KU LEUVEN

FACULTY OF SCIENCE

Intra-opus Discovery of Polyphonic Patterns Using a Sequential Approach

Wout MESKENS

Supervisor: Prof. Dr. Marie-Francine Moens
Mentor: Dr. Juan Carlos Gomez Carranza
Assessor: Dr. Ivan Vulic
Assessor: Dr. Tias Guns

Thesis presented in
fulfillment of the requirements
for the degree of Master of Science
in Applied Informatics

Academic year 2014-2015

Foreword

To write a thesis is a huge task. It is easier to complete this task if you have support from people around you. I am lucky that I have had such people around me. So first of all, I want to thank my parents and friends for giving me support during my time at the university. I also want to thank my mentor Juan Carlos Gomez Carranza and supervisor professor Marie-Francine Moens for their guidance and their time. I really appreciate all the help and advice you have given me. I also really want to thank Jaap Le Mair for proofreading my thesis and giving me feedback. And a special thanks goes to Rielke Le Mair for helping me to keep believing in myself. Last but not least, I want to thank in advance my assessors for spending their time reading my thesis.

Contents

Abstract

Repetitions are one of the most important aspect in the structure of music. There are many types of repetitions in music; patterns, themes, sections, ... It would be very useful if these repetitions can be automatically found, because it would help a lot in analysing music. Despite its importance, there has not been a lot of research about finding repetitions in music. In an attempt to give it more attention, the retrieval of patterns in music is added as a task to Mirex [4]. Mirex is a platform that makes it possible to exchange results of music information retrieval tasks.
One of the goals of my thesis is to implement a system that follows the rules of the task and then compare my results with the results of participants from a previous year. I decided not to start from scratch but improve an existing system. I chose the PatMinr algorithm which is developed by Olivier Lartillot [17]. I chose this pattern because it showed potential and because it could only solve the monophonic task. I also saw opportunities to attack its problems, like the slow runtimes and the bad structure. It was interesting if it would be possible to improve the algorithm, and to adapt it so it can also solve the polyphonic task.
I implemented the algorithm based on a better architecture using object oriented techniques. I introduced new concepts like sequences and times into the structure to make it more general so it could also handle the polyphonic task.
At the end, I created a program which is very different from the original algorithm. The biggest similarity and also the most important property of both systems is that they use a one pass approach. They both go trough all notes once and while they go trough them, they extend some kind of pattern library. This pattern library contains all patterns that were found in the notes of the musical file. My system is evaluated using evaluation measures that are developed for the task. This makes that my system can be compared with systems that have already participated in the task because the results are publicly available. I have searched for patterns in the monophonic task and in the polyphonic task. The evaluation revealed that the quality of the output of my system is comparable with the participants of the task, which all are current state-of-the-art systems. The system has still some weak points; it can only find monophonic patterns in the polyphonic task and the system can not find all occurrences. These weak points can be solved rather easily by adding extensions to the system. The real big advantage of the new system is that it is very fast. It is multiple times faster than the fastest system, while having the same quality of results.
My system has a good structure which makes that it is possible to extend the system rather easily in the future and solve the problems of the current system.

List of Figures

List of Tables

Chapter 1

Introduction

There are a lot of patterns that can be found in music. These patterns are called motifs, themes and sections. Often these patterns are the parts that define a musical piece. For example, almost everyone will recognize the pattern shown in figure 1.1 as a part of the fifth symphony of Beethoven when it is played on a piano.
This is an intuitive example why repetitions are a crucial part of music. Schenker [25], for example, said that repetitions are the 'basis of music as an art'. Despite this being such an important part of a musical piece's structure, it is a rather neglected field in Music Information Retrieval (MIR). It would therefore be very useful if there is a program that could automatically find all patterns and return the structure of a musical piece.
At the moment there are systems that can find repetitions in musical pieces, but these systems are still worse than humans. The main reason behind this is that there is currently not a perfect definition for patterns. Some repetitions are perceived by humans as important. These patterns will be recognized by almost everyone. Despite that people can recognize patterns, they can not define what a pattern exactly is.
There are multiple problems when searching for themes and sections, like finding all possible patterns, which not only can be shifted in time, but also transposed. Another problem is filtering the important patterns from all found possible pattern. A program that could find all the patterns that are important to human listeners, could be very useful for analysing the piece and maybe could be used in the future to help composing new pieces.
The notion that repeated patterns and parallelism in music is important for the analysis, has been said by multiple people like Cook [8] and Lerdahl and Jackendoff [12].

1.1 Mirex

Mirex (Music Information Retrieval exchange) is a platform that is used to exchange multiple solutions for different music information retrieval tasks. These solutions are open-source and are compared with other solutions by providing the platform evaluation methods that every participant needs to execute on the submitted solution. Because these solutions are open-source, people can use different parts from different solutions to improve their own solution. Therefore this will help with improving the speed of advancement in the Music Information Retrieval field.// My thesis is inspired on the 2015 Mirex task "Discovery of Repeated Themes & Sections" [4]. The task is explained in detail in chapter 2.

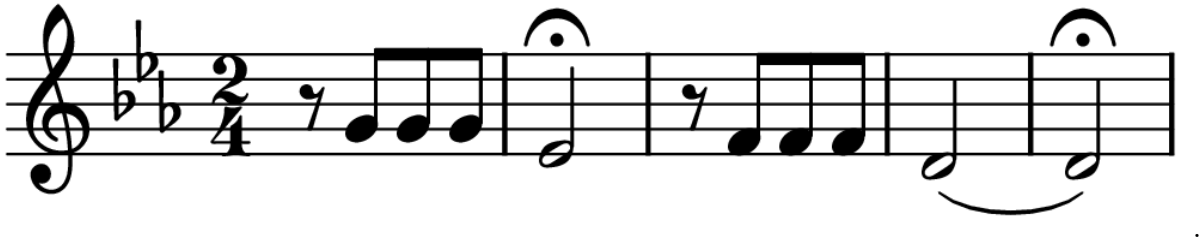

Figure 1.1: Pattern in fifth symphony of Beethoven

1.2 History

Themes and sections have always been a part of music. The examples I use in this thesis are western classical music tunes, which first appeared in the sixteenth century. Repetition has been used since the beginning in almost every classical piece, because it sounds good. In multiple styles of classical music, the structure is mainly based on parts of music that are repeated multiple times in some form in several parts of the musical piece. In 1950, Barlow and Morgenstein [1] composed a dictionary of known musical themes in classical music. They were the first to do something like this and they needed to look for patterns manually or need to use annotations made by the composer.

1.3 Pattern searching systems

The first automated pattern discovery systems appeared in the early 2000's. Most of these systems were still rather basic and tried different strategies to discover the patterns. There is still not a strategy that is accepted to be the "best". 2013 was the first year when the pattern discovery task was published as a Mirex task. This is also an indicator that the task is still rather new and has not received that much attention in comparison to some other MIR tasks.
Two of the early pattern discovery systems were developed by Hsu, Liu and Chen [11] which first find small patterns and then combine these patterns by using a correlative matrix and a string-join method. There have been some systems that used segmentation to find patterns in the musical pieces. When using this, especially when searching for polyphonic patterns, segmenting the piece and using these pieces to structure the music gave good results for a basic algorithm. Therefore there are still some methods that use these techniques and try to improve these techniques. The main advantage of these systems is that they work with both audio and symbolic input. Some examples of systems that use segmentation are: Cambouropoulos [2], and some participants of the Mirex 2014 task. These participants were Nieto and Farbood [23] and Meredith et al. [21]. Meredith has also developed a point-set algorithm [19] and an extension of this point-set algorithm that uses compression [20]. Tom Collins, the publisher of the Mirex task, has also done some important research on finding structure in music. He has, for example, developed an extension on the point-set algorithm from Meredith [6] and has done some research on pattern importance.

1.4 Structure of the document

My thesis is based on a Mirex task. I have therefore explained the Mirex task in chapter 2. It gives some general information about the task and explains what kind of problems the participating systems need to solve. Chapter 3 explains the PatMinr algorithm. Because I used PatMinr as the basis for my program, it is necessary to know what I have started with. I explain the different components and the algorithms they contain in chapter 4. The structure of the program, that contains an overview of the entire algorithm and the architecture of the program, is explained in chapter 5. Because I put effort in designing a good architecture for the system, I want to prove that this has some advantages. I do this by explaining the multifunctionality of the system and an example how easy it is to extend the algorithm. The explanation of the extendability of the system can be found in chapter 6. To evaluate the output of the program, evaluation measures are used. These measures are explained and the results of the program are compared with results of the participants of the Mirex 2014 task. All the information about the evaluation of my algorithm can be found in chapter 7. A summary about my thesis can be found in the conclusion, which can be found in chapter 8.

Chapter 2

Mirex task

This thesis is based on the 2014 Mirex task "Discovery of Repeated Themes & Sections" [4]. I chose this task because it is a rather new task so it is possible to make more advancements than in an older task. It is a task that is published yearly since 2013. The task is written by Tom Collins, who already has done research on pattern finding in music. Some of the insights in this thesis also come from the PhD thesis of Tom Collins [3] which gives a good overview of the current status of pattern finding algorithms. They also give a detailed explanation about a self-developed algorithm.

2.1 Explanation of the task

The goal of this task is to develop an intra-opus theme and section discovery algorithm. Intra-opus theme and section discovery is the discovery of all themes and sections inside a musical piece. The counterpart is inter-opus discovery where these elements are found in different musical pieces. Not all patterns need to be returned, only the "important" patterns like the themes. A theme is a pattern that can be recognized by people and helps define the structure of the musical piece. A pattern is defined in the task as "a set of ontime-pitch pairs that occurs at least twice in a piece of music". All the information in the data needs to be used to get a better algorithm. Because the symbolic version contains all the exact note information, this needs to be used instead of using a segmentation task that only searches for similarities between parts of the piece but does not compare single notes.

2.1.1 Sub-tasks

There are 4 different sub-tasks published wherefore people can submit a solution. There are two audio tasks which use audio as input. They are no part of my thesis. There are also two sub-tasks I will discuss in my thesis. These sub tasks use symbolic input which is a symbolic representation of the sheet music. The two symbolic sub-tasks work with monophonic and polyphonic music and will generally give better results. Especially using polyphonic music, because the conversion from audio to sheet music is not perfect.

2.1.2 Main Problems

When finding patterns in music, there are two main problems that need to be solved to get a decent algorithm. The first one is the finding of all possible patterns in the musical piece. It is impossible to exhaustively find all patterns because there are a lot of notes in a musical piece and patterns can possibly be rather long. The occurrences of a pattern does not need to be exactly the same. The occurrences can also be shifted in time or be transposed, which makes it even harder to recognize all possible occurrences of all patterns. Therefore the constructed algorithm needs to use a smart way to find all possible patterns and their occurrences.
The second problem is that not all possible patterns are "important" patterns. The possible patterns need to be filtered so only "important" patterns remain. It is also quite hard to define the filter parameters. Collins, Laney, Willis, & Garthwaite [7] for instance, have suggested using the frequency of the patterns or the length of the pattern.

Examples of inexact occurrences and patterns Figure 2.1 illustrates the different types of patterns that can occur in a musical piece. This figure was also used to help explain the results of the Mirex task [5]. By analysing the short musical piece, it is clear that occurrences can differ from each other in multiple ways. One of the main problems is to make the system capable of handling these deviations. There are therefore multiple ways how occurrences of the same pattern can be different:

- P2 is the same pattern as P1, but it has been transposed up one note.
- R2 is the same pattern as R1, but there is an upbeat that is part of R1, but not part of R2

Figure 2.1 represents a piece which is composed by Mozart. Bar A shows the sheet music of this piece. Mozart has indicated on his sheet music what the different sections and themes are. These different themes and sections can be seen in bars B-D. This piece shows that the structure of patterns is not always the same. Especially in polyphonic music, it is possible that patterns are very different. Some examples how patterns can differ from each other:

- The size can be different. Q1 is bigger than P1.
- The patterns can contain a different amount of voices. Q1 only contains one voice, P1 contains two.

Figure 2.1 also shows that patterns can overlap, which will also make it harder to find all the different patterns. For example Q1 overlaps with P1 and P2. This gives a small indication that the algorithm to be developed needs to be flexible enough to find all these patterns and occurrences.

2.2 Mirex data set

Mirex provides a small annotated database which can be used to evaluate the algorithms. The name of the database is "Johannes Kepler University Patterns Development Database (JKUPDD)". The JKUPDD is a representative small sample from the database that is

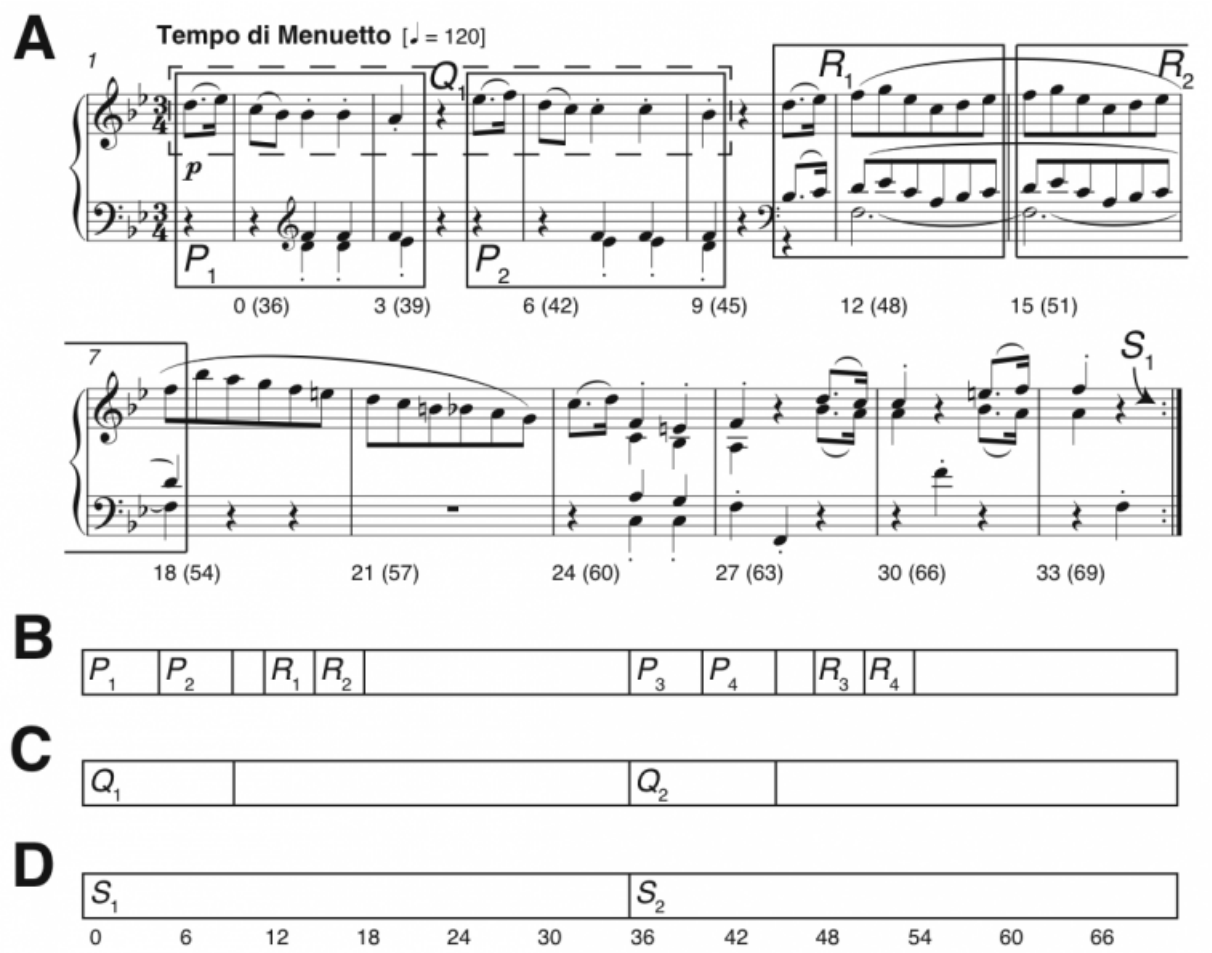

Figure 2.1: Pattern discovery and segmentation. (A) Bars 1-12 of Mozarts Piano Sonata in E-flat major K282 mvt.2, showing some ground-truth themes and repeated sections; (B-D) Three linear segmentations. Numbers below the staff in Fig. A and below the segmentation in Fig. D indicate crotchet beats. These value of crochet beats starts at 0 at bar 1 beat 1.

being developed at the Johannes Kepler University in Austria.
The basic principle inside the database for the themes and motifs is based on Barlow and Morgenstern's Dictionary of Musical Themes [1], Schoenberg's Fundamentals of Musical Composition [26], and Bruhn's J. S. Bachs Well-Tempered Clavier: In-depth Analysis and Interpretation [28].

The JKUPDD contains parts of five musical pieces. These pieces are composed by five well known composers. The pieces are:

- Bach: Prelude and Fugue in A minor, BWV 889 from Well-Tempered Clavier
- Beethoven: Piano Sonata No.1, Op. 2 No. 1 Mvt 3 (Menuetto. Allegretto)
- Chopin: Mazurka in B-flat minor, Op. 24 No. 4
- Gibbons: Silver Swan
- Mozart: Sonata in E-flat major, K. 282 Mvt 2 (Menuetto 1)

These data is chosen because these are described in relative uncontroversial sources. There is a relatively good consensus about what the themes in these pieces are. These pieces also give a rather good representation about western classical music. This is important because when determining if an algorithm is solid, it needs to be evaluated using data where no one can disagree. Using such data can give a result that is an accurate representation of the quality of an algorithm.

2.2.1 Structure

The information in the JKUPDD has a general structure based on the different tasks. The database contains the information which is needed to compute the patterns and it contains the ground truth which is needed to evaluate a given algorithm. For each musical piece there is a folder with all the relevant information. This folder has two sub-folders representing the mono and poly task for that piece. Inside these folders there are multiple representations of the piece: an audio file, a csv file, a kern file, a lisp file for the task, a MIDI representation of the piece, a logic file (a format that can be opened by the eponymous apple program) and the score. It also contains the ground truth patterns in these same formats.

information in symbolic formats The symbolic formats, which will be used in the symbolic tasks, contain information for each note. This information is:

- Ontime: indicates when the given note starts. The time is measured in "quarter notes", this corresponds to the "beat" of most of western classical music. So in most cases ontime zero represents a note that starts right on the first beat, one represents a note that starts right on the second beat and so on.
- MNN: This stands for MIDI Note Number. MIDI is the Musical Instrument Digital Interface, a communication protocol between electronic devices to communicate musical information to each other. Originally MIDI was developed as a universal synthesiser interface. MNN represents the pitch of a note.

- MPN: The Morphetic Pitch Number represents the height of the note on the staff. The difference between MNN and MPN is that MNN represents the absolute note height (a higher MNN will also be perceived as higher when played). The MPN represends the place of the note on the staff. A higher MPN not necessarily means that the note is higher than some lower MPN. For example B#3 (a half note higher than B) and C4 represent the same note height (same MNN) but have a different MPN. The difference in note height between B3 and C4 is a half note. If the note height of B3 is increased with a half note, then they represent the same note height(MNN).

- Duration: represents the duration of the given note. This is like the ontime given in the amount of beats that have passed.

- Staff number: represents the staff that corresponds to the given note.

Chapter 3

PatMinr

In section 1.3 some algorithms were mentioned to give an overview of the different methods that have been used to solve the problem of finding repetitions in musical pieces. It shows that there are multiple methods that solve the problem in several diverse ways. In the next section I will introduce another algorithm that also solves this problem. This algorithm, which was developed by Olivier Lartillot, uses a tactic other than the systems that have already been mentioned. The system is called PatMinr and uses a sequential approach, where it goes one time through all the notes and while doing this, it develops the pattern library. This algorithm gives a solution to the monophonic task. In this thesis I use PatMinr as my basis and I changed its monophonic functionality and adding a polyphonic functionality.

3.1 General information

The PatMinr algorithm is a part of the MiningSuite [16] open-source project by Olivier Lartillot. The MiningSuite is the successor of the MIRToolbox [18]. Both the MiningSuite and the MIRToolbox contain multiple tools that help with MIR tasks. Theme and section discovery is one of these tasks. In the MiningSuite, the algorithm that does this discovery is called PatMinr. The algorithm uses also techniques that are described in [14] as methods for efficient pattern extraction in music . Lartillot has also submitted this algorithm for the 2014 Mirex monophonic symbolic task [17]. An extended explanation about the techniques used in this algorithm are explained in the paper he published in the proceedings of the 15th ISMIR conference [15] (International Society for Music Information Retrieval). I have based my overview of the PatMinr algorithm on these articles.

3.2 Problems

The reason that a single pass method is developed for the discovery task is because it is impossible to compare all possible patterns to each other as there would be a combinatorial explosion. Some systems solve this problem by using heuristics or statistics to find the most frequent or longest patterns. Using such methods will always involve losing some information which is deemed “not important” by the heuristics, therefore it is almost impossible to find all patterns and occurrences with such a method.

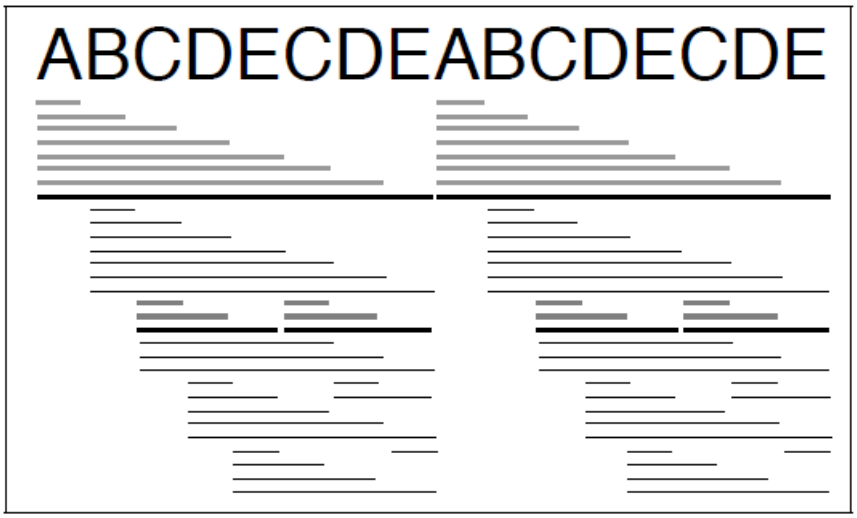

Figure 3.1: A figure indicating which patterns can be viewed as closed patterns by underlining them with thick black lines, other lines represent possible patterns that are not closed.

PatMinr solves this problem by solving two core problems that will produce pattern redundancy:

- **closed pattern mining**: a pattern is closed when it has more occurrences than all its more specific patterns. A pattern is more specific than another pattern when it completely contains the more general pattern, but also has an extra prefix or a suffix. Figure 3.1 shows a string that represents a piece of music. The patterns `abcdefcde` is the biggest closed pattern and this will be much more important than, for example, `abcde`. They both have the same amount of occurrences, but the first pattern is longer, which makes it more important. `cde` is also an important pattern because it has four occurrences. Because `cde` has no specific patterns with the same amount of occurrences, this is also a closed pattern. Therefore it can be an important pattern.

- **cyclic pattern mining**: a pattern is repeated directly after the last occurrence, this is a cause of a lot of redundancy. Figure 3.2 shows a cyclic sequence where the pattern ABC is repeated cyclically. If there would not be cycle detection there would be a lot of possible patterns.

Here it shows that the problem of finding all possible patterns is not a trivial task. Not all repetitions of notes are possible patterns, and the problem of finding all possible patterns can not be simplified into finding the longest patterns or the most frequent patterns. In the next section there is an explanation about how the algorithm works and how it solves these problems.

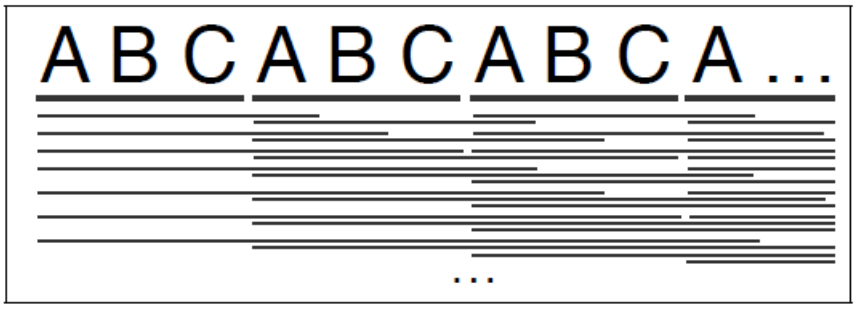

Figure 3.2: A figure that shows what patterns exist without overlap in a cyclic sequence.

3.3 Algorithm Explanation

3.3.1 Pattern representation

Like I explained in section 2.1.2, patterns and their occurrences do not have a straightforward structure. For example, occurrences of patterns can be transposed. PatMinr deals with these differences by having a flexible structure. Figure 3.3 shows an example of how patterns are viewed in PatMinr. Instead of representing a pattern as a sequence of notes, the differences in melody and the length of the notes are used. In the example the difference in the melodic value of the two first notes is equal to +1. This difference is the same as in the second occurrence. The difference in melodic values is the same, which is important because these are two occurrences of the same pattern. The pattern should therefore be represented by the difference in note values instead of the note values to make it possible to define transposed occurrences by the same pattern representation. The example in figure 3.3 shows two occurrences of the pattern that are not exactly the same. The last two notes differ rhythmically but because the melody is the same, it can be considered a possible pattern. With the PatMinr algorithm it is possible to define patterns as sequences of notes with the same melody. That makes that these two occurrences would belong to the same pattern. PatMinr not only needs to be able to describe the pattern. It also use an efficient storage system. This storage system makes it easy to find out if a sequence of notes has occurred already in the past, and if so, the system add the sequence as a pattern.

3.3.2 Pattern storage

To find the different occurrences of a pattern, some systems use numerical distance minimization to predict the similarity. Some studies [10] have shown that this is not the best technique, but it would be better if a strategy is used that is based on “exact identification along multiple musical dimensions of various specificity levels”. Some systems already follow this strategy, because the most important patterns that are perceived by listeners are repeated exactly or are transposed. But they can have some occurrences that have small differences like an upbeat. The PatMinr uses a prefix based tree extension algorithm that works quite well. This tree is constructed by using hash maps. Each note in a pattern has a hash map which links to all current suffixes of the pattern represented by the following note. You can find this in figure 3.4 where the storage of the pattern `abcde` is shown. Each note of the pattern contains a hash map for each property that is

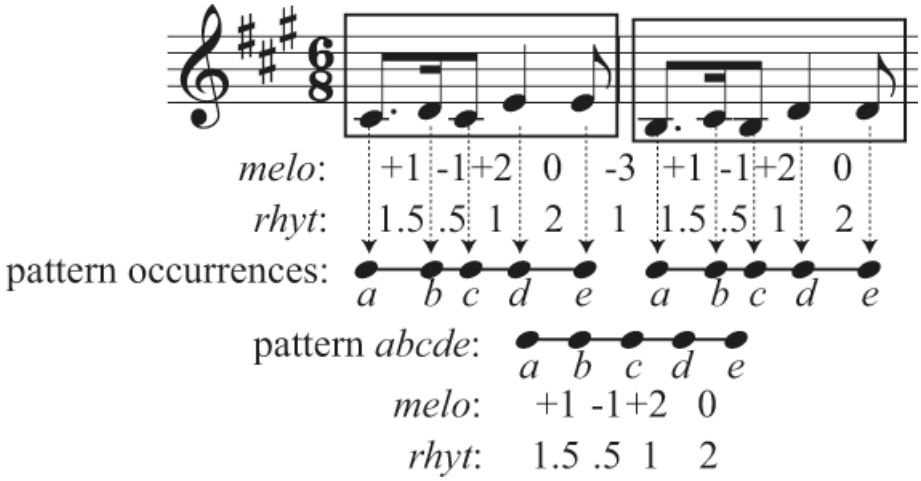

Figure 3.3: The pattern structure in PatMinr

evaluated. In this case there are two hash maps, a melody hash map and a rhythm hash map. In the example the note `a` will have such hash maps and each property of b will provide a link to `b`. So the melodic difference +1 and the rhythm 1.5 will both direct to `b`. It is important to notice that only the properties that are part of the pattern will link to the note. For example, the rhythm between `d` and `e` is not part of the pattern (the rhythm is different in both occurrences) so only the melodic difference of 0 will link to `e`. The algorithm will try to keep the patterns as specific as possible for as long as possible. This is done by adding as much connections as possible between sequential notes. If a pattern is more general, for example, only the melody is the same, then only a part of the connections will be present. It is important to understand that a specific pattern is inherently also a general pattern. A specific pattern can be a pattern which represents a sequence of notes where both the melody and the rhythm are the same. This is also inherently a more general pattern where only the rhythm is the same and a more general pattern where only the melody is the same.
A pattern is only stored as a pattern when two occurrences of the pattern are found. There needs to be information stored to be able to extend these patterns. Therefore a continuation memory is added to each pattern. This continuation memory stores the notes that have followed the pattern in the past, but where the continuation has not happened twice. When this continuation would happen for a second time, it is stored in the continuation memory and this way the pattern will be extended.

3.3.3 Pattern construction

To discover all patterns in the musical piece, all notes need to be processed so that they can be added to the pattern storage. The algorithm uses an incremental single pass approach to try to keep the comparison complexity low. Each note will be evaluated in the same way. This construction will extend the found patterns and occurrences by adding the current note to the possible occurrences.
The methods that are called in the main method can be found below. There are two possibilities:

- If there are currently pattern occurrence(s) that can be extended with the current

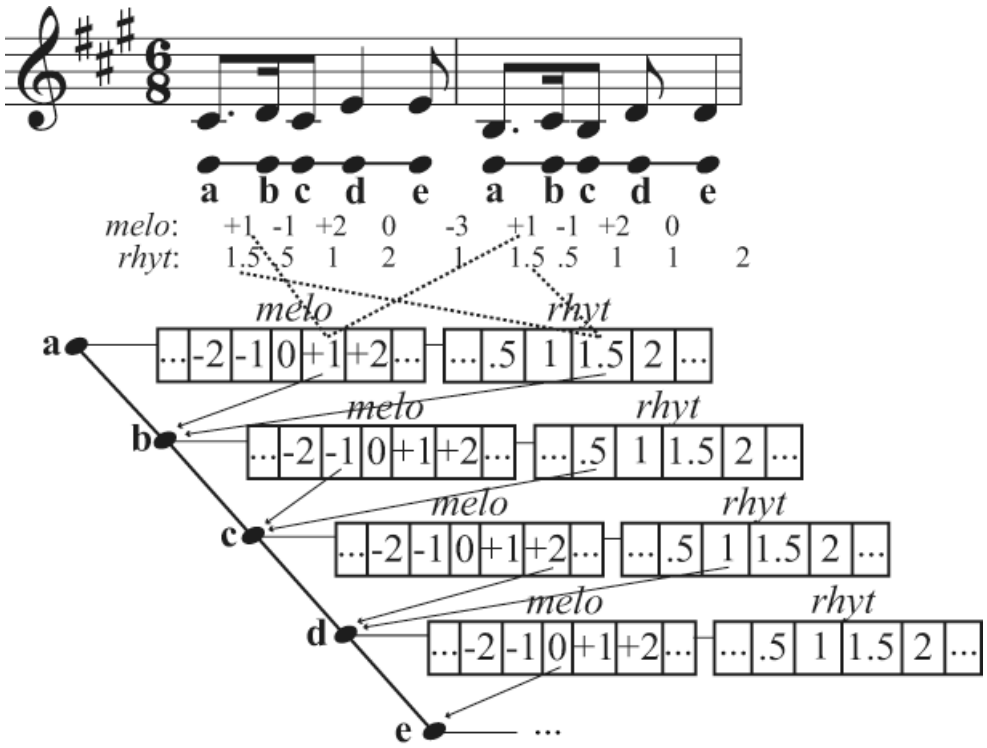

Figure 3.4: The Hashmap Pattern Tree in PatMinr

note, extend the occurrence(s).

  - If one of these occurrences is a cycle then this cycle will be extended.
  - Otherwise it will be normally extended.

- Otherwise begin a new pattern occurrence.

Create new pattern occurrence When there exists no pattern occurrence that can be extended with the current note, a new occurrence needs to be created. This new occurrence can be seen as an extension of the most general pattern, the root pattern. It is useful to use a root as the basis of all new pattern occurrences because this is the reason why it shows a tree and not a collection of patterns. All occurrences are an extension of the root and these occurrences can have multiple extensions, like the leaves of the tree.

Normal pattern occurrence extension When the occurrence is not currently in a cycle, the pattern will be extended normally. For each extension of the pattern it needs to be checked if the description is partly or exactly the same:

- If the description of the extension is partly the same and if there is no extension with that same partial description, then a new extension will be created with this partial description and added to the occurrence.

- If the description of the extension is exactly the same as the description of the following note, then the occurrence will be extended.

The system also needs to check if the note description is similar to the description of a note that is stored in the continuation memory of the pattern. A note is stored in the continuation memory when this note has followed the pattern once. If the current note is similar to a note stored in the continuation memory then the pattern can be extended (the pattern has been followed by the note twice). When this happens, a new extension needs to be made and this extension needs to be added to the current occurrence and the occurrence of the older note. Lastly the note needs to be removed from the continuation memory. If there does not exist any extension similar to the current note and if there is not a similar note in the continuation memory, then the current note needs to be added to the continuation memory.

Cycle pattern occurrence extension When the occurrence is currently in a cycle, other steps need to be taken. It needs to be checked if the description of the current note is the same or partially the same as an extension of the pattern. If it is, then it will act like a normal pattern extension. Otherwise it will increment the phase of the cycle pattern. If it is needed then the cycle pattern is also extended.

Chapter 4

Program components

This thesis is an extension of the PatMinr algorithm. One of the main disadvantages of the original PatMinr algorithm is that it only works with monophonic music. In this thesis the algorithm will be extended to also handle polyphonic music. To do this, I changes both the structure of the program and the algorithm. The monophonic algorithm is changed and extended to be able to handle monophonic and polyphonic music. Because the algorithm is an extension of a monophonic algorithm, the quality of the monophonic algorithm impacts the quality of the polyphonic algorithm. If improvements in the algorithm improve the quality of the monophonic task, then the quality of the output of the monophonic task will also improve.

Evolution of the program PatMinr is implemented in MatLab, but because this code was not documented and lacked some structure, I implemented the new system, from scratch, in Java. This new system is based only on the papers which were described in chapter 3 and the Matlab code. The system was first implemented like it was described in the papers. It was not an exact copy because I used a different structure and made some assumptions, because some parts were not clear. I have also changed some parts from the system, even from the start because I had a different view about solving some problems PatMinr has faced.
The representation differs from the representation of the Mirex program. The obvious difference is that everything is represented as an object because object oriented programming is used instead of the unstructured Matlab code where everything is stored in arrays. The different types of components used in the system will be discussed in this section. Every time I refer to an object instead of a concept, I have indicated it by starting the name with a capital and using a different font. An example of an object is `Time`. `Time` describes the object and time describes a type of measurement.

4.1 Main

The `Main` class is placed at the highest level of the program. It initialises the program and handles the output.

4.1.1 Initialisation of the program

The program needs to be initialized and this is done in the `Main` class using the initialisation algorithm 1. Algorithm 1 first creates a file which contains all output strings (all output patterns). An `InformationReader` (see section 4.3) is created using the meta data which stores which files needs to be analysed. This `InformationReader` returns a list of files which are then analysed using `Analysers` (see section 4.2.1). For each `Analyser` a `Filter` is created. This filter goes trough the entire pattern tree and uses a scoring system to determine if the `Pattern` needs to be returned. This filter process returns all the `Occurrences` of a `Pattern`, not only the `Occurrences` that are stored in the sub-pattern, but also the `Occurrences` found in the specific patterns. These `Analysers` can be converted in a string which represents all the patterns found in the file. The type of `Analyser` determines which types of patterns the program searches and returns.

Algorithm 1 Initialisation of the program

1: $outputFile \leftarrow$ createOutputFile()
2: $informationReader \leftarrow$ new InformationReader(metaData)
3: $fileList \leftarrow$ files retrieved using *informationReader*
4: **for all** File *file* in *fileList* **do**
5: $\quad analyser \leftarrow$ new Analyser(*file*)
6: $\quad analyser$ processes *file*
7: $\quad filter \leftarrow$ new Filter($analyser$)
8: $\quad filterString \leftarrow$ output of filter method in *filter*.
9: $\quad outputFile \leftarrow currentOutputString + filterString$
10: **end for**

4.2 Analyser

There are multiple definitions to be given to musical patterns. It is therefore necessary to make the program flexible enough to be able to find these different types of patterns. In PatMinr this is done by searching in the tree sequences of notes which have some elements in common like having the same mpn differences. The alternative for searching everything and then filtering out the information you want is to directly search for the patterns that are interesting. The new system would build a pattern database with only patterns that correspond to a certain description. This description would, for example, be a mnn translatable pattern. The advantage of this technique is that it will do a more targeted search for interesting patterns. The advantage of PatMinr is that when the pattern database is completed, different queries can be done on this database to find different kinds of patterns without needing to go through all the music again. In the future, when a precise pattern description will be available, my system will be able to use this description to perform a targeted search only these repetitions. My system will then be able to achieve very good results at a very high speed. This is one of the reasons why my algorithm has more potential than PatMinr. It will be easier to find imperfect patterns when using a designated analyser. This is because this analyser can, for example, store the max value of imperfection that is excepted. If an occurrence then would exceed this limit then is the occurrence rejected for that pattern.

The `Analysers` are given a file that needs to be analysed and turned into `Patterns`. The `Analysers` will go through the file and manage the actions that the rest of the program needs to make to include the entire file in the pattern tree. The `Analysers` use a `MusicReader` to convert the file into usable objects, in this case, it will turn the file into `Times`. The `Analysers` will use a sequential one-pass algorithm, which is explained in section 4.2.1, to evaluate all `Times` from the `MusicReader` and add them to the system. To add them to the system a processing algorithm is used which is explained in section 4.2.2.
Both algorithms are are implemented in the `Analyser` class because these algorithms are general enough to be used by all `Analysers`.

4.2.1 One-pass sequential loop

In algorithm 2, the `Analyser` creates a loop which contains all data from the file and processes this to extend the pattern tree. When all data is processed, the entire pattern tree is complete and can be used by other classes to retrieve patterns that occur in the music piece.
The `Analyser` first creates a `MusicReader` which converts the file into a list of lists of `Times` (see section 4.3.1). For each staff exists a list of `Times`. All these times need to be processed, so it uses a double loop to process the `Times` one by one (see section 4.2.2). It is important that the `Analyser` goes through all these `Times` sequentially because musical `Patterns` are composed of sequential notes. If a `Pattern` or a `Sequence` would be extended with a `Note` that is not subsequent to the last `Note` of this `Pattern` or `Sequence`, then this would not be correct. The program can handle if `Times` of different staffs iterate, but each processed `Time` of a staff needs to have a starting time that is later than all already processed `Times`. When all `Times` are processed, then it is possible that there are some `Sequences` left with partial `Occurrences` of `Patterns` in. Therefore at the end, all remaining `Sequences` need to be checked if they already contain an `Occurrence` of a `Pattern`.

Algorithm 2 One-pass processing of `Times` in musical piece

```
1: memory ← memory created by musicReader
2: for all staffMemory in memory do
3:     for all time in staffMemory do
4:         processTime(time,memory)
5:     end for
6: end for
```

4.2.2 Processing of `Time`

Each `Time` is processed using the processing algorithm 3 which is found in the `Analyser` class.
Because a `Time` may only extend `Sequences` with the same staff, it first looks up the staff of the given `Time` and gets the list of all relevant `Sequences`. If there does not already exist such a list, then the list will be created. To add a `Note` to the pattern tree it always needs to be added to a `Sequence`. To make sure that this always happens, the

`Analyser` checks if there are any `Sequences` in the current list of `Sequences`. If the list is empty, a new `Sequence` with the given staff will be created. The extension algorithm 5 is then executed on all `Sequences` in the `Sequence` list. This extension gives back a list of `Sequences` that are still active and that need to be extended with the next time. It is also possible that by extending a `Sequence` with the given `Note` the `Sequence` stops and then an empty list is returned. If no `Sequences` are ready to be extended in the future because the `Time` caused all `Sequences` to stop, then a new `Sequence` will be started and will be extended with the `Time`. It is possible that the given `Note` is not a possible extension of the past `Sequences`, but it can be the start of a new `Occurrence` of a `Pattern`. Because a one-pass approach is used, the `Sequence` with the extension based on the `Time` needs to be done at the end because after the processing of the `Time`, the `Time` will never be processed again if it is not part of an `Occurrence`.

Algorithm 3 Processing of a `Time`

```
1: function processTime(time, memory)
2:     staff ← staff of time
3:     if list of sequences that belongs to the given staff exists then
4:         sequences ← list of sequences that is currently been extended and belongs to the given staff
5:     else
6:         initialise sequences for the given staff
7:         sequences ← list of sequences that belongs to the given staff
8:     end if
9:     if sequences is empty then
10:        sequences ← new sequence
11:    end if
12:    for all sequence in sequences do
13:        sequences ← sequence extended using time, memory, base pattern and the current analyser
14:    end for
15:    if sequences is empty and at the begin of the processing it was not empty then
16:        sequences ← new sequence which is extended with the given time
17:    end if
18: end function
```

4.3 Reader

There are two readers defined in the program. These convert data in a predefined structure to objects that can be used in the system. The `InformationReader` reads meta data and can return a list of files that needs to be analysed. The music reader can take this file and convert it in data formats that the program can use, like `Notes`. In the current system, the reader converts the files into `Times` and stores them.

4.3.1 Convert music file into `Times`

Because the system is not designed to constantly work with strings, but with objects, the `MusicReader` converts the data from the music files into `Times` using algorithm 4.
The reader will first make sure that there is a line left to process. If there is, the reader will read that line to get note information out of it. The reader will then create the `Note` and will decide to which `Time` the `Note` needs to be added. The reader will first look at the last `Time` that is created because this has the highest possibility. A `Note` can be added to a `Time` if both the begin time of the `Note` is the same as the time stored in the `Time` and the staff is the same in the `Note` and the `Time`. If the `Note` can not be added to the most recent `Time` then a new `Time` needs to be created with only the `Note`. A new `Time` needs to be created because all `Notes` are ordered based on their begin times. When there are `Notes` with the same begin times, then the `Notes` are ordered based on their staff.

Algorithm 4 Read music file

```
1: function readFile()
2:     while input has a next line do
3:         Turn the data found on the next line into information that can be used in a
   note.
4:         note ← new Note(time,mnn,mpn,length,staff,memoryId)
5:         if note can be contained by the last found time then
6:             time ← note added to all already found notes
7:         else
8:             newTime ← a new time based on the current note
9:             newTime is added to the list of times that correspond to the staff of note.
10:        end if
11:    end while
12: end function
```

4.4 Sequence

A `Sequence` is a abstraction of sequences of `Notes` that can be part of a `Pattern`. Each `Sequence` consist out of an `Occurrence` and a `Pattern`. The program tries to extend these `Occurrences` to find possible `Patterns`. When the `Occurrence` can be matched with a `Pattern`, the `Occurrence` is stored in the `Pattern`, and there will be tried to also find an extension to the `Occurrence`. This extension can be a suffix `Pattern` or an extension of the current `Pattern`. This component is introduced to make it possible to create an `Occurrence` with `Notes` that are not stored directly after each other in the file. This is necessary in the polyphonic task because when different voices are played together, the `Notes` of the different voices are mixed and sequential `Notes` are not always candidates to extend an `Occurrence`.

4.4.1 Extend a Sequence with a Time

Algorithm 5 shows the actions that are taken when a Sequence needs to be extended by a Time.
First is the last Note of the Occurrence searched. If the Occurrence is still empty then it needs to search for the Time that is previous to the given Time. Because Occurrences can not store halve PatternValues, the program needs to search previous Note relative to the given Note to determine the PatternValue. Because all Times are stored in the memory, this can easily be searched. The problem that needs to be solved when there are two consecutive Times are found is what Note from the second Time follows what Note from the first Time. It is not easy to determine this because the amount of Notes is not constantly the same so there is not a constant order of Notes which can be used to determine if a Note is possibly part of a Pattern. Because I have found no better alternative, a combination of all Notes of the first Time with the second Time is used. Not all these combinations are possible, but the wrong combinations will be filtered out naturally because they are not normal extensions of the Pattern. For each of these combinations a PatternValue is created and then the system tries to extend the Sequence with the found PatternValue using algorithm 6. The PatternValue that is created can be a Note or a DifferenceValue. It depends on the kind of Sequence that needs to be extended.
If the current Occurrence contains Notes, then the sum of the begin time and length of the last Note is retrieved and compared with the begin time of the given Time. If there is not overlap between the Notes, then the program tries to extend the Sequence with the PatternValues which are created out of the previous Note and the Notes of the Time.

4.4.2 Extend a Sequence with a PatternValue

A Sequence consists out of an Occurrence and a Pattern. This algorithm tries to add a PatternValue to the Occurrence.
First is the Pattern position (patPos) determined. This patPos represents the place of total pattern where a new extension will take place. If the current Pattern of the Sequence is equal to the base Pattern (basePat), then the actions that are needed are different because the base Pattern stores only child Patterns and no Notes. When a Sequence with the base Pattern as Pattern needs to be extended, the algorithm will call algorithm 8. If the Pattern of the Sequence is different from the basePat, then the algorithm determines if a possible extension could still be part of the current Pattern. This is determined by comparing the patPos with the size of the current Pattern. If an extension can be part of the current Pattern, then the PatternValue at the patPos is determined. This PatternValue from the Sequence Pattern is then compared with the given PatternValue. If they are comparable, then the algorithm extends the Occurrence with the given PatternValue. If they are not the same, then the Pattern is split at position patPos. The Pattern is split because there are more Occurrences of the Pattern before the split than after the split because the current Occurrence of the Sequence is an Occurrence of the small Pattern, but not of the big Pattern. This split involves removing Notes from the Pattern and placing them in a specific Pattern. All Occurrences of the Pattern are then switched to the specific Pattern and all the Pattern that previously were specific Patterns, are added as specific Patterns of the new generated specific

Algorithm 5 Extend sequence with a time

```
1: function extendTimeSequence(time, memory, basePat, analyser)
2:     previous ← last note in the current occurrence
3:     if occurrence currently has no notes then
4:         previousTime ← the Time previous to time
5:         if previousTime exists then
6:             previousNotes ← notes that are stored in previousTime
7:             nextNotes ← notes that are stored in time
8:             for all Note prevNote in previousNotes do
9:                 for all Note nextNote in nextNotes do
10:                    patVal ← a PatternValue created using prevNote and nextNote
11:                    sequences ← extendPatValSequence(patVal,basePat,memory,analyser)
12:                end for
13:            end for
14:        end if
15:    else
16:        previousTime ← begin time of previous
17:        previousLength ← length of previous
18:        nextBeginTime ← begin time of time
19:        if previousTime + previousLength ≤ nextBeginTime then
20:            nextNotes ← notes that are stored in time
21:            for all Note nextNote in nextNotes do
22:                patVal ← a PatternValue created using prevNote and nextNote
23:                sequences ← extendPatValSequence(patVal,basePat,memory,analyser)
24:            end for
25:        end if
26:    end if
27: end function
```

`Pattern`. If the size of the current `Occurrence` is equal to the size of the current `Pattern`, than there needs to be looked for a different kind of extension than just an extension of the `Occurrence`. This extension is done in algorithm 7.
During this algorithm, there was a moment where `PatternValues` are compared with each other. This comparison depends on the type of `Pattern` that is stored in the `Sequence`. When the `Notes` are exactly the same, except for their begin time, then it's an exact `Pattern`. If it would be an `MnnTransPattern`, then only the mnn `DifferenceValue` needs to be the same.

Algorithm 6 Extend sequence with a `PatternValue`

```
1: function extendPatValSequence(patVal, basePat, memory, analyser)
2:     patPos ← size of the occurrence of the sequence
3:     if curPat is the same as basePat then
4:         sequences ← add all sequences of basePatExtension(patVal,curPat)
5:     else
6:         patSize ← size of the pattern of the current sequence
7:         if (patPos + 1) ≤ patSize then
8:             curPatVal ← pattern value at position patPos in the pattern of the sequence
9:             if patVal is comparable with curPatVal then
10:                Add patVal to the current occurrence of the sequence
11:                sequences ← add the current sequence
12:            else
13:                Split the current pattern at patPos
14:                Add the current occurrence to the current pattern
15:            end if
16:        else
17:            sequences ← add all findOutsidePatExtensions(patVal,memory, curPat, analyser,basePat)
18:        end if
19:    end if
20:    return sequences
21: end function
```

4.4.3 Find extensions to the `Sequence` when the current `Occurrence` has the same size as the current `Pattern`

If both the current `Occurrence` and the current `Pattern` have the same length, then the program tries to extend the current `Pattern` using algorithm 7. The program first tries to find specific `Patterns` that start with a similar `PatternValue` using algorithm 9. If there are such `Patterns`, then for each specific `Pattern`, the new `Pattern` in the `Sequence` is the specific `Pattern` and the `Occurrence` is the current `Occurrence` extended with the `PatternValue`. If no such specific `Patterns` are found, then the program tries to extend the current `Pattern` with the `PatternValue` by using the memory in algorithm 10. There are two types of possible extensions: if the `PatternValue` is added to the current `Pattern`, or if there is found a new specific `Pattern`. If the current `Pattern` is extended,

then the current `Occurrence` is extended with the `PatternValue` and the `Sequence` is kept active. If the `Pattern` is a new specific `Pattern`, then the new `Pattern` will be the current `Pattern` and the `PatternValue` will be added to the current `Occurrence`. The `Sequence` will again be kept active. If no extensions are found, then the current `Occurrence` is added to the `Pattern`.

Algorithm 7 Searching extensions of the sequence when the current occurrence and current pattern have the same size

```
1: function findOutsidePatExtensions(patVal, memory, curPat, analyser)
2:     specPats ← all specific patterns of the curPat that are possible extensions of the occurrence with the given patVal
3:     if there are specPats then
4:         curOcc ← curOcc extended with patVal
5:         for all Pattern pat in specPats do
6:             sequences ← new Sequence with pat as Pattern and curOcc as Occurrence
7:         end for
8:     else
9:         newPat ← call findExtendPat(patVal,memory)on curPat
10:        if curPat can not be extended using the given patVal then
11:            add curOcc to curPat
12:        else if curPat and newPat are the same when the pattern value is added at the end of curPat then
13:            currOcc is extended with patVal
14:            sequences ← this sequence
15:        else
16:            curPat ← newPat
17:            currOcc is extended with patVal
18:            sequences ← this sequence
19:        end if
20:    end if
21:    return sequences
22: end function
```

4.4.4 Base `Pattern` extension with a `PatternValue`

Algorithm 8 takes care of the actions that need to be made when the current `Pattern` is the same as the base `Pattern`. Because the base `Pattern` does not contain `PatternValues`, it needs to be handled differently than other `Patterns`. In the normal case, when the current `Pattern` is not the base pattern, the algorithm would check if the `PatternValue` equals the next `PatternValue` in the `Pattern`. Because the base `Pattern` does not contain `PatternValues`, this is not possible. First the algorithm checks if the base `Pattern` has specific `Patterns` that begin with the given `PatternValue` using algorithm 9. If there are such `Patterns`, then for each of these `Patterns` the algorithm creates a new `Sequence` with the specific `Pattern` as `Pattern` and a new `Occurrences` with the `PatternValue` as `Occurrence`. This `Sequence` is then added to the list with extendable `Sequences`. If there is not a specific `Pattern` found, then this `PatternValue` is added as a new `Pattern`

with the `PatternValue` also as `Occurrence`. Note that this new `Pattern` is not really a `Pattern` because it has only one `Occurrence` and one `PatternValue`, but this `Pattern` is needed to be a seed that can be extended with other `PatternValues`. If one of these `Patterns` is not extended, then it will not be returned because it will be filtered out before the `Patterns` are returned.

Algorithm 8 Searching extensions to a sequence which has the base pattern as pattern

```
1: function basePatExtension(patVal, basePat)
2:     specPats ← all specific patterns of the curPat that are possible extensions of the occurrence with the given patVal
3:     if there are specPats then
4:         curOcc ← curOcc extended with patVal
5:         for all Pattern pat in specPats do
6:             sequences ← new Sequence with pat as Pattern and curOcc as Occurrence
7:         end for
8:     else
9:         create a new pattern based on the pattern value and add it to the specifics of the basePat
10:    end if
11: end function
```

4.5 Pattern

In the new system a `Pattern` exist as an object. This is directly opposite to PatMinr where the patterns are generated out of the "pattern hash map".

Patricia Trie In figure 4.1 there is a simple example of the pattern structure. All `Patterns` are part of the pattern tree. This pattern tree can be seen as a variation on the Patricia search trie [13]. The Patricia trie is usually used with strings, but it is possible to use it in this context because we need to store data that is both sequential and used a prefix based approach. This new trie contains a sequence of `Note` elements in each node. These sequences are called patterns. These patterns are sub-patterns of the eventual returned patterns. The root of the tree is called the base `Pattern` and can be seen as the most general `Pattern` which is extended by other patterns. Each child of a `Pattern` is a possible suffix sub-pattern. This means that the addition of a suffix `Pattern` will always represent a more specific `Pattern`. All the patterns that can be found in this pattern tree are closed patterns. This is a solution that solves the closed pattern problem which is described in section 3.2.

4.5.1 Find specific `Patterns` that are a possible extension with a given `PatternValue`

Sometimes it is necessary to know which possible specific `Patterns` are extended by the current `Pattern` and which start with a given `PatternValue`. This is necessary when the program extends an `Occurrence` and needs to know what `Patterns` are possible

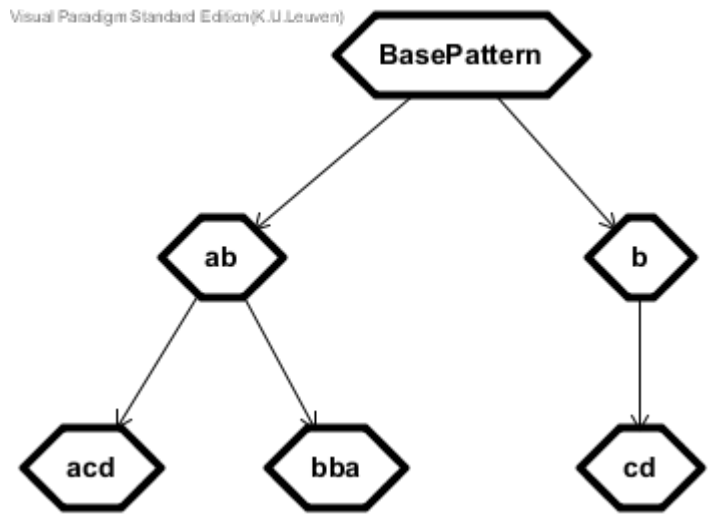

Figure 4.1: Pattern structure in the new system

extensions. To retrieve these specific `Patterns`, the algorithm 9 is used.
The algorithm first looks up all specific `Patterns` of the current `Pattern`. The first `PatternValue` of each of these `Patterns` is then compared with the given `PatternValue`. If these `PatternValues` are comparable, then the specific `Pattern` is added to a list. At the end the full list of possible specific `Patterns` is returned. The comparison of `PatternValues` again depends on the type of `Patterns` the program searches.

Algorithm 9 Find specific patterns that are a possible extension with a given `PatternValue`

```
1: function findExtendPat(patVal)
2:     specPats ← all specific patterns of the current pattern
3:     extendPats ← new list with patterns
4:     for all Pattern pat in specPats do
5:         newPatVal ← the first pattern value of pat
6:         if currentPatVal is comparable to newPatVal then
7:             add pat to extendPats
8:         end if
9:     end for
10:    return extendPats
11: end function
```

4.5.2 Find all possible `Pattern` extensions by looking at what `Occurrences` can be extended using the memory

To determine if a `Pattern` can be extended by a `PatternValue`, the `Occurrences` are checked if they can be extended. Based on which `Occurrences` can be extended, the decision is made if and how the extension is made.
First the extendOccVals and extendOccs are calculated by determining which `Occurrences` can be extended using algorithm 11. The extendOccs and extendOccVals respectively represent the `Occurrences` that can be extended and the values that follow the extendOccs.

An Occurrence can be extended if the next Note in the memory is similar to the last Note of the given PatternValue. If all the Occurrences are extended, then the algorithm extends the current Pattern with the given PatternValue and extends all extendOccs with their extendOccVals. The Pattern that needs to be returned is the current Pattern. If some Occurrences are extended, then these Occurrences are Occurrences of a new specific Pattern. The algorithm creates a new specific Pattern and adds all extended Occurrences to this Pattern. The Pattern that will be returned is the created specific Pattern. If no Occurrences can be extended, then no Pattern is returned. So if no Pattern is returned, then the Pattern is not extended. If the same Pattern is returned, then the current Pattern is extended. If a different Pattern is returned, then this Pattern is a new specific Pattern of the current Pattern.

4.5.3 Determine if an Occurrence can be extended with a given PatternValue

The program uses algorithm 11 to determine if an Occurrence can be extended. It gets the memory that belongs to the staff of the given PatternValue. Using this memory, all possible PatternValues of the Occurrence are retrieved. If one of these PatternValues is comparable to the given PatternValue, than the Occurrence can be extended and the Occurrence and the found PatternValue are added to lists and these lists will be returned.

4.6 Basic components

There are some basic components that are used by the other algorithms. These components help with making the system more understandable but do not have extended algorithms like the other components.

4.6.1 Occurrence

An Occurrence is represented as an object in the new system. A Pattern needs to have multiple Occurrences before being considered a pattern. These Occurrences store a list of sequential elements. These sequential elements are represented by PatternValues. The system currently supports two types of PatternValues: Notes and DifferenceValues. These Occurrences are also used in Sequences. A Sequence also stores an Occurrence. This Occurrence represents an Occurrence of sequential Notes that possibly are part of a Pattern. In a different context they have a different meaning. Generally they just store a list of PatternValues.

4.6.2 Pattern values

Pattern values represent like the name suggests data out of which Patterns are constructed. The Notes are used when searching exact Patterns, the DifferenceValues are used when searching translated Patterns.

Algorithm 10 Find all possible pattern extensions by looking at how occurrences can be extended using the memory

```
1: function findMemoryExtension(patVal, memory)
2:     extendOccVals ← new list of PatternValues
3:     extendOccs ← new list of Occurrences
4:     for all Occurrence occ in occurrences of the current Pattern do
5:         findMemoryExtension(memory,extendOccVals, extendOccs, patVal, occ)
6:     end for
7:     extended ← a boolean which represents if the size of extendOccs is bigger than 0
8:     AllOccsExtended ← a boolean which represents if the size of extendOccs is bigger
   then the size of the occurrences of this pattern
9:     if extended then
10:        if AllOccsExtended is false then
11:            newPat ← a new pattern which is a specific pattern of the current pattern
   an which contains all occurrences of the current pattern that can be extended by the
   pattern value. These occurrences are all extended with the given pattern value.
12:        else
13:            if the current pattern contains no specific patterns then
14:                add patVal to patValues
15:                amount ← the size of extendOccs
16:                for all i between zero and the size of extendOccs do
17:                    currOcc ← occurrence at position in extendOccs
18:                    currOccVal ← pattern value at position i in extendOccVals
19:                    add currOccVal to currOcc
20:                end for
21:                newPat ← the current pattern
22:            else
23:                newPat ← a new pattern which is a specific pattern of the current
   pattern an which contains all occurrences of the current pattern that can be extended
   by the pattern value. These occurrences are all extended with the given pattern value.
24:            end if
25:        end if
26:    end if
27:    return newPat;
28: end function
```

Algorithm 11 Determine if an occurrence can be extended

```
1: function findMemoryExtension( memory, nextPatVal, occ)
2:     staff ← staff of the current occ
3:     staffMemory ← memory with the given staff
4:     patVals ← the next pattern values for the given occurrence
5:     if there exist patVals then
6:         for all PatternValue patVal in patVals do
7:             if nextPatVal is comparable to patVal then
8:                 add patVal to extendOccVals
9:                 add occ to extendOccs
10:            end if
11:        end for
12:    end if
13: end function
```

Note representation A note is represented in the program as object. This object stores all the information that is known about that note; the ontime, mnn, mpn, duration and staff.

Difference value representation A `DifferenceValue` is represented as a combination of two sequential `Notes`. This `DifferenceValue` is used when discovering translated `Patterns`. It is different from the PatMinr algorithm because PatMinr represents the differences inside the last note. It is more flexible to use both `Notes` to represent the `DifferenceValue` because it is easier to get additional difference information. After all the differences are stored in a note, then no additional information about the difference with the previous note can be calculated. This is possible with the new system because a link to both `Notes` is stored.

4.6.3 Time

A `Time` is used to represent a collection of `Notes` on the same staff and that start on the same moment. This component is necessary in the polyphonic system because there is no way to know what `Note` of a `Time` will be used to extend a `Pattern`. By using `Times`, it also makes sure that the list in which the `Times` are stored will always progress in time. This is an addition that helps structure the data to convert it into a multidimensional representation of music.

4.6.4 Filter

Not all `Patterns` that are found are important. To return only a subset of `Patterns` that are more likely to be important, a filter is used. This filter takes the base `Pattern` of an `Analyser` and traverses this to find all possible important `Patterns`. It also searches for all `Occurrences` of the given `Pattern` by searching the `Pattern` extension. This is necessary because the entire `Occurrences` are stored in the relevant `Pattern` extension. By definition these `Pattern` extensions also have the general `Pattern` as a prefix because the extensions are suffixes of the `Pattern`. Inside the filter there is also a scoring algorithm

used to determine if a `Pattern` is important enough to return. Currently this algorithm is quite basic and is a point in which the algorithm can be improved in the future.

Chapter 5

Program Structure

5.1 Algorithm

5.1.1 Algorithm overview

The algorithm uses a loop to go trough all `Time` of the musical piece (See section 4.2.1. Each `Time` is processed using the algorithm which is described in section 4.4.1. The algorithm that can be found in section 4.4.2 is then executed once for each `Note` in the `Time`. The algorithm will try to extend the `Sequence` with a `PatternValue` which is based on the `Note`.

Extend a `Sequence` with a `PatternValue` Figure 5.1 shows an overview how the algorithm tries to extend a `PatternValue`. The figure shows all the actions that can be taken while a `Sequence` is extended by a `PatternValue`. The starting point is the same as the algorithm that is described in section 4.4.2. The figure is a flow diagram because a lot of decisions are made during the process of extending a `Sequence`.

Filtering When the pattern tree is finished, The algorithm will filter out only the most important `Patterns` by taking in account the amount of `Occurrences` of the `Pattern` and the size of the `Pattern`. The algorithm will return all the `Occurrences` of the filtered `Pattern`.

5.1.2 Complexity

The algorithms is not a greedy algorithm. The program determines each time which step will be taken. Because the state of the program can determine the steps that are taken, it is more difficult to determine the complexity of the program. I will evaluate the complexity of the program by evaluating the different parts of my system.

Amount of `Sequences` The program extends all active `Sequences` that have the same staff as the `Time`. The amount of `Sequences` can be different for each new `Time`. The amount is always bigger than one, and does not have a maximal value. The only part of the system that can cause the system to increase the amount of `Sequences` is when `Times` are added to a `Sequence`. This has been explained in section 4.4.1. Theoretically, the amount of `Sequences` can explode. If n is the maximal amount of `Notes` that exists

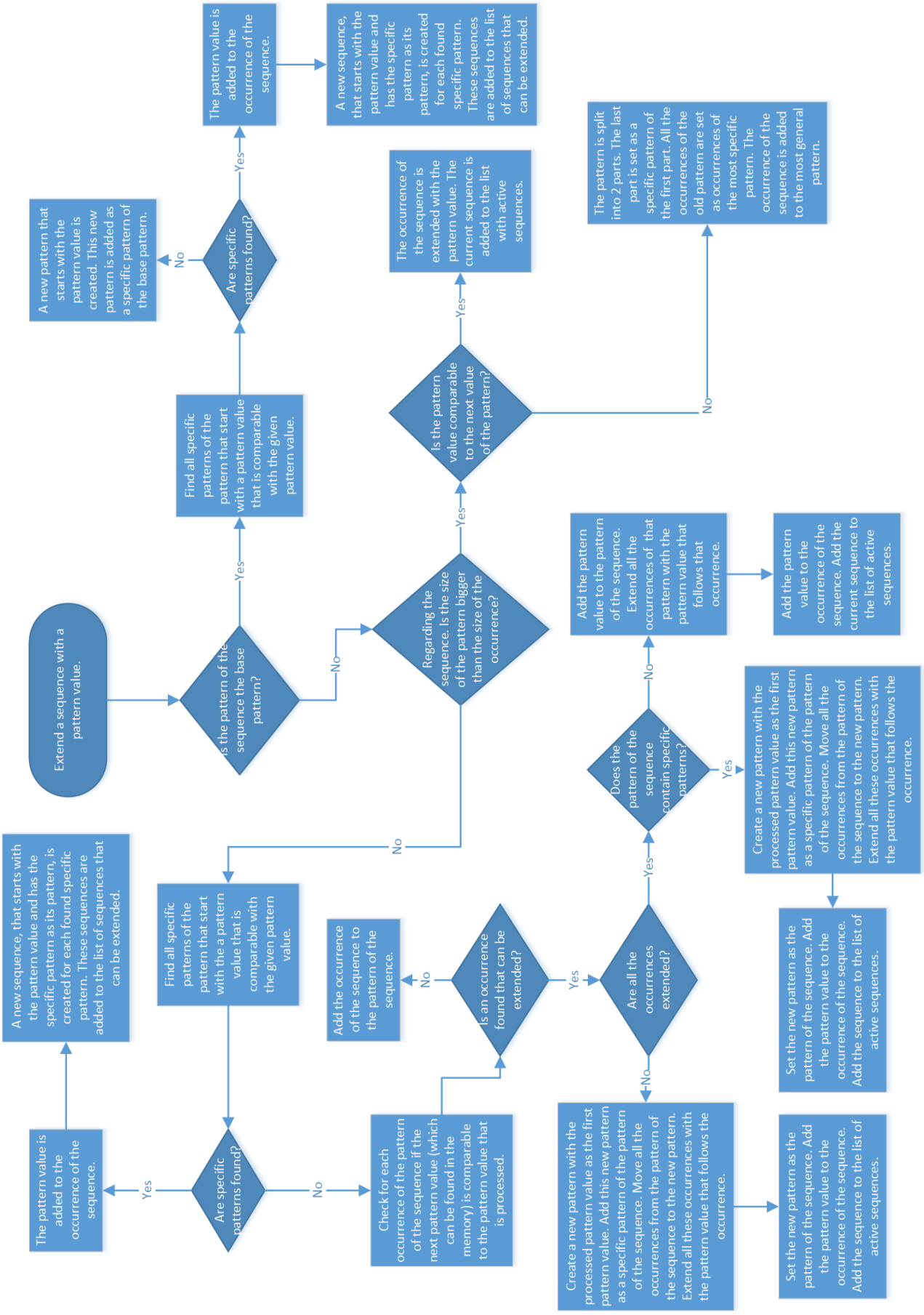

Figure 5.1: Overview of the structure of the algorithm

in a `Time` and t the amount of `Times` with a given staff then the amount of `Sequences` after t `Notes` can be n^t. The normal amount of `Sequences` is often much lower than the maximal amount. This normal amount is lower because when `Sequences` can not be extended, they are removed from the list of `Sequences`. It also frequently happens that the amount of `Notes` in a `Time` is less than the maximal amount. While solving the polyphonic task the amount of `Sequences` has never been higher than five on any given moment. The maximal amount of `Sequences` is one. This can be explained by looking at the formula for the maximal amount of `Sequences`. The general complexity is therefore constant.

Amount of specific `Patterns` The amount of the specific `Patterns` has an influence on the amount of comparisons and lookups of the algorithm. Algorithms 7 and 8 both go trough all specific `Patterns` of the `Pattern` of the current `Sequence`. The algorithm looks up the first `PatternValue` of the specific `Pattern`. This `PatternValue` is then compared with the `PatternValue` that is currently being processed. Because one of both algorithms will always be executed, the amount of specific `Patterns` will always have an impact on the complexity of processing a `Note`. The complexity originating at the amount of specific `Patterns` while processing a `Note` is: O(s) memory lookups and O(s) comparisons. The s represents the amount of specific `Patterns`.

Amount of `Occurrences` of the `Pattern` of the current `Sequence` When the size of the `Pattern` of the current `Sequence` is bigger than the size of the `Occurrence` of the current `Sequence` and when the `Pattern` of the `Sequence` doesn't have specific `Patterns` that are a possible extension of the `Occurrence`, then the algorithm checks all the `Occurrences` of the `Pattern` of the `Sequence`. If the next `PatternValue` which can be found in the memory is comparable to the `PatternValue` that is processed. This causes O(o) lookups in the memory with o an `Occurrence` of the `Pattern` of the `Sequence`. This complexity is additive to the complexity caused by the amount of specific `Patterns`.

Conclusion The complexity of the algorithm is quite low. Processing `PatternValues` will has a maximal memory complexity of O(s+o). The amount of comparisons is maximally O(s). The letter s represents the amount of specific `Patterns` of the `Pattern` of the current `Sequence`. The letter o represents the amount of `Occurrences` of the `Pattern` of the current `Sequence`. The processing of a `PatternValue` is done for each active `Sequence`. I have indicated that the amount of `Sequences` is nearly constant. The complexity needs to be multiplied with this constant. The complexity will therefore stay almost the same. It is very important to notice that this complexity is very low. In the simplest programs all possible sequences of a given length are compared with each other. This would give a comparison complexity of around O(n^2) to find all comparable `Patterns` of a given size. If large `Patterns` need to be found, these comparisons needs to be done for all possible lengths. This example indicates that there is a big difference in complexity between 'simple' algorithms and my algorithm.

5.2 Architecture

I designed an architecture for the new system to make it more structured and most importantly more flexible and adaptable. To make the new system more understandable, and to see what the changes are to the original system, the monophonic version will be explained first and after that are the changes explained to make the system work with polyphonic music. These were also the steps that were taken when implementing the system. First a monophonic version was made which included already some changes to the original system, and when the monophonic algorithm provided good enough results the extension to a polyphonic algorithm were made. Because the polyphonic algorithm is based very hard on the monophonic algorithm, sometimes while extending the algorithm to handle polyphonic music the system needed to change the monophonic version first to be able to handle this change to the polyphonic algorithm.
During this thesis, not only improvements to the algorithm are made, but also structural improvements to the program. This was necessary because the original program was undocumented and was difficult to understand. One of the reasons why it was difficult to understand is because it is part of a collection of algorithms instead of a solitary program. Because it is part of a collection, the program also used parts of different algorithms to make the collection more manageable. This is a advantage for the collection, because it has parts that are general enough to be reused, but it also makes the algorithms more scattered and therefore more difficult to understand.
Because I also wanted to work on the extendability and adaptability of the program, there was also a focus on having a good architecture. Figure 5.2 is a diagram that shows the architecture of the program. It shows all the classes that are used, except the test classes, and all the links that exist between these classes. The responsibilities of the system are split between the different classes. Because they all have a small set of responsibilities, the amount of links between the classes is low. One of the major rules of software design is to have low coupling and high cohesion. The program has both because it has a low amount of links and because each class represents a real world entity and only contains methods relating to this entity.

5.2.1 Inheritance

The program needs to be able to find different kinds of `Patterns`. The program uses different kinds of analysers to find these `Patterns`. These different kinds of `Patterns` are represented by objects of different classes. To build the `Patterns`, different kinds of `Sequences` are used. Inheritance is used because the analysers, `Sequences` and `Patterns` have a lot in common. This inheritance makes it possible to store general algorithms as high as possible in the hierarchy to avoid duplication. When there are parts that are different in the different subclasses, then these parts are overridden in methods in the appropriate classes. This structure makes it easier to add different types of `Patterns` in the future because it can use common methods from higher in the hierarchy. Because the different types of `Patterns` overlap for a big part only a few small methods need to be overridden. This makes the structure interesting because introducing a new pattern type requires three small classes which implements these differences.
Because the specific parts are also implemented in the super classes makes it interesting to create objects of these classes. Higher in the pattern hierarchy are `Patterns` that have an

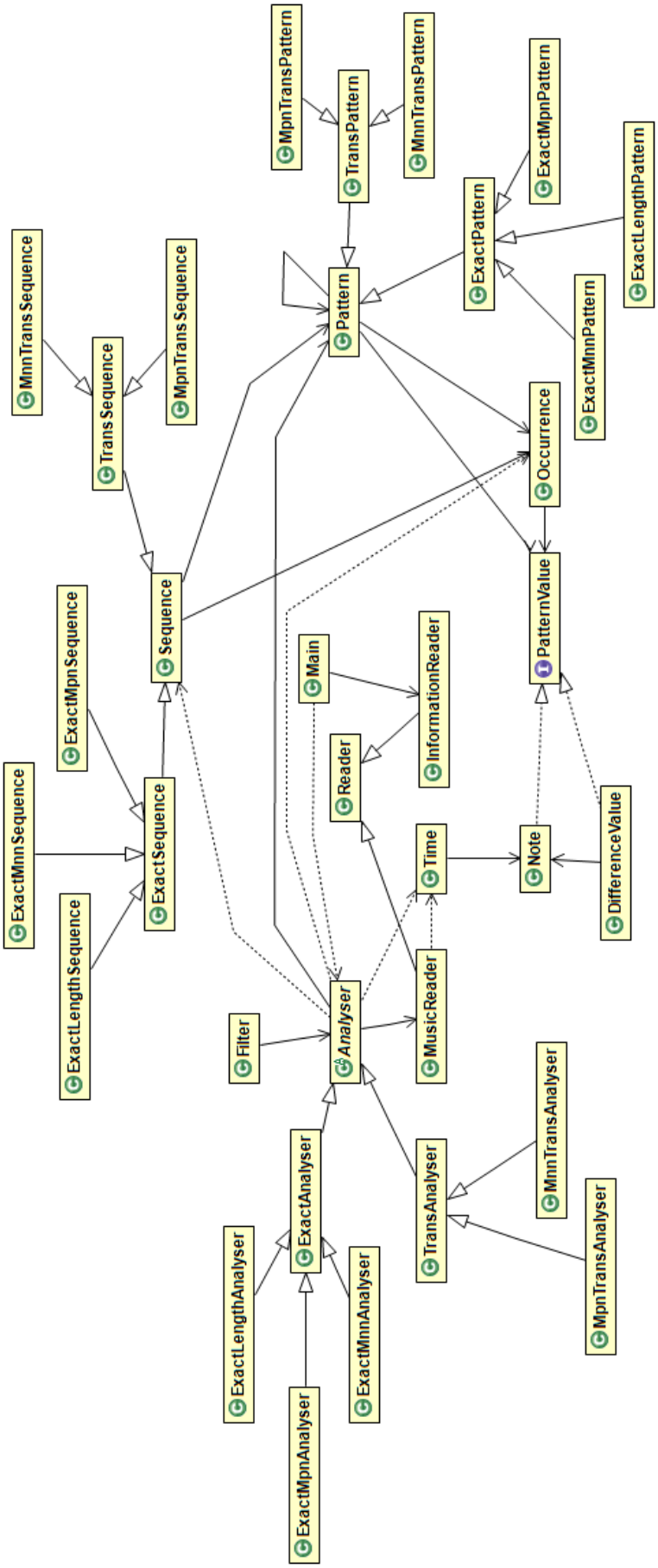

Figure 5.2: Diagram that shows all the classes of the program and the links between the classes

equal or more strict description. For example: a MpnTransPattern contains Occurrences where the mpn difference between successive Notes is the same. A TransPattern contains Occurrences where the difference in mnn and mpn between successive Notes are the same. A Pattern contains Occurrences where all the Notes are exactly the same. A Pattern is more specific than a TransPattern and a TransPattern is more specific than a MpnTransPattern.

5.2.2 Memory

Because Notes are processed in a one-pass approach, it is impossible to know which Notes follow each other if this is not stored. For example, when the program tries to extend a Pattern, it needs to know what the next Note after the last Note of the Pattern is. PatMinr does this by storing the next Note in the continuation memory of the Pattern. In this new system, some kind of memory is used. This memory stores the order of all the Notes in a list. The advantage of using such a list is that it knows the sequence of all Notes and can use it for the Pattern extension. By storing the sequence of Notes in a separate element, then all the information is stored once and it is stored in a structured way. It is therefore easier to retrieve information, but the side effect of having a separate element is that each time information needs to be retrieved from the memory instead of using the continuation memory. A Pattern exists out of Notes that do not start before the previous Note has stopped. A lot of Patterns are monophonic, but these needs to be found in a polyphonic environment. The polyphonic file is a list of Notes that are part of the musical piece. The file is sorted in time, but it is possible that sequential Notes start at the same moment. To make the data more intuitive, two extra features are added to the memory.

Multiple staffs Because monophonic Patterns need to be found using this polyphonic representation, the memory is split in different lists that each represent the Notes that are stored in a staff. When the program tries to extend a Pattern it will used the memory that corresponds to the staffs of its Occurrences. Only the relevant staff memory will be used because monophonic Patterns will only have Occurrences that stay in a staff.

5.2.3 Different lists of Sequences

Because sequential notes in a file are not always notes of the same voice is one of the reasons why PatMinr can not handle the polyphonic task. To solve this problem, Sequences are introduced which contains a Pattern and a list of PatternValues which is a candidate to become an Occurrence. The big advantage is that there it is possible to decide when to try to extend a Sequence. When a new PatternValue can not possibly extend the Sequence, then the program will not try to extend the Sequence. This is especially necessary for the polyphonic task because the notes of different voices are mixed together. Because different staffs always contain different voices, Sequences can only be extended by Notes of the same staff. To make it easier to follow this rule there are different lists of Sequences where a list represents all Sequences that can be extended by Notes of a certain staff.

5.2.4 PatternValue

To make it possible to search both `ExactPatterns` and `TransPatterns` the system needs to be able to find `Patterns` using both `Notes` and `DifferenceValues`. To avoid duplication an interface `PatternValue` is used which is implemented by both `Note` and `DifferenceValue`. These methods that use `PatternValues` are overriden by specific classes to determine what kind of `PatternValue` is used.

5.2.5 Collaboration between classes

The different interactions between the different classes can also be seen in figure 5.2. These connections give a good overview how the program works and how all the classes interact with each other.

5.2.6 Main program

The `Main` class is used to supervise the tasks that need to be done to retrieve `Patterns` out of a musical piece. An `InformationReader` is used to read meta data which contains the files that need to be read. It defines what type of `Analyser` needs to be used and to start the searching algorithm. It then filters the results using a `Filter` and returns the results.

5.2.7 Analysis of `Notes`

The `Analysers` control the search for `Patterns`. For each file that needs to be analysed, a different `Analyser` is used. The type of `Analyser` determines which kind of `Patterns` that are searched. The `Analysers` have a link to `Pattern` because each `Analyser` has a base `Pattern` which is expanded to become the pattern tree. The type of `Analyser` determines which type of `Pattern` is used. The `Analysers` also have a link to a `MusicReader` which converts the file into `Times` which are used in the analysis of the file. There is a link to Sequence because a list of `Sequence` lists is stored in the `Analyser` and used to extend candidates to become `Occurrences` of `Patterns`.

5.2.8 Pattern tree

Section 4.5 contains an explanation about how a pattern tree is structured. To make this possible, `Pattern` has a link to itself. This link shows that `Patterns` can have other `Patterns` as children. These children are the found extensions of the `Pattern`. These `Patterns` each contain a list of `PatternValue` which are the `Notes` of the `Patterns`. When the pattern tree is traversed, then each node represents the `Pattern` which consists of the `PatternValues` of all the general `Patterns` (parents of the node) and the `PatternValues` of the current `Pattern` node.

Chapter 6

Extensions

Chapter 4 explains the different components of the program and the algorithms that are stored in these components. Chapter 5 explains the structure of the program by giving an overview of the entire algorithm and explaining the architectural decisions. I have included a section about the architectural decision of my program because it is very useful for an algorithm to be adaptable and extendable. If an algorithm has a good structure, then it is easier to experiment with different changes and improve the algorithm. This chapter shows the advantages of having a good architecture by explaining important extensions that can be added in the future. This chapter also contains a section proving that extending the functionality of an algorithm instead of changing it has some advantages. Section 6.2 illustrates one of these advantages by proving that the system can handle both monophonic and polyphonic music. If the system would been changed instead of extended, which is not possible.

6.1 Searching polyphonic `Patterns`

The system currently searches for monophonic `Patterns` in a polyphonic environment. I have decided to only search for monophonic `Patterns` because this is a more natural extension of the PatMinr algorithm. Another reason why I decided to look for monophonic `Patterns` instead of directly searching for polyphonic `Patterns`, is that a lot of the ground truth `Patterns` are monophonic. My algorithm can find these `Patterns`, if it would search for polyphonic `Patterns`, it would be nearly impossible to find the `Occurrences` of monophonic `Patterns`.
It is possible to find polyphonic `Patterns` if all repetitions of monophonic `Patterns` are found. Finding polyphonic `Patterns` can be done by combining these monophonic `Patterns`. Because the algorithm has found all `Occurrences` of the monophonic pattern and can therefore look up when all these `Occurrences` happen, it is possible to calculate if a combination of monophonic `Patterns` also happens multiple times.
Another advantage of combining monophonic `Patterns` instead of looking for repetitions of sections is that the algorithm will be able to find polyphonic `Patterns` where only a subset of the voices are repeated.

6.1.1 Adjusting the system to search sections

In my opinion it is not possible, or at least be very hard to find monophonic `Patterns` when an algorithm returns sections. The current system is only capable of searching monophonic `Patterns`, but can be adjusted to search for sections instead. This would be interesting when only the repeated sections need to be returned. It is possible that the same proposed extension is used for finding polyphonic `Patterns` to find sections. When only sections need to be returned, then it would be faster to search sections instead of searching monophonic `Patterns` and then combining them. When searching for sections, it may be easier to work with repetitions in the file than converting the file into objects like `Times` and then try to find `Patterns` of `Times`. Because the polyphonic file is very structured, this can be used to find sections. When `Notes` start at the same time then the `Notes` of the lowest staff are mentioned first, ... When a section is repeated exactly, then the repetition will also be exact in the file. When searching for a repetition of a section, it will be easiest to let the program like with monophonic `Patterns`, but some structures need to be removed or changed.
There are currently a lot of systems in use that handle `Notes` of different staffs differently. Notes of different staffs are stored in different lists in memory. These systems try to extend only `Sequences` with `Notes` of the same staff, ... These structures need to be removed so all `Notes` are handled the same and they have to be stored in the same place. There are also rules implemented to avoid the creation of bad `Patterns`. An example of such a rule is that `Sequences` can only be extend with `Notes` which do not start before the end of the last `Note`. When searching for sections, the system needs to try to extend a `Sequence` with a consecutive `Note` from the file.
It would be easier to search for sections than it is searching for monophonic `Patterns`. An indication that this statement is true, is the fact that a previous version of my system, which was an adaptation of the PatMinr algorithm, could solve the monophonic task. :y previous system was almost able to find sections in the polyphonic task because the monophonic task also tries to extend `Sequences` with consecutive `Notes`. Finding sections in the polyphonic task can therefore be solved by systems that searches `Patterns` for the monophonic task.
The current system can not find section `Patterns`, but by using a combination algorithm on the monophonic `Patterns` or by using an algorithm that is similar to the algorithm that can solve the monophonic task, these `Patterns` can be found.

one pass approach The system needs to combine monophonic `Patterns` that happen at the same time to get sections. To find all possible combinations of monophonic `Patterns` and find all repetitions of these sections, a new algorithm needs to be constructed, because a file can have a lot of `Occurrences` and if all these `Occurrences` need to be compared with each other, then it can take a lot of time.
I propose to use an algorithm that is an adaptation on the sweep line algorithm like the one that is described by Cormen [9]. The `Occurrences` need to be ordered first by begin time and then by end time. The algorithm needs to 'sweep trough' this list so it can efficiently combine monophonic `Patterns` into sections.
To find if sections are repeated, `Occurrences` need to check if other `Occurrences` of the same `Pattern` are part of a similar section as the current `Occurrence` is in. The simplest way to do this is to go trough all sections and compare them with all other sec-

tions. Because this has a complexity of O(n^2) this best needs to be done more efficient. If `Occurrences` store a list of sections that they are part of, then another `Occurrence` can just check if they have a common section by comparing each others sections. The complexity of the checking for the same sections would then be O(n).

6.2 Handling the monophonic task

My program is able to handle the monophonic task. This is possible because the system is evolved from a system for the monophonic task to a system that can also handle the polyphonic task. The functionality is added by generalising the system and extending the functionality of different parts and by adding new parts to handle the new requirements. I have chosen to extend the system instead of changing it to handle the new task because it can keep solving the monophonic task and by improving the monophonic system, then the polyphonic system will also improve. An overview of the functionality that is added to make the system able to solve the polyphonic task.

- The system got different lists for storing `PatternValues` in the memory.
- There are different lists of `Sequences` where each list represents `Sequences` that are extended using `PatternValues` with the same staff.
- The system works with `Times`, which collects all `Notes` that begin on the same time and are placed on the same staff.
- The system will try to extend `Sequences` with a 'time'. Because `Sequences` contain a sequence of `PatternValues`, not `Times`, the system will try to extend the system with each of the `PatternValues` that the `Time` contains.
- The system will only try to extend a `Sequence` with a `PatternValue` if the `PatternValue` has the same staff value and starts later or on the same time as the time when the last `Note` of the `Sequence` stops.

All these extensions have an influence on how the system handles the monophonic task. Despite the differences that are introduced to handle with an extra difficulty of having different voices happen at once and having the `Notes` in a different order, it is still possible to find pattens in the monophonic task.
The system does not have problems with handling the monophonic task because the differences do not change anything or do not have any influence on the processing of the `Notes`.

- Because the monophonic task also only has `Patterns` where the `Notes` are placed on the same staff, all the `Notes` of the `Pattern` will always be placed consecutively in the memory of the same staff.
- If `Notes` follow each other, then they are possible candidates for extending all current `Sequences`. The file in the monophonic task is structured differently than the polyphonic task. All notes of a voice are placed after each other in the file and these sequences of notes are sorted by staff. The consecutive notes in the file will also happen after each other. There is not an overlap between consecutive notes. The

monophonic task therefore only needs to look for Patterns in in consecutive notes. This will automatically be done while iterating over the file and all the consecutive notes will pass the basic tests to check if they are possible PatternValues of the same voice.

- When solving the monophonic task, all the Notes will also be added to Times. The difference between the polyphonic and monophonic tasks is that the monophonic task can only contain one Note. This can be derived from the fact that monophonic music does not have Notes that overlap in time and Times only contain Notes that begin on the same time and therefore overlap. The advantage that Times only contain one Note is that Sequences only need to be extended using this Note. It is not possible that that multiple Sequences will be created out of a single Sequence because the extension of Notes is the only point in the algorithm where this can happen, and this does not happen in the monophonic task.

This indicates that the system is general enough to keep handling the monophonic task.

6.3 Improving Occurrence amount and Pattern length

There are still parts that can be improved and where the improvements will cause a big improvement in the results.

Sequence strategy One of these part is the strategy when Sequences are added and removed from the Sequence list. The Sequence list stores only Sequences that the program still needs to try to extend. This strategy is one of the main parts of my system and a basic strategy has been implemented in the system. The strategy is important because the quality of the strategy is one a main factor for the quality of the found Patterns.

Sequence extension rules The addition of a new Sequence with only one Note decides when a Pattern will begin, which is a very important decision. In the current system, Notes can only be added as the beginning of a new Sequence when there are no other Sequences left in the list of Sequences. The program is therefore very strict about when new Sequences can be created. It is rather natural to have such strict rules because Patterns are spread in time to help people notice them. Therefore Patterns almost never overlap with other Patterns.

Consequences There are other consequences involved with having such strict rules. Because Sequences only start when other Patterns have stopped, it is possible that the start of a Sequence is too early and started expanding based on the wrong Notes. When a Sequence is started too early, the probability of expanding it for longer than a few Notes is rather low. This happens because the Sequence probably will not have happened before. A bigger problem is when a Sequence is started too late. It is possible that another Sequence also begun too late in a previous Occurrence of the Pattern. If that has happened, the Sequence will be extended with the remaining Notes of the Pattern and the Occurrence will be added to the incomplete Pattern.

Longest `Patterns` Another problem with trying to keep extending a `Sequence` until no previous `Occurrence` is found, is similar. The program will try to search for the longest `Patterns`. Trying to find them can often lead to finding important `Patterns`, because the longer a pattern is, the lower the probability of it happening by chance. Therefore longer `Patterns` are often important `Patterns`. The disadvantage of this method is that it neglects sub-patterns of these longer patterns that possibly are more important than the longer pattern. For example, when the patterns `abcdef` and `ghadef` are found, the algorithm will not have noticed the sub-patterns `def`.

Solution for find `Occurrences` of sub-patterns One method to find the sub-patterns would be to go trough all `Notes` of each `Pattern` and see if the are the start of an existing `Pattern`. If a sub-pattern is found, then all `Occurrences` of the `Pattern` will also be added as `Occurrences` of the sub-pattern. The advantage of this method is that it will find all `Occurrences` of sub-patterns, but the big disadvantage is that it is rather computation intensive to do this. A more efficient way to handle this problem can be developed by using the structure that is already available. The `Patterns` in the system do not represent `Notes` or the complete `Pattern`, but sub-patterns with `Notes` that will always follow each other. It is not therefore necessary to go trough all the `Notes`, only the `Notes` that start a new sub-pattern.
Finding sub-patterns can also be done during the one-pass algorithm. This can be done by using the `Sequence` functionality. Every time a `Sequence` is extended and needs to look at a specific `Pattern` if it can be extended by the `Note`, it will also create a new `Sequence`, which begins with the current `Note`. If it will then also be extended using the standard algorithm and the system will be able to automatically add all sub-patterns. If all `Occurrences` need to be found, then this needs to be done at each transition. One kind of these transitions is also done when `Patterns` are split. When a `Pattern` is split, a new `Sequence` will be created which begins with the current `PatternValue`. All the existing `Occurrences` also need to be checked if they are part of a sub-pattern. The advantage of using the second strategy is a higher efficiency, but it is possible that it will not find all `Occurrences` of the sub-patterns. If the `Pattern` will not split at the beginning of the sub-pattern, then the algorithm will not find the sub-pattern `Occurrences`.
In my opinion is the second solution a good solution to find a lot of sub-patterns and sub-pattern `Occurrences` while only adding a little computational intensity. It is important for the quality of the output to find all sub-pattern and sub-pattern `Occurrences`, because these sub-patterns can be more important than the full patterns. It will also start more `Sequences`, so it can also cause an increase in pattern length because it is possible that patterns will start at a better time.

6.4 Conclusion of this chapter

This chapter shows that the current system is multifunctional and can be extended with features to improve the quality of the output. Section 6.1 explains that the system currently searches monophonic patterns. This is done because monophonic patterns can be combined in sections. The other way around is much harder, if not impossible. It is much easier to find the sections than it is to find the monophonic patterns. Even a previous version of my algorithm could find sections in polyphonic music by finding repetitions in

the the file. This method is very simple but there is no control over it. For example, it would never be able to find sections if only a part of the voices are repeated instead of all the voices. The section also briefly describes an algorithm that could be used to find those sections. This algorithm would use an adaptation of the sweeping line algorithm to find overlapping monophonic patterns to combine them in sections. The algorithm will then need to find all repetitions of these sections. This can all be done very efficiently.
Section 6.2 explains how the system is an extension of the algorithm that is used to participate in the monophonic task. It explains that these are extensions and generalisations rather than changes. Because these are not changes, the system can still handle the monophonic task and improvements on the monophonic task will have an positive effect on the results of the polyphonic task.
Section 6.3 explains that not every occurrence of a pattern is found. The biggest problem is finding occurrences of sub-patterns. The current system is capable in finding long patterns, but is not capable in finding all sub-patterns and their occurrences. An adaptation to the algorithm is proposed by using existing parts of the system for finding these occurrences. The system needs to create new `Sequences` each time a sequence is extended by using a specific pattern.
These are all important extensions that need to be implemented in the future to improve the quality of the output. I have explained what currently the weakest points of my algorithm are and how they can be fixed. Because I have created a good structure for the algorithm. These changes will be rather easy to implement. I already implemented them in a new version of the system, but because this version still has some bugs, it can not be used to generate results.

Chapter 7

Evaluation

7.1 Evaluation measures

All the algorithms that were submitted to the Mirex task are evaluated using the same evaluation measures. Almost all of these measures are based on general measures that are used in different machine learning tasks. The standard precision, recall and F1 scores are determined, but also adapted versions of these algorithms to make them more suitable and robust to evaluate the music pattern searching algorithms. Adaptations of these standard measures use the establishment of patterns or look at the occurrences of the patterns. There is also a three layer measure defined that is based on precision, recall and F1. The last measures use the runtime, the time to return the first five patterns, and the establishment of the first five patterns. The establishment and occurrence versions of precision, recall and F1 are developed by Tom Collins [4]. He also proposed the use of runtime.

7.1.1 Standard

The measures to calculate the standard precision, recall and F1 scores are generally accepted as solid tests. To calculate these metrics, some auxiliary values are calculated and combined. The total amount of ground truth patterns is represented by Tg., the amount of found ground truth patterns is represented by Fg. and the total amount of found pattern is represented by Tf. The precision P is defined as $P = Fg/Tf$, the recall R is defined as $R = Fg/Tg$ and the F1 score is defined as $F1 = 2PR/(P + R)$.

Problems with standard measures The standard measures are not really useful for deciding the value of an algorithm. The biggest problem with the standard measures is that it uses the value Fg, the found ground truth patterns. In these measures, a pattern is defined as a found ground truth pattern when the found pattern is exactly the same as the ground truth pattern. This definition of found ground truth pattern is much too strict to be useful. When an algorithm finds a pattern that differs only one note from the ground truth, it will not be accepted. Often it happens that patterns are found that only cover a part of a ground truth pattern. Because these partial patterns are also valuable the metrics need to show that they are. This is done by adjusting the measure and making them more robust to small differences.

7.1.2 Robust versions of the standard measures

Because the standard measures can not handle inexact found occurrences, which are still interesting to find, the output of the algorithms is also evaluated using adapted versions of the normal measure.

Element representation The following measures use the patterns that are stored in the ground truth and patterns that are found, as well as the occurrences that are stored in ground truth patterns and occurrences that are stored in found patterns using the algorithms. To represent these elements abbreviations are used.

- The patterns in the ground truth P: $\{P_1, P_2, ..., P_{np}\}$
- The patterns Q that are found using an algorithm: $\{Q_1, Q_2, ..., Q_{nq}\}$
- The occurrences of the ground truth pattern P_o: $\{P_{o1}, P_{o2}, ..., P_{mop}\}$
- The occurrences of a pattern Q_o found in the output of an algorithm: $\{Q_{o1}, Q_{o2}, ..., Q_{moq}\}$

Cardinality score To make the measures more robust, a cardinality score is used to define symbolic musical similarity. Patterns do not need to be found exactly to be recognized, they need to be similar. If only a part of the pattern is found, then the cardinality score of the occurrence is a positive number instead of zero. The similarity score is defined as $s_c(P_i, Q_j) = |P_i \cap Q_j|/max|P_i|, |Q_j|$. To get an overview of the scores that are given to the found occurrences and the stored occurrences, a score matrix s(P_i, Q_j) can be used. Each cell inside the matrix represents the similarity score of the two patterns it represents.

Measures adaptation based on establishing that patterns are found in at least one occurrence It is useful to know algorithms can find a least one occurrence of patterns. To know how much patterns are found, the establishment scores represent the amount of patterns where at least one occurrence was found. To determine the most similar occurrence that is found in for a pattern, S(P,Q) is used. S(P,Q) is the maximal value in the score matrix.
The establishment precision is defined as: $P_{est} = (1/nq) \sum_{j=1}^{nq} max\{S(P_i, Q_i)|i = 1, ..., np\}$.
The establishment precision process first searches for each found pattern what the max S(P,Q) is. The establishment precision is defined as the sum of all these max values and normalized by dividing by the amount of found patterns. The establishment precision is therefore maximal if each found pattern contains an occurrence that is exactly the same to an occurrence of a ground truth pattern.
The standard recall is defined as: $R_{est} = (1/np) \sum_{i=1}^{np} max\{S(P_i, Q_i)|j = 1, ..., nq\}$. This is very similar to the precision process. It will look for the maximal S(P,Q) for each ground truth pattern, and calculates the normalized sum of these values. The standard recall is maximal if each ground truth pattern contains an occurrence that is exactly the same to an occurrence of a found pattern.
The standard F1 score is similar to the standard F1 but uses the standard precision and recall.

Measures adaptation based on the ability of finding all occurrences of a pattern The measure tries to find out how well an algorithm performs at retrieving all occurrences of discovered patterns. It does not take in account patterns that are not discovered. Even if it finds all occurrences of only a subset of patterns, then it will return a good result. Because the measure only needs to handle found patterns, it will look in the establishment matrix for values above a threshold value. If it is over, then the measure presumes that the pattern is found. The measure will then construct an occurrence matrix with the found patterns and the relative ground truth patterns. For each of the cells inside the matrix, it will calculate the precision of the score matrix that belongs to the given $s(P_i, Q_j)$. The recall precision matrix can be calculated in the same way, but is every time the recall value used instead of the precision score. The F1 occurrence value is again a combination of the precision and the recall.

7.2 State-of-the art systems

To add context to the evaluation of the output, my developed system is compared with current state of the art systems. My system is compared with participants of the 2014 Mirex task and the best participants of the 2013 Mirex task. If the system contains "13" in its name, then the system participated in the 2013 Mirex task. Some of the systems are capable of handling the monophonic task as well as the polyphonic task. If this is the case for a given system, then this system is mentioned in the list of both tasks.

7.2.1 Monophonic systems

NF1 This algorithm is developed by Oriol Nieto and Morwaread Farbood [24]. The algorithm uses segmentation techniques and a greedy path finder algorithm to discover the patterns. This is the only algorithm that participated in every sub-task of the Mirex 2014 task.

OL1 This code represents the algorithm developed by Olivier Lartillot. This submission is the PatMinr algorithm which is discussed in detail in chapter 3.

VM1 VM1 is one of the two algorithms developed by Gissel Velarde and David Meridith [27]. They use musical segmentation and classification based on filtering with wavelets.

VM2 VM2 is the second algorithm developed by Gissel Velarde and David Meridith [27]. It uses the same techniques as VM1, but it uses different values as parameters.

NF1'13 This is the previous version of the NF1 algorithm by Oriol Nieto and Morwaread Farbood [22]. The algorithm extracts a chromagram from the given input. The algorithm uses this chromagram to produce a key-invariant self-similarity matrix using the Euclidean distance.

DM10'13 This submission is developed by David Meredith. This submission contains 2 greedy compression algorithms COSIATEC and SIATECCompress [20]

7.2.2 Polyphonic systems

NF1 NF1 is explained in section 7.2.1.

NF2'13 NF2'13 is comparable with NF1'13 (see section 7.2.1, only this system is developed to handle the symbolic polyphonic music.

DM10'13 DM10'13 is explained in section 7.2.1.

7.3 Results

This section shows the results of the experiments I have conducted with my system. First I compare my results of the polyphonic task with the 2014 Mirex task participants. The systems of the participants are discussed very short at section 7.2.2. At the end of the section my results of the monophonic task are compared with the monophonic participants of the 2014 Mirex task. The participants of the monophonic task were discussed at section 7.2.1. PatMinr, the algorithm where I based my algorithm on, is one of the participants of the monophonic task. I will therefore also compare my results with the results of PatMinr. The results are evaluated using the metrics which are described in section 7.1. The graphs with the information about the results of the Mirex task are taken from the site of the task [5]. These graphs contain all the results of the systems that are described in section 7.2. Each line in the graphs represents a participant of the task. The tables in this section represent the results of my algorithm for the different measures. The tables are created using data that is generated by evaluating the output of my algorithm with the provided tests. The setup that is used to generate the results is described in section 7.3.1.

7.3.1 System setup

The results that I will refer to are generated using my system which I have described in chapter 4 and chapter 5. For the polyphonic task I have searched patterns where the difference of the Mnn or Mpn values between consecutive `Notes` is the same. This type of pattern is called `MnnOrMpnTransPattern` inside the system. The system has filtered the patterns before returning them. The system used a basic filter based on the amount of found occurrences and the length of pattern. The score of the pattern needs to be above ten, the amount of occurrences needs to be more than one and the pattern size needs to be bigger than three. The score is a weighted sum with the weight one for the pattern size and two for the occurrence amount, in order to make the amount of occurrences more important. The runtimes are achieved by executing the algorithm on a normal laptop. I do not know what machines were used for executing the other algorithms.
I also executed the monophonic task using the same setup.

7.3.2 Polyphonic systems comparison

This section discusses the different aspects of evaluating the polyphonic task. The ground truth of the polyphonic task is discussed. The results of the measures of my system for

the polyphonic task are also discussed and compared with the participants of the Mirex 2014 polyphonic task.

Polyphonic ground truth

To generate a good output, the system needs to be able to handle the problems it is faced with. By focusing on the results there it is possible to what the strong and the weak points of the system are. First here is a general overview of the different files and what kind of ground truth patterns they contain. When the system tries to retrieve the different types of patterns, it is faced with different problems.

Bach The piece from Bach contains three ground truth patterns. All these pattern are monophonic. The system is able to find all these patterns, but the main problem with these patterns is that the occurrences of the same pattern can have different lengths. It is even possible that there exist occurrences with a size different from the other occurrences of that pattern. The length of the patterns varies; two of the patterns are rather small with around five notes. There is also a pattern with around twenty notes. The system can find these patterns, but has difficulty with finding the entire patterns because they do not have the same size. The biggest problem is the different sizes of the occurrences. The notes of the longer occurrences often have notes that are placed before the rest of the pattern that they have in common. The system extends the patterns with suffixes, so it is easy to find extensions at the end of a pattern. It is currently not possible for my system to find occurrences with extensions as a prefix.

Beethoven The piece from Beethoven does not only contain monophonic ground truth patterns, but also polyphonic ground truth patterns. The monophonic patterns are rather easy to find because they are not very long and have many occurrences. The piece also contains polyphonic patterns, which can be a section or a normal pattern with a limited amount of notes. The system has the most problems with finding polyphonic patterns. Because the system is currently developed to find only monophonic patterns. The system will find parts of the polyphonic patterns, because polyphonic patterns are a combination of monophonic patterns.

Chopin The mazurka of Chopin also contains monophonic and polyphonic patterns as well as sections. The monophonic patterns overlap is more difficult to retrieve. There are a lot of occurrences of the pattern, which makes it easy to recognize the pattern, but difficult to find all the occurrences. The problem with the sections and the polyphonic patterns is the same as with the piece from Beethoven.

Gibbons Silver swan is the shortest piece of the five. It is also very hard to define patterns for this piece. Three patterns are very short because they only have four or five notes. They are recognizable because there are multiple occurrences. The last pattern is a section that only has two occurrences. The patterns are very different and this makes it harder to find all the patterns and their occurrences because it is not clear what the definition of a pattern is. It is also hard to find all the occurrences because most of the occurrences are very small. Because they are so small, if a note is missed, it will be regarded as an unimportant occurrence and will not be returned.

type/piece	1	2	3	4	5
recall	0.60714	0.39599	0.30741	0.37277	0.44848
precision	0.25837	0.23014	0.07464	0.2502	0.25292
F1	0.36249	0.2911	0.12012	0.29943	0.32344

Table 7.1: Establishment recall, precision and F1 of output of my system

Mozart The sonata from Mozart is one of the easiest pieces, because it is rather 'normal'. The patterns all have a decent length and a decent amount of occurrences. It contains polyphonic patterns, but these can be described as monophonic patterns with some additions from other voices. The system will perform well on these pieces because a lot of notes are on the same voice. The only difficult pattern is the sectional repetition because it is very long and because the notes are spread over multiple voices.

Establishment measures of polyphonic task

Figures 7.1, 7.2 and 7.3 show the results of the establishment metrics of the polyphonic algorithms. Respectively the figures show the establishment recall, establishment precision and establishment F1. Table 7.1 shows the results of my algorithm.

Recall The values of the establishment recall are comparable with the values of the state-of-the-art systems. The establishment recall of my system regarding the pieces of Bach and Mozart is better than any of the three systems. The establishment recall of the system regarding the other pieces is higher than the weakest state-of-the-art system of each piece. Every piece except Bach contains polyphonic patterns. The current system can find one voice of these patterns and therefore the recall is still comparable with the state-of-the-art systems.

Precision The precision is comparable with the worst submitted system. It is low because the found occurrences are monophonic, not polyphonic, which makes them less similar to the ground truth patterns. The patterns are filtered by a basic filter. This basic filter only filters out patterns that can not be important based on the occurrence amount and pattern size. Because of this basic filtering, the output still contains a lot of unimportant patterns.

F1 Because the F1 depends on the recall and precision, it is logical that F1 scores are also comparable to the current systems. Because the precision is quite low, the F1 score is sitting between the second and third place.

Occurrence measures of polyphonic task

The occurrence recall, precision and F1, evaluate how similar the returned occurrence are to the most general occurrence and how many occurrences are found when a pattern is found. The graphs with the occurrence metric values in it are figure 7.4 with occurrence recall, figure 7.5 with occurrence precision and figure 7.6 with occurrence F1 scores. The occurrence measure values of my system are given in table 7.3.2.

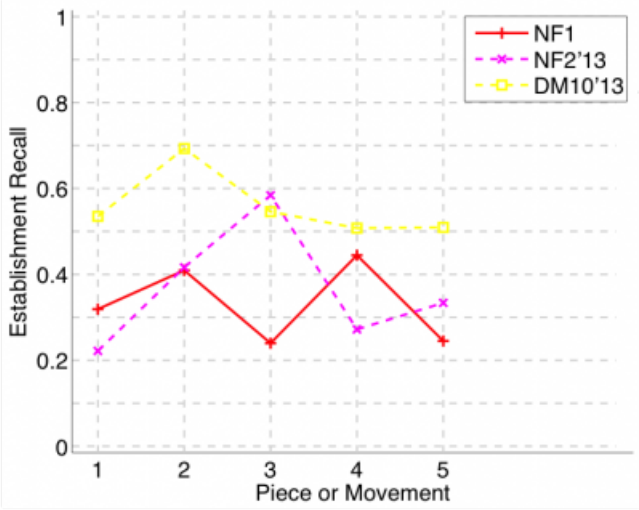

Figure 7.1: Establishment recall of participating polyphonic systems

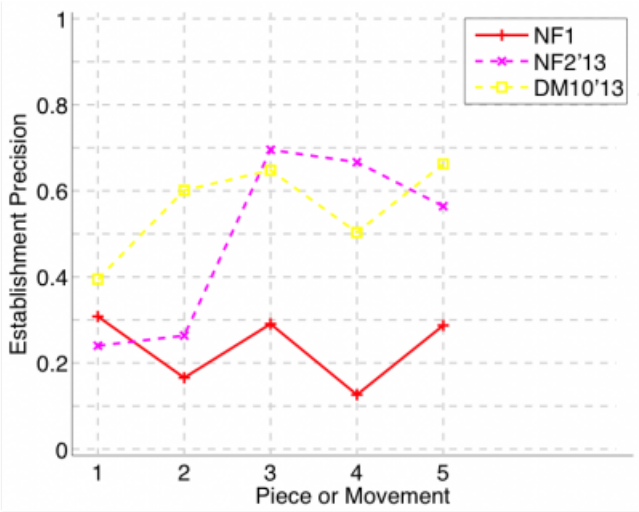

Figure 7.2: Establishment precision of participating polyphonic systems

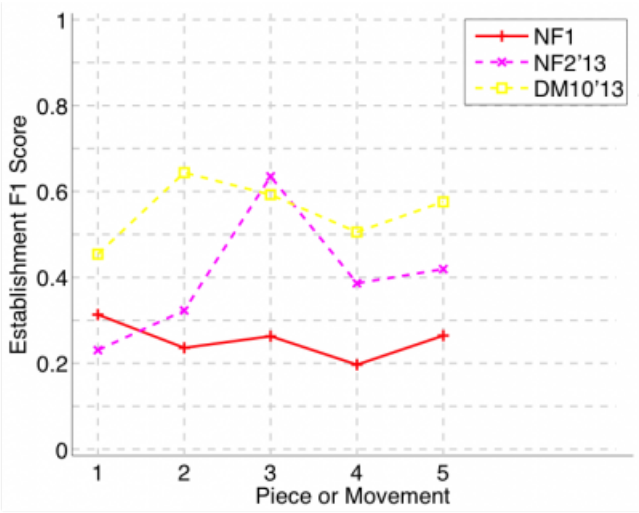

Figure 7.3: Establishment F1 of participating polyphonic systems

type/piece	1	2	3	4	5
recall	0.23937	0	0	0	0.39286
precision	0.83208	0	0	0	0.78571
F1	0.37179	0	0	0	0.52381

Table 7.2: Occurrence recall, precision and F1 of my system for the polyphonic task.

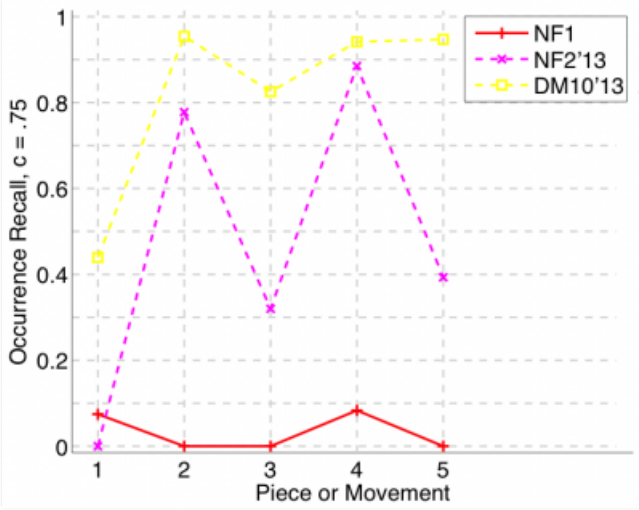

Figure 7.4: Occurrence recall of submitted polyphonic systems

Recall The recall is rather low. This is because the system is not yet capable of finding sub-patterns and therefore cannot find occurrences of sub patterns. The system can currently only find monophonic patterns and their occurrences in polyphonic pieces. The occurrences of monophonic patterns will only contain a small part of the notes of the polyphonic pattern. Because the similarity between the occurrences is lower than the threshold value, monophonic occurrences will not be recognized by the occurrence measure as an occurrence of the polyphonic pattern. Because some of the music pieces only contain polyphonic patterns, the occurrence measures are equal to zero. The occurrence recall will therefore be zero for the pieces 2-4.

Precision The precision is rather high, because the patterns are filtered. The output only contains patterns that has at least two exact occurrences. The rules are very strict before occurrences can be returned. The occurrences are therefore probably rather important. Therefore the precision is also rather high because only high quality occurrences are returned. Because the occurrences of the monophonic patterns are not recognized as occurrences of the polyphonic pattern, the occurrence precision of the system of pieces 2-4 equals zero.

F1 The occurrence F1 values of my system are zero for pieces 2-4 because monophonic occurrences can not be recognized. The occurrence F1 values of piece 1 and piece 5 are comparable with the current state-of-the-art systems. This indicates that the system is capable of finding monophonic occurrences in a polyphonic piece.

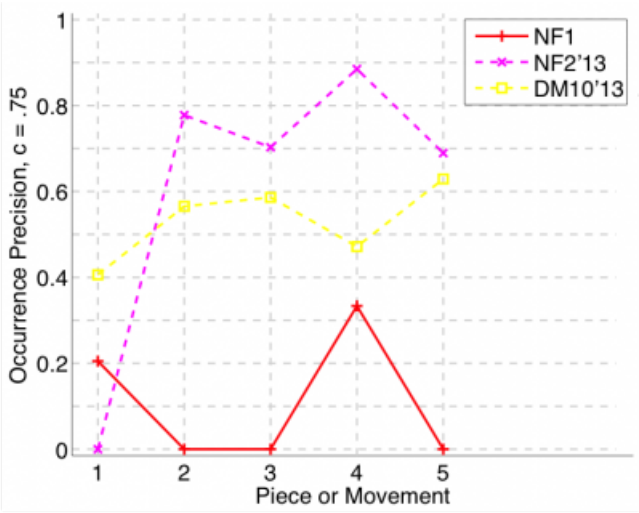

Figure 7.5: Occurrence precision of submitted polyphonic systems

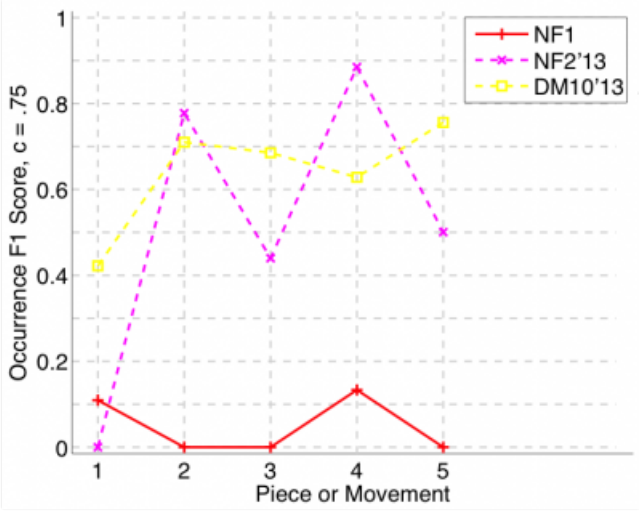

Figure 7.6: Occurrence F1 of submitted polyphonic systems

type/piece	1	2	3	4	5
runtime in seconds	0.269	0.440	0.342	0.018	0.333

Table 7.3: Runtime of my system to generate the output of the polyphonic pieces.

Runtime of polyphonic task

Figure 7.7 shows the runtimes of the different submitted systems for the polyphonic Mirex task. It is very clear that my system is a lot faster than the current state-of-the-art systems. It takes less than 0.5 seconds to calculate the patterns of any piece. If you compare it to the runtimes of the other pieces, then this is incredibly fast. It takes them between three and eleven seconds to retrieve the patterns out of the smallest piece, piece four. It takes more than 1000 seconds to retrieve the patterns out of the second piece.

Complexity handling polyphonic task

The second piece contains around 1500 notes, the fourth piece around 300. Piece four havs five times as many notes, the runtime of the second piece is more than 100 times as high as the runtime of the fourth time. This means that the computational complexity of the systems will likely be $O(n^3)$. The runtime of my system to process the fourth piece is around 20 milliseconds. the runtime of my system to process the second piece is around 400. The complexity of my system will be less than $O(n^2)$.
The submitted systems basically compare matrices with each other. The $O(n^3)$ is therefore a logical complexity. My system goes trough all the notes sequentially and processes them. The complexity is higher than constant, because my system works with `Times`. When a `Sequence` gets extended by a `Time`, then a sequence will be created for each of the notes in the `Time`. This can be compared with a combination of notes, which causes a complexity of (n^2). Because not all `Times` contain multiple notes, not every time will create multiple `Sequences`.
There is not a step in the algorithm while processing a `Note` that requires a lot of computational power. This cannot be said about the other systems. Therefore my system is a lot faster than the submitted systems.

7.3.3 Monophonic systems comparison

Section 6.2 explains that the systems is based on the monophonic task and that achieving good results in the monophonic task help with also getting good results in the polyphonic task. The section also explains that the current system is capable of participating in both the monophonic and polyphonic task. Because the system is very hard based on the monophonic task and because it was inspired by the PatMinr, it is interesting to compare the current system with participants of the monophonic task. The results of my system for the monophonic task can all be found in table 7.4.

Establishment measures of the monophonic task

The establishment recall, precision and F1 of the participants can be seen in figures 7.8, 7.9 and 7.10. The results of the establishment measures of my system of the monophonic

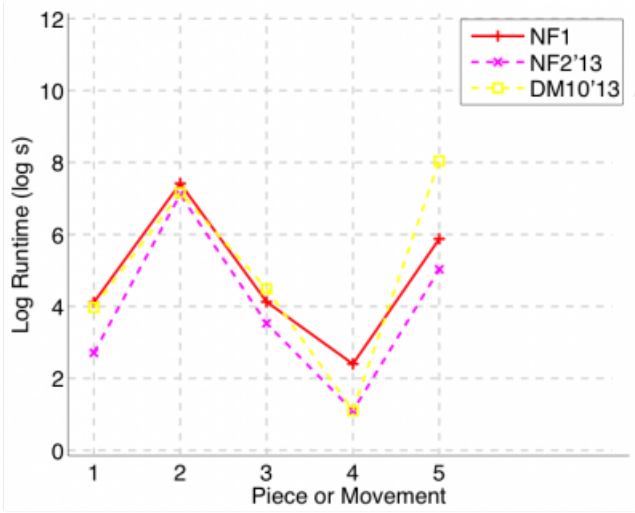

Figure 7.7: Runtime of submitted polyphonic systems

type/piece	1	2	3	4	5
establishment recall	0.87249	0.82272	0.85548	0.70435	0.61202
establishment precision	0.32388	0.48316	0.27619	0.65435	0.31559
establishment F1	0.4724	0.60879	0.41757	0.67843	0.41644
occurrence recall	0.36334	0.69491	0.36264	0.87696	0.86118
occurrence precision	0.77463	0.90712	0.86528	0.87696	0.87696
occurrence F1	0.49466	0.78696	0.51108	0.87696	0.86118
runtime	0.294	0.167	0.250	0.027	0.313

Table 7.4: Establishment, occurrence and runtime measures of my system

task can be seen in table 7.4. The establishment recall of the current system is high, compared to the other systems. For the first three pieces it has the highest establishment recall of all the systems. The establishment recall of the last two pieces is the second best of all the participants.
The quality of the establishment precision is not that good. It is comparable with the worst participant for each piece, except the second piece where it was the second best.
These results indicate that the system can find all important patterns, but that it also returns a lot of unimportant patterns. The system can therefore be improved by making the filtering more strict.
The establishment F1 of my system is comparable with the establishment F1 of the state-of-the-art systems.

Occurence measures of monophonic task

Figures 7.11, 7.12 and 7.13 show the occurrence recall, precision and F1 of the participating monophonic systems in the Mirex 2014 task. The occurrence measures of my system of the monophonic task can be seen in table 7.4.
The occurrence precision of the system is very good. It is comparable with the occurence precision of PatMinr which is the best system in occurrence precision. The occurrence

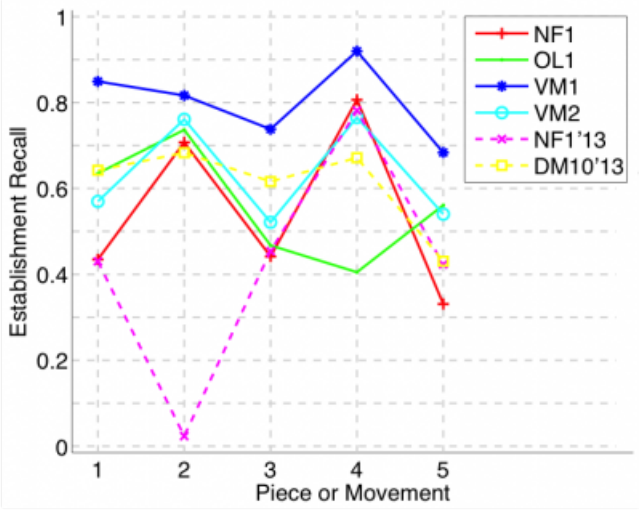

Figure 7.8: Establishment recall of the participants of the Mirex monophonic task

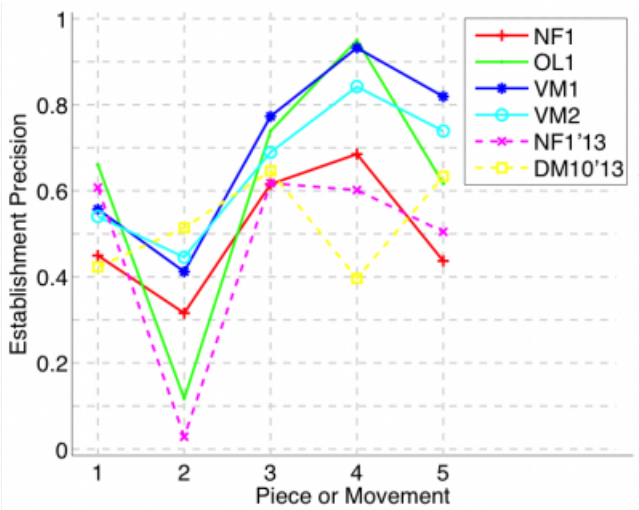

Figure 7.9: Establishment precision of the participants of the Mirex monophonic task

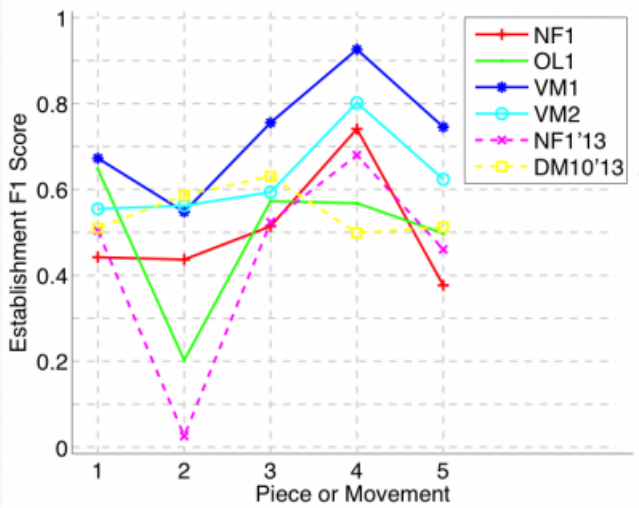

Figure 7.10: Establishment F1 of the participants of the Mirex monophonic task

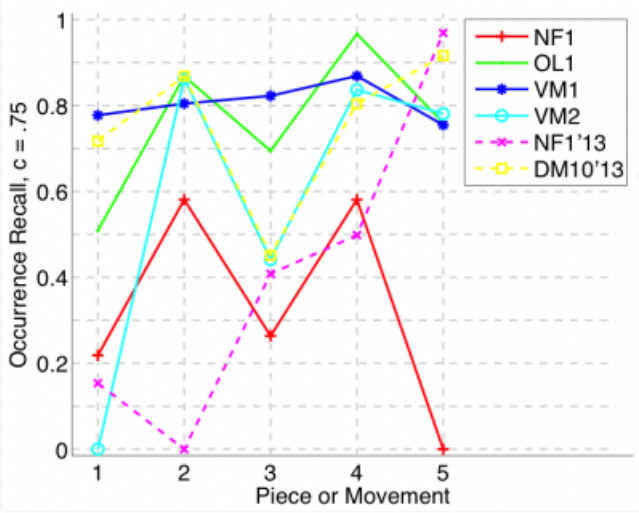

Figure 7.11: Occurrence recall of the participants of the Mirex monophonic task

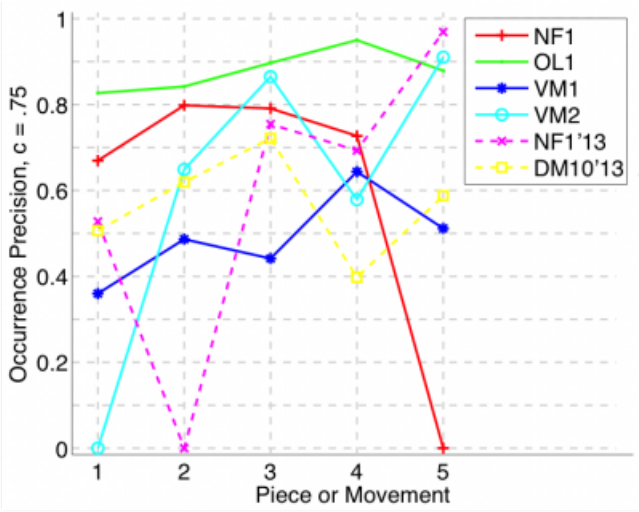

Figure 7.12: Occurrence precision of the participants of the Mirex monophonic task

recall is mediocre. It is around the median for each piece.
The system can therefore find the most important patterns and the occurrences of these patterns are very precise. The system currently can do better in finding more occurrences of these patterns. It is important that sub-patterns need to be added as occurrences in the future to increase the occurrence recall.
The occurrence F1 of my system is comparable with the occurrence F1 of the other systems. The occurrence F1 value is for some pieces one of the best.

Runtime Like with the polyphonic task, the difference in runtime is very big. My system is a lot faster than the rest of the systems. The runtimes of the participating algorithm are shown in 7.14. The return time of the patterns found in the polyphonic version or in the monophonic version are comparable. The monophonic version is returned faster because each `Time` contains only `Note`. The system will therefore only need to expand `Sequences` with only one `Note`. This is faster than having to expand the sequence

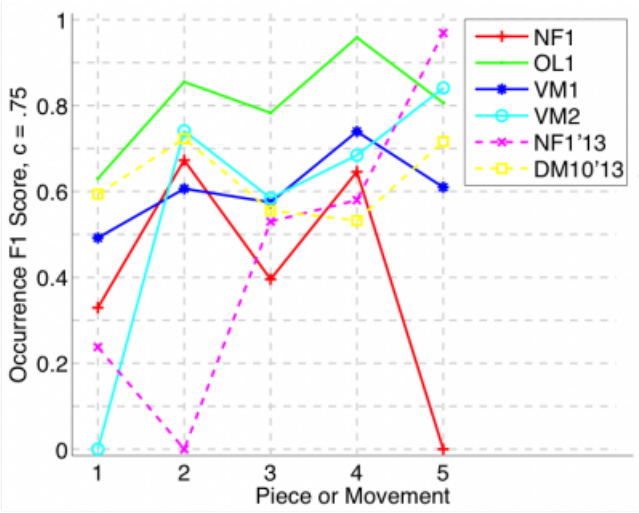

Figure 7.13: Occurrence F1 of the participants of the Mirex monophonic task

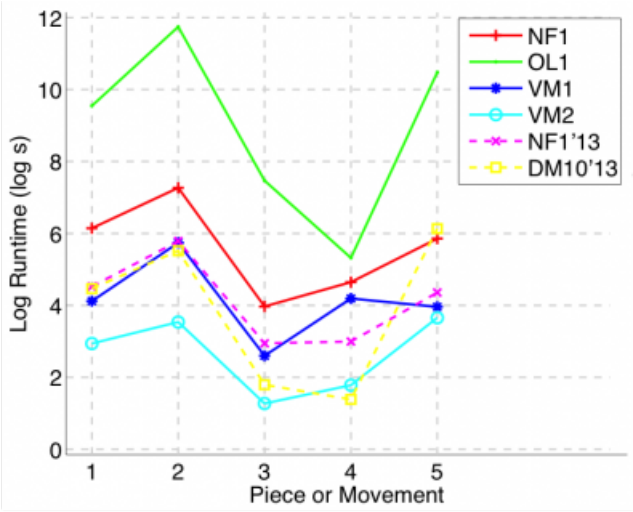

Figure 7.14: Runtimes of the participants of the Mirex monophonic task

with different `Notes`. The monophonic task contains systems that are faster and have a lower computational complexity than the polyphonic versions like VM2. Even VM2 is 10 to 100 times slower than my system. There is especially a huge difference in runtime with the original PatMinr algorithm (The code of PatMinr is OL1). The time it takes for PatMinr to find patterns in the second piece is more than a day. It takes my system around a second to find all patterns in all pieces.

7.4 Conclusion of this chapter

The algorithm has still a lot of room for improvement, which can be seen in chapter 6. Because the current algorithm can not find sections, this lowers a lot of the metrics in the polyphonic task. Only parts of the sections can be found (monophonic patterns that are part of the section). If occurrences of sub-patterns can be found, then this would help increase the occurrence recall, which is one of the areas where the algorithm can do better.

The current system is very good in finding the important patterns. The establishment precision is one of the points the system scores better. One of the biggest problems is that only a part of the occurrences of the pattern can be found. It would also be helpful if there could be done research on the best way of filtering because the establishment recall is too low because a lot of unimportant patters are returned together with the important patterns.
Despite that there are areas that can be improved, the algorithm got results that are often comparable with the state-of-the-art systems. If the extensions can be implemented in the future, then it is quite possible that the algorithm can have good results for all metrics.
This evaluation shows that the quality of the current system is comparable with the quality of results of the other systems. The current system can find these results in a fraction of the time that the other systems need. Especially when the runtime of my system is compared with the algorithm it is inspired by, PatMinr, a huge difference in runtime is seen.
In the future, there needs to be tried to increase the occurrence and establishment measures without increasing the runtime too much.

Chapter 8

Conclusion

In my thesis I researched if it was possible to extend the functionality of the PatMinr algorithm so it can also find patterns in polyphonic music pieces. To do this, I developed a new program which was inspired by the algorithm, but was better designed and was a lot more efficient. I paid a lot of attention to the design because it made it possible to extend the functionality of the program more easily and made the program more general. I decided to use object oriented programming instead of working with matrices to make the program more understandable and to make it possible to extract functionality, which then can be changed. A good example was the introduction of sequences in the program. In the original PatMinr algorithm each note will immediately try to extend the pattern tree. This has a lot of disadvantages. Because a note is added immediately, it made it impossible to handle the polyphonic task because there are notes in between notes that need to form a pattern. By introducing the concept of sequences, this is possible. Sequences do not need to be extended by the each note. A note will only try to extend a sequence for which it has a chance. For example, it will not try to extend sequences that happen on a different staff or sequences where the last note has not yet ended. It is also more easy to extend a sequence than the pattern tree of PatMinr because only the last note needs to be checked. PatMinr needs to go trough the entire pattern tree for each note it adds which is much more intensive.
By defining the hierarchy in the different kinds of patterns, analysers and sequences, it is possible to search for a specific kind of patterns. PatMinr adds all information of a note to the pattern tree, also the information that is not needed. This was also in my opinion one of the reasons why the algorithm is slow.
Newly introduced concepts like `Time` and `Sequence` made it possible to handle the polyphonic task. The system is multifunctional; it can find monophonic patterns in the polyphonic environment, but it is also capable of finding patterns in the monophonic environment. This is very useful because the algorithm is based upon the monophonic algorithm. Improvements in the monophonic algorithm will also cause improvements in the polyphonic algorithm. It is therefore interesting that the system can still participate in the monophonic task, to check how the monophonic algorithm can be improved.
The results show that the quality of the output of the monophonic algorithm can be compared to the quality of the output of the current state-of-the-art systems. The quality of the output of the polyphonic algorithm has comparable results to the state-of-the-art systems for pieces with a low amount of polyphonic systems. Because currently only monophonic patterns can be found in polyphonic musical pieces, the results of the mea-

sures of my system about finding patterns in musical pieces with a lot of polyphonic patterns is bad. The current system is very good in finding the important monophonic patterns. The occurrences that belong to these important patterns are also of a good quality. In the evaluation of the quality of the output, the system was in these areas better than the participants of the polyphonic task. The areas that can be improved are finding polyphonic patterns and sections instead of only monophonic parts, the amount of occurrences that can be found, and the filtering. These areas are discussed in this thesis and showed the good design. I have build a system that can handle both the monophonic and polyphonic task. During the process it can create a good structure which helps with understanding the repetitions in the music. The quality of the monophonic patterns that are found using my system is comparable with other state-of-the-art systems. It is especially nice that this is also the case for the polyphonic task. The best improvement that I have managed is the improvement in the run time. The run time of the system is a lot lower than the run time of the other systems. The system is able to get comparable results in only a fraction of the time. The system is also very extendable and there are some areas where the system can easily be improved. I think that I therefore can conclude that I have created a system which already shows some results, but has still some room for improvement. It has potential to become better. One of the best properties of my system is that it can find the results at a very high speed. The biggest challenge for the future for this system is to improve the measurements while keeping the speed.

Bibliography

[1] Harold Barlow and Sam Morgenstern. *A dictionary of musical themes*. Crown Pub, 1975.

[2] Emilios Cambouropoulos. Musical parallelism and melodic segmentation. 2006.

[3] Tom Collins. *Improved methods for pattern discovery in music, with applications in automated stylistic composition*. PhD thesis, The Open University, 2011.

[4] Tom Collins. Discovery of repeated themes & sections. `http://www.music-ir.org/mirex/wiki/2014:Discovery_of_Repeated_Themes_%26_Sections`, 2014. Online; accessed 7-Jan-2014.

[5] Tom Collins. Discovery of repeated themes & sections results. `http://www.music-ir.org/mirex/wiki/2014:Discovery_of_Repeated_Themes_%26_Sections_Results`, 2014. Online; accessed 22-Oct-2014.

[6] Tom Collins, Andreas Arzt, Sebastian Flossmann, and Gerhard Widmer. Siarct-cfp: Improving precision and the discovery of inexact musical patterns in point-set representations. In *ISMIR*, pages 549–554, 2013.

[7] Tom Collins, Robin Laney, Alistair Willis, and Paul H Garthwaite. Modeling pattern importance in chopin's mazurkas. 2011.

[8] Nicholas Cook. *A guide to musical analysis*. G. Braziller New York, 1987.

[9] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, Clifford Stein, et al. *Introduction to algorithms*, volume 2. MIT press Cambridge, 2001.

[10] W Jay Dowling and Dane L Harwood. *Music cognition*. Academic Press, 1986.

[11] Jia-Lien Hsu, Chih-Chin Liu, and Arbee LP Chen. Discovering nontrivial repeating patterns in music data. *Multimedia, IEEE Transactions on*, 3(3):311–325, 2001.

[12] Ray S Jackendoff and Fred Lerdahl. *A generative theory of tonal music*. Cambridge, Mass.: MIT Press, 1983.

[13] Konstantin Knizhnik. Patricia tries: A better index for prefix searches. *Dr. Dobb's Journal*, 2008.

[14] Olivier Lartillot. Efficient extraction of closed motivic patterns in multi-dimensional symbolic representations of music. In *Web Intelligence, 2005. Proceedings. The 2005 IEEE/WIC/ACM International Conference on*, pages 229–235. IEEE, 2005.

[15] Olivier Lartillot. In-depth motivic analysis based on multiparametric closed pattern and cyclic sequence mining. In *ISMIR*, pages 361–366, 2014.

[16] Olivier Lartillot. Miningsuite:a comprehensive matlab framework for signal, audio and music analysis, articulating audio and symbolic approaches. `https://code.google.com/p/miningsuite/`, 2014. Online; accessed 13-Feb-2014.

[17] Olivier Lartillot. Patminr: In-depth motivic analysis of symbolic monophonic sequences. *Music Information Retrieval Evaluation eXchange*, 2014.

[18] Olivier Lartillot, Petri Toiviainen, and Tuomas Eerola. A matlab toolbox for music information retrieval. In *Data analysis, machine learning and applications*, pages 261–268. Springer, 2008.

[19] David Meredith. Point-set algorithms for pattern discovery and pattern matching in music. *Content-Based Retrieval*, (06171), 2006.

[20] David Meredith. Cosiatec and siateccompress: Pattern discovery by geometric compression. In *International Society for Music Information Retrieval Conference*, 2013.

[21] David Meredith, Kjell Lemström, and Geraint A Wiggins. Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music. *Journal of New Music Research*, 31(4):321–345, 2002.

[22] Oriol Nieto and Morwaread Farbood. Discovering musical patterns using audio structural segmentation techniques. *Music Information Retrieval Evaluation eXchange, Curitiba, Brazil*, 2013.

[23] Oriol Nieto and Morwaread M Farbood. Mirex 2014 entry: Music segmentation techniques and greedy path finder algorithm to discover musical patterns.

[24] Oriol Nieto and Morwaread M Farbood. Music segmentation techniques and greedy path finder algorithm to discover musical patterns.

[25] Heinrich Schenker. Harmony, ed. oswald jonas, trans. elisabeth mann borgese, 1954.

[26] Gerald Strang and Arnold Schönberg. *Fundamentals of musical composition*. Faber & Faber, 1967.

[27] Gissel Velarde and David Meredith. A wavelet-based approach to the discovery of themes and sections in monophonic melodies. *Music Information Retrieval Evaluation eXchange*, 2014.

[28] Colin Watts. Siglund bruhn, js bach's well-tempered clavier-in depth analysis and interpretation volume i, hong kong: Mainer international, 1993; 264 pp., isbn 962-580-017-4. siglund bruhn, js bach's well-tempered clavier-in depth analysis and interpretation volume ii, hong kong: Mainer international, 1993; 250 pp., isbn 962-580-018-2. *International Journal of Music Education*, (1):87–87, 1994.

AFDELING INFORMATICA
Celestijnenlaan 200A
3000 LEUVEN, BELGIË
tel. +32 (0)16 32 77 00 www.kuleuven.be