5.5.

First of all, we have informal requirement IF01.0, stating that the output size should be linear to the input size. Since requirement IF01.1 states that no pattern occurrences in the output can overlap, the output can contain at most 1 of each tuple. Therefore, we do not need to make a formal requirement FU01 for requirement IF01.0, as the output is strictly smaller than the input. Requirement IF01.1 will be discussed further in Section 5.3.

Requirement IF01.0 is also superseded by requirement IF01.1. This requirement states that we should not output slight variants on existing patterns. The supersede is also partially caused by all our requirements that allow pattern occurrences to vary a bit without being considered different. Since we have no overlap in patterns and any slight variations are grouped into the same pattern, variant patterns are eliminated entirely, as is this formal requirement FU02.

For formal requirement FU03, we have no formalization of informal requirement IG03. This requirement gives priority to patterns that occur a lot across the tune family, rather than being abundant in a few songs. Since requirement IG06 states that patterns must occur in a majority of the songs, we naturally gain this prioritization.

Requirement IF03 gives us both a formal requirement and an effect across other requirements. For the formal requirement FU04, we have that the output tuples should be of the form \((o, c)\), containing the onset and chromatic pitch. The informal requirement also states that all calculations should be done using morphetic pitch. To this end, whenever we use the pitch in the formal requirements, we use morphetic pitch \(m\).

Finally, we have requirement IG01, bounding our resource usage to low polynomial complexity. Since SIATEC [10] was used as a base for algorithms that run well enough, we can use its processing time as a bound. This gives us the formal requirement FU05, stating that the running time, and by extension memory usage, should be bounded by \(O(n^3)\), where \(n\) is the number of tuples in the input, so \(O(\text{Size}(F)^3))\).

5.3 Interplaying Requirements

In this section, we discuss the formal requirements that build on each other. We discuss the requirements from bottom to top, such that the later requirements depend on the earlier ones. The exact formal notation of these requirements is found in Table 5.2 in Section 5.5.

Our first formal requirement FI01, based on requirement IF04, describes how pattern occurrences may not contain large gaps. This means that occurrences \(O\) must satisfy a boolean function GapLess\((O, gs)\), in which \(gs\) is the input parameter for the size of gap we want to allow. The function simply returns whether for each tuple \(t\) in \(O\), the tuple is either the start, or has some preceding tuple within distance \(gs\).

The next requirement FI02 builds directly on the definition of what it means for a pair of sets of tuples to be part of the same pattern. The basic criterion for two of these sets, or pattern occurrences, to be of the same pattern, is for them to be identical in pitch and rhythm. This means each pair of tuples in one occurrence must have a pair of tuples in the other occurrence with the same morphetic pitches and distance between their onsets.

We add onto requirement FI02 the informal requirement IF05, stating that some rhythmic variation should be allowed. From this, we get a boolean function RhythmicValidPair\((O_1, O_2, rr)\), where \(O_1, O_2\) are our occurrences and \(rr\) is a parameter for the ratios that two matching tuple pairs have to be within with their onset distances. The distance between the onsets of a set of tuples in occurrence \(O_1\), \(di_1\), should be between \(di_2 * rr\) and \(di_2/rr\), where \(di_2\) is that same distance for the tuple in \(O_2\). Since we use ratios this way, the input parameter \(rr\) work consistently regardless of how long or short the tuple intervals are, as well as working symmetrically between occurrences.
Building upon the definition of valid pairs of occurrences, we now want to allow transposition, as suggested by informal requirement IF06. To describe this formal requirement FI03, we define the boolean function TransposedValidPair\((O_1, O_2, rr)\), where we see \(rr\) again, simply to transfer it down to our base function. The function determines whether there exists some transposition \(tr\), such that if we shift the morphetic pitches \(m\) of all notes in \(O_2\) by \(tr\), the pair satisfies RhythmicValidPair.

For the next informal requirement IF07, we want to allow patterns that have notes that do not match in between, while still keeping those notes in the pattern. For this formalization to the requirement FI04, we need to build on both our functions TransposedValidPair and GapLess. We do this in the form of a boolean function ValidEmbellishedPair\((O_1, O_2, rr, gs)\), where \(rr\) and \(gs\) are both passed down to the functions it builds upon. This function determines whether a pair of occurrences \(O_1, O_2\) have subsets \(O'_1, O'_2\) that are a valid pair according to TransposedValidPair. In order for these subsets to not be empty, GapLess must apply to both, and the length of each subset must be at least the length \(O_1\) or \(O_2\), minus \(gs\). This ensures that the actual matching notes must span the original patterns and sufficiently represent them.

Finally, informal requirement IG02 states that pattern occurrences from different songs should still be valid if they are simply stretched in the time-domain. Since the primary aim here is that songs as a whole are considered stretched in the time-domain, we apply this requirement on the scope of pairs of songs, rather than pattern occurrences. The formal requirement FI05 for this is described by the boolean function ValidStretchedSet\((F, R, rr, gs, ss)\). This is the top level function of this section, so we apply it to the tune family \(F\) and the set of patterns \(R\). The parameters \(rr\) and \(gs\) are passed down once again, and \(ss\) is a parameter between 0 and 1 determining how much stretch we allow. The function simply checks for all pairs of songs \(S_1, S_2\) if there is a stretch \(st\) between \(ss\) and 1 such that we can compress one and have all pattern be valid given that compression. With compressing we mean that the onset of each tuple is multiplied by \(st\). The validity in this case means that all pairs of occurrences of that pattern that occur in these two songs must be valid according to ValidEmbellishedPair while always having the same stretch. Since we compress down either of the songs for this, our gap size parameter \(gs\) is applied normally to one song, and effectively scaled up for the other, making it allow the larger gaps that the larger song normally has.

### 5.4 Independent Requirements

In this section, we discuss the last set of formal requirements. Each requirement here may borrow functions from the interplaying requirements, but otherwise stands on its own. The exact formal notation of these requirements is found in Table 5.3 in Section 5.5.

Our first requirement FS01 is a formalization of informal requirement IF01.1, stating that no pattern occurrences should overlap. The boolean function NonOverlap\((R)\) for this simply states that for any pair of occurrences \(O_1, O_2\), regardless of pattern, if the occurrences are in the same song, one should end before the other starts.

A similarly simple formal requirement FS02 formalizes requirement IG04. The informal requirement here is that an occurrence must contain all tuples in its time range. This is easily formalized in the boolean function CompleteSubsequence\((R)\). This function states that for all occurrences, all tuples between their start and end that exist in its song must also be in the occurrence.

The last simple formalization is requirement FS03. This requirement formalizes requirement IG06, stating that patterns should occur across the majority of the tune family. We do this with the boolean function MajorityPresence\((F, R, mr)\), in which the input parameter \(mr\) denotes what portion of the songs should at least represent each pattern. The function states whether for all patterns, the number of unique songs the pattern has occurrences in is greater or equal to \(mr\) times the number of songs in the family.

Requirement IG07 states that a pattern should occur more frequently inside the tune family than outside. For the formalized requirement FS04, we have to bring in a new input variable
\( U \), which denotes a set of songs as sample space for pattern frequency. That is, we take \( U \) as a representative for the set of all songs in the universe. We then check how frequent a pattern is in \( U \) to determine if the pattern is significantly more frequent in \( F \). We formalize this requirement into the function \( \text{FamilyIdentifying}(F, R, U, rr, gs, ss, ir) \), in which we use parameters \( rr, gs, ss \) to allow inexact pattern occurrences from \( U \), and use \( ir \) as a parameter for how significant the frequency inside the family should be. The function states that for each pattern \( P \), the number of occurrences per tuple in the family should be at least \( ir \) times the number of possible valid occurrences we could find in \( U \) per tuple in \( U \). A possible occurrence \( O \) in \( U \) is valid if for all occurrences \( O' \) in \( P \), there is some stretch \( st \) such that we can compress either occurrence to make it valid according to \( \text{ValidEmbellishedPair} \). This is slightly lenient for accepting patterns from \( U \), but this can be counteracted by selecting a lower \( ir \).

Our last two requirements, requirement \( FS05.0 \) and requirement \( FS05.1 \), are based on requirements \( IG05.0 \) and \( IG05.1 \), respectively. In the exact formalization, these are rather extensive, but the core is relatively straightforward.

For requirement \( FS05.0 \), we must denote that patterns occur in a consistent order across songs. We do this with the boolean function \( \text{ConsistentOrder}(F, R, rr, ss, ac) \), where \( ac \) denotes how large the sequence should be. Basically, we state that the size of the largest common subsequence between any two songs is less than \( ac \) times smaller than the number of tuples in the pattern set for those songs. A sufficiently large sub-sequence indicates that the patterns are both similarly ordered and have high similarity within pattern occurrences. In the exact formalization, we use the parameters \( rr \) and \( ss \) for an interpretation of the largest common sub-sequence on our data format.

Finally, for requirement \( FS05.1 \), we need to formalize that all patterns should occur in similar positions across each song. For this, we define the function \( \text{ConsistentPosition}(F, R, ss, pa, pg) \), where \( pa \) denotes the leniency with absolute positions, after allowing stretching within \( ss \), while \( pg \) denotes the leniency with global positioning. The function states that for all patterns, if they have any occurrences in a pair of songs, there should be at least one pair of occurrences in similar position. Similar position applies when the starting onsets of both occurrences are at most \( pa \) apart. Alternately, a position is similar if the start onsets divided by the song lengths are at most \( pg \) apart, that is, considering the duration of a song as a spectrum from 0 to 1.

### 5.5 Requirements Combined and Compiled

In this section we briefly present the full formal function for determining if an input-output pair is valid. We follow this up with the exact formalizations of each requirement described in the preceding sections, in tables in the order of their respective sections.

As to be expected, the full formal boolean function can simply be defined as the conjunction over all preceding functions. For this we take the top-level function \( \text{ValidStretchedSet} \) from the interplaying requirements and all the functions denoting independent requirements. The result is the conjunction below:

\[
\text{ValidSolution}(F, R, U, rr, gs, ss, mr, ir, ac, pa, pg) = \\
\text{ValidStretchedSet}(F, R, rr, gs, ss) \land \text{NonOverlap}(R) \land \text{CompleteSubsequence}(R) \land \\
\text{MajorityPresence}(F, R, mr) \land \text{FamilyIdentifying}(F, R, U, rr, gs, ss, ir) \land \\
\text{ConsistentOrder}(F, R, rr, ss, ac) \land \text{ConsistentPosition}(F, R, ss, pa, pg)
\]
<table>
<thead>
<tr>
<th>ID</th>
<th>Informal ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FU01</td>
<td>IF01.0</td>
<td>Superseded by requirement IF01.1</td>
</tr>
<tr>
<td>FU02</td>
<td>IF02</td>
<td>Superseded by IF01.1</td>
</tr>
<tr>
<td>FU03</td>
<td>IG03</td>
<td>Superseded by requirement IG06</td>
</tr>
<tr>
<td>FU04</td>
<td>IF03</td>
<td>Output tuples of the form ((o, c)), morphetic pitch used in all requirements</td>
</tr>
<tr>
<td>FU05</td>
<td>IG01</td>
<td>The worst case time and space complexity should be (O(\text{Size}(F)^3))</td>
</tr>
</tbody>
</table>

Table 5.1: Formal requirements without significant formal component, based on informal requirements, with related identifiers.

<table>
<thead>
<tr>
<th>ID</th>
<th>Informal ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FI01</td>
<td>IF04</td>
<td>(\text{GapLess}(O, gs) = \forall t \in O</td>
</tr>
</tbody>
</table>
| FI02 | IF05        | \(\text{RhythmicValidPair}(O_1, O_2, rr) = \)  
|       |             | \(\text{RhythmicSubset}(O_1, O_2, rr) \land \text{RhythmicSubset}(O_2, O_1, rr)\)  
|       |             | \(t_{11}.m = t_{21}.m \land t_{12}.m = t_{22}.m \land \)  
|       |             | \((t_{11}.o - t_{12}.o) \cdot rr \leq t_{21}.o - t_{22}.o \leq (t_{11}.o - t_{12}.o)/rr)\) |
| FI03 | IF06        | \(\exists tr[\text{RhythmicValidPair}(O_1, \text{Transpose}(O_2, tr))]\)  
|       |             | \(\text{Transpose}(O, tr) = \{t | ![t] \in O \land t = (t'.a, t'.c, t'.m + tr, t'.d, t'.v)\}\) |
| FI04 | IF07        | \(\text{ValidEmbellishedPair}(O_1, O_2, rr, gs) = \exists O_1', O_2' | ![O_1', O_2']\|
|       |             | \(\text{ValidlyUnEmbellished}(O_1', O_1, O_2, gs) \land \)  
|       |             | \(\text{ValidlyUnEmbellished}(O_2', O_2, O_1, gs) \land \text{TransposedValidPair}(O_1', O_2', rr)\)  
|       |             | \(\text{ValidlyUnEmbellished}(O', O_1, O_2, gs) = O' \subset O_1 \land \text{GapLess}(O', gs) \land \)  
|       |             | \(\text{Length}(O') \geq \min(\text{Length}(O_1), \text{Length}(O_2)) - gs\) |
| FI05 | IG02        | \(\forall S_1, S_2 | ![S_1, S_2] | ![S_1, S_2] \in F | ![S_1, S_2]| ss \leq st \leq 1\)  
|       |             | \(\text{ValidStretchedSet}(F, R, rr, gs, ss) = \forall S_1, S_2 | ![S_1, S_2] | ![S_1, S_2] \in F | ![S_1, S_2]| ss \leq st \leq 1\)  
|       |             | \(\text{ValidStretchedPair}(S_1, S_2, R, rr, gs, st)\)  
|       |             | \(\text{ValidStretchedPair}(S_2, S_1, R, rr, gs, st)\)  
|       |             | \(\forall P | ![P] | ![P] | ![P] \in R | ![P] | ![P] | ![P] \land \text{Song}(O_1) = S_1 \land \text{Song}(O_2) = S_2\)  
|       |             | \(\text{ValidStretchedPair}(S_1, S_2, R, rr, gs, st) = \)  
|       |             | \(\forall P | ![P] | ![P] \in R | ![P] | ![P] \land \text{Song}(O_1) = S_1 \land \text{Song}(O_2) = S_2\)  
|       |             | \(\text{ValidStretchedPair}(O_1, \text{Stretch}(O_2, st), rr, gs)\)  
|       |             | \(\text{Stretch}(O, st) = \{t | ![t] \in O \land t = (t'.o + st, t'.c, t'.m, t'.d + st, t'.v)\}\) |

Table 5.2: Interplaying formal requirements based on informal requirements, with related identifiers.
| FS01 | IF01.1 | NonOverlap($R$) = $\forall P_1, P_2 \in R[\forall O_1, O_2 \in P_1 \land O_2 \in P_2$
| | | $\Rightarrow$
| | | $\forall t \in \text{Song}(O_1) \Rightarrow \text{Song}(O_2) \Rightarrow$
| | | $(\text{Start}(O_1) > \text{End}(O_2) \lor \text{Start}(O_2) > \text{End}(O_1))$]
| FS02 | IG04 | CompleteSubsequence($R$) = $\forall P \in R[\exists O \in P[\forall t \in \text{Song}(O)$
| | | $(t \cdot o \geq \text{Start}(O) \land t \cdot o \leq \text{End}(O)) \Rightarrow t \in O]]$
| FS03 | IG06 | MajorityPresence($F, R, mr$) =
| | | $\forall P \in R[[\exists O \in P[\text{Song}(O) = S]]] \geq mr * |F||$
| FS04 | IG07 | FamilyIdentifying($F, R, U, rr, gs, ss, ir$) = $\forall P \in R$
| | | $|P|/\text{Size}(F) \geq$
| | | $ir * |\{ O \exists S [S \in U \land O \in S \land$
| | | $\forall O'[O' \in P][\exists st] [ss \leq st \leq 1]$
| | | $\text{ValidEmbellishedPair}(O, \text{Stretch}(O', st), rr, gs) \lor$
| | | $\text{ValidEmbellishedPair}(O', \text{Stretch}(O, st), rr, gs)])]|/ | Size(U)|
| FS05.0 | IG05.0 | ConsistentOrder($F, R, rr, ss, ac$) = $\forall S_1, S_2, S_1, S_2 \in F$
| | | $ac \leq \text{LargestCommonSubstring}(S_1, S_2, R, rr, ss) /$
| | | min(|TuplesInPatterns($S_1, R$), |TuplesInPatterns($S_2, R$))]
| | | LargestCommonSubstring($S_1, S_2, R, rr, ss = \max(|O_1||$
| | | $O_1 \subset \bigcup \{O|\exists P \in R[O \in P \land \text{Song}(O) = S_1]\})$
| | | $O_2 \subset \bigcup \{O|\exists P \in R[O \in P \land \text{Song}(O) = S_2]\}$)
| | | $\text{TransposedValidPair}(O_1, O_2, rr \ast ss)$
| | | TuplesInPatterns($S, R$) = $\{|t|\exists P \in R[\exists O \in P|t \in O \land t \in S]\}|$
| FS05.1 | IG05.1 | ConsistentPosition($F, R, ss, pa, pg) = \forall S_1, S_2, S_1, S_2 \in F|$\exists st [ss \leq st \leq 1]$
| | | $\text{StretchedConsistentPosition}(S_1, S_2, R, st, pa, pg) \lor$
| | | $\text{StretchedConsistentPosition}(S_2, S_1, R, st, pa, pg)$]
| | | StretchedConsistentPosition($S_1, S_2, R, st, pa, pg) =$
| | | $\forall P \in R[\exists O_1, O_2 \in P \land \text{Song}(O_1) = S_1 \land \text{Song}(O_2) = S_2 \Rightarrow$
| | | $\exists O_1, O_2 \in P \land \text{Song}(O_1) = S_1 \land \text{Song}(O_2) = S_2 \land$
| | | $(\text{SimilarAbsolutePosition}(O_1, \text{Stretch}(O_2, st), pa) \lor$
| | | $\text{SimilarGlobalPosition}(O_1, O_2, pg)])]$}
| | | SimilarAbsolutePosition($O_1, O_2, pa) = \text{Start}(O_2) - pa$
| | | $\text{Start}(O_1) \leq \text{Start}(O_2) + pa$
| | | $\text{SimilarGlobalPosition}(O_1, O_2, pg) = \text{Start}(O_2)/\text{Length}(O_2) - pg \leq$
| | | $\text{Start}(O_1)/\text{Length}(O_1) \leq \text{Start}(O_2)/\text{Length}(O_2) + pg$

Table 5.3: Independent formal requirements based on informal requirements, with related identifiers.
Part III

Methods
Chapter 6
Solution Pipeline

The set of requirements we have determined are varied enough to make a single-step algorithm that would address them overly complex. In order to address this issue, we opt to utilize a pipeline format, to break up our set of requirements into smaller subsets. Each task in the pipeline has the responsibility of solving one of the subsets of the problem.

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

In this chapter we describe the goals and requirements of each task in our pipeline. We also include the input and output format requirements. The chapters that follow will expand on each task, its challenges and their solutions. We will not dedicate a full chapter to pattern cropping, this will simply be an extension of the chapter on pattern clustering.

6.1 Input and Output

For both in- and output, we use the same format as used for the formal requirements, Chapter 5. The exact description for this is given in Section 3.1.1 under definitions 3.1.1 and 3.1.2, with again the exception that the extensive tuples from songs will also be used in patterns.

Every task gets both the set of songs, or tune family, $F$ and a pattern set $R$. In the case of the normalization and pattern discovery tasks, the set $R$ should still be empty or omitted as input, as no patterns are discovered up to that point. The normalization and reversion tasks will transform the pattern set and tune family slightly, so during all other tasks, the transformed versions must be used. Each task may have its own set of parameters associated with it, these will be inputted separately.

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

Figure 6.1: The algorithmic solution pipeline.
6.2 Normalization

The first task manipulating the data in the pipeline is normalization. In this task, we take the tune family \( F \) and transform all tuples \( t \) in each song \( S \), such that any pair of songs \((S_i, S_j)\) can be processed better by the discovery algorithms. Songs in a family have a relation in the form of derivation from one another, they have a family tree. With this normalization, we work around two forms of change between nodes in the family tree. First, songs may be reworked into a different key, meaning the pitches change in a consistent manner. Second, songs may be shifted in tempo or metre, meaning the onsets and durations will be stretched or contracted, while retaining the same order. Sequence-based discovery algorithms sometimes have trouble with the pitch shifts, while geometry-based algorithms often have trouble with the stretch of onsets. In this task, we take care of requirement \( FI05 \), as well as partially covering requirement \( FI03 \).

Since we have to use the reversion task to undo the changes normalization makes to all notes, we need a two-way function to both make the changes and undo them. Normalization changes the onset \( o \), the duration \( d \), and the pitches \( c \) and \( m \) of tuples, which are the time and pitch related properties, respectively. The voice \( v \) has no reason to change in this task. How something is normalized also depends on what song \( S \) it is in, as \( S \) provides the context of the tuples. This gives us the following tuple transformation function:

\[
\text{Normalize}(S, o, c, m, d, v) = (o_{\text{norm}}, c, m_{\text{norm}}, d, v)
\]

The time-based values, \( o \) and \( d \), will be multiplied, \( o_{\text{norm}} = o \times x \) and \( d_{\text{norm}} = d \times x \). Optionally, these may be translated as well, but this rarely makes a difference. The pitch-based values, \( c \) and \( m \), will be translated, \( m_{\text{norm}} = m + y \).

The specific mapping we want to find is one that allows discovery algorithms to find the most representative patterns to identify a family of songs. In Chapter 7, we discuss what a good quality normalization does, and how to achieve one. We do, however, note that it is important to retain a song’s structure after normalizing. A song cannot simply change each note in an entirely different manner, as this would lose the actual song in the process. By default, we should go for a single \((x, y)\) pair that applies its stretch and translation over all notes in the song.

This task has no immediate influence on the performance of tasks other than reversion. If we decide on a more complex normalization, and make the storage of the mapping non-trivial, the reversion task might perform significantly worse. The reversion task is discussed in Section 6.2.1.

6.2.1 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 task in the pipeline, before output is written. Our goal here is to transform each tuple in each pattern discovered back to how it was in the original song, before normalization.

In order to do reversion correctly, we need to apply the inverse function created by the normalization step, creating the original tuple \( t = (o, c, m, d, v) \):

\[
\text{Normalize}^{-1}(S, o_{\text{norm}}, c_{\text{norm}}, m_{\text{norm}}, d_{\text{norm}}, v) = (o, c, m, d, v)
\]

6.3 Pattern Discovery

The second task, the discovery task, runs a pattern discovery algorithm on the normalized family \( F \) supplied by the normalization task. The pattern discovery algorithm returns a set \( R \) with patterns \( P \in R \). Each pattern consists of occurrences \( O \in P \), each of which consist of tuples \( t \in O \) and a reference to the song \( S \in F \) they belong to, so also \( O \subset S \).

The pattern discovery task is more challenging than the normalization task. First of all, the output must be kept small, despite needing to find many interesting patterns. This output is potentially the largest set of data the pipeline must deal with, which then might be non-linearly...
processed by later tasks. Second, the algorithm in question must find patterns without being obstructed by additional tuples in the time range of a pattern occurrence, as per requirement \textit{FI04}. This problem is typically circumvented by geometric approaches. The final problem is rhythmic variation, patterns with slight variation in subsequent timing of the tuples, between different occurrences, as per requirement \textit{FI02}.

Pattern discovery should also discover transposed patterns, as it helps the normalization task with fully covering requirement \textit{FI03}. In most cases, it is also beneficial for this task to include all tuples in the time range of each occurrence, as per requirement \textit{FS03}. This should provide the clustering task with more grounds on which to merge patterns. This task can make use of requirement \textit{FI01} to prune its potential search space, as patterns with large gaps can be ignored outright. If any patterns with gaps are still discovered, these can simply be removed or split up using basic post-processing. We will go into more detail on these challenges in Chapter 8.

Most algorithms are not immediately suitable to run with this task’s input. For the MGDP algorithm, we need to supply an additional set of songs, which are used to check if patterns are specific to the tune family. For algorithms intended for single songs, we concatenate the songs as done by D.P. Boot [3]. This essentially stitches each song to the last, and treats the entire strung-together family as one song. This kind of approach ignores factors such as the positions of patterns within songs, but as this problem is delegated to another task, this approach should work fine. We will, however, opt to use our own solution, which is intentionally built with these requirements in mind. More details on handling the input and output are given for each specific solution in Chapter 8.

There are two ways for this task to become the bottleneck of the pipeline. The most straightforward way is that pattern discovery itself simply takes too long in terms of the input size, since the input has not changed in size in earlier tasks, this is directly based on the algorithm itself. The other way is the output size this task generates. If the number of patterns found is too large and abundant with uninteresting patterns, the final result will likely be worsened, and all subsequent steps will be significantly slowed down. This can be addressed by having a filter at the end of this task to remove obviously unsatisfactory patterns.

6.4 Pattern Clustering

The third major part of the pipeline deals with clustering the discovered patterns together. The goal here is to take the output from Section 6.3, and group similar patterns together in a smart way. The problem being solved here is that even within the flexibility of the discovery algorithms, pattern occurrences that belong together might get separated. The main factor here is that patterns may occur only in a small subset of the songs in a family, while slightly altered versions exist in a separate subset. These sets should be merged properly, but the selection of sets to merge should be done without requiring too much processing. A notable algorithm that already employs this type of process is SIARCT-CFP [4].

The challenges here are arguably the toughest to handle, outside of pattern discovery. The main challenge of this part is actually preventing the clustering from being a performance bottleneck. Clustering is the most likely task to take long in terms of its input size, and due to the input size
Figure 6.3: Two occurrences of what may be considered the same pattern. The black notes are part of the pattern, including the bottom note as embellishment in the right pattern. The grey notes are more of an extension to the pattern, and may be removed if it occurs rarely.

being the largest throughout the pipeline. If possible, for input size $n$, the performance should stay sub-quadratic, ideally near-linear. The second challenge is selecting the right patterns to group together, since similarity is not transitive. Finally, as third challenge, it is important to take the position of patterns within songs into account. Whether or not two similar patterns should be grouped together more often than not depends on whether they represent the same segment of their respective songs' structures.

Although this task is directly responsible for only one requirement, it does provide a great help in dealing with many of them. The requirement being addressed here is requirement FS03, making sure that patterns occur in the majority of the tune family. Applying clustering at all allows the discovery task to skip out on merging patterns together, as this task should make sure to combine occurrences that should be part of the same pattern.

As for requirements being supported by this task, many of pattern discovery algorithms go for exact repetitions of patterns, making requirements FI02, FI03 and FI04 hard to satisfy. Although the requirements would technically be satisfied, we would not be grouping the patterns together in similar batches according to the leniency provided by these requirements, worsening the end result. This task helps by grouping patterns together that are really the same according to our leniency together. This task can also help by already addressing the positioning factor of requirement FS05.1, position can be used as a criterion for clustering.

The output of this task is of the same format as the pattern discovery task. The major difference is that there are fewer patterns, but more occurrences per pattern. The output size remains identical, unless the clustering algorithm decides to discard outliers. The data size may also slightly grow, if patterns that form super sets of smaller patterns are assigned to multiple groups. Of course, its input is exactly that of the pattern discovery task described in Section 6.3.

6.4.1 Pattern Cropping

After clusters have been determined, it is likely the case that some pattern occurrences in the same cluster are significantly larger than the rest. In this case, often these occurrences form an excessive super-set of the occurrences they are grouped with. In the event that these large sets only occur in few songs of the family, they should be shrunk down. This focuses the pattern down to the core that actually occurs throughout the family, rather than having a few special cases with added bits that are not very representative. Furthermore, the pattern selection task, described in Section 6.5, will work better in most cases with the patterns properly cropped down. The requirement that is indirectly being helped here is requirement FS01. Removing superfluous extensions to patterns prevents the overlap we want to avoid, while sacrificing minimal important information.

This task faces one primary challenge. The pattern occurrences must be shortened to a meaningful degree, without being excessive. The pattern occurrences should still retain some variation, as otherwise the remaining patterns would likely become too small to be interesting. Part of this challenge is also to pick the right tuples to remove. Tuples that extend a pattern occurrence are to be removed, but not the tuples that simply embellish it. The performance is likely not an issue, as determining the cropped pattern occurrences can simply be done with two linear passes over the data, in most cases.
In our case specifically, we merge the cropping process into the clustering task. This allows us to cluster the parts of a large pattern with smaller patterns.

### 6.5 Pattern Selection

The final task before reversion occurs is the selection of patterns. This step has the primary function of filtering the output. The intention is partially to remove any uninteresting patterns or occurrences. The other part is that the patterns and occurrences are selected in order for the output to be a cohesive whole, rather than a jumbled mess of discovered patterns. Typically, as seen in the problem analysis in Chapter 4, we want the selected patterns to not overlap, and essentially have each important, recurring piece of each song represented by a single pattern.

Notable examples of an algorithm that already apply a selection step are COSIATEC [10], and in some sense, MGDP [7]. However, bases its selection primarily on compression qualities, in a lexicographic fashion. In order to do more suitable selection, a weighted sum of metrics indicating how interesting a pattern is should be in order. MGDP also somewhat performs selection during the pattern discovery itself, pruning its search tree as it goes. For this, it uses properties indicating how representative a pattern is for a tune family, as compared to some other set of songs. This could prove to be a useful metric, although it is likely better to embed into pattern discovery solutions.

Since pattern selection is a very heavy filtering step, it comes with the challenge of retaining just the right patterns and occurrences. When selecting a pattern, that pattern should persist as selected throughout the whole family, rather than having a variant pattern take over in some songs. This keeps the representation of the tune family consistent. The decision to select a certain pattern over others requires both looking at the pattern’s quality over the entire family, but also comparing that to the quality of the other patterns. Doing all this would require an excessive amount of resources, so we will likely have to use a fast heuristic approach to prevent blown up processing times. Selecting the right patterns under limited information may be both difficult to do well, and have great impact on the quality of the final results.

The requirements addressed in this last task mainly revolve around pattern quality and position. Requirement FS03 must be taken care of, by making sure the selected patterns do occur in enough songs. The non-overlap requirement FS01 gets covered here. The position-based requirements FS05.0 and FS05.0 must be fully satisfied by this task. In the case that no previous tasks make use of the non-matching tuples in the time range of occurrences, these tuples should still be added during this task, as per requirement FS03. Finally, if we can find the right approach, this task can strive towards satisfying requirement FS04, such that the patterns we end up with actually represent the tune family.

### 6.6 Conclusion

This pipeline approach allows us to flexibly fill in solutions to the individual problems delegated to its tasks. Aside from filling in the tasks with solutions to their problems, we also have to tailor a task selection to the desired quality-performance trade-offs. Various algorithms have already addressed some of the problems the pipeline needs to solve, which we can use to build solutions for each task of the pipeline.
Chapter 7

Family Normalization

In this chapter we analyze the problem of tune family normalization, and provide an attempt at solving it. Tune family normalization is the pre-process leveling out the playing field between songs in a tune family.

We first discuss the challenges of the process of normalization, including the requirements it has to account for. After laying out the challenges, we explain our attempt at solving the normalization problem.

7.1 Challenges

In this section we discuss all the challenges we must overcome to make a proper normalization of a tune family. We first discuss the normalization of pitch, then tempo on pairs of songs. Finally, we go over how to choose an anchor point to normalize to for an entire tune family, rather than pairs.

7.1.1 Pitch Normalization

Most pattern discovery algorithms find recurring patterns using relative pitches. Some algorithms, however, only detect patterns that are of the exact same pitch. These algorithms might be very effective at discovering patterns, if they are given a level playing field. For these algorithms we need pitch normalization to make sure the desirable patterns are actually detected. This aspect of normalization covers requirement F103 for those algorithms.

We might also not always desire that patterns of different pitch within a song are considered the same pattern. This may be due to preference on what patterns are desirable, or due to lowering the algorithmic complexity for discovery.

In order to keep a normalized song true to its original, we must shift the pitch of the entire song at once, rather than translating individual segments. An added advantage of this is that the problem of normalization becomes a lot simpler, as we only require a single offset for the pitches per song. Figure 7.1 shows a rough illustration of two songs being normalized to the same pitch.

Since we are trying to discover patterns, using such patterns as a frame of reference for a song’s pitch is natural. As such, we should perform pattern discovery on the pair of songs, and base our pitch difference on the patterns shared by both songs. Since pattern occurrences are internally consistent in their pitch, but not necessarily with respect to other occurrences, we need an appropriate way to filter the inconsistent occurrences.
The most reasonable approach to get the right pitch shift from the patterns is to look at all pairs of occurrences $O_1, O_2$ such that they are from different songs $O_1 \subset S_1 \land O_2 \subset S_2$, and take the most common pitch difference between the songs. This does of course require that any two occurrences have their notes mapped to each other one-to-one. Since not all occurrences are equal, we should also incorporate the size of the patterns as a weight to the calculation.

We call the pitch shift between two occurrences $O_1, O_2$ by the function $\text{Shift}(O_1, O_2)$. This function returns the shift $ps$ that must be applied to all notes in $O_1$ to be of the same pitches as the notes in $O_2$. Given a set of patterns $R$, and a weight function $\text{ShiftWeight}(O_1, O_2)$, we can then define the weight of a pitch shift $ps$ as follows:

$$\text{ShiftWeight}(ps, S_1, S_2) = \sum_{P \in R} \sum_{O_1, O_2 \in P \land O_1 \subset S_1 \land O_2 \subset S_2} \begin{cases} \text{ShiftWeight}(O_1, O_2) & \text{if } \text{Shift}(O_1, O_2) = ps \\ 0 & \text{otherwise} \end{cases}$$

The weight function $\text{ShiftWeight}(O_1, O_2)$ should be defined as either $|O_1|$ or $|O_1|^2$, as both occurrences contain an equal number of notes, and we can either value the number of consistent tuples, or consistent vectors between tuples. Another weight function could also be fine, but we try to ground our choice in the concepts of tuples or vectors. The shift with the greatest weight should be applied for normalization.

An important factor to consider is that a pair of songs might radically differ from each other. This may come in pitch, tempo, rhythm and embellishment, all at the same time. It is important that any algorithm we use to determine our frame of reference can at least handle pitch and tempo at the same time. Handling rhythm and embellishment as well would, however, give a better basis for normalization.

The reason we do not necessarily just use the discovery algorithm here directly is to narrow down the search space with the normalization first. This search space may still be large in this step, but in that case we can use a faster algorithm that takes less into account here, as the results are not as crucial.
7.1.2 Tempo Normalization

The normalization of tempo comes with the same problems as pitch normalization. This includes requiring an algorithm to perform both at the same time. This is especially important in making sure that the frame of reference for both normalizations is the same. Tempo normalization also has the factor that it is fully responsible for requirement \textit{FI05}. An illustration of tempo normalization is shown in Figure 7.2.

Tempo normalization specifically does bring its own specific problems. The first is the fact that tempo is a continuous factor, rather than a discrete one. The second is the factor of rhythmic variations, as these can differ between matching pattern occurrences, and as such provide even more difficulty pinning down the correct normalization.

Luckily, the same general solution as with pitch normalization still applies. One change is that we can no longer limit the search space to discrete values, which is an implementation problem. The second is that we are now looking at individual matching tuples. To denote that two tuples \(t_1, t_2\) are mapped to each other as the same tuple in their respective pattern occurrences, we use the boolean function \textit{MatchInPattern}(\(t_1, t_2\)). We use this to define the function \textit{GetMatchOnset}(\(t_1, O_1, O_2\)), which takes \(t_1\) from \(O_1\) returns onset \(t_2, o\) of \(t_2\) from \(O_2\) such that \textit{MatchInPattern}(\(t_1, t_2\)). This allows us to make a new function for the weight of a tempo stretch \(ts\) which applied to one song \(S_1\) makes it the same tempo as another song \(S_2\):

\[
\text{StretchWeight}(ts, S_1, S_2) = \sum_{P \in R_{O_1}} \sum_{O_2 \in P} \sum_{S_1 \subseteq O_1} \sum_{S_2 \subseteq O_2} \sum_{t_1 \in O_1} \sum_{t_1' \in O_1} \\
\begin{cases} 
1 & \text{if } (t_1, o - t_1', o) * ts = \text{GetMatchOnset}(t_1, O_1, O_2) - \text{GetMatchOnset}(t_1', O_1, O_2) \\
0 & \text{otherwise}
\end{cases}
\]

This function basically computes the number of matching distances between any pair of pairs of tuples across any two occurrences, given that the tempo stretch is applied. This should give a good idea of what tempo stretch is most likely to be correct.

This is, of course, a quadratic number of additions in terms of each pattern occurrence. In order to save on processing time, only sequential pairs of tuples can be considered, at the expense
Again, we cannot properly apply tempo normalization without pitch normalization. Figure 7.3 illustrates the combination of the two normalizations being applied.

### 7.1.3 Choosing an Anchor Point

The final challenge to overcome is choosing the right song as an anchor point for the normalization. Since we must normalize for a whole tune family, we need a song that is representative of the correct tempo and pitch to use for pattern discovery. For pitch, we simply need a song that is consistent, as no parameters depend on having a certain pitch. The tempo, however, does need to anchor to the right point, as the gap size we want to allow within patterns would lead to different results depending on how we stretch the time domain.

In our requirements FI01 and FI05, we essentially used the fastest song as anchor point for any pair of songs. Since the comparison here is pairwise, do not strictly take the fastest song in the family as anchor for most occurrences. Rather, we take the fastest song of each pair. This means the most common tempo we want to take as reference is a combination of what tempo is common and what tempo is fast. Additionally, if we have one song faster than the rest, and we simply took it as anchor point, the whole family would be skewed towards that song’s tempo. For these two reasons, the better choice would be to simply settle for the most common tempo in the tune family.

In order to find the right song, we can simply determine all pair-wise normalizations, and pick out the set of songs that have the most pairs without a need for tempo normalization. From this set, we need to find the most reliable song to use as anchor. We can do this by counting the number of paths of length 3 we can take through the graph of pair-wise normalizations, which loop back to the same song with all the normalizations in the path adding up to 0. This means we would normalize each song \( S_0 \) to the first song in the path \( S_1 \), then from the frame of reference of \( S_1 \) translate it to \( S_2 \) in the path, and finally from \( S_2 \) to \( S_0 \). If the normalizations of \( S_0 \) are consistent, then the only possible point of incorrectness can stem from the normalization between \( S_1 \) and \( S_2 \). By going over all possible paths, the normalization errors for \( S_0 \) subtract quadratically from the total count, while the mistakes in \( S_1 \) and \( S_2 \) would only count linearly. This should give the highest total count the best consistency, making the song with that count the logical anchor.
CHAPTER 7. FAMILY NORMALIZATION

7.2 Solution

In this section we will discuss how we solve the normalization problem. We first introduce the algorithm we use, then we defend its performance. Finally, we briefly go over the normalization reversion solution.

7.2.1 Algorithm

In order to solve the normalization problem, we construct the Normalization Algorithm 1 from the steps discussed in Section 7.1. It goes through the steps of computing patterns and extracting how the tempo and pitch changes from song to song according to these patterns. The algorithm then determines which song has the most reliable normalizations towards the other songs, and applies normalizations to match the pitches and tempos of the family to this song. Since all pair-wise data is symmetrical, the algorithm may be implemented with skipping of symmetrical pairs, as the data needed is already there with one of the two.

Algorithm 1: Normalization

\[
\text{Input : } \text{Tune Family } F, \mathbb{R} ss, \mathbb{R} gs \\
\text{Output: } \text{Tune Family } F, \text{Mapping}(\text{Song } \rightarrow \mathbb{Z}) \text{ pitchChangeMap}, \text{Mapping}(\text{Song } \rightarrow \mathbb{R}) \text{ tempoChangeMap} \\
\text{Result: } \text{Normalizes } F, \text{ with tempo at most divided or multiplied by } ss, \text{ and storing the normalizations in pitchChangeMap and tempoChangeMap.}
\]

1 \text{ begin} \\
1.1 // Discover non-embellished patterns, using } ss \text{ as rhythmic variation ratio parameter.} \\
1.2 \text{ Pattern Set } R \leftarrow \text{Discovery}(F, ss, gs); \\
1.3 \text{ Mapping}((\text{Song, Song } \rightarrow \mathbb{Z}) \rightarrow \mathbb{N}) \text{ pitchWeights, Mapping}((\text{Song, Song } \rightarrow \mathbb{R}) \rightarrow \mathbb{N}) \text{ tempoWeights} \leftarrow \text{ComputeNormalizationWeights}(F, R); \\
1.4 \text{ Mapping}((\text{Song, Song } \rightarrow \text{Integer}) \text{ pitchPairMap, Mapping}((\text{Song, Song } \rightarrow \text{Double}) \text{ tempoPairMap} \leftarrow \text{DeterminePairWiseNormalization}(F, \text{pitchWeights, tempoWeights}); \\
1.5 \text{ Song } S_0 \leftarrow \text{DetermineAnchor(Tune Family } F, \text{pitchPairMap, tempoPairMap}); \\
1.6 \text{ Mapping}((\text{Song } \rightarrow \text{Integer}) \text{ pitchChangeMap, Mapping}((\text{Song } \rightarrow \text{Double}) \text{ tempoChangeMap} \leftarrow \text{ComputeChangeMaps}(F, \text{pitchPairMap, tempoPairMap, } S_0); \\
1.7 \text{// Apply the normalization to the tune family.} \\
1.8 F \leftarrow \text{ApplyNormalization}(F, \text{pitchChangeMap, tempoChangeMap});
\]

The first step in this algorithm is determining a pattern set to compute from. We leave this to our discovery algorithm, which we will elaborate on in Chapter 8. There are a few important properties the discovery algorithm used here must have. One is that it has to discover patterns with rhythmic variation that allow for one occurrence to be entirely scaled in the time-domain compared to another occurrence in the same pattern. This allows us to use our maximum stretch parameter $ss$ as input for the rhythmic variation, giving us all the patterns that are stretched as such. The discovery algorithm should of course also discover patterns with differing pitch among the occurrences. It is also required that the patterns returned are not embellished with additional notes, such that we can match individual notes between occurrences for our computations. Finally, it is highly desirable that the patterns returned only consist of two occurrences each. Our discovery algorithm conveniently does this, making computations in the rest of the normalization more straightforward in performance. We use the gap size parameter $gs$ in our normalization purely to reduce the search space in the discovery step.
In the second computation on line 3, we use a weight computation subroutine, described in Algorithm 2 to determine what translations and stretches are frequent among the patterns. To do this, we simply check each pattern for how its occurrences translate and stretch from one another. For each translation or stretch we encounter for a pair of songs, we keep track of how much we have encountered it. In the case of pitches, we simply add the size of the pattern occurrences to the weight. For tempo, we count each individual vector between subsequent tuples, as rhythmic variation might throw off our computations if only take the vector from the start to the end of each occurrence.

**Algorithm 2**: ComputeNormalizationWeights

**Input**: Tune Family $F$, Pattern Set $R$

**Output**: Mapping((Song, Song, Z) → N) pitchWeights, Mapping((Song, Song, R) → N) tempoWeights

**Result**: Computes for each pair of songs how prominently each existing normalization applies, stored in the weight maps.

1 begin
   // Initialize the normalization weight maps, every entry starts at 0.
   2 Mapping((Song, Song, Z) → N) pitchWeights ← 0;  
   3 Mapping((Song, Song, R) → N) tempoWeights ← 0;     
   // Fill the normalization weight maps.
   4 for each pair of occurrences $O_1, O_2$ of the same pattern $P$ in $R$ do
      5 Increment pitchWeights(Song($O_1$), Song($O_2$), $O_2[0].m - O_1[0].m$) by $|O_1|$;
      6 for i with 0 < i < $|O_1|$ do
         7 Increment tempoWeights(Song($O_1$), Song($O_2$), $(O_2[i - 1] - O_2[i])/ (O_1[i - 1] - O_1[i])$);

In the third computation on line 4, we take the weights of all the normalizations, and determine how songs should translate and stretch to one another, using Algorithm 3. This step boils down the weighted information we computed previously by taking the most highly weighted translation and stretch for each pair of songs as the correct one. Due to the nature of the weights, we are now likely to have an accurate map of how to normalize any song to the pitch and tempo of another song.

**Algorithm 3**: DeterminePairWiseNormalizations

**Input**: Tune Family $F$, Mapping((Song, Song, Z) → N) pitchWeights,  
          Mapping((Song, Song, R) → N) tempoWeights

**Output**: Mapping((Song, Song) → Integer) pitchPairMap, Mapping((Song, Song) → Double) tempoPairMap

**Result**: Puts the most highly weighted normalizations from the weight maps into the pair maps.

1 begin
   // Initialize the pair-wise normalization mappings.
   2 Mapping((Song, Song) → Integer) pitchPairMap ← ∅;
   3 Mapping((Song, Song) → Double) tempoPairMap ← ∅;
   // Fill the pair-wise normalization mappings.
   4 for each pair of songs $S_1, S_2$ in $F$ do
      5 Put the maximum weight over all pitchWeights($S_1, S_2, x$) for any $x$ in pitchPairMap($S_1, S_2$);
      6 Put the maximum weight over all tempoWeights($S_1, S_2, x$) for any $x$ in tempoPairMap($S_1, S_2$);
In the fourth computation on line 5, we use our complete mapping of normalizations to determine which song has the most suitable pitch and tempo to work with, which is done by Algorithm 4. This algorithm looks for the largest set of songs with the same tempo, as this tempo is most representative of the tune family. From this set it computes which song is most likely to have reliable normalizations in our complete mapping. We do this as discussed in Section 7.1, by applying the normalizations in a short loop and counting the number of times the resulting pitch and tempo was the same as the original. The song that comes out with the highest count here is taken as our anchor point.

Algorithm 4: DetermineAnchor

| Input : Tune Family $F$, Mapping($(Song, Song) \to \text{Integer})$ pitchPairMap, Mapping($(Song, Song) \to \text{Double})$ tempoPairMap |
| Output: Song $S_0$ |
| Result: Determines the song $S_0$ most suited as an anchor point for the family to normalize to, such that $S_0$ has the most consistent normalizations among the songs with the most common tempo. |

1 begin  
  // Determine the set of songs with the most common tempo, as potential anchor points.  
  Set(Song) commonSongs $\leftarrow \emptyset$;  
  Add to commonSongs all songs in $F$ with the maximum number of 1 entries in tempoPairMap;  
  // Determine the best song as anchor point.  
  Determine the Song $S_0$ in commonSongs with the maximum number of unique pairs $S_1, S_2$ in $F$ satisfying $\text{pitchPairMap}(S_0, S_1) + \text{pitchPairMap}(S_1, S_2) + \text{pitchPairMap}(S_2, S_0) = 0$ and $\text{tempoPairMap}(S_0, S_1) \cdot \text{tempoPairMap}(S_1, S_2) \cdot \text{tempoPairMap}(S_2, S_0) = 1$;  

On line 6, we apply our fifth sub routine, shown in Algorithm 5, which takes our complete normalization map and anchor point, and reduces the data down to one normalization per song. This is a straightforward process of taking the normalizations from each song to our anchor point.

Algorithm 5: ComputeChangeMaps

| Input : Tune Family $F$, Mapping($(Song, Song) \to \text{Integer})$ pitchPairMap, Mapping($(Song, Song) \to \text{Double})$ tempoPairMap, Song $S_0$ |
| Output: Mapping(Song $\to \mathbb{Z}$) pitchChangeMap, Mapping(Song $\to \mathbb{R}$) tempoChangeMap |
| Result: The pitchChangeMap and tempoChangeMap are filled with the normalizations for each song in $F$ that make them match the pitch and tempo of $S_0$, using the data in pitchPairMap and tempoPairMap. |

1 begin  
  // Initialize the normalization mappings.  
  Mapping(Song $\to$ Integer) pitchChangeMap $\leftarrow \emptyset$;  
  Mapping(Song $\to$ Double) tempoChangeMap $\leftarrow \emptyset$;  
  // Determine the normalization mappings, based on anchor $S_0$.  
  for each song $S$ in $F$ do  
    pitchChangeMap($S$) $\leftarrow$ pitchPairMap($S, S_0$);  
    tempoChangeMap($S$) $\leftarrow$ tempoPairMap($S, S_0$);  

Finally, our last step is to apply the normalizations we have determined, using Algorithm 6. This process is a matter of going through all songs and changing the onsets, durations and morphetic pitches of all tuples according to the map we have computed. This should make it so that if
we were to discover patterns again, most discovered patterns would have occurrences consistent in their pitch and tempo.

Algorithm 6: ApplyNormalization

<table>
<thead>
<tr>
<th>Input</th>
<th>Tune Family $F$, Mapping(\text{Song} \rightarrow \mathbb{Z}) \text{pitchChangeMap}, Mapping(\text{Song} \rightarrow \mathbb{R}) \text{tempoChangeMap}</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>Tune Family $F$</td>
</tr>
<tr>
<td>Result</td>
<td>Normalizes $F$ using pitchChangeMap and tempoChangeMap.</td>
</tr>
</tbody>
</table>

\begin{verbatim}
begin
  for each song $S$ in $F$ do
    for each tuple $t$ in $S$ do
      $t \leftarrow (t.o * \text{tempoChangeMap}(S), t.c, t.m + \text{pitchChangeMap}(S), t.d * \text{tempoChangeMap}(S), t.v);$
\end{verbatim}

7.2.2 Performance

In this section we discuss how the performance of our normalization algorithm is within $O(\text{Size}(F)^3)$, given that the discovery algorithm is as well.

As our first computation is the discovery algorithm itself, we will discuss its performance in Section 8.3.2. The only different factor in applying it here is that we use the stretch parameter $ss$ for rhythmic variation. Since both stretch and rhythmic variation carry similar range for similar purposes, we will assume that these will assume the same values and hence result in similar performance. The most common value to be used here would be $1/2$, as songs are not typically changed to a time signature more than double or less than half of the original. The same goes for rhythmic variation, unless very radical changes are involved, $1/2$ should be sufficient.

The second computation by Algorithm 2, assuming each pattern has only 2 occurrences, should be linear in the size of the pattern set. This would make sure that it cannot be slower than the discovery algorithm. With 2 occurrences per pattern, we loop over each occurrence exactly once, and compute its weight in linear time. What remains is that we have to do a look up and a potential insertion on the mappings. Since computing and maintaining the pitch weights is strictly faster than doing the same for tempo, we skip over the pitch weights. The tempo weights are computed a linear number of times over the size of the pattern set, as each pair of subsequent tuples is considered. These computations require look ups and insertions, and the cost for these should be at most $\log n$, with $n$ being the size of the mapping, which is at most linear in the size of the pattern set. Due to this, we require that the size of our pattern set is at most $O(\text{Size}(F)^3 / \log \text{Size}(F)^3)$.

We discuss this is the case for the pattern discovery in Section 8.2.5 and Section 8.3.2.

The third computation, done by Algorithm 3, is strictly faster than the previous. We are looping over pairs of songs once more, but in doing so only looking at the mappings relevant to the pair being considered each time, evaluating to one look up per entry in the weight mappings. This does require that the mappings are implemented such that we can fetch all entries satisfying a subset of the look up parameters. This is easily accomplished by nesting the mappings.

Algorithm 4, executing our next computation, does not perform any particularly resource intensive tasks. One task involves counting the number of specific entries in the pair-wise mapping of tempo stretches we have determined. This task requires going over each pair a constant number of times, making it only quadratic in the number of songs, times the look up time. The bigger task is performing a constant time computation on every possible set of 3 songs, making it cubic in the number of songs, again multiplied by look up time. Unless the songs have extremely few notes to them, cubic in number of songs stays far enough below cubic in number of tuples. We can safely assume this, as such short songs would render the entire algorithmic goal pointless, and significantly reduce the search space for pattern discovery.
Our last two sub routines, Algorithm 5 and Algorithm 6 are insignificant in resource usage compared to the rest. The former only has to retrieve and insert a number of items linear in the number of songs in the tune family. The latter performs look ups on the created linear-size mapping, to perform a linear number of computations on the tune family.

7.2.3 Reversion

The Reversion Algorithm 7 uses the preserved normalization data to undo the changes done by the normalization. Rather than applying to a tune family like the Normalization Application Algorithm 6, this applies to the discovered pattern set. The operations done to all tuples are the exact inverse of what the normalization originally applied to the tune family, making the discovered patterns match the original tune family.

Algorithm 7: RevertNormalization

Input : Pattern Set $R$, Mapping(Song $\rightarrow \mathbb{Z}$) pitchChangeMap, Mapping(Song $\rightarrow \mathbb{R}$) tempoChangeMap

Output: Tune Family $F$

Result: Reverses normalization on $R$ using pitchChangeMap and tempoChangeMap.

```
1 begin
2   for each pattern $P$ in $R$ do
3     for each occurrence $O$ in $P$ do
4       for each tuple $t$ in $O$ do
5         $t \leftarrow (t.o/\text{tempoChangeMap(Song}(O))$, $t.c$, $t.m -$
6           $\text{pitchChangeMap(S,} t.d/\text{tempoChangeMap(Song}(O))$, $t.v)$;
```
Chapter 8

Family Pattern Discovery

In this chapter, we discuss solving the pattern discovery problem in the context of our pipeline. We first highlight the ways in which our pipeline helps the pattern discovery task. Then, we go over the challenges brought by the requirements delegated to this task. Finally, we give our solution to the problem in the form of an algorithm.

8.1 Helping Factors

While pattern discovery is the core task of the entire pipeline, having the other tasks present allows us to approach the discovery problem more freely.

Having normalization allows discovery to ignore a larger portion of potential patterns. Most notably, the rhythmic variation requirement significantly increases how broad we need to search for patterns, this possibility space is made smaller by tempo normalization. We also get the option to consider transposed patterns within the same song as separate patterns, which would also shrink the possibility space.

Having clustering gives us two very important advantages for discovery. Since all similar patterns will end up together anyway, we can simply restrict our search to only patterns with 2 occurrences. This, by extension, also allows us to search for patterns in pairs of songs at a time, lowering the required complexity of our algorithm. Applying cropping after or during clustering also allows us to always take the largest valid version of a pattern we can find. Only when clustering patterns together do we have reason to take subsets such that the entire cluster is consistent.

Finally, pattern selection simply allows us to ignore pattern quality and pattern position while searching for them. We can, however, still choose to use these factors to shorten our searches.

8.2 Challenges

In this section we go over the individual challenges the requirements give us. For each of these challenges, we also touch on the performance aspects involved, as processing time should satisfy $O(\text{Size}(F)^3)$ for an input tune family $F$. First we discuss discovering transposed patterns. We will then take into examine the implications of taking into account that patterns may have additional notes in between. This is followed by how to deal with, and make use of, the fact that patterns may not have large time gaps between tuples. Finally, we address the two most difficult problems, those of allowing rhythmic variations and keeping the output size low.

8.2.1 Transpositions

The problem of transposition, requirement $FI04$, is a fairly straightforward one to solve. For geometric discovery algorithms it is simply a matter of finding patterns of the same shape, regardless of their position on the pitch axis. For sequential discovery algorithms this becomes a bit more
CHAPTER 8. FAMILY PATTERN DISCOVERY

involved. In this case, rather than looking at the songs as a sequence of tuples representing notes, it must instead be a sequence of pitch differences between the subsequent notes. This does, however, make embellishments harder to deal with again, as a one note between two others in the song completely changes the differences in the sequence. This is one of the reasons we will stick to using a geometric approach.

An important factor to take note of with transpositions is that allowing them means that every pattern occurrence of the same shape becomes a candidate for the same pattern. It is possible to restrict transpositions partially or entirely, if we ever need a way to improve performance at the cost of completeness of our discovery. Even if we restrict transposition, we should still do an unrestricted run for the normalization task’s use of discovery, to make sure we are not missing complete songs worth of patterns.

8.2.2 Embellishments

The problem of embellishments, requirement FI04, is similar to the problem of transpositions. Geometric discovery algorithms handle the extra tuples easily, as these algorithms see the tuples as points in a plane, freely allowing them to ignore tuples. With sequential discovery algorithms this is a problem again, as now we need to consider that any pair of pattern occurrences may have a number of tuples in the middle of the sequence, which have to be skipped over intentionally. This quickly diminishes the performance advantage some sequence-based algorithms might have. As discussed in Section 8.2.1, the combination with transposition makes this especially difficult to handle. This is our second reason for taking a geometric approach.

Since the clustering task allows us to limit discovery to pairs of occurrences, and the combination with cropping encourages taking the largest common subset, we should not only find the longest matching patterns, but also the most matches within the embellishments. We specifically choose to incorporate all matching embellishments, while ignoring the non-matching ones, rather than sticking to requirement FS02 already and including all embellishments. We do this to give the clustering task a clear handle on what tuples match from one occurrence to the other. It is also rather easy to maintain or look up the tuples we skipped for each pattern occurrence. Including the remaining embellishments in the actual patterns is up to the selection task.

8.2.3 Gap Size

Adhering to a maximum gap size in patterns, requirement FI01 is the easiest challenge to overcome. Since we decided to find the largest valid version of each pattern, we can actually ignore gaps entirely. We can then take the large patterns and split them up at the points where the gaps are too large.

The real challenge here is not to adhere to the gap size, but to use it in limiting the search space. We can do this by only comparing tuples up to the tuples through their own properties and only the properties of tuples within the next gap size $gs$ worth of time. The rest of the information can be ignored, as there would have to be a tuple within $gs$ in the pattern for a tuple outside of $gs$ to be allowed in. The information before a tuple $t$ can be ignored, since preceding tuples $t'$ that would be of the same pattern will look ahead and find $t$.

8.2.4 Rhythmic Variation

The rhythmic variation allowed by requirement FI02 is the cause for the most potential drop in performance and increase in output size. Here, we briefly discuss the process of allowing for rhythmic variations, searching for them efficiently, and keeping the output from becoming too large from it.

The process of constructing patterns with rhythmic variation is simple on its own. If we use some form of constructing patterns tuple by tuple, all we have to do is relax the conditions for a new tuple to be added to the pattern. This should only preserve the order of the pattern, and stick within the rhythmic variation window for all notes in the pattern. Due to mathematical
construction of requirement \textit{FI02}, the set of tuples \( t \) allowed by one tuple \( t' \) in the pattern is a strict subset of the tuples allowed by a tuple \( t'' \) closer in onset to \( t \). This allows us to only look at the immediately surrounding tuples of a potential addition to a pattern.

The performance of constructing patterns with rhythmic variation is hampered by related factors. First, we need to consider strictly more tuples to add for each pattern, and vice versa. The second factor is that we get more and larger patterns than would be allowed without rhythmic variation, which of course take longer to construct.

The output size becomes much larger with rhythmic variation, as stated by having more and larger patterns. We also have to deal with there being multiple permutations of patterns. Adding one tuple \( t \) to a pattern tightens the conditions for nearby tuples to be added. As such, another tuple \( t' \) might not be allowed anymore, making for two versions of the pattern.

In order to keep processing time short, we can take a measure to filter which tuples we compare to one-another. This is a matter of finding out for each tuple \( t_1 \) what tuples \( t_2 \) share tuples \( t'_1, t'_2 \) nearby with the same pitch distance and time distance in each other’s rhythmic range. This can be done without having to check every set of 4 tuples individually.

The output, as far as rhythmic variation goes, can also be kept small by sacrificing pattern variations. When building patterns and coming across multiple potential additions to the pattern that are incompatible with each-other, we can choose one through a heuristic.

The main trade off here is between tuples close to the existing tuples in the pattern, and those far away. Far away tuples mainly provide a farther starting point for the maximum gap size to get subsequent tuples from. Close by tuples share a relatively large space with their window for rhythmic variation with the existing pattern. Since large gaps within the maximum gap size are relatively less desirable, a good heuristic to take would be to go with the tuple closest to the pattern.

Alternative heuristics are more concerned with rhythmic consistency. One is to always take the tuple providing the least rhythmic variation between the two occurrences of a pattern. This would favour the detection of patterns that are rhythmically the most similar. The other alternative is to keep the rhythmic variation between the occurrences consistent. This would favour the detection of stretched patterns.

\subsection{Output Size and Performance}

As per requirement \textit{FU05}, the processing time of this task should be bounded by \( O(\text{Size}(F)^3) \). Since this required also applies to the normalization and clustering tasks, which build on the output of this task. Hence, the output size should be bounded lower still. To give our clustering some room to work with, we are best off at least limiting our output size to \( O(\text{Size}(F)^2) \).

If we want to sacrifice finding potentially important patterns, we can limit the number of tuples within the maximum gap size or rhythmic variation range to some constant. Doing so in particular would give us a clear handle on the maximum processing time.

The number of tuples within the maximum gap size is directly linked to the tuple density of the songs on their time domain. The density typically does not vary more than a factor 4. Lowering the maximum gap size or taking a maximum gap size based on the tuple density are valid approaches here. If we want to consider at most a constant number of tuples, we should stick to taking the closest tuples as a heuristic, as done in SIAR \cite{5}.

For rhythmic variation we can do something similar, in only taking the least rhythmically varied tuples. This goes well with the heuristic of always taking the least rhythmically varied tuples when there are multiple options. This is simply a matter of dynamically scaling down the rhythmic variation parameter \( rr \) to enforce the desired number of considered tuples.

\subsection{Solution}

In this section we discuss our solution to the discovery problem, given the requirements we must adhere to and the help from the surrounding pipeline. We go over our algorithm, which is heavily
CHAPTER 8. FAMILY PATTERN DISCOVERY

8.3.1 Algorithm

We made our discovery algorithm out of two major components, split up over three subroutines. In this section, we will explain the top level algorithm and each subroutine. The first two routines make up the pre-process to set up the search environment, while the last uses it to construct patterns.

In the routine, algorithm 8, we first collect all the tuple pairs that are up to our maximum gap size apart. Then, we go over each pair of songs, and use the tuple pairs to determine what tuples from the different songs share a similar tuple pair, making them a potential match for in patterns. Finally, we build patterns from the ground up, using only the potentially matching tuples as search space. We make sure to include a run of each song with itself, to detect pattern occurrences within the same song, which will help later with our clustering algorithm, explained in Section 9.2.1. At the end of this process, we filter out the patterns with less than minSize length, which we take to be 3, as patterns of length 2 occur extremely often and carry little significance.

Algorithm 8: Discovery

\begin{itemize}
\item \textbf{Input :} Tune Family $F$, $R$, $rr$, $R$, $gs$, $N$, maxEmbellish, $N$, patsPerTuple, $N$, minSize
\item \textbf{Output :} Pattern Set $R$
\item \textbf{Result:} Discovers largest versions of patterns in $F$ of at least size $minSize$ as pairs of occurrences, such that the time distance between subsequent tuples is at most $gs$ and the time between tuples varies at most a ratio $rr$ between occurrences.
\end{itemize}

\begin{algorithm}
\begin{algorithmic}
\State Pattern Set $R \leftarrow \emptyset$;
\State Mapping$(Song \rightarrow Mapping(Z \rightarrow Set((Tuple,Tuple))))$ shortVectors $\leftarrow$
\State GetShortVectors$(F, gs, patsPerTuple)$;
\For{each song $S_i$ in $F$, with $0 \leq i < |F|$}
\For{each song $S_j$ in $F$, with $i \leq j < |F|$}
\State Mapping$(Z \rightarrow Set((Tuple,Tuple)))$ tupleMatches $\leftarrow$
\State GetTupleMatches$(shortVectors(S_i), shortVectors(S_j), rr)$;
\State Add all patterns from GetPatterns$(S_i, S_j,$ tupleMatches, $rr, gs,$ patsPerTuple) to $R$;
\EndFor
\EndFor
\State Remove patterns with occurrences with less than minSize tuples;
\end{algorithmic}
\end{algorithm}

In the first sub-routine, getting the vectors within the maximum gap size, is shown in algorithm 9. Here we get all the vectors, denoted as pairs of tuples from the same song, such that the onset of the first tuple is at most our maximum gap size $gs$ less than the onset of the second. Each such vector is a potential subset of a pattern occurrence. Since every vector is at most time-length $gs$, the tuples of such vectors can be valid subsequent tuples in a pattern while satisfying the gaplessness condition. We also limit this to a maximum number of vectors from each tuple, given by maxEmbellish. This enforces that we are less likely to get patterns with many non-matching
tuples between subsequent matching tuples. Note that due to our pattern construction at the end of the main process, this does not enforce that heavily embellished patterns are prevented entirely.

We collect these vectors by going through the tuples in ascending order, and for each tuple looking ahead until a tuple is more than $gs$ away from it. We use mappings to segregate the sets, lowering the amount of time spent searching and clearly distinguishing between unrelated vectors. Figure 8.1 shows this sub-routine in the middle of its process. All the points are tuples and all the arrows are vectors between these tuples we are collecting. The grey area denotes the distance $gs$ from the tuple currently being processed. This tuple at the start of the grey area, and gets the vectors to other tuples in this range. We also show that the fourth tuple in this image does not get a vector to the seventh, as this would exceed the maxEmbellish parameter given for this particular instance.

**Algorithm 9: GetShortVectors**

```
begin
    shortVectors ← ∅;
    Sort $S$ on ascending order of tuple onset;
    for each song $S$ in $F$ do
        for each tuple $t_i$ in $S$ do
            N_j ← i + 1;
            while $t_j.o - t_i.o \leq gs$ and $j < |S|$ and $j - i \leq \text{maxEmbellish} + 1$ do
                Add $(t_i, t_j)$ to shortVectors($t_j.m - t_i.m$);
            Increment $j$;
end
```

In the second sub-routine of the pre-process, shown in algorithm 10, we find all the tuples that have sufficiently similar vectors for them to be matching tuples in the same pattern. We take the vectors from two songs, created by the previous sub routines and separately loop over the vectors for each possible pitch-distance. We sort each of the two vector sets by how long they are on the time domain. We then loop over the vectors $v_1$ of one song, and maintain the vectors $v_2$ of the other song that are within the rhythmic variation range of it. Each vector $v_2$ within the rhythmic variation range has its first tuple matched with that of $v_1$, and the same for the second, which are added to the set of matching tuple pairs.

Since every vector was essentially a length 2 pattern occurrence, and the matched vectors satisfy the conditions of being another occurrence for the same pattern, their individual tuples are candidates for being matching pairs in a pattern. Since the vectors are also all relative, we ignore the base pitch altogether, catching transposed matches as well. Finally, since each vector
may skip over notes in the song, this also works with embellishments.

In Figure 8.2 we show one sequence of vectors with consistent pitch distance being used to construct the tuple matches. The dashed lines indicate pointers between matching tuples. The grey area contains all the vectors that are in the same rhythmic range as the bolder vector at the start of it, which is being processed here. The vectors farther right in the image have not yet been processed.

We want to point out that it may be worth doing more with this sub-routine for forming patterns. Some tuples may be matched multiple times, and some matches may have more consistent rhythm. We can store this information without increasing the processing time, and use it as a scoring heuristic. This scoring heuristic could then be applied to find patterns more efficiently, or discern good and bad patterns.

Algorithm 10: GetTupleMatches

Input : Mapping($Z \rightarrow \text{Set}((\text{Tuple}, \text{Tuple})))$ vectors$_1$, Mapping($Z \rightarrow \text{Set}((\text{Tuple}, \text{Tuple})))$ vectors$_2$, $r r$

Output: Mapping($Z \rightarrow \text{Set}((\text{Tuple}, \text{Tuple})))$ tupleMatches

Result: Finds all pairs of tuples with one tuple from vectors$_1$ and one tuple from vectors$_2$, such that they both have a tuple of the same relative pitch and a time distance within $rr$ from one another.

1 begin
2 Mapping($Z \rightarrow \text{Set}((\text{Tuple}, \text{Tuple})))$ tupleMatches ← ∅;
3 for each key $k$ in vectors$_1$ do
4 // Sort the vectors on their length in the time-domain.
5 // VectorTimeLength of a pair of tuples $t_1, t_2$ returns $t_2.o - t_1.o$.
6 Sort vectors$_1(k)$ and vectors$_2(k)$ on vectorTimeLength(vectors$_1(k)[i]$) for elements $i$, in ascending order;
7 Set((Tuple,Tuple)) status ← ∅;
8 for $N i$ from 0 to $|vectors_1|$ do
9 // Update the rhythmic variation range.
10 Add tuple pairs $p$ from vectors$_2(k)$ to status that satisfy
11 vectorTimeLength(vectors$_1(k)[i]) \geq \text{vectorTimeLength}(p)/rr$;
12 Remove tuple pairs $p$ from status that satisfy
13 vectorTimeLength(vectors$_1(k)[i]) > \text{vectorTimeLength}(p) * rr$;
14 // Add all tuples with a similar enough vectors to tuple matches.
15 Add (vectors$_1(k)[i][0], p[0])$ and (vectors$_1(k)[i][1], p[1])$ to
16 tupleMatches($p[0].m - \text{vectors$_1(k)[i][0].m}$) for all tuple pairs $p$ in status;

The final sub-routine of our algorithm, algorithm 11, takes care of actually discovering the pairwise patterns. For a pair of songs, we go through the matching tuples from the start of one of the songs. We then check for each tuple match $(t, t')$ if it fits into any patterns $P$ according to the last pair of matching tuples added to $P$. If there is no such pattern, we create a new pattern $P'$, starting with the occurrences $\{t\}$ and $\{t'\}$. To enforce the maximum gap size $gs$, we also start a new pattern $P'$ if the additional tuple match goes outside of $gs$ from the pattern $P$ it would otherwise add to. Pattern $P$ can then no longer gain additions within $gs$, so it is removed from the set of possible extendable patterns, and added to the finished patterns in $R$. Any patterns not broken off this way will be added to $R$ at the end of the process.

It is important to note that this way of constructing patterns is a heuristic. It is possible to abandon the construction of a pattern prematurely, if an added tuple $t$ prevents the addition of a tuple $t'$, which would be required to accept the next tuple $t''$. This results in two patterns with slight overlap, in which one can be the continuation of another. It should be fairly easy to change the heuristic being used, by for instance sorting the matching tuples in a different order.

In Figure 8.3 we see a visual representation of a pattern being constructed. The dashed
arrows denote tuple matches previously computed, and the grey bounding boxes denote the two occurrences of the pattern. Note how the tuple matches do not necessarily match exactly, as the construction allows for rhythmic variation. The tuple match currently being added in the image is the bold arrow and points at the right. This match is outside of \( gs \) distance, and hence the previously constructed pattern is completed.

In order to prevent having excess patterns within the exact same time range, we also limit our construction to a number of longest constructed patterns per endpoint. Whenever we extend a set of patterns with a new pair of tuples, we keep a number of longest patterns constructed, given by \( \text{patsPerTuple} \), which we have as 3 by default. This shrinks the output while keeping the most important patterns around. Note that this is not a flawless approach, as patterns may share the same end points, but be otherwise entirely different. For these cases, it is worth allowing a few more patterns than our default.

8.3.2 Performance and Output Size

The performance of our algorithm is largely based on the output of each part of the pre-process. As such, we will discuss both subjects in this section.

In algorithm 9, we loop over each song \( S \) in the family, and for each tuple \( t \) look ahead for a number of tuples \( t' \) based on the maximum gap size \( gs \). The number of tuples within the maximum gap size \( gs \) should typically be at most \( 4gs \), as songs do not tend to use faster notes than 16th notes. If we place an additional constant limit, with \( \text{maxEmbellish} \), on the number of tuples we look at within \( gs \), we can reason about an lower processing time and output size than otherwise for this sub-routine.

For the performance, we sort the songs in a total of \( O(\text{Size}(F) \times \log(\text{Size}(F))) \) time. We then loop over all tuples, which in turn may look at up to \( \text{Size}(F) \) tuples each, or this can be limited by \( c \). This causes a processing time of \( O(\text{Size}(F) \times \log(\text{Size}(F))) + \text{Size}(F) \times \text{maxEmbellish}) \) or \( O(\text{Size}(F)^2) \) pessimistically. The same number applies to the output size, as the operation in the nested is the addition of a tuple pair for the output.

Since the next sub-routines are done on all pairs of songs, we get that the worst-case performance is at least \( O(\text{Size}(F)^2) \). This is due to getting the situation of all songs having 1 note. Then there is the inverse situation, where the family contains one song. In this case, the worst-case running time is that of the sub-routines themselves. For these subroutines, we will reason in terms of the performance for the entire family, as opposed to the single songs, since that is what the worst case comes down to.
Algorithm 11: GetPatterns

**Input**: Song $S_i$, Song $S_j$, Mapping($\mathbb{Z} \rightarrow \text{Set}((\text{Tuple}, \text{Tuple}))$) tupleMatches, $\mathbb{R}_{rr}$, $\mathbb{R}_{gs}$, $\mathbb{N}$ patsPerTuple

**Output**: Pattern Set $R$

**Result**: $R$ contains patterns between songs $S_i$ and $S_j$, such that any matching tuples come from tupleMatches, and all patterns satisfy the rhythmic and gap size conditions indicated by $rr$ and $gs$.

```plaintext
1 begin
2 Pattern Set $R \leftarrow \emptyset$;
3 for each key $k$ in tupleMatches do
4    // Sort tupleMatches($k$) in lexicographic order of tuples in $S_i$, then $S_j$.
5    Sort tupleMatches($k$) in ascending order by (tupleMatches($k$)[i][0].o, tupleMatches($k$)[i][1].o) for elements $i$;
6    Mapping((Tuple, Tuple) $\rightarrow$ Set(Pattern)) status;
7    // Go through all potentially matching tuples in order, building up patterns in the process.
8    for $N_i$ from 0 to |tupleMatches($k$)| do
9       Set(Pattern) newEntry $\leftarrow \emptyset$;
10      // Find the pairs with the same relative pitch and within bounds the approximate same relative onsets.
11      Set((Tuple, Tuple)) contenderKeys $\leftarrow$ all tuple pairs $p$ in status such that
12      $(p[0].o - \text{tupleMatches($k$)[i][0].o}) \ast rr \leq p[1].o - \text{tupleMatches($k$)[i][1].o} \leq$
13      $(p[0].o - \text{tupleMatches($k$)[i][0].o}) / rr$;
14      for each (Tuple, Tuple) $p$ contenderKeys do
15         // Finalize patterns outside $gs$ distance from the current tuple match, gather the patterns within $gs$.
16         if $(\text{tupleMatches($k$)[i][0].o - p[0].o}) \leq gs \text{ and}$
17            $0 < (\text{tupleMatches($k$)[i][1].o - p[1].o}) \leq gs$ then
18            Add status($p$) to newEntry;
19         else
20            Add status($p$) to $R$;
21            Remove $p$ from status;
22      // Start a new pattern with the current tuple match, if needed.
23      if newEntry $\neq \emptyset$ then
24         Add new pattern $P$ with empty occurrence $P[0]$ from $S_i$ and empty occurrence $P[1]$ from $S_j$;
25         // Extend the gathered patterns with the current tuple match.
26         for each Pattern $P$ in newEntry do
27            Add tupleMatches($k$)[i][0] to $P[0]$ and tupleMatches($k$)[i][1] to $P[1]$;
28            Remove all but the patsPerTuple patterns with the most tuples from newEntry; // Replace the old tuple matches by the current one as last added to its patterns.
29            Add (tupleMatches($k$)[i] $\rightarrow$ newEntry) to status;
30       Add remaining patterns in status to $R$;
```
The next sub-routine, algorithm 10 builds on the number of tuples output by the previous. This process loops over each possibly pitch distance separately, to split up the data into smaller sets and improve performance on average. The worst-case running time and output are based on the event that all vectors made by the previous sub-routine are in the same set vs. In the inner loop, we go over all elements in vs, and maintain a window of vectors vw, which we want to use to create matching tuples. For each vector in vs, we make matches with each vector in vw. We can again opt to use a constant limit \( l \) on the size of vw, and only keep the most rhythmically similar vectors in vw. To do this, simply enhance the removal step by removing until at most \( l \) elements remain.

In the worst case, using neither constant bound leads to processing \( O(\text{Size}(F)^4) \) pairs, of which only \( O(\text{Size}(F)^2) \) will be unique. This at least tells us that \( O(\text{Size}(F)^2) \) is the maximum of the output size for this sub-routine. Since we are getting rid of duplicates on insertion, we incur another factor \( O(\log(\text{Size}(F)^2)) \) insertion time, making a total processing time of \( O(\text{Size}(F)^4 \cdot \log(\text{Size}(F)^2)) \). If we account for the constant bounds \( c \) and \( l \), or treat them as approximates of the results of our parameters, our processing time is cut down to \( O(\text{Size}(F) \cdot c \cdot \log(\text{Size}(F) \cdot \text{maxEmbellish}) + \text{Size}(F) \cdot \text{maxEmbellish} \cdot l) \cdot \log(\text{Size}(F) \cdot \text{maxEmbellish} \cdot l) \). This includes the sorting of \( O(\text{Size}(F) \cdot \text{maxEmbellish}) \) elements and looping over them with \( l \) insertions for each.

The output size is then also at most \( O(\text{Size}(F) \cdot \text{maxEmbellish} \cdot l) \), and likely lower as recurring patterns imply that a lot of the tuple matches will be created multiple times.

Our last sub-routine, algorithm 11, uses only the output of algorithm 10 to process on. This is done through a main loop going over the matching tuples, giving \( O(\text{Size}(F) \cdot \text{maxEmbellish} \cdot l) \) or \( O(\text{Size}(F)^2) \) iterations. During each iteration, involving tuple pair \( p \), we find all pairs \( p' \) in the set of pairs representing extendable patterns, such that \( p' \) satisfies the rhythmic variation conditions. We can maintain \( st \) by performing the insertions such that the order is based on the distance between the matching tuples \( p' \). This allows us to find the right pairs in \( O(\log |st| + \text{patsPerTuple}) \) time, where \( \text{patsPerTuple} \) is the number of patterns that can be extended by \( p \). We end up adding every tuple pair onto up to \( \text{patsPerTuple} \) patterns.

In the worst case we have all tuple pairs separately in \( st \), or we have one tuple pair representing all patterns, making \( \text{patsPerTuple} \) potentially linear in the number of tuple pairs. This puts the processing time between \( O((\text{Size}(F) \cdot \text{maxEmbellish} \cdot l)^2) \) and \( O(\text{Size}(F)^4) \), with the same counting for the output size.

We can use one final constraint parameter \( \text{patsPerTuple} \) for the maximum number of patterns a tuple pair can be made part of. Limiting this would change the running time to between \( O((\text{Size}(F) \cdot \text{maxEmbellish} \cdot l) \cdot \log(\text{Size}(F) \cdot \text{maxEmbellish} \cdot l) + \text{patsPerTuple})) \) and \( O(\text{Size}(F)^2 \cdot (\log(\text{Size}(F)^2) + \text{patsPerTuple})) \), depending on input size. The output size would be between \( O((\text{Size}(F) \cdot c \cdot l) + n) \) and \( O(\text{Size}(F)^2 \cdot \text{patsPerTuple}) \) depending on input size. We intend to take all notes in the time range of patterns at the end, according to requirement FS02. We also know that how we extend patterns purely depends on the last added tuple pair. As such, we can simply discard the shortest patterns that exceed our allowed number of patterns when merging sets of patterns under the same tuple pair. Alternatively, we could also remove patterns based on how much they are represented by the other patterns already.

It is worth noting that the parameters we’ve introduced for limiting the processing time and output size have different relations to the data. The space being limited by \( \text{maxEmbellish} \) is purely tied to \( gs \) and the tuple density of the songs. Since tuple density tends not to differ much in relation to the input size, keeping \( \text{maxEmbellish} \) as a constant based on \( gs \) is sufficient to let it function effectively. The parameter \( l \) limiting the number of tuple matches between a pair of songs is tied to \( rr \) and the size of the songs, as well as the number of vectors allowed by \( \text{maxEmbellish} \). A wider range given by \( rr \), and a larger number of vectors given by \( \text{maxEmbellish} \) both increase the space \( l \) is limiting linearly. The number of tuples in songs also limits \( l \) linearly, so simply multiplying these factors would give a guideline to what \( l \) should be to function well. Depending on the data, output size might increase due to increasing number of songs, or increasing size of the songs, the former is more easily handled than the latter by \( l \). Finally, our number of patterns per note, limited by \( \text{patsPerTuple} \) likely tends to be linear in the number of tuple matches, and thus shares the recommended growth rate with \( l \). As such, to keep the output and processing time balanced.
with our quadratic bottleneck of comparing songs, we recommend to treat maxEmbellish as a constant, and \( l \) and patsPerTuple as relative to the square root of the input size, \( O(\sqrt{\text{Size}(F)}) \).
Chapter 9

Family Pattern Clustering

In this chapter, we discuss the third task in our pipeline: clustering patterns together. This task aims to take over the responsibility of making patterns span the tune family, as this would be difficult to achieve directly from the discovery task.

We first go over the challenges involved in this task, particularly with respect to how we approach the preceding tasks. We then give our solution to the pattern clustering problem in the form of an algorithm. Since pattern cropping in our case is merged into this task, we will also discuss this problem during each main section.

9.1 Challenges

In this section, we discuss what challenges come up in pattern clustering. First we discuss how to handle similarity between patterns. Second, we address the need for patterns to have sufficient occurrences. We then go over how we can prevent pattern overlap from the clustering perspective, which will be expanded on with pattern selection in Chapter 10. Finally, we discuss the most difficult problem of this task, keeping the processing time low.

9.1.1 Pattern Similarity

A difficult factor in clustering patterns together is that we cannot map the patterns to any low-dimensional space. We have to essentially resort to a similarity graph, connecting every pair of patterns that is sufficiently similar. This would come down to performing a pairwise evaluation of the similarity requirements from Chapter 5.

Since patterns from the pattern discovery task already come with at least two occurrences, we have an alternative for determining similarity. If we find patterns $P_1, P_2$ which have occurrences $O_1, O_2$ from the same song sharing a sufficient subset $O_{12}$, we can see these two occurrences as the same. This way, we let the pattern discovery algorithm take care of approximate similarities involving transposition and rhythmic variation, covering requirements $FI03$ and $FI02$.

In order to also make sure the gaplessness requirements $FI01$ is covered, we must make sure that the shared subsets between two patterns cover this condition as well. This simply comes down to making sure that the shared subset does not contain gaps beyond the permitted size. Since we are already taking subsets here, the embellishment requirement $FI04$ is already taken into account.

It is important to note that just looking at the shared subset of one occurrence in each pattern does not mean the other occurrences $O'_1, O'_2$ are sufficiently similar. Particularly, the rhythmic variation may be too far removed. In this case, it would be most correct to also check the occurrences from the different songs, verifying that they satisfy the conditions pair-wise as well. Doing this for an entire cluster of patterns would require a lot of computation. Instead, it might be worth sacrificing completeness of the discovery process by shrinking the rhythmic variation window.
CHAPTER 9. FAMILY PATTERN CLUSTERING

The appropriate shrunken parameter $rr'$ to take would be $\sqrt{rr}$ for the original parameter $rr$. Doing so would cause $O_1'$ and $O_2'$ to stay within the desired variation. This is, however, something we only can do if we center a cluster on a single shared subset of occurrences.

9.1.2 Pattern Spread

At the end of our pipeline, we want to end up with patterns that at least appear in $mr|F|$ songs of the family, as stated in requirement FS03. In order to provide for this, every single pattern output by this task should cover at least that many songs with pattern occurrences. As such, we should increase the size of a cluster until there is enough coverage. The selection task will take care of making sure the better patterns will get into the final output.

We do not want to restrict our clustering to just adding patterns from different songs to one another. It is also important to add occurrences of patterns that are in different places in the same song to a cluster. Since we have no reason to split up patterns beyond lack of similarity, we do not need to pay special attention to only including patterns from different songs.

We also need to consider that pattern selection may involve only taking a subset of the occurrences of a pattern. To this end, adding additional occurrences could be helpful.

9.1.3 Pattern Overlap Reduction

As we are integrating cropping into our clustering, we have to consider the pattern overlap problem. We require that patterns in our final output do not overlap in the time-domain, as per requirement FS01. Since we are clustering patterns together, we will run into cases where some pattern occurrences extend farther than the rest. The additional pattern length is not likely to be indicative of sufficiently recurring tuples. Due to this problem, we want remove the excess tuples from pattern occurrences.

The degree to which we remove tuples should be in service of the pattern spread requirement FS03. This relation to that requirement gives us the parameter $mr$, indicating how many songs a pattern should occur in, to use. We can simply remove tuples $t_s$ from the start or $t_e$ from the end of a pattern, until there are $mr|F|$ songs represented by pattern occurrences spanning from $t_s$ to $t_e$ in the cluster. We can of course choose which of these two to remove, based on how many occurrences we would add towards $mr$.

Since often smaller patterns can be chained together as a valid pattern in some subset of songs, it can be the case that the discarded tuples would have been the sole representation of some patterns. To save these tuples, but increase processing time, we can reinsert these partial pattern occurrences as full occurrences. It is unlikely that these are important, however, as the pair-wise pattern discovery we use will likely provide alternate instances of these discarded partial occurrences.

It is important to note, that whenever we group together a set of patterns, partially or otherwise, we will be taking away from the total pool of patterns for subsequent clustering. The order in which we do our clustering becomes important, so we should appropriately select which patterns or sets of tuples we will start with clustering. We should aim to prioritize patterns found in areas where most patterns are found, as these areas indicate that they are similar across the entire tune family. If we have the resources to look more closely, we specifically want to cluster patterns that are very consistent first. That is, if the similarity between each pattern occurrence in a cluster is high, we want to first form these clusters. This should reduce the risk of missing out on important patterns due to cropping at the wrong points.

9.1.4 Performance

In Section 9.1.1, we stated that we can only properly group patterns together with a pair-wise similarity graph. Determining clusters from this, in which every pattern has to be similar to each other pattern in the cluster, would become a clique problem. We can not afford computing clusters as cliques normally, as this problem is severely hard to solve quickly.
Instead, we can use the idea of using a shared subset of tuples as core of a cluster. This would involve picking one pattern with a good occurrence for clustering around. Good patterns to start with are patterns that contain a large portion of the tuples in their time-range. We also want to start off with large patterns, as the results of cropping them might come back as part of smaller patterns later. The number of tuples in a pattern multiplied by the fraction of tuples it has in its time range is likely a good option here.

Specifically finding sets of patterns with a large shared subset of tuples may still cost too much processing time. Since we are giving priority to patterns containing a large portion of the tuples in their time-range, we may look at the time-range of pattern occurrences, rather than the exact subsets. In doing this, we are making the assumption that patterns that contain most tuples have sufficient overlap with patterns in their same time-range.

When a single pattern contains most tuples in its time-range, this may not indicate that the other patterns do the same. As such, we may want to choose our order of the clusters we make based on time-ranges in songs, rather than patterns. We would be basing our order on what time-range in a song contains patterns with the largest portion of tuples in their time-ranges. This would also immediately take care of figuring out where we should crop patterns.

9.2 Solution

In this section, we discuss our algorithmic solution to the clustering problem. We first explain our algorithm. We then discuss the performance of our algorithm. Finally, we briefly go over covering the pattern cropping problem, as this was delegated to the clustering task.

Our solution for the clustering problem is very basic. Since we are working with the output of the discovery task, we first want to prioritize performance over quality. As such, it serves more as part of the proof of concept for the pipeline, than a high quality example of clustering in this context.

9.2.1 Algorithm

In our algorithm, we combine patterns that have an occurrence that share particular tuples with each other. We determine where enough pattern occurrences still go on for a long time, and take these long-lasting pattern occurrences from this common starting point, to an end point they can all meet.

Algorithm 12 is our overarching routine. This largely covers the preprocess needed for cluster extraction. We first determine based on our parameter $mr$ exactly how many songs each cluster should be represented in. We then make a mapping of tuples $t$ to the lists of pattern occurrences contain these $t$. Each of these lists is then sorted on the end time of these pattern occurrences, such that the pattern occurrences that still go on the longest after $t$ are first. We use this sorted structure to do most of our computations. We then determine the greatest time $l$ that pattern occurrences go on past $t$, such that at least the required number of songs is represented by the patterns these occurrences are from. Finally, we sort all tuples based on these lengths $l$, having the tuples with pattern occurrences reaching the longest first, and we start extracting patterns in that order. We do this so the longest patterns are extracted first. This can be replaced by a different ordering heuristic.

We use Algorithm 13 to compute the aforementioned lengths. Since the list of pattern occurrences is already sorted by endpoint, the task is very simple. We loop down the list of pattern occurrences $O$, and add the songs that the occurrences in the pattern of $O$ to a set of represented songs. When the set of represented songs reaches the required number of songs, we take the end time of the currently considered pattern occurrence and subtract the onset time of the tuple we are computing for, giving us our length $l$.

Finally, in Algorithm 14, we extract a pattern by clustering the patterns containing a certain tuple $t$ together. Since we are removing patterns from our lists later in this sub-routine, we must first recompute the length $l$, in case it has changed. From here, we can loop down the sorted list of
CHAPTER 9. FAMILY PATTERN CLUSTERING

Algorithm 12: Clustering

Input : Tune Family $F$, Pattern Set $R_{in}$, $mr$, $N$ maxOccsPerSong

Output: Pattern Set $R_{out}$

Result: Combines patterns with 2 bijective occurrences from $R_{in}$ into patterns with many occurrences in $R_{out}$, such that each pattern occurs in at least $|F| \times mr$ different songs.

1 begin
2 Pattern Set $R_{out} \leftarrow \emptyset$;
3 $N \text{ numReqSongs} \leftarrow |F| \times mr$ rounded up;
4 // Initialize mapping indicating what Pattern Occurrences contain what tuples and what the length of a cluster starting at each tuple is.
5 Mapping(Tuple $\rightarrow$ ($R$, List(Pattern Occurrence)) $\text{tupData} \leftarrow \emptyset$;
6 for each Pattern Occurrence $O$ in $R_{in}$ do
7   for each Tuple $t$ in $O$ do
8     Add $O$ to $\text{tupData}(t)[1]$;
9   for each Tuple $t$ in the keys of $\text{tupData}$ do
10      Sort $\text{tupData}(t)[1]$ in descending order of $\text{End}(\text{tupData}(t)[1][i])$;
11 // Compute the lengths of clusters.
12 for each Tuple $t$ in the keys of $\text{tupData}$ do
13   $\text{tupData}(t)[0] \leftarrow \text{ComputeClusterLength}(\text{tupData}, t, \text{numReqSongs})$;
14 // Compute the sorted list of tuples based on the length of clusters starting at each tuple.
15 List(Tuple) sortedTuples $\leftarrow$ all tuples in $F$;
16 Sort sortedTuples in descending order of $\text{tupData}(t)[0]$ for tuples $t$;
17 for each Tuple $t$ in sortedTuples, in order do
18   Add ExtractCluster($F$, $R_{in}$, $\text{tupData}$, $t$, $\text{numReqSongs}$, $\text{maxOccsPerSong}$) to $R_{out}$;

Algorithm 13: ComputeClusterLength

Input : Mapping(Tuple $\rightarrow$ ($R$, List(Pattern Occurrence)) $\text{tupData}$, Tuple $t$, $N$ numReqSongs

Output: $R$ length

Result: Finds the distance between onsets from $t$ to the last tuple $t'$, such that there are at least numReqSongs songs represented by pattern occurrences containing $t$ and ending at or after $t'$, using data from $\text{tupData}$ sorted by endpoints of occurrences.

1 begin
2 $R$ length $\leftarrow 0$;
3 Set(Song) representedSongs $\leftarrow \emptyset$;
4 $N$ index $\leftarrow 0$;
5 while index $< |\text{tupData}(t)[1]|$ and $|\text{representedSongs}| < \text{numReqSongs}$ do
6   length $\leftarrow \text{End}(\text{tupData}(t)[1][\text{index}]) - t.0$;
7   Add Song($\text{tupData}(t)[1][\text{index}]$) to representedSongs (if it is not already contained);
8   Increment index;

Discovering Audibly Recognizable Patterns within Tune Families
pattern occurrences, until a pattern occurrence no longer goes as far as \( t \). At this point, we know that the pattern occurrences we have considered for \( t \) together provide patterns covering sufficiently many songs. For each of these pattern occurrences \( O_t \), we go over all pattern occurrences \( O \) in its pattern. So if we have two pattern occurrences involving \( t \), then the even pattern occurrences elsewhere from their respective patterns are used for the same cluster. From \( t \) through the tuples within \( l \) from \( t \), if a tuple in this range is contained in \( O_t \), we take its equivalent in \( O \) and add it into the cropped version of \( O \), \( O_{\text{cropped}} \). After going over all these tuples, we add \( O_{\text{cropped}} \) to the cluster, and remove \( O \) from all the lists it appears in. After doing this for all relevant pattern occurrences, this gives us a cluster of pattern occurrences that are all approximately similar, from sharing the same time-range. We show this extraction process visually in Figure 9.1, in which two songs must be covered by the extracted patterns, so the four indicated occurrences are extracted, while the occurrences of the remaining pattern end earlier and are thus left behind.

9.2.2 Performance

The performance of this task depends both on the size of the tune family, and the size of the pattern set received from the discovery task. We aim for the size of our discovered pattern set \( R_{\text{in}} \) to be at most \( O(\text{Size}(F)^2) \). In order for our running time to remain sub-cubic, we require a running time of \( O(\text{Size}(R_{\text{in}})\sqrt{\text{Size}(R_{\text{in}})}) \). We also need to make sure that our output does not grow too large for pattern selection. To this end, our output size should be linear in \( R_{\text{in}} \), and each individual cluster should contain at most \( O(\text{Size}(F)\sqrt{\text{Size}(F)}) \) occurrences.

In Algorithm 12, we largely perform computations linear in the size of the input. Lines 5 through 7 perform one un-ordered insertion into some list per tuple in \( R_{\text{in}} \), which can be done within \( O(\text{Size}(R_{\text{in}})\log \text{Size}(R_{\text{in}})) \) total. The same total time goes for lines 8 and 9, in which the resulting lists are sorted. In lines 10 and 11, we perform a number of computations with Algorithm 13, equal to the size of the tune family. As such, this sub-routine cannot take longer than \( O(\text{Size}(R_{\text{in}})) \). Finally, a sort is applied to all tuples in the tune family, and Algorithm 14 is executed on every tuple in the tune family.

Algorithm 13 involves a simple computation through at most as many pattern occurrences contain the tuple it is called upon. This computation is fully linear in nature, we only query the
Algorithm 14: ExtractCluster

\textbf{Input}: Tune Family \( F \), Pattern Set \( R_{in} \), Mapping(Tuple \( \rightarrow \) (R, List(Pattern Occurrence))) \( \text{tupData} \), Tuple \( t \), \( N \) numReqSongs, \( N \) maxOccsPerSong

\textbf{Output}: Pattern \( P_{ext} \)

\textbf{Result}: Extracts a pattern \( P_{ext} \) starting with \( t \) such that the occurrences in \( P_{ext} \) come from at least a number of songs equal to \( \text{numReqSongs} \), with maximum length and at most \( \text{maxOccsPerSong} \) occurrences per song.

1 begin
2 \( \text{R length} \leftarrow \text{ComputeClusterLength}(\text{tupData}, t, \text{numReqSongs}); \)
3 Mapping(Song \( \rightarrow \) N) \( \text{representedSongs} \leftarrow 0; \)
4 \( N \) index \( \leftarrow 0; \)
5 Pattern \( P_{ext} \leftarrow \emptyset; \)
6 // Add pattern occurrences to \( P_{ext}. \)
7 \textbf{while} \( \text{index} < |\text{tupData}(t)[1]| \) and \( 0 < \text{length} \leq \text{End(\text{tupData}(t)[1][\text{index}])} \) \textbf{do}
8 \text{Pattern currentPattern} \leftarrow \text{Pattern(\text{tupData}(t)[1][\text{index}]);}
9 \textbf{for} each Pattern Occurrence \( O \) in currentPattern \textbf{do}
10 \text{if} \ \text{representedSongs(\text{Song}(O))} < \text{maxOccsPerSong} \text{ then}
11 \text{PatternOccurrence croppedOcc} \leftarrow \emptyset;
12 \textbf{for} each Tuple \( t' \) in \text{tupData}(t)[1][\text{index}], from \( t.o \) through \( t.o + \text{length} \) \textbf{do}
13 \quad \text{Add the equivalent tuple in } O \text{ to croppedOcc;}
14 \quad \text{Add croppedOcc to } P_{ext};
15 \text{Increment representedSongs(\text{Song}(O));}
16 \textbf{for} each Tuple \( t' \) in \( O \) \textbf{do}
17 \quad \text{Remove } O \text{ from } \text{tupData}(t)[1];
18 \text{Increment index;}
19 end
pattern occurrences in constant time. As such, even if we apply this computation to all tuples in
the tune family, we are performing one operation per tuple in each pattern, which is $O(\text{Size}(R_n))$.

Finally, Algorithm 14 involves a more complex computation, with implications for future com-
putations. We first see a call of Algorithm 13, which is done once, so we already know it performs
sufficiently well. As with the other sub-routine, we loop over at most a number of tuples linear
in $R_n$. Now, however, we are also looping through all associated pattern occurrences for each
pattern occurrence, including all its notes. This time around, our operations within the loop are
not constant time. We perform insertions into a new pattern occurrence for, at most, each tuple in
a pattern occurrence we are considering, seen in line 12. In line 16 and 17, however, we remove all
tuples we could have possibly used for computation from the data set we are working on. As such,
we end up only performing our insertions into the new pattern occurrences, and the removals from
our data set, a total of $O(\text{Size}(R_n))$ times. Since we can do each of these operations in logarithmic
time, our total running time for all calls of Algorithm 14 combined is $O(\text{Size}(R_n) \log \text{Size}(R_n))$.

We also require for the selection task, that no pattern has occurrences greater than linear in
the tune family size. During Algorithm 14, in which we construct the new patterns, we use the
parameter maxOccsPerSong, which we default to 3. Whenever we gather pattern occurrences for
a cluster, we keep track of how many times each song is represented by an occurrence in our new
pattern. If we exceed maxOccsPerSong, we simply do not add the extracted pattern occurrence
to the new pattern. This way, we are guaranteed to have at most $O(|F| \times \text{maxOccsPerSong})$
occurrences per pattern, which is at worst linear in the size of the tune family.

### 9.2.3 Cropping

We decided to cover the need for pattern cropping within our clustering algorithm. The cropping
we perform takes place upon pattern extraction, in Algorithm 14. Specifically, in line 10 through
13, we construct the cropped version of each pattern occurrence for the output. Here, only the
subset of each occurrence is being taken from a specified starting note to a common time-distance
after that starting note. This ensures that all pattern occurrences within a single cluster share
the same maximum endpoints.

It is important to note that there should be a bijective mapping maintained between the
tuples of occurrences within the same pattern from the pattern discovery task. During cropping,
we only use a specific range of tuples in one song to build our cropped patterns from. One pattern
occurrence is present in this specific range, as our point of reference. Any other occurrence has its
tuples $t$ taken, based on whether or not the bijectively mapped note $t'$ from our point of reference
is within the desired range.
Chapter 10

Family Pattern Selection

In this chapter, we discuss the fourth and last task in our pipeline, the task of selecting patterns for the final output. This task takes the responsibility of limiting the output to one coherent set of patterns.

We once again discuss the challenges presented by the requirements and role of this task. We then give our algorithmic solution to the pattern selection problem.

10.1 Challenges

In this section we discuss and address the challenges involved in pattern selection. We start with the problem of overlapping patterns. We then discuss selecting patterns with consistent positioning. Third, we will discuss selecting patterns that occur sufficiently across the tune family. Finally, we go over how to determine what patterns are most important to have in the output.

10.1.1 Overlap Prevention

The first primary task of pattern selection is to find a subset of discovered patterns, such that there is no overlap in these patterns, as per requirement FS01. This largely comes down to determining the best subset of a song in the form of a sequence of pattern occurrences. This can be as simple as picking one pattern occurrence $O_1$, then picking the next pattern occurrence $O_2$ in the song without $O_1$, and so on.

In preventing overlap, we could still apply an additional pattern cropping step. We would have to account for consistency across all pattern occurrences when doing so. As such, this is likely an expensive step to allow. If a pattern is worth enough to warrant taking a subset, we are likely better off prioritizing it over other patterns. We will discuss this more in Section 10.1.4.

10.1.2 Pattern Order and Position

Making sure the positions and order of patterns stays consistent across the tune family is both a simple and a difficult problem. We first need an approach for having consistency across a pair of patterns, which is the easier task. We then have to enforce consistent positioning across the entire family, making this task more difficult.

For a pair of songs $S_1, S_2$, we can take the songs as the axes of a grid, with each grid cell denoting a pair of tuples $t_1, t_2$ from $S_1, S_2$, respectively. We now want to find a path that always advances close to the diagonal of this grid, going through the pattern occurrences we will take as output. Since we only advance towards the ends of both songs in our path, the patterns we encounter must be in the same order. Additionally, because the patterns are found close to the diagonal, the two occurrences from each song must be in similar positions in their respective songs. We can accomplish this with a simple dynamic programming approach, optimizing for the path...
CHAPTER 10. FAMILY PATTERN SELECTION

with the highest score. We can have patterns give some points, following a diagonal step to the next cell costs some points, and following a straight step to the next cell costs yet more points.

A problem arises when we try to apply the pair-wise approach to an entire tune-family. This problem is either that we need the grid to have a number of dimensions equal to the number of songs, which we can not afford. The other option is to perform the pair-wise approach on all pairs of songs, and trying to get each song to be consistent for the pairs it is in.

The first thing to address here is that a song may have different resulting pattern sequences on the different pair-wise paths. In this case, we may want to give every pattern occurrence that does show up a score. Based on this score, we can then do the same dynamic programming approach, but in one dimension, to select a sequence of pattern occurrences for the song that best represents all pair-wise sequences.

The second problem is that even when we have a sequence, the sequence is still likely far removed from many pair-wise results. In order to solve this problem, we simply need to make the pair-wise sequences similar in advance, while keeping them reliable. This involves somehow finding sequences pair-wise that are somehow consistent with the other sequences. We can do this by doing an initial run of the whole pair-wise process. Then, rather than getting a final sequence based on how pattern occurrences scored, we can filter out patterns that did not score sufficiently. From here we can rerun the computation to find sequences with less potential variation in patterns. We can do this re-run a number of times with increasing score threshold for removal. The more re-runs we use, the more likely it is that the final sequence is a reliable estimate.

10.1.3 Pattern Spread

In pattern selection, we are also selecting specific pattern occurrences within patterns. As such, we need to make sure that we select sufficient occurrences for each pattern in the output, such that requirement \text{FS03} is met. This can once again be made difficult by having to apply to the entire tune family. Even if we select a pattern for one song, we may not select it consistently for another. We need some way of controlling that patterns that are selected are selected for a large part of the family at a time.

As proposed in for order and positioning consistency in Section 10.1.2, we can hone in on a good and consistent set of patterns by filtering and re-evaluating our selection. We can simply incorporate the frequency of pattern occurrences we wish to have in the filtering step. For this it would be primarily important to prevent cutting out potential final patterns early, while removing enough patterns to advance with each re-run. This may become a delicate matter of tweaking parameters for vastly different results, although any of these results could be technically valid.

A problem that arises with this approach is that the number of occurrences for a pattern may not strictly increase with the removal of competing patterns. We may, for instance, have the case that two patterns have occurrences \(O_1, O_2\) covering a certain part of a song, while two other patterns have occurrences \(O_3, O_4\) covering that same exact part. If \(O_1, O_2\) scores high initially, but \(O_2\) does not occur in other sequences, \(O_2\) may be removed, making the overall better sequence in the next iteration \(O_3, O_4\). To address this, we may want to skew the score a pattern adds to the sequence in one iteration, by increasing the score given by prominent patterns in the previous iteration.

10.1.4 Determining Importance

When choosing the right patterns, we need some way of determining which patterns get priority over others. Our requirement \text{FS04} states that a good way to determine the importance of a pattern is to compare how often it occurs in the tune family, with how often it occurs outside the tune family. Sadly, since our patterns can vary in rhythm, pitch and embellishments, determining how often a pattern may occur outside the tune family is a very demanding task.

We do have some properties for patterns that may indicate that a pattern occurs more within the family than elsewhere. One such property is the number of tuples in a pattern. The more tuples a pattern contains, the more specific the pattern is, and by extension, the less the pattern
CHAPTER 10. FAMILY PATTERN SELECTION

would occur in arbitrary songs. The other property is the number of occurrences the pattern has within the family already. The more occurrences there are in the family, the more there would have to be outside the family to counter balance it.

Since we are largely processing with heuristics throughout the pipeline, we should also prioritize patterns that are most likely to satisfy our requirements. First of all, for patterns to be strong candidates for the final output, they have to appear a lot throughout the tune family. As such, we should prioritize patterns that have a lot of occurrences, particularly when these are from different songs. Another factor we have to take into account comes from the possible inaccuracies in the clustering process. To account for this, we should prioritize pattern occurrences with a large portion of the tuples in their time-range, since these are more likely to satisfy our requirements.

All these priorities require us to make some sort of scoring function for pattern occurrences. Along with the parameters, the form of this function can account for a wide range of outcomes for our pipeline.

10.2 Solution

In this section, we discuss our solution to the pattern selection problem. We will first discuss our algorithm for performing pattern selection. Afterwards, we will discuss the running time of our algorithm.

10.2.1 Algorithm

Our pattern selection algorithm revolves heavily around dynamic programming, iteration and scoring. We select patterns based on how they fit in a path from the start to the end of a song together, with a weight for each pattern in the form of a song. We then iterate on this, removing patterns that do not occur sufficiently, to clean up patterns that might be in the way of more important ones. This iteration happens a number of times until the remaining patterns satisfy the right conditions.

In our overarching algorithm, algorithm 15, we perform the entire iteration process. We first start with a single pair-wise selection step, using algorithm 16, which gives us a sub-set of our initial set of patterns. This step gets rid of any patterns that are extremely unlikely to be candidates for the final output, preventing them from obstructing the rest of the process. We then iterate a specified number of times over the full selection process. Each iteration, we remove patterns that did not occur sufficiently in our selection, with each iteration requiring more occurrences. This iteration process again is intended to gradually remove patterns that obstruct our selection process. At the end of this process, by having removed patterns with insufficient occurrences, we know that our final output only has patterns that occur enough in the tune family. For all intents and purposes, we will default to using 10 iterations.

The selection process itself uses two selection sub-routines. The first sub-routine determines in broad strokes what pattern occurrences can be in our final pattern set. It does this for each pair of songs. This sub-routine uses the last known output from the selection process to have a slight bias towards including previously selected pattern occurrences, for stability. This sub-routine can only use patterns from the last previous selection, but can use any pattern occurrence of these patterns, encouraging further selection of occurrences from those patterns.

The second sub-routine determines exactly what pattern occurrences should be taken for the output, such that there is no overlap between patterns. This sub-routine can only select pattern occurrences from the rough selection of the first. In this case, there should be a rather heavy bias towards patterns that occur many times in the results of the first sub-routine. This causes pattern occurrences occurring in many pair-wise selections to more likely be selected for both songs in the pair.

Our first selection sub-routine, algorithm 16, takes care of the broad, pair-wise pattern occurrence selection, as stated before. Although it is a fairly long algorithm, the intuition behind it is a fairly simple dynamic programming approach. We simply maintain a grid for each pair
Figure 10.1: A visual representation of the pair-wise and individual selections for Song 1. The thick line segments denote pattern occurrences, the dashed line denotes an optimal path.
Algorithm 15: Selection

**Input:** Tune Family $F$, Pattern Set $R_{in}$, $mr$ numIterations, $pwsBias$, $isBias$, skipCost

**Output:** Pattern Set $R_{out}$

**Result:** Selects a set of non-overlapping patterns $R_{out}$ from $R_{in}$, such that each pattern occurs in $|F| \cdot mr$ different songs, and occurs in similar position and order for each song.

1 begin
  // Determine an initial pair-wise set of patterns.
  2 for $N_i$ from 1 through numIterations do
    3 $R_{out} \leftarrow$ PairWiseSelection($F$, $R_{in}$, $R_{in}$, 0, skipCost);
    4 $R_{out} \leftarrow$ PairWiseSelection($F$, $R_{in}$, $R_{out}$, pwsBias, skipCost);
    5 $R_{out} \leftarrow$ IndividualSelection($F$, $R_{in}$, $R_{out}$, $isBias$, skipCost);
    6 $R_{minSpread} \leftarrow mr \cdot |F| \cdot i/$numIterations;
    7 Remove patterns occurring in fewer than minSpread songs from $R_{out}$;
    8 Add the remaining tuples in the time-range of each Pattern Occurrence to that occurrence;

of songs, with each cell indicating a pair of tuples from these songs. Every such cell contains the best possible score of a path from the start of the pair of songs to that pair of tuples. The cell also contains information on what the previous cell in the path was and, if any, what pair of pattern occurrences formed the bridge from the previous cell. Once we fill in the entire grid, we can back-track from the end of a pair of songs to find what patterns were part of the best scoring path. Figure 10.1 shows examples of two grids with Song 1 and Song 2, and Song 1 and Song 3. Here the thick lines denote a pair of pattern occurrences, bridging to later points in the grid. The dashed line is an example of the optimal path found in these grids.

We start by placing all pairs of occurrences from the same pattern in the grid as bridges. This provides a reference of one cell, at the ends of the two pattern occurrences, to a preceding cell, the ends of the two pattern occurrences. Each such bridge has a score equal to the sum of the scores of its two pattern occurrences. A higher score for a bridge makes it more likely for the bridge to be used in the highest scoring path, so pattern occurrences considered better are given a higher score. We use 0.05 as default value for our bias here, as this selection largely does not come down to specific occurrences from a previous iteration.

In the main for-loop, we go over each pair of songs and build the highest scoring paths. For a pair of songs, we initialize the cell in the grid indicating the starts of both songs. We also initialize the cells indicating the start of one song with the advancement of the other. For each step taken in the grid from the starting point of both songs without using a bridge, we enforce a penalty.

We then start the core nested loop, filling in the grid one row at a time, in advancing order through the songs. This way, we always know the top possible score for any cell earlier in the songs than the cell we are filling. For each cell, we determine the cost of doing a single step from the previous tuple in either song, or both at a same time, a diagonal step. A diagonal step incurs the same penalty as a regular step, encouraging diagonal advancement, the penalty is given by the skipCost, which we default to 1. This makes it more likely for the optimal path to be close to the diagonal, indicating consistent pattern positioning. After determining the scores for these direct steps, we check the scores given by any bridges to the current cell. The bridge both takes the score of an earlier cell and adds the score of the pattern occurrences, highly encouraging the use of these bridges.

Finally, when the cell indicating the end of both songs is filled in, we determine the output. We always indicate what the last cell in a path was and what bridges were used, and every cell only contains this information for the path with the highest score. As such, we back-track across this information to find all pattern occurrences used to create the optimal path. This gives us a
set of non-overlapping pattern occurrences for each pair of songs, such that the patterns always occur in the same order, and likely in similar position.

In our second selection sub-routine, algorithm 17, we select exactly the pattern occurrences the output should have for each song, such that nothing overlaps. This builds heavily on the information given by the other selection sub-routine. A high number of occurrences of a pattern in pair-wise selection indicates to this selection process that a pattern is more likely to be represented across many songs. Figure 10.1 shows how the individual selection at the bottom could look, based on the output of the two pair-wise selections above it. The patterns in this individual selection image coincide with non-overlapping patterns from the individual selections.

Similarly to algorithm 16, this sub-routine computes scores for patterns and enters bridges for pattern occurrences into a grid. In this case, the grid has only one dimension, as we are selecting for a single song. Everything else is adjusted to the one-dimensional grid.

There are two core differences in this sub-routine from the previous. In this, we only take pattern occurrences that occurred in the previously computed pattern set, as these were specifically selected in the rough pair-wise selection step. Since we need to select pattern occurrences that have a high chance of appearing in more songs, we also increase the bias towards these patterns to a default of 0.5.

Finally, our last sub-routine, algorithm 18, determines the scores of pattern occurrences, used by the selection sub-routines. This sub-routine loops over all patterns and maps it to a score based on a number of its properties. This requires the input of a previous set of selected patterns, the full set of patterns, and a bias parameter.

Our score function values a few straightforward properties. First, we value the number of tuples in an occurrence, $|O|$, which indicates a pattern unlikely to be found outside of the tune family. We value this property by $\log |O| \ast |O|$, which is slightly above linear, as otherwise two halves of the occurrence would still give the same score together. Second, we multiply by the number of songs the pattern is in, fullPatSpread, as this makes it a more likely candidate for the final output. Third, we multiply by the ratio between tuples in the occurrence and tuples in the time-range of the occurrence, $|O|/\text{fullOccSize}$. Containing most tuples in the time-range indicates that the occurrence is more likely consistent with other occurrences of the same pattern. Lastly, we multiply by 1, plus the number of songs we previously selected the pattern for, scaled by the bias parameter.

### 10.2.2 Performance

Our algorithm starts with a pair-wise selection sub-routine, using all patterns from the input. Since worst-case running-time would involve all patterns from the input to be involved in every iteration of our selection process, this initial step is insignificant compared to iterative usages of the pair-wise selection.

The iteration process itself is regulated by the numIterations parameter. This parameter purely increases the likelihood of a good final selection, but can be considered a constant. In order to get the maximum possible quality out of the iterations, the number of iterations should be equal to $mnr \ast |F|$, which is linear in the number of songs.

For the removal of patterns from the output, we can claim at most linear processing time in the size of our initial pattern set. As such, the bottleneck must lie with either of the selection sub-routines.

In both selection sub-routines, we start by calculating the scores of pattern occurrences. To do this with algorithm 18, we go over each pattern occurrence once. In each pattern occurrence, we use the number of notes in the occurrence and the number of notes in the time-range. These two properties can be stored after being computed once, and used later, making them effectively constant to linear time. We also need the number of songs a pattern occurs in in the last selection and in the input. We can keep a set of these songs and insert the songs of each occurrence, making it the same number of operations as looping over the occurrences, except that the operation may take logarithmic time in the number of songs. As such, this calculation costs us roughly linear time in the input pattern set.
Algorithm 16: PairWiseSelection

Input: Tune Family $F$, Pattern Set $R_{in}$, Pattern Set $R_{prev}$, $\mathbb{R}$ bias, $\mathbb{R}$ skipCost
Output: Pattern Set $R_{out}$
Result: Selects a set of patterns $R_{out}$ from $R_{in}$, such that each pattern occurs in $R_{prev}$ and in similar position in at least one pair of songs for each occurrence.

begin

Mapping(Pattern Occurrence $\rightarrow \mathbb{R}$) scores $\leftarrow$ CalculateScores($F$, $R_{in}$, $R_{prev}$, bias);
Mapping((Tuple, Tuple) $\rightarrow$ (Pattern Occurrence, Pattern Occurrence)) patternBridges;

// Determine all pattern bridges.
for each Pattern $P$ in $R_{in}$ such that $P$ occurs in $R_{prev}$ do
  for each pair of Pattern occurrences $O_1, O_2$ in $P$ such that $Song(O_1) \neq Song(O_2)$ do
    $e_1 \leftarrow$ the tuple after the last tuple of $O_1$, if it exists, otherwise, the last tuple of $O_1$;
    $e_2 \leftarrow$ the tuple after the last tuple of $O_2$, if it exists, otherwise, the last tuple of $O_2$;
    patternBridges(($e_1, e_2)$)) $\leftarrow$ ($O_1, O_2$);

// Compute the optimal path through each pair of songs.
for each pair of songs $S_1, S_2$ in $F$, such that $S_1 \neq S_2$ do
  // Initialize the start row and column.
  topTotalScores(($S_1[0], S_2[0]$)) $\leftarrow$ (0, null, null);
  for $i$ from 1 to $|S_1|$ do
    topTotalScores(($S_1[i], S_2[0]$)) $\leftarrow$ ($j \ast$ skipCost, ($S_1[i-1], S_2[0]$), null);
  for $j$ from 1 to $|S_2|$ do
    topTotalScores(($S_1[0], S_2[j]$)) $\leftarrow$ ($j \ast$ skipCost, ($S_1[0], S_2[j-1]$), null);

  // Compute the optimal path.
  for $i$ from 1 to $|S_1|$ do
    for $j$ from 1 to $|S_2|$ do
      topTotalScores(($S_1[i], S_2[j]$)) $\leftarrow$ (topTotalScores(($S_1[i-1], S_2[j-1]$))$[0]$ $-$ skipCost, ($S_1[i-1], S_2[j-1]$), null);
      if topTotalScores(($S_1[i], S_2[j]$))$[0]$ $<$ topTotalScores(($S_1[i-1], S_2[j]$))$[0]$ $-$ skipCost then
        topTotalScores(($S_1[i], S_2[j]$)) $\leftarrow$ (topTotalScores(($S_1[i-1], S_2[j]$))$[0]$ $-$ skipCost, ($S_1[i-1], S_2[j]$), null);
      if topTotalScores(($S_1[i], S_2[j]$))$[0]$ $<$ topTotalScores(($S_1[i], S_2[j-1]$))$[0]$ $-$ skipCost then
        topTotalScores(($S_1[i], S_2[j]$)) $\leftarrow$ (topTotalScores(($S_1[i], S_2[j-1]$))$[0]$ $-$ skipCost, ($S_1[i], S_2[j-1]$), null);

  // Check possible pattern bridges.
  for each pair of Pattern Occurrences ($O_1, O_2$) in patternBridges(($S_1[i], S_2[j]$)) do
    $s_1 \leftarrow$ the first tuple of $O_1$;
    $s_2 \leftarrow$ the first tuple of $O_2$;
    if topTotalScores(($S_1[i], S_2[j]$))$[0]$ $<$ topTotalScores(($s_1, s_2$))$[0]$ $+$ scores($O_1$) $+$ scores($O_2$) then
      topTotalScores(($S_1[i], S_2[j]$)) $\leftarrow$ (topTotalScores(($s_1, s_2$))$[0]$ $+$ scores($O_1$) $+$ scores($O_2$), ($s_1, s_2$), ($O_1, O_2$));

Get optimal path by backtracking from the pair of last tuples in $S_1, S_2$ through totalTopScores;

$R_{out} \leftarrow$ all pattern occurrences appearing on optimal paths;
Algorithm 17: IndividualSelection  
**Input:** Tune Family $F$, Pattern Set $R_{in}$, Pattern Set $R_{prev}$, $\mathbb{R}$ bias, $\mathbb{R}$ skipCost  
**Output:** Pattern Set $R_{out}$  
**Result:** Selects a set of non-overlapping patterns $R_{out}$ from $R_{in}$, such that each pattern occurs in $R_{prev}$.  

1. **begin**  
2. Mapping(Pattern Occurrence $\rightarrow \mathbb{R}$) scores $\leftarrow$ CalculateScores($F$, $R_{in}$, $R_{prev}$, bias);  
3. Mapping(Tuple $\rightarrow$ Pattern Occurrence) patternBridges;  
4. Mapping(Tuple $\rightarrow$ ($\mathbb{R}$, Tuple, Pattern Occurrence)) topTotalScores;  
5. // Determine all pattern bridges.  
6. for each Pattern Occurrence $O$ in $R_{prev}$ do  
7. end $\leftarrow$ the tuple after the last tuple of $O$, if it exists, otherwise the last tuple of $O$;  
8. patternBridges(end) $\leftarrow O$;  
9. // Compute the optimal path through each song.  
10. for each Song $S$ in $F$ do  
11. topTotalScores($S[0]$) $\leftarrow$ (0, null, null);  
12. for $N$ $i$ from 1 to $|S|$ do  
13. topTotalScores($S[i]$) $\leftarrow$ (topTotalScores($S[i-1]$)[0] $-$ skipCost, $S[i-1]$, null);  
14. for each Pattern Occurrence $O$ in patternBridges($S[i]$) do  
15. start $\leftarrow$ the first tuple of $O$;  
16. if topTotalScores($S[i]$)[0] $<$ topTotalScores(start)[0] $+$ scores($O$) then  
17. topTotalScores($S[i]$) $\leftarrow$ (topTotalScores(start)[0] $+$ scores($O$), end, $O$);  
18. Get optimal path by backtracking from the last tuple in $S$ through totalTopScores;  
19. $R_{out}$ $\leftarrow$ all pattern occurrences appearing on optimal paths;  

Algorithm 18: CalculateScores  
**Input:** Tune Family $F$, Pattern Set $R_{in}$, Pattern Set $R_{prev}$, $\mathbb{R}$ bias  
**Output:** Mapping(Pattern Occurrence $\rightarrow \mathbb{R}$) scores  
**Result:** Computes the quality scores for all patterns occurrences in $R_{in}$ of which the pattern exists in $R_{prev}$, with positive bias towards occurrences in $R_{prev}$.  

1. **begin**  
2. Mapping(Pattern Occurrence $\rightarrow \mathbb{R}$) scores $\leftarrow$ $\emptyset$;  
3. for each Pattern $P$ in $R_{in}$ such that $R_{prev}$ has an occurrence from $P$ do  
4. partPatSpread $\leftarrow$ the number of songs with an occurrence from $P$ in $R_{prev}$;  
5. fullPatSpread $\leftarrow$ the number of songs with an occurrence from $P$;  
6. for each Pattern Occurrence $O$ in $P$ do  
7. fullOccSize $\leftarrow$ the number of tuples in Song($O$) between Start($O$) and End($O$);  
8. scores($O$) $\leftarrow$ $(1 + \text{bias} \times \text{partPatSpread}) \times (\log |O| \times |O|^2 / \text{fullOccSize}) \times \text{fullPatSpread}$;
In our individual selection sub-routine, algorithm 17, we compute an optimal path for each song. For each tuple $t$ in the song, we have to check the previous tuple and all pattern occurrences ending in $t$. Since each pattern occurrence can only end in one tuple, this process is linear in the size of the tune family and the size of the pattern set.

Our most resource-intensive sub-routine is algorithm 16, performing the pair-wise selection process. Just as with the individual selection, we are computing paths. The crucial difference here is that we are computing paths pair-wise and using bridges made of pairs of occurrences. We now get that the number of cells we are computing is quadratic in the size of the tune family, $O(\text{Size}(F)^2)$.

The bridges of the pair-wise selection sub-routine are the most potentially costly element in our algorithm. Before, we only used each pattern occurrence once, but now, we must use each pattern occurrence up to the number of occurrences in its pattern. This number of occurrences for good patterns should be about linear in the number of songs. We also only apply the constant-time information from the occurrences. A problem arises when we have small patterns that occur many times. In the worst case, we could technically run into the problem of having $O(\text{Size}(R_n)^2)$ bridges. Since we can limit our pattern set to quadratic in the size of the tune family, this would still give us quartic running time. Depending on the data, it may be worth filtering out insignificant, overly abundant patterns.
Part IV

Discussion
Chapter 11

Results

In this chapter, we discuss the results of applying our pipeline approach to the same data set as our problem analysis of Chapter 4. We will briefly examine the behaviour of our pipeline in isolation. We will then explore the effect of varying our primary parameters. Finally, we will evaluate how well the results satisfy our requirements.

11.1 Discovered Pattern Properties

In this section, we briefly discuss the general behaviour of our approach. We will mainly look at the immediately apparent properties that discovered patterns display. This will somewhat mirror the analysis done in Chapter 4. Aspects immediately related to our requirements will largely be discussed in Section 11.4.

First and foremost, we see that in most cases, such as in Figure 11.3b, our output is small. The number of patterns remaining after our selection process tends to be comparable to the ground truth. Typically, the greater the number of patterns discovered, the fewer songs each pattern occurs in. We do notice, however, that the number of songs a pattern occurs in tends to be higher than the required number given by the parameters. This is likely due to the clustering task trying to find more pattern occurrences than strictly necessary. The selection task, since it removes patterns rather aggressively, can then end up with a small set of patterns, which allow each pattern to occur in more songs.

In Figure 11.1, we see the purple pattern, which occurs many times in the tune family, as a

Figure 11.1: A large pattern in purple appearing once in the output, with two subsets occurring, undiscovered.
11.2 Exploration of Parameters

In this section, we examine the effects of changing the parameter configurations of our main parameters.

In Figure 11.3, we show what the pattern sets for a tune family, and for details in songs, are like with our default configuration. This will serve as a basis for our reasoning. Our default parameter configuration uses a maximum gap size of 2, allowing rhythmic variation down to a factor 0.7, a time-stretch down to a factor 0.5, a maximum number of embellishments per note of 1 and the demand for patterns to occur in 0.4 of the tune family. We will address our choice of configuration as we discuss each parameter.

We will first discuss the usage of different pattern spread requirements, illustrated in Figure 11.4. When we set the pattern spread parameter to 0.8, illustrated in Figure 11.4b, we get very aggressive filtering behaviour from our pattern selection approach. This is clearly reflected in how very few patterns are present, with one song containing no patterns at all. When we use the higher leniency of setting the parameter to 0.2, we get results similar to our default of 0.4. Our clustering still combines patterns sufficiently to cause significant spread. Our selection, meanwhile, still selects reasonable patterns thanks likely to its scoring function. More leniency given by 0.2 does cause patterns to not spread as much as with 0.4, which is why we stick to the slightly stricter configuration.

The effects of differing gap size without changing how much embellishment is allowed is fairly straightforward. In Figure 11.5, we see that a larger gap size allows for larger patterns, whereas a smaller gap size allows only smaller patterns. The smaller gap size of 1.0 prevents a lot of patterns from being discovered, as notes have to be too close together to be considered for discovery. If we...
CHAPTER 11. RESULTS

(a) A pattern set for a song selected to show key differences in parameter configuration.

(b) A pattern set determined by our default parameter configuration.

Figure 11.3: A comparison between the pattern sets from human sources, and our algorithm with our favoured parameter configuration.

(a) A pattern set determined with lower required spread of patterns.

(b) A pattern set determined with higher required spread of patterns.

Figure 11.4: A comparison between the pattern sets with higher or lower requirements for pattern spread in the tune family.

increase this parameter to 4.0, we see a lot of larger patterns appear, which are not necessarily less valid than with our default value of 2.0. The problem with larger patterns is that these typically take priority in the clustering task, and due to taking up more space cause selection to result in relatively few patterns. This is becomes a problem, as some songs may not have any patterns in the resulting pattern set. Typically, we do not lose out on any important patterns with 2.0 as our configuration, yet we also avoid the disadvantages of allowing all the larger patterns. We may want to change the priority of clustering to something other than pattern length, such that large patterns can be discovered alongside the smaller ones.

The effects of varying the parameter for the number of embellishments per note in patterns are similar to those of varying the gap size. In Figure 11.6, we see results resembling those in Figure 11.5, with somewhat more severe results. The low embellishment value set is 0, indicating that no embellishments can be contained in matching patterns. This almost entirely removes differences patterns in the same cluster can have. The downside is that the number of patterns discovered and the length of these patterns is severely lowered, as seen by Figure 11.6a being nearly devoid of patterns.
CHAPTER 11. RESULTS

(a) A pattern set determined with smaller gap size.

(b) A pattern set determined with larger gap size.

Figure 11.5: A comparison between the pattern sets allowing for smaller and larger gaps in patterns.

Allowing for almost any number of embellishments within the gap size is done using the value 5. This causes a lot larger patterns to be discovered, but we lose a lot of similarity between pattern occurrences. The patterns being too large also causes our selection process to remove patterns from the output more aggressively, causing a relatively small set of patterns. Our default embellishment tolerance is 1, as usually a pattern does not contain more than 1 embellishment between subsequent notes, while still being considered the same pattern. Depending on clustering quality and personal preference, 2 or 3 embellishments should also be reasonable configurations.

(a) A pattern set determined with lower embellishment tolerance.

(b) A pattern set determined with higher embellishment tolerance.

Figure 11.6: A comparison between the pattern sets allowing more of fewer embellishments in patterns.

Our rhythmic variation parameter is a relatively difficult one to decide on a default for. In Figure 11.7, we see what our approach discovers for higher tolerance of rhythmic variation, or the lack of tolerance entirely, in Figure 11.7b. The lack of tolerance for rhythmic variation, given by a ratio of 1.0, requires patterns to vary primarily based on embellishment. This results in relatively few larger patterns between songs, as similarities are cut short during discovery. We do however get an impression of how our algorithm compares in other aspects to algorithms with inherently no rhythmic variation tolerance, which is convenient for analysis.

When we increase the tolerance to allowing a ratio of 0.5, we start discovering large patterns, however with a higher likelihood of lacking proper similarity. This is seen in how the top patterns in Figure 11.7a do not necessarily resemble those in the bottom. Our choice of 0.7 gives us a middle ground in terms of accuracy and discovery. We particularly chose this, because our clustering approach groups together pattern occurrences 2 steps of similarity removed from each other. Applying 0.7 twice over allows most variations to be discovered, while preventing the similarity from becoming too low. This does not completely remedy the problem, as seen in Figure 11.3a. In order to improve on this, we will likely have to be stricter with similarity during clustering. As long as rhythmic variation tolerance is kept fairly lenient, the choice of this parameter is largely
up to personal preference.

(a) A pattern set determined with higher tolerance for rhythmic variation.

(b) A pattern set determined with lower tolerance for rhythmic variation.

Figure 11.7: A comparison between the pattern sets determined with higher or lower tolerance for rhythmic variation.

We now discuss the effect of allowing less or more stretch in the time-domain, using a different tune family that actually has varying tempo. When we use 1.0 as a ratio here, we essentially disallow normalization from performing any stretch on the tune family. In Figure 11.8a, we see the usage of this ratio, and two songs missing patterns entirely. These songs have a stretch in the time domain, as opposed to the rest of the tune family. When we use 0.1 as our ratio, we see in Figure 11.8b that these songs do, in fact, contain patterns. These patterns also show us that the patterns here have larger gaps between notes, evidenced by them being longer. The resulting pattern set of our default parameter 0.5 gives us the exact same results, as songs typically only stretch by such a factor. The downside of using a lenient configuration such as 0.1 is that the normalization task may take longer to complete.

(a) A pattern set determined with less stretch tolerance in the time domain.

(b) A pattern set determined with more stretch tolerance in the time domain.

Figure 11.8: A comparison between the pattern sets allowing less or more stretch in the time domain.

We have many less important parameters working under the hood for our sub-routines. For our exploration, these parameters are fixed. These parameters are considered in the same category...
of configuration as applying different scoring functions for selection, or doing different selection and clustering approaches altogether.

### 11.3 Comparison to Annotated Patterns

In Figure 11.9, we see the tune family overview with annotated patterns of the family from Figure 11.3b. Our discovered patterns display a similar, yet not identical spread when compared to the annotated patterns. We can see that some patterns match directly, for example, the beige annotated pattern matches the dull orange pattern from our discovered patterns. The discovered patterns, although they display an ordering, are not as consistent on this front as the annotated patterns. This is likely due to how our clustering approach tends to sometimes combine the wrong patterns together. With regards to mimicking human annotation, we still require improvements, but we are coming closer to doing so.

![Figure 11.9: The annotated patterns of the same tune family as in Figure 11.3b.](image-url)
11.4 Evaluation of Problem Solving

A large portion of our established requirements in Chapter 4 have been met. We will first discuss which requirements have been met fully, then the requirements that have been partially satisfied, and finally the requirements we still need to meet.

Requirements IF01.0, IF01.1, and IF02.0 have been satisfied by our selection approach. All our output contains non-overlapping patterns, and by extension a pattern set linear in the size of the tune family. The lack of overlap also immediately makes sure that no multiple variants of the same pattern are in the output. We can, however, do better on clustering, as we do see different patterns representing highly similar sets of notes.

We have also met requirements IF04 and IG04, as all discovered patterns are highly dense. No discovered patterns contain gaps or miss notes in their time range. It is however the case that in satisfying requirement IF07 and allowing embellishments, we do get patterns that have occurrences that do not match on a significant portion of their notes.

Thanks to the strictness of our pattern selection approach, we have met requirement IG06. All patterns in the final output occur in at least a requested portion of the tune family. This strictness does have its downside in that it typically causes many patterns to be filtered out. As such, the larger the portion of songs a pattern should appear in, the more likely it is that the output will contain only 1 pattern.

We see in Figure 11.10 and from looking at the output files, that we meet requirement IF03. Patterns such as the green pattern do not match properly on chromatic pitch, but they do on morphetic pitch, and we do pick up on that. We specifically discovered this pattern while disallowing embellishments, so the only possible way of detecting it would be through morphetic pitch.

The stretch in time-domain, requirement IG02, due to being limited to full songs rather than individual patterns, does not appear to cause any negative side-effects. Songs with patterns that would otherwise be missed are treated properly, as seen in Figure 11.8b. Depending on the exact parameterization used for the normalization task, the correct time-stretch may stay undetected. This is, however, easily remedied by trying a slightly different configuration.

Through our theoretical performance analysis, we have argued that our algorithms run in low polynomial time, as per requirement IG01. Although the heuristic nature of the algorithms allow for potentially quartic running time, in practice this largely lands in low super-quadratic range. The running time is largely helped by the currently quick approach to clustering, and its side effect of removing a large portion of the originally discovered data. This allows the selection task to largely avoid running in its potentially quartic time.

Although we have discovered all manner of varying pattern occurrences, as required by re-

Figure 11.10: A pattern in green shows two transposed occurrences with difference in their relative chromatic pitches, but the same relative morphetic pitches.
requirements IF05, IF06 and IF07, this is not done perfectly. Patterns such as the orange pattern in Figure 11.7a show that our clustering is still too lenient with pattern similarity. We get pattern occurrences with embellishments, different pitch and different rhythm, but in allowing all of these, patterns in the same cluster can vary too much.

We also run into the problem that pattern occurrences that should be in the same pattern end up in different patterns. Attaining the right balance in occurrence similarity within a pattern and selecting occurrences appropriately should be largely a matter of improving the clustering approach.

Patterns occurring in consistent position and order across the tune family, as required by requirement IG05.0 and IG05.1, is largely covered. We tend to see the order be consistent, with incorrect order largely being attributed to the lack of additional occurrences surrounding the selected patterns. Positioning of patterns is also seen to be highly consistent, with the same slight flaws as the ordering.

As seen in most figures in this chapter, when patterns occur in a song, they typically occur multiple times. This slightly clashes with requirement IG03, as the multiple occurrences of one pattern get selected in favour of having a different pattern occur in more songs. This is likely due to our selection approach. When we iterate over selected patterns, patterns with presence in the majority of the tune family get prioritized. This can cause these patterns to be prioritized multiple times within the same song, causing other patterns to be filtered out.

Due to potential complexity, we could not yet meet requirement IG07. The extent to which this is harmful to the resulting pattern sets is not yet quantifiable. As far as our pattern selection goes, there are a lot of factors at play in what patterns get selected. As such, the problem of pattern selection in its entirety is somewhat of a separate problem space.

11.5 Conclusion

Our approach largely covers requirements set, and can successfully mimic parts of the ground truth. In particular, the tasks done by the discovery and normalization tasks in our pipeline appear to deliver accurate results. The clustering and selection tasks have some problems, which cause inconsistencies in the resulting output. If these inconsistencies were to be remedied, the resulting output would improve significantly.

The parameter selection provides a highly extensive set of possible configurations, many of which can cause the output to lower in quality. Selecting the right parameters can become a sensitive task, but a single selection of parameters typically works well for several different tune families. The pipeline also allows for more intricate configuration in the form of slightly changing heuristics all together. We can likely explore these configurations long-term to find a way of consistently delivering good results.
Chapter 12

Conclusion

We have developed an visual analysis tool for determining the qualitative properties of patterns in tune families. Based on the analysis of existing pattern discovery data with this tool, we determined a set of requirements to be met for high quality pattern discovery in tune families. We then split the requirements over four major tasks: normalization, discovery, clustering and selection. For each of these tasks, we have provided a solution. The quality of these solutions and the surrounding framework have been evaluated using our analysis tool. Our analysis lead to the conclusion that our requirements were accurate, and our solutions at least partially met all but one of the requirements.

In going through these steps, we have met our research goal of finding a way to properly perform pattern discovery on tune families, while mimicking qualities of pattern discovery by human experts. Based on our results, we claim that our set of requirements is a reasonable approximation of the qualities of human pattern discovery. These requirements include allowing for rhythmic variation, ignoring tempo differences between songs, valuing positioning of patterns and accommodating minimal embellishment within patterns. Our pipeline splitting the requirements across multiple tasks served as a good way of lowering the complexity of the problem space. Based on our results, it seems feasible to algorithmically mimic the qualities of pattern discovery by human experts, although we do not have strict evidence that it is.

Our visual analysis tool consists of a number of large and smaller views, each of which can be filled in with some representation of songs and patterns. We include various options for filling these views, in the forms of views mimicking the intuitive geometric layout used in sheet music, and a more global overview of multiple songs or sets of patterns. Each of these views can indicate patterns directly or form a heat map of pattern frequency per note. The global overview can also vary on whether it shows the same song with patterns discovered under different settings, or a set of songs with all the same discovery settings. This analysis tool has been verified through an expert study, to be a useful insight into discovered patterns.

Our algorithmic solution returns a non-overlapping set of patterns for a given tune family. It accomplishes this by first discovering simple patterns as frame of reference, and adjusting each song’s dimensions to a common ground based on these patterns. We then perform pattern discovery through geometric structure, in which we find pairs of tuples with tuples in their neighbourhood that have similar relative position. We then use these matching tuples to build a set of sequential matching tuples, forming a pattern. We then cluster these patterns based on containing an occurrence within the same time-range as one another. Finally, we use a dynamic programming approach to select a small set of tuples. We do this through the creation of a path through a grid mapping tuples of two songs to each other, in which pairs of pattern occurrences function as favourable bridges for the path.
12.1 Improvements and Future Work

Since our approach allows for a large amount of configuration, and our analysis is limited to a single data set, we should aim to further explore the results our pipeline can produce. This extended evaluation should be applied to different data sets with different priorities on what pattern qualities are most desirable. We should also further explore the entire set of parameters, and test different revisions of the scoring and ordering heuristics used in the discovery, clustering and selection tasks. It is also worth exploring what happens when we do not use the same parameter across multiple tasks, and adjust the parameters independently for each task in the pipeline. Experimentation during development has shown that requiring a larger coverage of the tune family in clustering is better for selection, for instance.

Our pipeline approach does its job well in terms of splitting the responsibility of all requirements over multiple tasks. Improving the individual tasks would be the next step in proper development. The first task to be improved is the clustering task. An improved clustering approach should be more careful with the assumptions of similarity among patterns. We should also consider clustering patterns into multiple clusters.

There is also a possible alternative for our combination of normalization, discovery and clustering, in which we would focus on pattern positioning early on. This would involve also performing normalization to translate to a common starting point in the time-domain. We can then limit pattern discovery between different songs to only patterns around similar positions. We should then keep normal discovery for patterns within the same song. In clustering, we would then cluster patterns around the same position, and then cluster the clusters based on recurrences within songs.
Discovering Audibly Recognizable Patterns within Tune Families 93

Bibliography


List of Figures

1.1 The first four measures of two songs in the same tune family, shared patterns according to human experts marked by coloured bounding boxes. 3

2.1 The primary two representations of music. 5
2.2 One octave worth of keys of the piano keyboard. The same pattern is repeated, with repetition having a number to distinguish them. 6
2.3 An example of a 4/4 time signature, and two measures. 7
2.4 The first four measures of Für Elise, as sheet music and as geometric representation. 8
2.5 Three steps of the SIAR algorithm illustrated. 9

3.1 The file system architecture to read songs from. The dotted folders might not be present. 15
3.2 The file system architecture to read patterns from. The dotted folders might not be present. IntraOpusDiscoveryTask has the same architecture as InterOpusDiscoveryTask. 16
3.3 The piano roll illustration of a song, including marked patterns and integrated song player. 18
3.4 The piano roll illustration of a song, as heat map. 18
3.5 The illustration of a tune family, marking the pattern positioning only, as simplified song representation. 19
3.6 The main screen of the tool, contains large and small panels to put visualizations in. 19
3.7 The container filler, the window through which all view changes are made. 20

4.1 The MGDP algorithm illustrating the problem of finding too many patterns. 23
4.2 The annotated motifs illustrating an instance of a pattern, in light green, being exact in morphetic pitch, but not chromatic pitch. 24
4.3 The COSIATEC algorithm illustrating the problem of finding wide, sparse patterns. 25
4.4 The annotated patterns over a tune family, showing a similar pattern distribution for each song. 27
4.5 Three different songs displaying the same annotated pattern in different forms. 27

6.1 The algorithmic solution pipeline. 38
6.2 Two occurrences of what may be considered the same pattern. The black notes are part of the pattern, despite the middle having different timing. The grey notes embellish the occurrences. 40
6.3 Two occurrences of what may be considered the same pattern. The black notes are part of the pattern, including the bottom note as embellishment in the right pattern. The grey notes are more of an extension to the pattern, and may be removed if it occurs rarely. 41

7.1 Modifying the pitch of the song on the left, to match the one on the right. Matching notes are in black. 44

Discovering Audibly Recognizable Patterns within Tune Families 95
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.2</td>
<td>Modifying the tempo of the song on the right, to match the one on the left. Matching notes are in black.</td>
<td>45</td>
</tr>
<tr>
<td>7.3</td>
<td>Applying both normalizations.</td>
<td>46</td>
</tr>
<tr>
<td>8.1</td>
<td>The process of Algorithm 9 applied to a song.</td>
<td>56</td>
</tr>
<tr>
<td>8.2</td>
<td>A visual representation of the innermost loop of Algorithm 10.</td>
<td>57</td>
</tr>
<tr>
<td>8.3</td>
<td>A visual representation of Algorithm 11 constructing a pattern with occurrences across two songs.</td>
<td>59</td>
</tr>
<tr>
<td>9.1</td>
<td>A visual representation extracting a cluster from a set of patterns containing the same tuple from Song 1. Each bounding box is a song, with the thick line denoting a pattern occurrence in it. If a bounding box connects to another, these denote occurrences of the same pattern. The thick vertical lines denote the extracted, cropped sub-segments of patterns. The vertical line intersecting the patterns indicates the tuple we are extracting from.</td>
<td>67</td>
</tr>
<tr>
<td>10.1</td>
<td>A visual representation of the pair-wise and individual selections for Song 1. The thick line segments denote pattern occurrences, the dashed line denotes an optimal path.</td>
<td>74</td>
</tr>
<tr>
<td>11.1</td>
<td>A large pattern in purple appearing once in the output, with two subsets occurring, undiscovered.</td>
<td>82</td>
</tr>
<tr>
<td>11.2</td>
<td>A large pattern in dark purple appearing twice in the song, in one of which is an approximate subset.</td>
<td>83</td>
</tr>
<tr>
<td>11.3</td>
<td>A comparison between the pattern sets from human sources, and our algorithm with our favoured parameter configuration.</td>
<td>84</td>
</tr>
<tr>
<td>11.4</td>
<td>A comparison between the pattern sets with higher or lower requirements for pattern spread in the tune family.</td>
<td>84</td>
</tr>
<tr>
<td>11.5</td>
<td>A comparison between the pattern sets allowing for smaller and larger gaps in patterns.</td>
<td>85</td>
</tr>
<tr>
<td>11.6</td>
<td>A comparison between the pattern sets allowing more of fewer embellishments in patterns.</td>
<td>85</td>
</tr>
<tr>
<td>11.7</td>
<td>A comparison between the pattern sets determined with higher or lower tolerance for rhythmic variation.</td>
<td>86</td>
</tr>
<tr>
<td>11.8</td>
<td>A comparison between the pattern sets allowing less or more stretch in the time domain.</td>
<td>86</td>
</tr>
<tr>
<td>11.9</td>
<td>The annotated patterns of the same tune family as in Figure 11.3b.</td>
<td>87</td>
</tr>
<tr>
<td>11.10</td>
<td>A pattern in green shows two transposed occurrences with difference in their relative chromatic pitches, but the same relative morphetic pitches.</td>
<td>88</td>
</tr>
<tr>
<td>A.1</td>
<td>The main screen</td>
<td>101</td>
</tr>
<tr>
<td>A.2</td>
<td>The container filler</td>
<td>102</td>
</tr>
<tr>
<td>A.3</td>
<td>The piano pattern view</td>
<td>104</td>
</tr>
<tr>
<td>A.4</td>
<td>The piano heat map view</td>
<td>105</td>
</tr>
<tr>
<td>A.5</td>
<td>The family pattern view</td>
<td>106</td>
</tr>
<tr>
<td>A.6</td>
<td>The family heat map view</td>
<td>106</td>
</tr>
<tr>
<td>A.7</td>
<td>The parameter pattern view</td>
<td>107</td>
</tr>
<tr>
<td>A.8</td>
<td>The parameter heat map view</td>
<td>107</td>
</tr>
<tr>
<td>A.9</td>
<td>The single pattern view</td>
<td>108</td>
</tr>
<tr>
<td>A.10</td>
<td>The single pattern family view</td>
<td>108</td>
</tr>
<tr>
<td>A.11</td>
<td>The pattern strips view</td>
<td>109</td>
</tr>
</tbody>
</table>
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Informal requirements from fundamental problems with pattern discovery.</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>Informal requirements from problems with pattern discovery on tune families.</td>
<td>28</td>
</tr>
<tr>
<td>5.1</td>
<td>Formal requirements without significant formal component, based on informal requirements, with related identifiers.</td>
<td>35</td>
</tr>
<tr>
<td>5.2</td>
<td>Interplaying formal requirements based on informal requirements, with related identifiers.</td>
<td>35</td>
</tr>
<tr>
<td>5.3</td>
<td>Independent formal requirements based on informal requirements, with related identifiers.</td>
<td>36</td>
</tr>
</tbody>
</table>
Listings
Appendix A

Visualizer Instructions

A.1 Main Screen

The main screen shows a number of empty panels of varying sizes.

Right click:

- Opens up a window we call the container filler, with the clicked panel as target container to fill. If there is already data present in the clicked panel, this, including highlighted data, is used to pre-select values in the container filler for ease of use.

- If the "Copy from this" button was clicked on the container filler, the clicked panel will receive a copy of the previously targeted panel.

- If the "Copy to this" button was clicked on the container filler, the clicked panel will send a copy of itself to the previously targeted panel.

- If the "Swap" button was clicked on the container filler, the clicked panel will swap its contents with the previously targeted panel.
A.2 Container Filler

Figure A.2: The container filler

Top buttons:
- Copy to this: As described under Main Panel, this button closes the container filler, and copies the contents from the currently targeted panel to the next right-clicked panel.
- Copy from this: As described under Main Panel, this button closes the container filler, and copies the contents from the next right-clicked to the panel currently targeted panel.
- Swap: As described under Main Panel, this button closes the container filler, and swaps the contents of the panel currently targeted panel with the next right-clicked.
APPENDIX A. VISUALIZER INSTRUCTIONS

Top three combo boxes:
- The top box selects the data set being used, derived from the file system structure set up using Peter Boot’s pattern discovery pipeline.
- The second box selects the tune family in the selected data set, these group together songs with the same name but different ID.
- The third box selects the specific song from the selected tune family to be used.

Bottom four combo boxes:
- The top box selects the scope of the pattern discovery being done, this can either be "intra opus", discovering patterns within the song itself, "inter opus", discovering patterns across songs in the tune family, or "annotated motifs", inter opus discovery done by actual musicologists.
- The second box selects which algorithm is being used for discovery. In the case of "annotated motifs", this is the only 'algorithm'.
- The third box selects the parameter configuration used by the selected algorithm, which may differ per algorithm. Again, "annotated motifs" has no different configurations.
- The final combo box selects a specific pattern ID from the pattern set present for the selected song, algorithm, and parameters.

Components between the combo boxes and bottom buttons:
- The checkbox, if checked, makes the bottom buttons use drag-and-dropped files as target data, rather than the data specified by the combo boxes. This data does lack in some functionality, since there is no file structure taken into account to infer more information.
- The left text field lists the files dropped that contain songs. When using all data from this list, a song is always paired with the pattern set of the same index on the right list. When using only one song-pattern set pair, the last elements of these lists are paired.
- The right text field lists the files dropped that contain pattern sets.

Bottom buttons:
- Piano Pattern View: Uses the selected song, algorithm and parameterization to display a Piano Pattern View on the targeted panel.
- Piano Heat Map View: Uses the selected song, algorithm and parameterization to display a Piano Heat Map View on the targeted panel.
- Family Pattern View: Uses the selected tune family, algorithm and parameterization to display a Family Pattern View on the targeted panel.
- Family Heat Map View: Uses the selected tune family, algorithm and parameterization to display a Family Heat Map View on the targeted panel.
- Parameter Pattern View: Uses the selected song and algorithm to display a Parameter Pattern View on the targeted panel.
- Parameter Heat Map View: Uses the selected song and algorithm to display a Parameter Heat Map Pattern View on the targeted panel.
- Single Pattern View: Uses the selected song, algorithm, parameterization and pattern to display a Single Pattern View on the targeted panel.
- Single Pattern View: Uses the selected song, algorithm, parameterization and pattern to display a Family Pattern View on the targeted panel.
- Pattern Strips View: Uses the selected song, algorithm and parameterization to display a Pattern Strips View on the targeted panel.
A.3 Views

A.3.1 General notes on views

Across the various views, there are a couple of properties that are commonly shared:

- Views illustrating the notes of a single song show the notes relative to the song’s pitch-range and time-range. This means that songs that start before the count of 1, have the left-most note representing the first note, not the note on the count of 1.

- Views illustrating the notes of a single song also come with a song player. Left-clicking the common representations of the play, pause and stop buttons makes the player act accordingly. The blue rectangle with the red stripe indicates how far the song is in playing, and can be moved as a seekbar. The seekbar and played music match the notes displayed at the same horizontal positioning.

- Views marking patterns in coloured bounding boxes allow left-clicking of these bounding boxes to highlight and select pattern occurrences. The occurrence clicked becomes brighter, while all occurrences not belonging to the same pattern become darker. The patterns also play when clicked. A pattern selected this way will be used by the container filler to pre-select.

- Views illustrating heat maps use grey for notes that occur in no patterns, green for notes that occur in few patterns, and red for notes that occur in many patterns. The reddest notes simply have the most occurrences for the data illustrated, and scale according to the data it is portrayed with.

- Every instance of a song and pattern set illustrated is transparently overlayed with the name of the song, the scope, algorithm and parameterization of the pattern set, and in the case of specific patterns, also the pattern ID.

A.3.2 Piano Pattern View

Figure A.3: The piano pattern view
This view simply shows a single song and marks patterns from the associated pattern set with colored notes and bounding boxes. In the event that a note falls into multiple patterns, this note is colored in divisions of different colours, to a maximum of 5, after which a white strikethrough denotes too many patterns to visualise.

A.3.3 Piano Heat Map View

This view shows a song the same way as the Piano Pattern View, but instead of marking the patterns, it indicates on the notes how frequently each note occurs in patterns, relative to the maximum number of times any note occurred, for a given pattern set.

A.3.4 Family Pattern View

This view illustrates a tune family by showing each song in the family on its own row, with just the bounding boxes of the patterns visible, for a given algorithm and parameterization.

A.3.5 Family Heat Map View

This view, as the Family Pattern View, shows a song on each row, denoting all notes without vertical indication of pitch. As with the Song Heat Map View, each note is colored depending on how often it occurs in patterns.

A.3.6 Parameter Pattern View

This view shows essentially the same as the Family Pattern View, except that rather than showing all songs in a family using pattern sets from the same configurations, it shows a single song in repeated for every possible parameterization of an algorithm.

A.3.7 Parameter Heat Map View

Similarly to the Parameter Pattern View, this view shows a single song using all different parameterizations of an algorithm, but this time as a heat map, similar to the Family Heat Map
APPENDIX A. VISUALIZER INSTRUCTIONS

A.3.8 Single Pattern View

This view takes into account the specifically selected pattern from the ContainerFiller. The view illustrates exactly the same as the Song Pattern View, except that only one single pattern is shown, instead of the full set.
APPENDIX A. VISUALIZER INSTRUCTIONS

A.3.9 Single Pattern Family View

This view takes into account the specifically selected pattern from the ContainerFiller. The view illustrates exactly the same as the Family Pattern View, except that only one single pattern is shown, instead of the full set. This only works correctly for annotated motifs and inter opus discovery tasks, as the pattern ID does not map to the same pattern across songs for intra opus discovery tasks. Whenever a song in the family does not have the desired pattern present, all other patterns are shown instead.
APPENDIX A. VISUALIZER INSTRUCTIONS

A.3.10 Pattern Strips View
This view shows rows similar to the Family Pattern View, but displays only the pattern information of a single song, as seen in the Song Pattern View. Each row here denotes a single pattern, as done in the Single Pattern View.

A.3.11 Warning
The algorithm SIARCT_CFP does not have any data under the InterOpusDiscoveryTask. If this algorithm and scope is used, it will indicate no patterns, this is not representative of the algorithm itself.

108 Discovering Audibly Recognizable Patterns within Tune Families
Figure A.11: The pattern strips view
Appendix B

Visualizer Evaluation Tasks

In this chapter, we list the tasks used for evaluation of the pattern viewer visualization tool. After each task, we note an estimate of the time taken on said task. These estimates proved to be accurate for users who quickly got the hang of the tool.

1. Find a tune family with at least 3 different annotated patterns. (Approx. 1 min)

2. Find two songs of the same tune family, one of which not featuring an annotated pattern the other does. Is the annotated pattern indeed not present? What is different about the song that makes it miss the pattern? (Approx. 2 min)

3. Find an algorithm that returns too many patterns. Try to find a parameterization of this algorithm such that there are a reasonable number of patterns (think a similar magnitude to the annotated motifs). Investigate the results using tune family, and one song specifically from that family. Can this algorithm find a proper number of patterns at all? If so, are the patterns found at all desirable or interesting? Keep the selected data for task 4. (Approx. 5 min)

4. For the algorithm, family, and song used in task 3, pick a parameterization that still returns too many patterns. Cherry pick 3 desirable patterns and pick another 3 patterns at random. Does the algorithm find good patterns? Are randomly selected patterns discovered by this algorithm at all desirable? Did cherry picking on a less strict parameterization allow you to find better patterns than when the algorithm found few patterns in task 3 (if it did at all)? (Approx. 5 min)

5. Find a pair of songs that has highly similar distribution of annotated patterns, but entirely different patterns discovered by the COSIATEC algorithm, running on some parameterization as an inter opus discovery task. How do the discovered patterns differ from the annotated patterns? Can the discovered patterns of one of the songs be seen in the other, despite COSIATEC not having found them? (Approx. 5 min)

6. Take one song, for the MGDP algorithm running an inter opus discovery task, find out if there is a parameterization, such that on average, patterns roughly match with the annotated patterns. If there is not, do the patterns center around some other areas you would consider good alternatives to the annotated patterns? (Approx. 5 min)

7. Find a sequence based discovery algorithm (MGDP, PatMinr, MotivesExtractor) and a parameterization that produces an overseeable number of patterns, think similar numbers to the annotated motifs. Find a song for which the annotated motifs indicate recurring patterns within the same song, such that at least one of the pattern occurrences is shifted in pitch. Does the discovery algorithm find part of the shifted recurrence? (Approx 8 min.)
APPENDIX B. VISUALIZER EVALUATION TASKS

8. With the same algorithm as task 7, find a pair of songs in the same family, such that the songs share annotated motifs, but they are stretched in duration with respect to each other, and both motifs have minimal embellishments. Does the discovery algorithm, applied as inter opus discovery task, recognize (part of) these motifs as the same pattern? (Approx. 5 min)