

Vinícius Augusto Pereira Queiroz

# **Inter-Opus Musical Motif Discovery in Symbolic Music**

Curitiba - PR, Brazil

2023

Vinícius Augusto Pereira Queiroz

# **Inter-Opus Musical Motif Discovery in Symbolic Music**

Master's Thesis submitted to the Graduate Program in Computer Science at Pontifícia Universidade Católica do Paraná in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Pontifícia Universidade Católica do Paraná - PUCPR

Graduate Program in Computer Science

Supervisor: Carlos Nascimento Silla Junior

Curitiba - PR, Brazil

2023

Dados da Catalogação na Publicação  
Pontifícia Universidade Católica do Paraná  
Sistema Integrado de Bibliotecas – SIBI/PUCPR  
Biblioteca Central  
Sônia Maria Magalhães da Silva – CRB 9/1191

Q3i 2023	Queiroz, Vinícius Augusto Pereira Inter-opus musical motif discovery in symbolic music / Vinícius Augusto Pereira Queiroz ; supervisor: Carlos Nascimento Silla Junior. – 2023 89 f. ; il. : 30 cm  Dissertação (mestrado) – Pontifícia Universidade Católica do Paraná, Curitiba, 2023 Bibliografia: f. 84-89  1. Música. 2. Motif (Arquivo de computador). 3. Emoções. 4. Mineração de dados (Computação). 5. Informática. I. Silla Junior, Carlos Nascimento. II. Nascimento. III. Pontifícia Universidade Católica do Paraná. Programa de Pós- Graduação em Informática. III. Título.
	CDD. 20. ed. – 004



Pontifícia Universidade Católica do Paraná  
Escola Politécnica  
Programa de Pós-Graduação em Informática

Curitiba, 13 de julho de 2023.

58-2023

## **DECLARAÇÃO**

Declaro para os devidos fins, que **VINICIUS AUGUSTO PEREIRA QUEIROZ** defendeu a dissertação intitulada “**INTER-OPUS MUSICAL MOTIF DISCOVERY IN SYMBOLIC MUSIC**”, no dia 31 do mês de março do ano de 2023, a qual foi aprovada.

Por ser verdade firmo a presente declaração.

---

Prof. Dr. Emerson Cabrera Paraiso  
Coordenador do Programa de Pós-Graduação em Informática

# Acknowledgments

Agradeço à minha família e aos meus amigos por todo o suporte durante essa fase da minha vida. Sem vocês este trabalho não seria concluído! À minha mãe Maria Helena, pelas palavras de carinho e conforto. À minha irmã Carla, por compartilhar minha empolgação, escutar a todos os meus descontentamentos, passar segurança e me incentivar a seguir em frente. À minha companheira Carol, por todo o amor, parceria, e pelo apoio durante dias e noites, enquanto eu focava neste trabalho. À minha filha Aurora, por me dar motivação. Ao meu irmão Felipe, por todos os ensinamentos - os quais sigo até hoje. Ao meu pai Carlos, por me passar resiliência. Ao meu psicólogo Thuan, pelo excelente acolhimento profissional. Aos meus amigos que me ouviram e me deram conselhos em relação ao mestrado - em especial, Rodrigo, Chelles, Manu, Guy, Kviar, Stan, Lenin, Gian e Henrique.

Agradeço também a todos aqueles que influenciaram na qualidade desse trabalho. Ao meu orientador e amigo Carlos Silla, por compartilhar sua vasta experiência, pelas oportunidades e por todas as conversas no meio da madrugada. Aos demais professores do PPGIa pelo excelente acompanhamento e didática. Aos colegas, por compartilharem as alegrias e as dores dessa jornada, mesmo que brevemente - em especial, André, Fábio e Eric. Ao Gian, pelos conselhos técnicos e acadêmicos inestimáveis. À equipe do Champ-Analytics, pela confiança. E à Pontifícia Universidade Católica do Paraná, pela excelente infraestrutura.

Por fim, agradeço à CAPES e aos contribuintes, que permitiram que esse trabalho fosse realizado com dedicação integral durante 2 anos.

*“You don’t have **to be** the best in  
whatever you choose to work with;  
but you should choose to work with something that  
will make you willing **to give** your best.”*

**Felipe Queiroz**

# Abstract

Musical motifs are short musical patterns found in one piece (intra-opus) or across multiple pieces (inter-opus). Music can not only convey emotions, but also provoke emotional responses in the listeners; and certain musical elements (such as the combination of mode and tempo) convey different emotions. However, there is an open research question in how music and emotions relate to one another: “Which elements from musical motifs are responsible to conveying which emotions?”. To answer this question, we might use computational methods to discover patterns in a music corpus labeled with the emotions that each musical piece was meant to convey. Our objective is to design and implement a system capable of finding motifs across multiple musical pieces, that outputs patterns that are understandable by humans, in a manner that is easy to visualize. This system could then be used in a dataset of musical pieces labeled with emotion tags. In this work we present reasons to use Sequential Pattern Mining algorithms for the task. We also find that the Maximally General Distinctive Pattern (MGDP) algorithm is currently one of the state-of-the-art algorithms for inter-opus musical motif discovery, although a lack of standardization in evaluation methods hinders the definition of state-of-the-art for the task. We extended the MGDP algorithm with novel methods to enable the discovery of musical patterns that have a variable number of events in-between the events of the pattern. We also designed and developed a method to automatically generate the visualizations of the patterns embedded into musical scores.

**Keywords:** Music, Motif, Emotions, Pattern, Discovery, Identification, Mining.

# List of Figures

Figure 1 – Relationship between the different kinds of musical motifs . . . . .	21
Figure 2 – Melody excerpts from three soundtracks of the Super Mario World game, which share the same melodic motif. The melodic contour is depicted in red. The diatonic intervals are presented below each note . . . . .	22
Figure 3 – Example of an intra-opus rhythmic motif, from Beethoven’s Symphony No. 5 . . . . .	23
Figure 4 – Summarized Entity-Relationship Diagram depicting basic Opera concepts	25
Figure 5 – Overview of the structure of the Demofonte opera, by Pietro Metastasio. Shapes represent the characters that are singing the corresponding Aria. Lines represent the recitatives. The number to the right represents the scene number in the act. Numbers to the left represent the position of the opera in the dramatic text, by number of poetic lines . . . . .	26
Figure 6 – MIDI Event List example . . . . .	27
Figure 7 – MIDI Piano Roll visualization example . . . . .	28
Figure 8 – Different types of MIDI controllers . . . . .	29
Figure 9 – MusicXML “Hello World” file excerpt, with visual indicators of each block’s purpose . . . . .	30
Figure 10 – Visualization of the MusicXML “Hello World” in MuseScore . . . . .	31
Figure 11 – Example of generation of a Lexicographical Sequence Tree . . . . .	35
Figure 12 – Viewpoint representation of a music piece excerpt. $\perp$ represents an undefined value. . . . .	45
Figure 13 – Depiction of the definition of distinctive patterns. . . . .	46
Figure 14 – Depiction of MGDPs in a Sequence Tree . . . . .	47
Figure 15 – Overall MGDP algorithm as a simplified block diagram . . . . .	47
Figure 16 – First step of the MGDP algorithm . . . . .	48
Figure 17 – Second step of the MGDP algorithm . . . . .	48
Figure 18 – Third step of the MGDP algorithm . . . . .	49
Figure 19 – Fourth step of the MGDP algorithm . . . . .	49
Figure 20 – Example of a Subsumption Graph of featuresets . . . . .	50
Figure 21 – Subsumption Graph from Figure 20 after pruning the infrequent nodes	50
Figure 22 – Examples of possible results from converting the Subsumption Graph of Figure 21 to a Subsumption Tree . . . . .	51
Figure 23 – Fifth step of the MGDP algorithm . . . . .	52



Figure 24 – Pattern $\hat{7}-\hat{1}-\hat{2}-\hat{6}-\hat{8}$ , found by the <code>MGDP_Bitmap()</code> algorithm, manually annotated in the two scores. Green notes indicate the first event of the pattern, red notes indicate the last event of the pattern, and blue notes indicate the rest of the occurrences of the events of the pattern. . . . .	57
Figure 25 – Pattern found by the <code>MGDP_Bitmap_limitMaxJumpSize(maxJumpSize)</code> algorithm with <code>maxJumpSize = 1</code> , manually annotated in the two scores’ excerpts. Depicts the pattern $\hat{2}-\hat{3}-\hat{5}-\hat{6}-\hat{8}$ , Green notes indicate the first event of the pattern, red notes indicate the last event of the pattern, and blue notes indicate the rest of the occurrences of the events of the pattern. . . . .	58
Figure 26 – Automatically annotated score output from the <code>MGDP_InstanceMap()</code> algorithm, visualized in MuseScore 4. Green notes indicate the first event of the pattern, red notes indicate the last event of the pattern, and blue notes indicate the rest of the occurrences of the events of the pattern. . . . .	63
Figure 27 – Output from the <code>MGDP_InstanceMap_limitMaxJumpSize(maxJumpSize)</code> algorithm, with <code>maxJumpSize=1</code> , visualized in MuseScore 3. Depicts the automatically annotated score. Each box corresponds to one event. Boxes are stacked up vertically from top down, corresponding to the order in which their associated events appear. Green notes indicate the first event of the pattern, red notes indicate the last event of the pattern, and blue notes indicate the rest of the occurrences of the events of the pattern. . . . .	64
Figure 28 – Introduction riff from Beethoven’s 5th Symphony written with two different time signatures, but sounding exactly the same . . . . .	70
Figure 29 – Complete block diagram representing the implemented MGDP algorithm. Blocks are steps of the algorithm, and arrows indicate the input hyper-parameters of the algorithm and the outputs from each step . . . . .	71
Figure 30 – One of the patterns sent to be analyzed by the musicologist . . . . .	74
Figure 31 – Pattern outputted by the <code>MGDP_InstanceMap_limitMaxJumps()</code> algorithm, with <code>maxJumpSize = 1</code> , with only one viewpoint . . . . .	76
Figure 32 – Pattern outputted by the <code>MGDP_InstanceMap_limitMaxJumps()</code> algorithm, with <code>maxJumpSize = 1</code> , but with 3 viewpoints . . . . .	77
Figure 33 – Box plot distributions of the execution time of Bitmaps and InstanceMaps algorithms’ runs, when setting <code>maxJumpSize</code> (lower is better) . . . . .	78
Figure 34 – Box plot distributions of memory consumption of Bitmaps and InstanceMaps algorithms’ runs, when setting <code>maxJumpSize</code> (lower is better) . . . . .	79

Figure 35 – Box plot distributions of memory consumption of Bitmaps and InstanceMaps algorithms' runs, when NOT setting <i>maxJumpSize</i> (lower is better) . . . . .	80
Figure 36 – Box plot distributions of execution time of Bitmaps and InstanceMaps algorithms' runs, when NOT setting <i>maxJumpSize</i> (lower is better) .	80

# List of Tables

Table 1 – Popular songs that contain the chord progression I-V-vi-IV, exemplifying an inter-opus harmonic motif . . . . .	24
Table 2 – Sequence Database example . . . . .	33
Table 3 – Sequential patterns found in the SDB of Table 2, with $minSup = 3$ . . . . .	34
Table 4 – Another Sequence Database example . . . . .	36
Table 5 – Verticalized SDB example . . . . .	37
Table 6 – Result of i-extension operation through vertical bitmaps . . . . .	37
Table 7 – Result of s-extension operation through vertical bitmaps . . . . .	38
Table 8 – Examples of some possible viewpoints . . . . .	45
Table 9 – MGDGP discovery algorithm combination of strategies . . . . .	55
Table 10 – MTC dataset’s genres, genres abbreviations, number of songs in each genre, and average( $\pm$ standard deviation) number of events per song, grouped by genre . . . . .	67
Table 11 – Pairwise memory consumption comparison (in MB) between algorithms using different data structures, when $maxJumpSize$ is not set. Values in the same row presents runs with the same hyper-parameter configuration (apart from the Data Structure). Lower is better . . . . .	81

# List of Algorithms

1	The i-extension algorithm for InstanceMaps . . . . .	60
2	The s-extension algorithm for InstanceMaps . . . . .	62
3	The s-extension algorithm for InstanceMaps, limiting the maxJumpSize . .	65

# Contents

<b>1</b>	<b>INTRODUCTION</b>	<b>14</b>
1.1	Motivation	14
1.2	Objectives	16
1.3	Contributions	17
1.4	Structure of the Document	17
<b>2</b>	<b>THEORETICAL FRAMEWORK</b>	<b>19</b>
2.1	Computational Musicology	19
2.2	Musical Motifs	21
2.3	Opera and Arias	24
2.4	Symbolic Music Representation	25
2.4.1	MIDI	27
2.4.2	MusicXML	28
2.4.3	MEI	31
2.5	Sequential Pattern Mining	32
2.5.1	SPAM	34
2.5.1.1	Vertical Bitmaps	36
<b>3</b>	<b>TOOLS AND METHODS</b>	<b>39</b>
3.1	Computational Musicology Tools	39
3.1.1	music21	39
3.2	Inter-Opus Musical Motif Discovery	39
3.2.1	Sequence Clustering VS Sequential Pattern Mining	39
3.2.2	Related Work	41
3.2.3	Viewpoint Data Representation	44
3.2.4	MGDP sequences	46
3.2.5	MGDP Algorithm	47
3.2.5.1	Read MusicXML and Apply Viewpoints	48
3.2.5.2	Split SDB	48
3.2.5.3	Verticalize SDB	48
3.2.5.4	Create Subsumption Tree	49
3.2.5.5	MGDP Discovery	51
<b>4</b>	<b>PROPOSED ALGORITHMS</b>	<b>54</b>
4.1	MGDP Algorithms Implementations	54
4.1.1	MGDP_Bitmap()	55

4.1.2	MGDP_Bitmap_limitMaxJumpSize(maxJumpSize)	56
4.1.2.1	Bitmap s-extension with maxJumpSize - the Bitmatrix	58
4.1.3	MGDP_InstanceMap()	59
4.1.3.1	The InstanceMap	59
4.1.3.2	InstanceMap i-extension	60
4.1.3.3	InstanceMap s-extension	61
4.1.3.4	MGDP_InstanceMap() output	62
4.1.4	MGDP_InstanceMap_limitMaxJumpSize(maxJumpSize)	63
<b>5</b>	<b>EXPERIMENTAL SETUP</b>	<b>67</b>
<b>5.1</b>	<b>Dataset</b>	<b>67</b>
<b>5.2</b>	<b>Implemented Viewpoints</b>	<b>68</b>
<b>5.3</b>	<b>Experiments</b>	<b>70</b>
<b>5.4</b>	<b>Hardware and Software</b>	<b>71</b>
<b>6</b>	<b>RESULTS AND DISCUSSION</b>	<b>73</b>
<b>6.1</b>	<b>RQ1</b>	<b>73</b>
<b>6.2</b>	<b>RQ2</b>	<b>75</b>
6.2.1	Setting maxJumpSize	76
6.2.2	Without setting maxJumpSize	78
<b>7</b>	<b>CONCLUSION</b>	<b>82</b>
	<b>REFERENCES</b>	<b>84</b>

# 1 Introduction

## 1.1 Motivation

Music has been historically present in various social contexts, such as in matrimonial rituals, poetic declamations, funerals, banquets, and parties. It is used alongside medical treatments, possibly improving the therapeutic efficiency, such as in treating Alzheimer’s disease and other mental disorders (MOREIRA; JUSTI; MOREIRA, 2018). Given certain conditions, music might be used to indirectly improve work productivity (HUANG; SHIH, 2011; LESIUK, 2005; SHIH; HUANG; CHIANG, 2012). Music might be used to foster 21<sup>st</sup> century’s high-demand soft-skills, such as creativity (ENGELMAN et al., 2017). There is a possibility that learning music in childhood might bring long-term positive impacts in IQ scores and academic performance (SCHELLENBERG, 2006; SCHELLENBERG; WEISS, 2013). But even with all these noble applications, it is arguable that the ultimate purpose of music is to make life more enjoyable.

It has been shown by the Music Psychology field that music is able to not only convey a musical emotion from the composer (simulating a language) (HUNTER; SCHELLENBERG; SCHIMMACK, 2010), but also to evoke emotional responses in the listeners (JUSLIN; VÄSTFJÄLL, 2008; LUNDQVIST et al., 2009). In fact, given certain conditions, music is able to evoke even peak-emotional responses, such as tearing and chills/goosebumps (MORI; IWANAGA, 2017). Music can also serve as a significant emotional enhancer in narrative contexts (i.e., movies, video games, theater, opera, storytelling, etc.) (BAUMGARTNER et al., 2006; LI; CHENG; TSAI, 2019; FU, 2015; BEZDEK; GERRIG, 2008).

Some researchers try to find correlations between musical features and emotions. Schellenberg, Krysciak and Campbell (2000) used music pieces that were labeled as sad, happy, or scary to indicate that variations in pitch are more relevant to express emotions than rhythm. Hunter, Schellenberg and Schimmack (2010) were able to show how the combination of mode (Major/Ionian or minor/Aeolian) with tempo (slow or fast) conveys and evokes different emotions (happiness or sadness). Ramos, Bueno and Bigand (2011) used the seven Greek modes in their research (i.e., Ionian, Dorian, Phrygian, etc.), and one of their findings was that the three major Greek modes might be related to both happiness and serenity, depending on the tempo. Arjmand et al. (2017) found that changes in musical motifs and instrumentation are the two primary factors associated to peak-emotional responses, followed by high pitches, and high loudness.

Most of the researchers aiming to link music and emotions that we found try to

correlate music to the emotions perceived and felt from the listener's perspective. However, instead of looking from the listener's perspective, we can also investigate how the music writers use the musical features as tools to express, convey, and evoke emotions.

The personal report from soundtrack composer [Douek \(2013\)](#) mentions that he believes most soundtrack composers do not know exactly how or why their compositions are congruent with the emotions they wanted to convey - he claims they just follow their intuitions. However, he gives some tips and tricks that composers can use for the purpose of conveying specific emotions and feelings. [Douek \(2013\)](#) mentions, for example, that changes in dynamics and tempo might convey the feeling of urgency and importance; harmony/cadence and rhythm can be used to convey tension and release; a lack of resolution might cause surprise or suspense; and musical motifs (i.e., any kind of repeating patterns in music, be it melodic, harmonic, or rhythmic) might have their meaning changed according to changes in mode or tempo.

As mentioned by [Juslin and Västfjäll \(2008\)](#), one of the psychological mechanisms that induce emotions through music is the Emotional Conditioning (a.k.a. Evaluational Conditioning) - i.e., the association of a music segment with a particular emotion through repetition of associated contexts. In other words, the listener might associate a certain musical element with a specific emotion because this musical element was heard paired with that emotion multiple times. As an example, if a listener has watched 100 movies in which some images that caused fear were associated with a back-and-forth minor second interval in increasing tempo (much like the movie *Jaws*' theme), the listener would associate the back-and-forth minor second interval in increasing tempo with fear, even without seeing the fearful images. Moreover, this Emotional Conditioning (EC) might be unconscious for the listener.

Since music writers are, first and foremost, listeners, we hypothesize that when they are creating music, they might be using their own EC patterns to analyze their work's emotional aspects, unconsciously perpetuating the EC from previous music writers that serve as their references, causing a kind of chain reaction. Therefore, finding out how exactly music writers map specific musical aspects into specific emotions could help composers, songwriters, arrangers, and music producers to take informed and conscious decisions while creating music, instead of blindly following anecdotal traditions, or relying solely on inspiration.

One of the open research questions in the field of Music and Emotions is: "Which elements from musical motifs are responsible to conveying which emotions?". As an example, one might wonder whether a melodic motif containing three ascending perfect fifths in three successive strong beats is often used by composers to convey a feeling of boldness. To answer this question, one might try to manually retrieve several pieces of music, label them according to the emotion following some kind of guideline, and then analyze each



one of them, measure by measure, trying to find patterns that are repeated across multiple songs, marking down and counting when they appear, while still considering the musical variations they might have. This would be a very burdensome and repetitive task, would take a very long time, would incur in a very high cognitive load for the human analyzers, and would be prone to many errors due to human factors - especially when dealing with a large corpus.

Fortunately, with the last century's advance in technology, we might be able to tackle the aforementioned research question through a computational approach. To do so, we need a dataset consisting of digitized pieces of music and the corresponding labels of the emotions meant to be conveyed in each piece of music. It could be, for example, a dataset of opera Arias (see [Section 2.3](#)) labeled with the emotions that each Aria should convey, according to the *libretto* - just like the one being done by [Torrente and Llorens \(2022\)](#). Besides the data, we also need a computational method to find the most relevant musical motifs for each emotion being analyzed. This computational method is what brings special interest to this work.

One computational method that is used to find sequential patterns in symbolic music is known in the literature as the Maximally General Distinctive Patterns (MGDP) algorithm proposed by ([CONKLIN, 2010a](#)). The MGDP algorithm is based upon the Sequential PAttern Mining (SPAM) algorithm ([AYRES et al., 2002](#)). One limitation of the original MGDP algorithm is in the way that it explores the patterns in the music score. In order to overcome this limitation in the MGDP algorithm it is necessary to change the way it internally uses the SPAM algorithm.

## 1.2 Objectives

Our main goal is to develop a novel approach for the discovery of musical motifs across pieces from a symbolic music corpus. In other words, we aim to develop a system that, given a dataset of MusicXML files of music pieces that are somehow related (e.g., different pieces of the same genre), it can discover, search, and return a set of sequences considered to be relevant musical patterns, repeated across two or more of these musical pieces, possibly presenting some degree of musical variations between them.

Our specific objectives are to:

- Summarize essential theoretical concepts required to understand the Automatic Musical Motif Discovery task;
- Search for existing methods of musical motif discovery available in the literature;
- Develop an end-to-end system capable of discovering musical motifs with musical variations, shared by multiple pieces;

- Test the musical motif discovery system using a real music corpus.

Overall, we want to continue the work that has been done in the Computational Musicology field. Ultimately, our objective is to aid humanity in reaching a better understanding of music using computational methods. This would enable music creators to consciously explore musical features to their fullest potential, turning what they envision into reality.

## 1.3 Contributions

This work has led to three main contributions:

- A Scientific Contribution. We have implemented four variations of the MGDGP algorithm - all aiming to enable the discovery of patterns with a dynamic number of musical events in-between the patterns. Two of these variations use the same underlying data structure as the original MGDGP algorithm - the Bitmap data structure. In the other two variations we use a different data structure - the InstanceMap. For each variation of the MGDGP algorithm we implemented a different s-extension algorithm. We performed a statistical comparison between these four modified MGDGP algorithms in terms of memory consumption and time complexity.
- A Technical Contribution, in the sense that we implemented variations of the MGDGP algorithm, considering that there were no public implementations of this algorithm beforehand. We plan to release the source-code of our implementations soon. Also, the code developed during this work was implemented with the utmost care for the best-practices in Software Engineering (namely, following Clean Code, Design Patterns, and Versioning), so that when it becomes publicly available, it is maintainable and extendable.
- A Social Contribution, in the sense that this work develops new ways to automatically generate visualizations of sequential musical patterns. This may aid musicologists to utilize the information from these visualizations to derive new knowledge regarding the correlation between sequential musical patterns and their topics of interest.

## 1.4 Structure of the Document

This master's thesis is structured as follows:

- The current [Chapter 1](#) informs the motivation for this project and its objective. We presented the relationship between music and emotions according to the literature,

explaining why we intend to do this work, and stating how this work can help bridge the gap of knowledge in music and emotions.

- [Chapter 2](#) presents a theoretical review, covering foundational concepts that are required to fully comprehend the remainder of this document. We explain what the purpose of the Computational Musicology field is, and its difference from the Music Information Retrieval (MIR) field. We also introduce some standard symbolic music formats, an introduction to musical motifs, and the foundations of Sequential Pattern Mining algorithms. In this chapter we explain the SPAM algorithm, which is fundamental to understand our contributions.
- [Chapter 3](#) presents some of the tools and methods that were useful during the project development. We formally introduce the problem of Inter-Opus Musical Motif Discovery, and we present a literature review on the few works that are related specifically to the Inter-Opus Musical Motif Discovery task. In this chapter we bridge the gap between the computational methods and the musicological aspects of this work, by introducing the notion of Viewpoints. We describe in detail the original MGDGP algorithm, which we based our proposed methods upon. We also give more details about the limitations of the MGDGP algorithm.
- [Chapter 4](#) introduces our proposed algorithms. We mention the opportunities of improvement that the original MGDGP presents, and we explain the four variations of the MGDGP algorithm that we implemented to address these opportunities of improvement. In this chapter we also introduce the InstanceMap data structure that we used in two of our novel MGDGP variations.
- [Chapter 5](#) presents the methodological process of the work. We give details on the dataset that we used in this work, the research questions that we wanted to answer with our experiments, and descriptions on the hardware and software environments that we ran our experiments on. We define the viewpoints that we implemented for the MGDGP algorithm, and add some details about our specific implementations.
- [Chapter 6](#) presents the results of the experiments and the answers to the proposed research questions. We present a comparison of some of the patterns discovered with different settings of the algorithms. We also show the results from statistical tests comparing the computational performance of the implemented algorithms.
- [Chapter 7](#) presents the conclusion of this thesis. We summarize what has been discussed along the document, the limitations of this work, and some proposals of future works.

## 2 Theoretical Framework

### 2.1 Computational Musicology

Volk, Wiering and Kranenburg (2011) define Musicology as the general study of music in all of its facets, including five areas:

- **Music Theory and Analysis** - studies patterns in music (such as note pitches, rhythm, tonality, harmony, form and structure, instrumentation), and their relationships, seeks to define novel patterns that describe processes and general principles in music composition, and uses these patterns to link with more abstract concepts related to music, such as semiotics and aesthetics;
- **Historical Musicology** - studies the relation of music with the historical context it is inserted in;
- **Ethnomusicology** - studies the relation of music with the cultural and social aspects it is inserted in;
- **Cognitive Musicology** - studies the relation of music and its effects in the individual human psyche;
- **Musical Performance Research** - studies the music performing activity, specially the cases in which there are no exact definitions in the score.

Digital Humanities is the name of the research field that incorporates any humanities subject that can be studied through the use of computational methods (BERRY, 2011), such as history, philosophy, linguistics, sociology, law, music, theater, fine arts, and others. The Digital Humanities discipline aims to extend these subjects' traditional research methods (which often rely solely on dialectics, or the results might be too subjective) into more experimental and empirical studies. This is possible since the diffusion of computers and computational methods enable the research to store and process a very large amount of data in a consistent and thorough manner, often extracting complex information that would take a lot of time if done manually. Although the computational methods do not substitute completely the traditional dialectic methods (and, arguably, should not), they serve as a complement to the research, either corroborating the results' analysis, or contradicting them - boosting the research's reliability. Digital Humanities enables novel research to be done with increasingly levels of complexity and scope, and enables traditional conjectures and assumptions to be confirmed (or discarded) in a practical manner.

Computational Musicology is enclosed in the broader field of the Digital Humanities (FRIELER, 2020), and may be defined as the scientific research area that aims to study music through the use of computational approaches (VOLK; WIERING; KRANENBURG, 2011). One may apply computational methods to any of the five areas of Musicology (VOLK; WIERING; KRANENBURG, 2011). Many different computational methods might be applied for different tasks within the Computational Musicology field. Some of these tasks include Melodic Similarity, Melody Reduction, Melody Evolution, Form and Structure Segmentation, Categorization (of genres, composers, rhythms, regions, and others), Composition, Harmonization, among others (MOR; GARHWAL; KUMAR, 2020).

Music Information Retrieval (MIR), as the name suggests, might be defined as the process of automatically extracting information directly from a piece of music. Frieler (2020) considers that MIR is mainly applicable to audio signals, and that Computational Musicology is only applicable to symbolic representations. However, Mor, Garhwal and Kumar (2020) consider that MIR is a task encompassed within the Computational Musicology field. The line between MIR and Computational Musicology seems to be ill-defined.

Traditionally, MIR research has the ultimate goal of creating a music query system that uses musical parameters (extracted directly from the piece of music) instead of relying on manually annotated metadata (DOWNIE, 2003), but it does not require a thorough understanding of the underlying patterns the query algorithm uses. As an example, one might use a Deep Learning approach to retrieve all Pop songs from a database and have 100% accuracy, being perfect for its MIR purpose - but since the Deep Learning approach is usually deemed as a black box system, it might not give any enlightenment to Musicology whatsoever. On the other hand, Computational Musicology (as in Musicology) has the goal of studying and understanding music and all of its different facets. With this in mind, Volk, Wiering and Kranenburg (2011) conclude that Computational Musicology and MIR are actually different fields that overlap, and should benefit from each other.

Frieler (2020) also defines the Digital Musicology research field, which comprises the process of digitizing music, including the editorial process of encoding and storing existing music from their available physical formats (e.g., Vinyl, Music Sheet) into digital representation formats (e.g., MP3, MIDI, MusicXML). One of the tasks included in the Digital Musicology field is the Optical Music Recognition (OMR) research line, which aims to automatically convert a music sheet image into a digitally encoded music representation (e.g., MusicXML) (BAINBRIDGE; BELL, 2001; CALVO-ZARAGOZA; HAJIČ; PACHA, 2019; HAJIC; PECINA, 2017; REBELO et al., 2012). Both MIR and Computational Musicology are greatly benefited by the OMR field, since it enables the digitization of very large corpora with all kinds of music; and is not limited to only western music notation.

## 2.2 Musical Motifs

In music, we might find certain patterns that repeat themselves along one song (i.e., intra-opus). These patterns can be melodic, harmonic, rhythmic, or a combination of these. Such patterns are usually referred to as a theme, or musical motifs<sup>1</sup>. Each recurrence of the musical motif might present some degree of variation, either in the intervallic structure, overall pitch, rhythm, harmony/accompaniment, instrumentation, or dynamics (DRABKIN, 2008). When a musical motif has the purpose of representing a person, object, place, idea, state of mind, supernatural force, or any other ingredient in a dramatic work, the musical motif is usually called a leitmotif (WHITTALL, 2001). Leitmotifs can be detected in any entertainment media that might involve music (e.g., theater, movies, TV shows, circus, video games, opera). Figure 1 depicts the relationship between musical motifs, leitmotifs, melodic motifs, harmonic motifs, and rhythmic motifs.

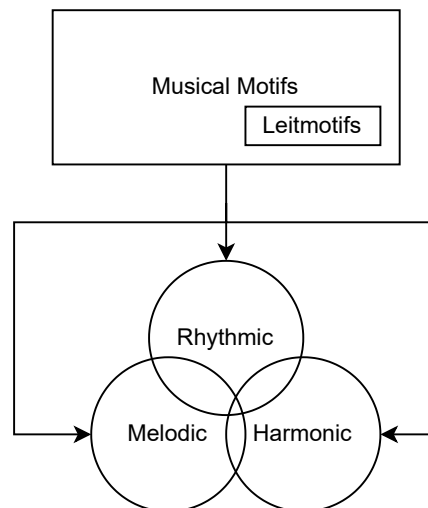


Figure 1 – Relationship between the different kinds of musical motifs

Musical motifs and leitmotifs are not restricted to the same piece of music. Figure 2 gives an example of a musical motif being applied to the melodies of three different soundtracks of a popular video game from the 1990s. It is visually observable that the same melodic contour (i.e., the shape that the melody forms with its variation in pitch) is present in the three melody excerpts, but with variations in mode, rhythm, tempo, time signature, and instrumentation. Moreover, if we analyze the diatonic intervals (i.e., the interval between the tonal center and the note being analyzed), and we discard the note qualities (i.e., minor, Major, Perfect, etc.), we can observe that all three tracks share the pattern  $\hat{3}-\hat{1}-\hat{5}-\hat{6}-\hat{1}$  in the first half, and then  $\hat{5}-\hat{1}-\hat{5}-\hat{3}$  in the second half (with slight variations in some cases, such as the repetition of P1 in the second half of track (b)). This

<sup>1</sup> Some musicologists consider that musical motifs consist in a very short pattern, and a theme would be a slightly longer (but still short) pattern. In this work, we do not differentiate motifs from themes, so we consider that sequential musical patterns with any length are considered motifs

depicts an example of an inter-opus musical motif, since the same musical motif happens in more than one musical piece (as opposed to being intra-opus, when the analysis is done in a single piece of music). Specifically, it is a melodic motif, since the similarities are, in this case, mostly in the melodic contour and in the interval structure of the melody.

## SMW Underground

Koji Kondo

(a)

## SMW Swimming

Koji Kondo

(b)

## SMW Fortress

Koji Kondo

(c)

Figure 2 – Melody excerpts from three soundtracks of the Super Mario World game, which share the same melodic motif. The melodic contour is depicted in red. The diatonic intervals are presented below each note

Looking further into the three tracks presented in [Figure 2](#), we can observe that the time signature for the track (b) is 3/4, while tracks (a) and (c) are in a 4/4 meter. The rhythm for each piece is different, although some rhythmic relationships are maintained, such as the duration of the last two notes in tracks (b) and (c), or the quarter length rest in the beginning of the third bar in tracks (a) and (c). Although all tracks use F as their tonal center, tracks (a)<sup>2</sup> and (c) use a Dorian Pentatonic Scale, while track (b) uses an Ionian/Major Pentatonic Scale - being one of the reasons that track (b) might be perceivable as more “positive”, “bright”, or “serene” than the other two. The tempo and instrumentation is different for each piece. Even with all these differences, all tracks share

<sup>2</sup> Track (a)’s scale is actually debatable. It could be also perceived as a Major Pentatonic, at least in the second half of the theme. Another interpretation is that it would be a Raga Mohanangi Scale, due to the bend in the last note that goes from m3 to M3

the same melodic motif (i.e., they all have a very similar melodic contour and/or a similar intervallic structure in the melody), so we can perceive the similarities in the melodies.

Figure 3 shows an example of a rhythmic motif. The rhythmic pattern shown in Figure 3a, consisting of three short notes followed by a longer one, is perceived during the whole piece of Beethoven's Symphony No. 5. Figure 3b shows some examples of the rhythmic motif being applied in the musical piece. We can see that some musical variations are present, such as in the contour, the contour direction, the overall pitch, the intervallic structure, and even in part of the rhythmic structure (e.g., instead of 3 eighth notes followed by a half note, we can see there are variations in which the half note is replaced by a whole note, or a half note linked with an eighth note).



(a) A rhythmic motif, consisting of three eighth notes, followed by a half-note



(b) The rhythmic motif denoted in (a), being applied in several passages from Beethoven's Symphony No. 5

Figure 3 – Example of an intra-opus rhythmic motif, from Beethoven's Symphony No. 5

Source: Adapted from (ANDERS, 2007)

As for the harmonic motifs, we could search for patterns in the chord progressions, vertical intervals, and/or harmonic cadences. It is important to note that chord progressions that are repeated along one musical piece, but only due to repeating sections (such as a pop song with a Verse A and then Verse B with the same chord progression) would not be considered as a harmonic motif - it is a consequence of a pattern in the structural



level. As an example, [Table 1](#) presents a list of a few popular songs that contain the chord progression I-V-vi-IV, being considered an inter-opus harmonic motif.

Table 1 – Popular songs that contain the chord progression I-V-vi-IV, exemplifying an inter-opus harmonic motif

<b>Artist</b>	<b>Title</b>
U2	With Or Without you
Journey	Don't Stop Believing
Jason Mraz	I'm Yours
Maroon 5	She Will Be Loved
The Beatles	Let It Be
Green Day	When I Come Around
Lana Del Rey	Love
Red Hot Chili Peppers	Under The Bridge
Ed Sheeran	Photograph
Avenged Sevenfold	Afterlife

## 2.3 Opera and Arias

In this section, we will give a brief overview of the structure of opera and arias. Our main reference for this section is the work by [Muñoz-Lago et al. \(2020\)](#). [Figure 4](#) presents an Entity-Relationship Diagram (ERD) following Chen's notation, which summarizes the basic concepts that will be introduced in this section.

Opera is a form of dramatic audiovisual performance. One opera's structure is defined by one *libretto*. The libretto contains the dramatic text of the narrative being told and is divided into acts. Each act contains a small set of scenographic settings that might be changed during the opera (e.g., a forest scenery, then a castle one). Each act is further divided into scenes, marked by the entrance or exit of characters from the stage, and/or changes of scenery.

An opera alternates between two main types of vocal performances: fully musical pieces (i.e., Arias, Duets, Choirs, etc.) with singers and, often, orchestral ensembles; and passages that are more closely related to dialogues (i.e., recitatives). Arias are sung by a soloist, whereas Duets have two singers with equal importance, and Choirs have many concomitant singers. Since Arias, Duets, Choirs, and other groupings (e.g., Trios) are the most “musical” sections of an opera, they are usually the musical highlights in an opera. In this work, we will use the term Aria interchangeably with Duets, Choirs and the other groupings, to represent the most “musical” sections of an opera. [Figure 5](#) depicts an overview of an opera, through one of the novel visualization approaches proposed by [Muñoz-Lago et al. \(2020\)](#).

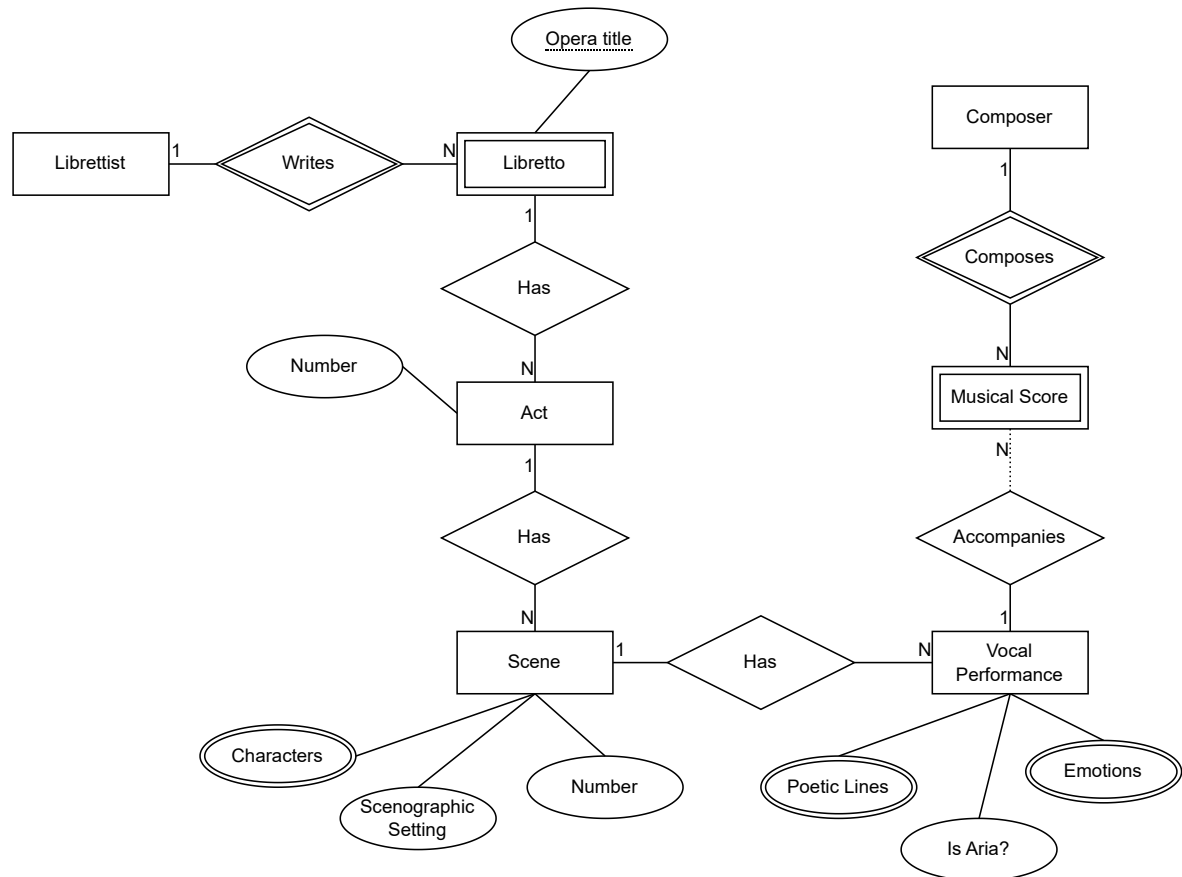


Figure 4 – Summarized Entity-Relationship Diagram depicting basic Opera concepts

The libretto itself does not contain a musical composition. There is also no explicit indication of what emotions should be conveyed in each Aria, although one can infer the emotion that is meant to be conveyed by interpreting the libretto’s narrative context (TORRENTE; LLORENS, 2022). Therefore, with the libretto in hands, composers analyze what emotion(s) should be conveyed by a particular Aria through the overall narrative context and the lyrics’ content, and then they write a fitting music score for that Aria. Notwithstanding, the same opera might have different music composers. This is what makes opera datasets specially interesting for analyzing musical motifs and their relationship with emotions: all composers must compose original scores for one Aria, but using the same lyrics, and trying to convey the same general emotion (TORRENTE; LLORENS, 2022).

## 2.4 Symbolic Music Representation

Since the advent of computers, a handful of different ways to store music have been invented and utilized for different purposes. These include, for example, the MIDI (Section 2.4.1), the MusicXML (Section 2.4.2), the MEI (Section 2.4.3), the WAV, and the ubiquitous MP3 file formats. MusicXML, MIDI, and MEI fall into the category of Symbolic

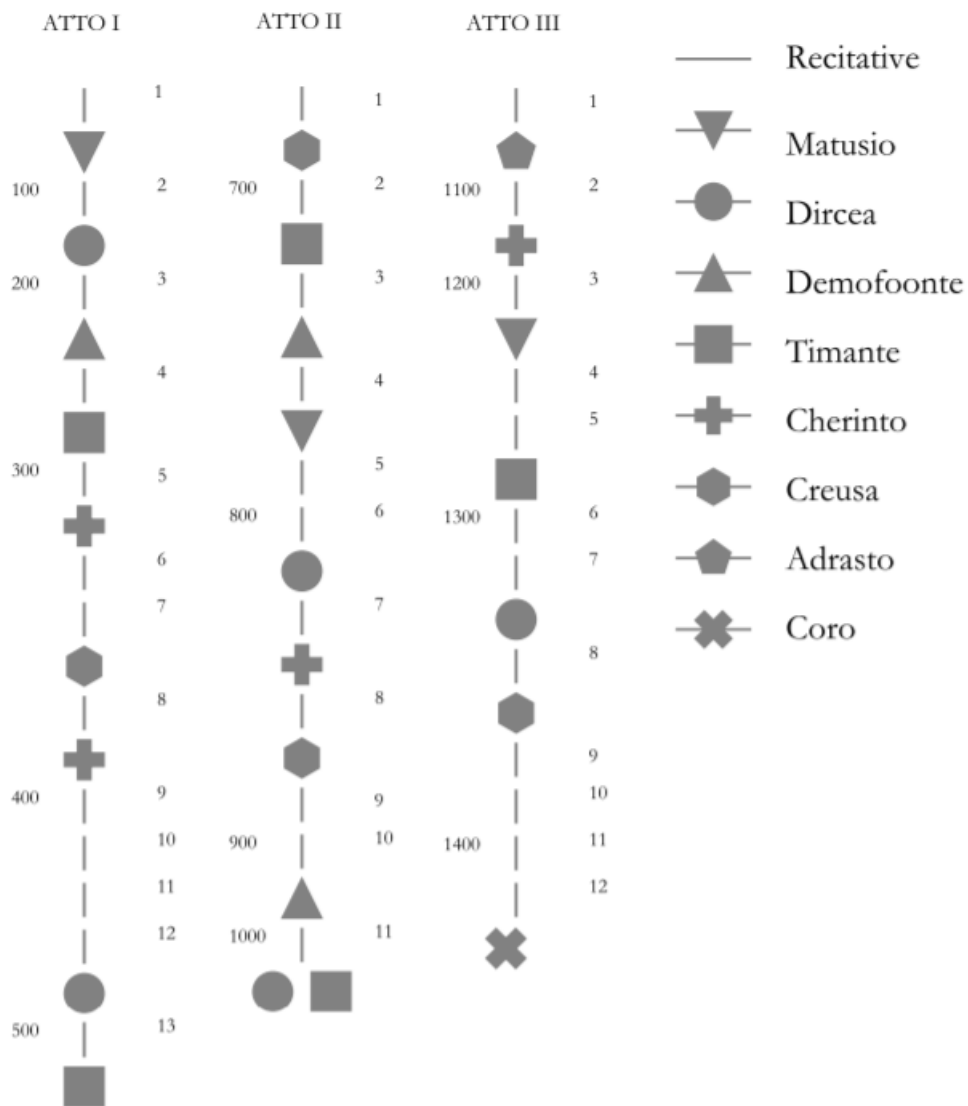


Figure 5 – Overview of the structure of the Demofonte opera, by Pietro Metastasio. Shapes represent the characters that are singing the corresponding Aria. Lines represent the recitatives. The number to the right represents the scene number in the act. Numbers to the left represent the position of the opera in the dramatic text, by number of poetic lines

Source: (MUÑOZ-LAGO et al., 2020)

Music Representations, while WAV and MP3 are Digital Audio Coding Formats that store digital audio signals. In this work, since we are dealing with the task of Computational Musicology, we will focus only on the symbolic formats, as they are, commonly, the formats used by musicologists.

## 2.4.1 MIDI

Musical Instrument Digital Interface (MIDI) is a complete technical standard that aims to enable playing, recording, and editing digital music. It includes a set of specifications for hardware implementation, electrical connectors, communication protocol, and digital music storage (LOY, 1985). Although it was created to standardize synthesizers communication, it has been through many updates, and is broadly used to this day, including in song arrangement and producing, and live electronic performances.

A piece of music is symbolically represented in MIDI format as a binary file, which contains an Event List of MIDI Messages, and other optional meta information (e.g., title, author, tonality, time signature). An Event List example is shown in Figure 6, presented in a human-readable fashion, where each line depicts one MIDI Message.

Trk	HMSF	MBT	Ch	Kind	Data		
1	00:00:07:00	5:03:000	1	Note	A 6	127	192
1	00:00:07:15	6:01:000	1	Note	A 6	127	192
1	00:00:08:00	6:02:000	1	Note	G 6	127	96
1	00:00:08:08	6:02:120	1	Note	F#6	127	96
1	00:00:08:15	6:03:000	1	Note	E 6	127	96
1	00:00:08:23	6:03:120	1	Note	G 6	127	96
1	00:00:09:00	7:01:000	1	Note	F#6	127	192
1	00:00:09:15	7:02:000	1	Note	E 6	127	96
1	00:00:09:23	7:02:120	1	Note	D 6	127	96

Figure 6 – MIDI Event List example

MIDI Messages are sequences of byte codes that contain information about how the piece of music should be performed. Some examples of these information are: when to start a note, when to stop a note, what instrument should be playing a given note, what is the velocity of that note (which is a parameter often used to control loudness), and other optional properties of that note (e.g., pitch bend, lyrics, sustain, modulation). As a quick example, if one wants to send the MIDI message “Turn on the note with pitch A6, in channel 0, with 127 velocity”, the message in binary code is ‘10010000 01101001 01111111’. More information about MIDI Messages can be found in the tutorials by Brink (1995) and Vandenuecker (2012), and detailed in the MIDI Specification (ASSOCIATION et al., 1996).

Musicians often work with MIDI using a Piano Roll visualization, which is usually embedded in a Digital Audio Workstation (DAW). Figure 7 shows the Piano Roll visualization in the Cakewalk<sup>3</sup> DAW. In a Piano Roll visualization, sequential notes are lined up horizontally, and simultaneous notes of different pitches are lined up vertically. Usually, the Piano Roll visualization software also enables editing and creation of MIDI files directly through the graphical interface, eliminating the necessity of the end-user knowing any of the byte codes for the MIDI messages used behind the scenes. Thus, the Piano Roll visualization not only facilitates the editing and creation of MIDI files for the end-user,

<sup>3</sup> <<https://www.bandlab.com/products/cakewalk>> - Accessed on 11 may 2021

but also enables anyone with a computer keyboard and mouse to compose, arrange, and produce music, with minimal music theory knowledge requirements - as long as they are up to a lot of experimentation in a trial-and-error fashion.

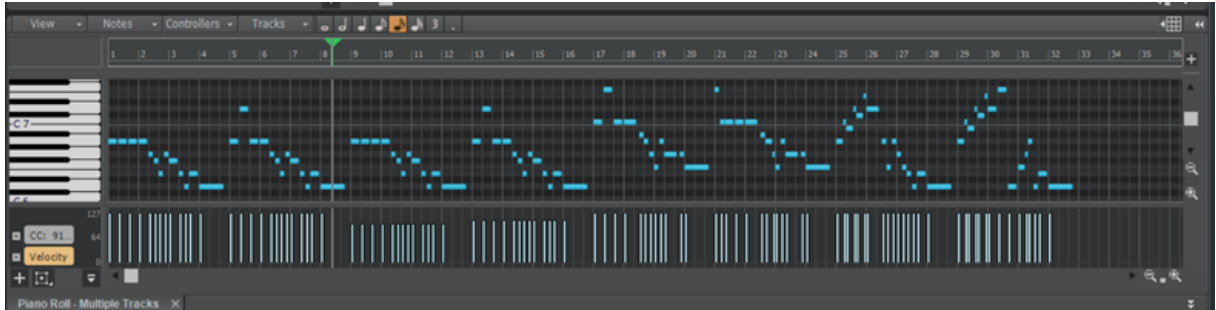


Figure 7 – MIDI Piano Roll visualization example

One could also choose to create a MIDI file by recording their performance using a MIDI Controller - an external hardware that aims to mimic a musical instrument. The MIDI controller often takes the form of a keyboard/digital piano, a drum pad, or a mixing console. [Figure 8](#) illustrates these types of MIDI controllers. However, a MIDI controller can take any physical form that the manufacturer chooses, as long as it follows the protocol for sending MIDI messages to the computer in “real-time”. After the recording, it is still possible to edit the stored MIDI file through the Piano Roll visualization - or directly through binary code.

## 2.4.2 MusicXML

MusicXML is a set of Extensible Markup Language (XML) definitions aiming to standardize the representation of music in western sheet notation for computers ([GOOD, 2001](#)). As any XML-based language, MusicXML is stored as a text file. MusicXML defines specific text strings to be used as tags, standardizing the properties we want to store, in a hierarchically structured fashion. [Figure 9](#) gives an example of a MusicXML file excerpt.

[Figure 9](#) defines one part named “Music”, containing a single measure with a G Major (or E minor) key signature, a 4/4 time signature, a treble clef, and has a C4 whole note. The XML header - which defines the file as a MusicXML format file - is omitted. It is also possible to define metadata, (e.g., the score title, composer name), Graphical User Interface (GUI) and appearance related attributes (e.g., font size, font type, page layout), tempo, lyrics, harmony, instrumentation, dynamics, and even commentaries through MusicXML - just as writing scores on paper.

MusicXML can also be used in an interactive fashion. Some MusicXML visualization software enables the creation and editing of music sheet directly through their graphical interface, very often offering keyboard shortcuts as well. These pieces of software are



(a) MIDI controller in keyboard shape



(b) MIDI controller in drum pad shape



(c) MIDI controller in mixing console shape



(d) MIDI controller with keyboard, drum pad and mixing console

Figure 8 – Different types of MIDI controllers

often called “scorewriters”. By using digital instruments, some software also enables the playback of the MusicXML, giving instant audible feedback for each modification in the music sheet. The playback function also enables multiple instruments to be played at once, even without any musicians available to play the physical instruments, facilitating the composition and arrangement for bands and orchestras. MusicXML visualization software include MuseScore, Finale, Guitar Pro, Sibelius, and many others. Figure 10 depicts the way that the MusicXML in Figure 9 is visualized using the scorewriter MuseScore<sup>4</sup>. We note that the part name (“Music”) is not shown since, by default MuseScore does not show part names when there is only one part.

<sup>4</sup> <<https://musescore.com/>> - Accessed on 1 Apr 2021

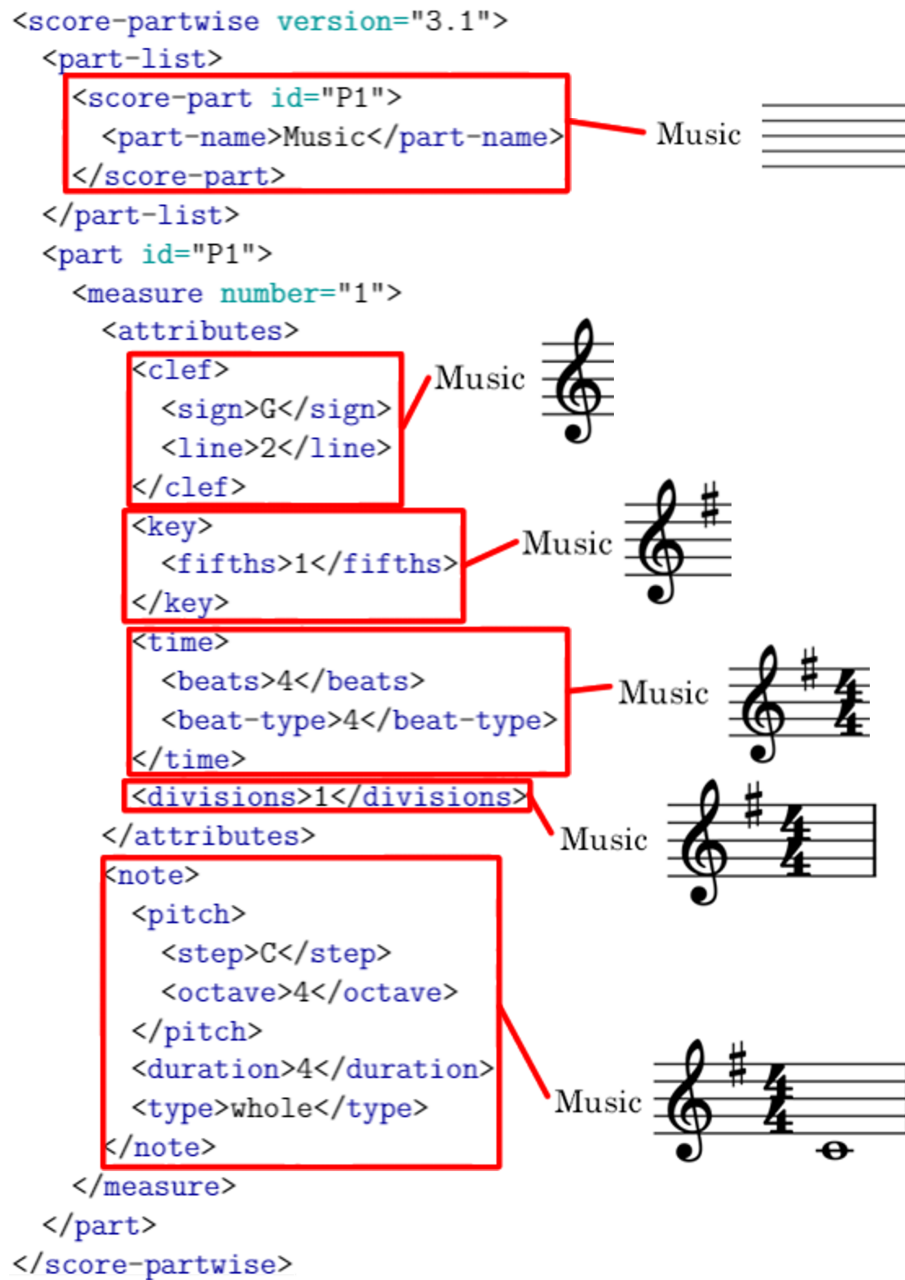


Figure 9 – MusicXML “Hello World” file excerpt, with visual indicators of each block’s purpose

The main advantage of MusicXML over MIDI as a symbolic music representation format is that the former explicitly annotates rests, notes duration, and other specific symbols (such as fermatas, mordents, and grupettos), while the latter does not annotate such information at all. MIDI is also incapable of differentiating enharmonics (i.e., notes with the same pitch frequency, but different pitch class names, such as  $A\flat$  and  $G\sharp$ ). They are heard exactly the same, but they might have a different musical meaning in the context of the piece). This means that when one tries to convert MIDI to music sheet notation (and vice-versa), the conversion algorithm will have to guess these missing

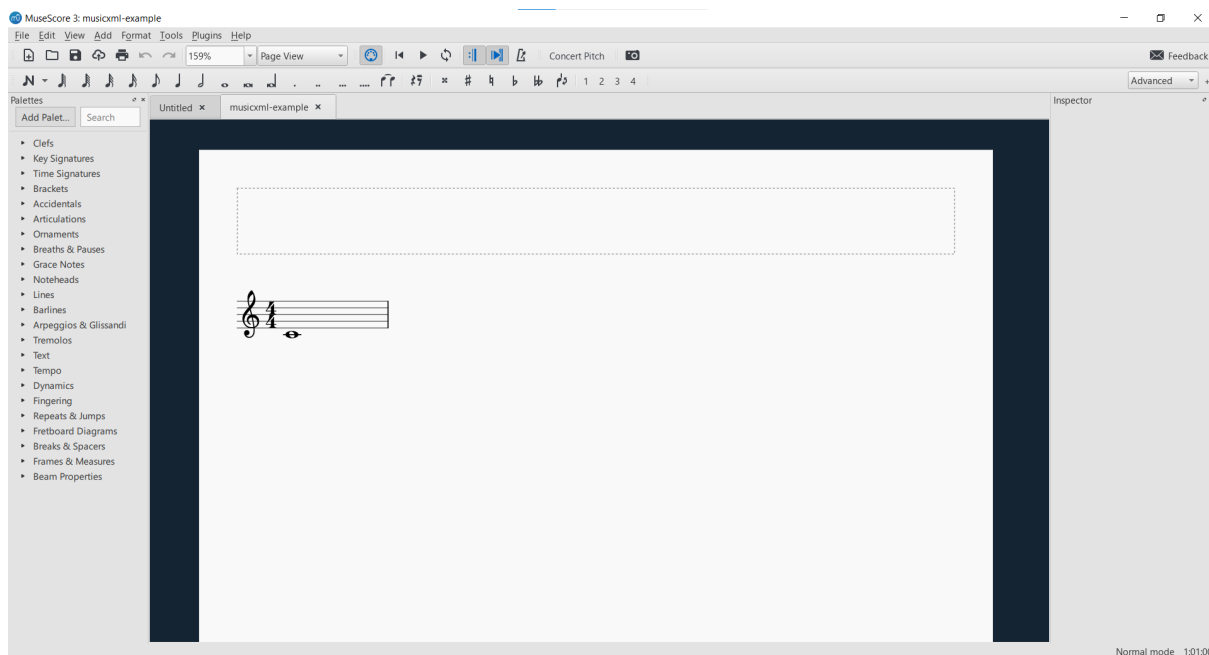


Figure 10 – Visualization of the MusicXML “Hello World” in MuseScore

information, occasionally making mistakes, and requiring manual review and correction afterwards (GOOD, 2001).

### 2.4.3 MEI

Music Encoding Initiative (MEI) (ROLAND, 2002) is another symbolic representation of music in sheet notation, encoded in XML - much like MusicXML. However, MEI has some defining differences, both in implementation, in philosophy, and in scope:

- While MusicXML prefers clarity over conciseness, MEI focus on the opposite - files written in MEI are less readable than MusicXML, but occupies less memory space in disk. Hence, MusicXML might be slightly easier to work with when developing applications - although this should not affect most of the end-users directly, since they will seldom work without visualization software.
- MusicXML was developed with the intention of standardizing music sheet notation across multiple music sheet visualization and editing software, and to enable easy transfer over the Web. Conversely, MEI was conceptualized to assist in musical analysis and musicology research, annotating information that usually would not be annotated in MusicXML.
- Currently, MusicXML is only able to represent music in Common Western notation, while MEI is able to represent other notations as well - such as the Neume nota-



tion (from the Medieval period), or the Mensural notation (from the Renaissance period) (ROLAND; HANKINSON; PUGIN, 2014).

Although MEI is meant to be more complete than MusicXML, MusicXML is still the most used format for music sheet notation representation (KEPPER, 2009). There is currently a lack of tools that use MEI - out of the popular music sheet visualization software, only Sibelius has a third-party plugin that enables the use of files in MEI format. Furthermore, the conversion between MusicXML and MEI may cause loss of information (PARADA-CABALEIRO; TORRENTE, 2020).

## 2.5 Sequential Pattern Mining

This subsection aims to be a quick introduction to Sequential Pattern Mining (a.k.a. Sequence Pattern Discovery), and to introduce the notation that will be used hereinafter. For a more detailed explanation on the topic, we recommend reading the survey by (FOURNIER-VIGER et al., 2017), which is the theoretical foundation for this whole subsection.

The traditional Sequential Pattern Mining problem might be defined as: having a sequence database  $SDB$  containing  $n$  sequences  $S_i$ , find the  $k$  most frequent subsequences  $s_j \sqsubseteq S_i | 1 \leq i \leq n \wedge 1 \leq j \leq k$  that occur in the  $SDB$ . An alternative to the hard-defined number of outputs  $k$  is to define the minimum support threshold  $minSup$  - i.e., the minimum appearance frequency in the  $SDB$  for a subsequence to be included in the output.

While Frequent Itemset Mining algorithms work with a dataset of unordered sets of items, sequences are defined as ordered sets of unordered sets of items. As a comparison, Frequent Itemset Mining can be used to find sets of items that are frequently bought together by the same person; whereas Sequential Pattern Mining can be used to find sets of items that are frequently bought some time after a set of items were bought by the same person.

As standard notation, sequences (ordered sets) will be enclosed by  $\langle \rangle$ , and unordered sets will be enclosed by  $\{ \}$ . Unordered sets contained in a sequence will be separated by commas, showing that the rightmost set succeeds the leftmost one. Items contained in unordered sets are also separated by commas, but items in the same unordered set occur concurrently. As an example, in Table 2, in the sequence with Sequence Identifier (SID) 1, the set  $\{a, b\}$  comes before the set  $\{c\}$  - i.e.,  $\{a, b\} \prec \{c\}$  - while the items  $a$  and  $b$  occur concurrently. The length of a sequence, denoted by  $|S_i|$ , is the number of unordered sets contained in the sequence. For example, in Table 2,  $|S_3| = 4$  and  $|S_4| = 2$ .

Table 2 – Sequence Database example

SID	Sequence
1	$\langle \{a, b\}, \{c\}, \{f, g\}, \{g\}, \{e\} \rangle$
2	$\langle \{a, d\}, \{c\}, \{b\}, \{a, b, e, f\} \rangle$
3	$\langle \{a\}, \{b\}, \{f, g\}, \{e\} \rangle$
4	$\langle \{b\}, \{f, g\} \rangle$

**Source:** adapted from (FOURNIER-VIGER et al., 2017)

There are multiple ways to define how to measure the support of a subsequence in a *SDB*. In this work, we will use the following definitions:

1. As shown in Equation 1, the absolute support of a sequence  $s_j$  in a *SDB*, denoted by  $absSup(s_j, SDB)$ , is the number of sequences  $S_i$  in the *SDB* that contain the subsequence  $s_j$  - only one occurrence of  $s_j$  is counted per sequence  $S_i$ .

$$absSup(s_j, SDB) = ||\{S_i | S_i \supseteq s_j\}|| \forall S_i \in SDB \quad (1)$$

where the  $||$  delimiters represents the cardinality (i.e., the number of items) of the enclosed unordered set.

2. The relative support of a subsequence  $s_j$  in a *SDB*, denoted by  $relSup(s_j, SDB)$ , is defined as in Equation 2. It is the proportion of sequences  $S_i$  that contain the subsequence  $s_j$ , relative to the number of sequences in *SDB*.

$$relSup(s_j, SDB) = \frac{absSup(s_j, SDB)}{n} \quad (2)$$

where  $n$  is the number of sequences in the *SDB*.

3. The total support of a subsequence  $s_j$  in a *SDB*, denoted by  $totalSup(s_j, SDB)$ , is defined in Equation 3. It is the total number of times the subsequence  $s_j$  appears in the *SDB*, taking into consideration if it happens more than once in the same sequence as well.

$$totalSup(s_j, SDB) = COUNT(s_j, SDB) \quad (3)$$

where  $COUNT(s_j, SDB)$  is the number of times the subsequence  $s_j$  appears in the dataset *SDB*.

We note that there might be cases in which two instances of the same subsequence might overlap. In such cases, we would count the total support up by only one. As an example, suppose we have a *SDB* with a single sequence  $S_1 = \langle \{a\}, \{b\}, \{a\}, \{b\}, \{c\} \rangle$ , and we want to calculate the total support of the subsequence  $s_j = \langle \{a\}, \{b\}, \{c\} \rangle$ . We might consider that one instance of  $s_j$  in  $S_1$  is the

first two itemsets and the last one from  $S_1$ ; and we also might consider that the other instance of  $s_j$  in  $S_1$  is the three last itemsets of  $S_1$ . However, since these two instances overlap, we consider only one of these instances, and so  $totalSup(s_j, SDB) = 1$ .

As an example for these support measures, the subsequence  $\langle\{a\}\rangle$  from Table 2 has  $absSup = 3$ ,  $relSup = \frac{3}{4} = 0.75$ , and  $totalSup = 4$ .

Considering  $minSup = 3$ , the  $absSup$  as the support measure, and without any other constraints, a Sequential Pattern Mining algorithm would output the subsequences (patterns) presented in Table 3. We can see, for example, that  $\{a\} \prec \{f\}$  in the sequences  $S_1$ ,  $S_2$ , and  $S_3$ , thus  $absSup(\langle\{a\}, \{f\}\rangle) = 3$ .

Table 3 – Sequential patterns found in the SDB of Table 2, with  $minSup = 3$

Pattern	absSup
$\langle\{a\}\rangle$	3
$\langle\{a\}, \{f\}\rangle$	3
$\langle\{a\}, \{e\}\rangle$	3
$\langle\{b\}\rangle$	4
$\langle\{b\}, \{g\}\rangle$	3
$\langle\{b\}, \{f\}\rangle$	4
$\langle\{b\}, \{f, g\}\rangle$	3
$\langle\{b\}, \{e\}\rangle$	3
$\langle\{e\}\rangle$	3
$\langle\{f\}\rangle$	4
$\langle\{f, g\}\rangle$	3
$\langle\{g\}\rangle$	3

**Source:** adapted from (FOURNIER-VIGER et al., 2017)

Sequential Pattern Mining algorithms differ between themselves by the way they model and search for the frequent patterns, and by the additional constraints that are imposed to find the most relevant patterns applied to a specific problem. For this work, we are interested in one algorithm, called SPAM (AYRES et al., 2002), since the MGDPA Algorithm (CONKLIN, 2010a) was based upon it.

### 2.5.1 SPAM

SPAM (AYRES et al., 2002) uses a lexicographical tree representation for sequences. It assumes that exists an underlying order between items in an unordered set, purely for algorithmic purposes (even if there is no actual order in reality). As an example, the letter “a” precedes the “b” in the dictionary, so the item  $a$  will precede  $b$  in an unordered set with a lexicographical order.

The root of the tree is denoted by  $\emptyset$ . Each node of the tree is a sequence that is the result of either a sequence-extension (s-extension) or an itemset-extension (i-extension).

A s-extension consists in appending an unordered set with a single item in it to the end of a sequence. For example, the result of a s-extension of the sequence  $\langle\{a, b, c\}\rangle$  could be  $\langle\{a, b, c\}, \{a\}\rangle$  or  $\langle\{a, b, c\}, \{b\}\rangle$ , but could not be  $\langle\{a, b, c\}, \{a, b\}\rangle$ . An i-extension consists in appending a single item to the last unordered set of a sequence, given that the item being appended succeeds all items that are already in that unordered set, following the lexicographical order. For example, one result of an i-extension of the sequence  $\langle\{a, b\}, \{a, c\}\rangle$  could be  $\langle\{a, b\}, \{a, c, d\}\rangle$ , but could not be  $\langle\{a, b\}, \{a, b, c\}\rangle$ . Figure 11 depicts the process of generating a Lexicographical Sequence Tree through a series of s-extensions and i-extensions, considering a *SDB* containing only two items (*a* and *b*), and presenting only sequences with a maximum sequence length of 2.

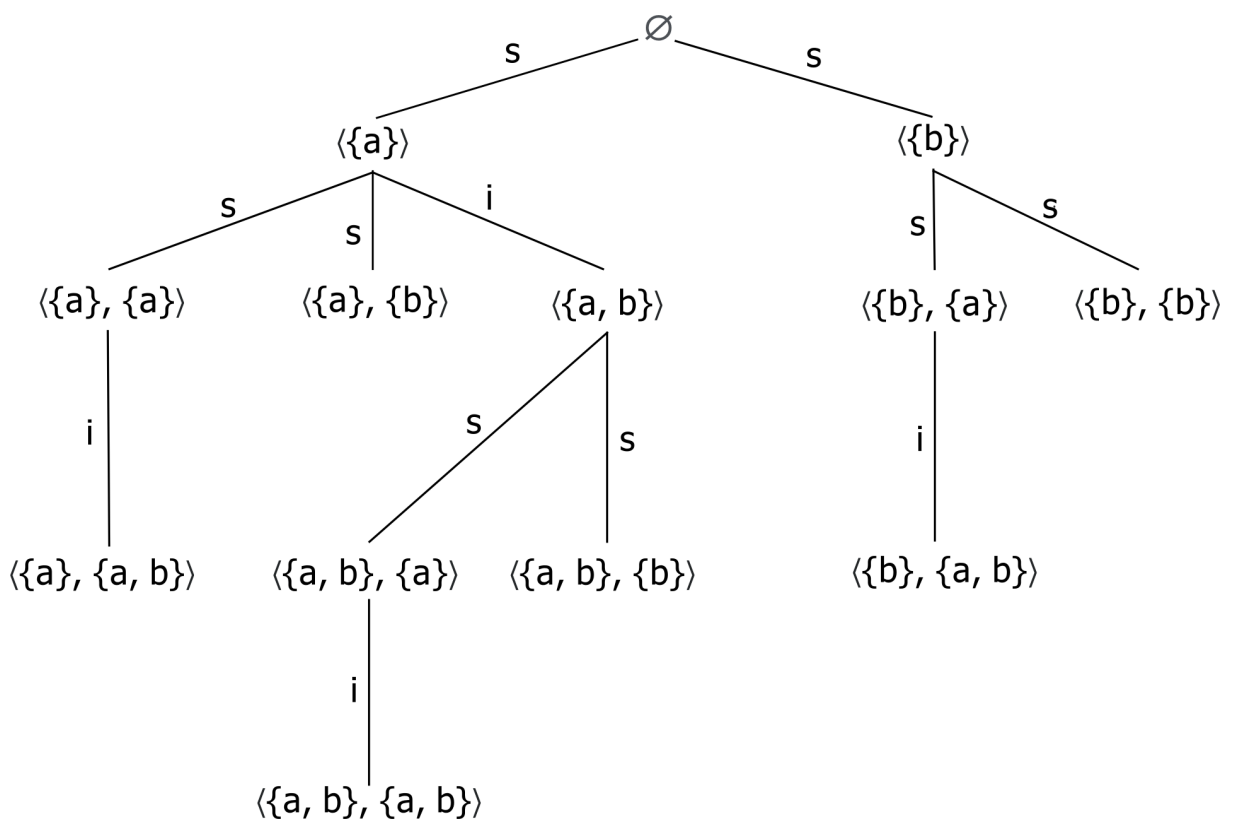


Figure 11 – Example of generation of a Lexicographical Sequence Tree

Source: adapted from (AYRES et al., 2002)

The Sequence Tree is traversed through a Depth-First Search approach. Following the Apriori principle (AGRAWAL; SRIKANT, 1994), if a node  $j$  contains a sequence  $s_j$  with a support smaller than  $minSup$ , then there is no need to verify any of the children of the node that contains  $s_j$ , since they will, consequently, also be infrequent.

Given a node  $j$  containing a sequence  $s_j$ , we can associate two sets with the node  $j$ : the set of candidate items to be appended to  $s_j$  during an s-extension, denominated  $\Sigma_j$ ; and the set of candidate items to be appended to  $s_j$  during an i-extension, denominated

$I_j$ . Using the Apriori principle, we can also reduce the size of both  $\Sigma_j$  and  $I_j$  during the generation of the tree in two cases.

In the first case, consider that we have a node  $j$  containing a sequence  $s_j$ . Now, suppose  $s_j$  has two s-extensions,  $s_j^a = \langle s_j, \{i_a\} \rangle$  and  $s_j^b = \langle s_j, \{i_b\} \rangle$ , and that  $s_j^a$  is frequent but  $s_j^b$  is not frequent. Following the Apriori principle, we can then assume that both  $\langle s_j, \{i_a, i_b\} \rangle$  and  $\langle s_j, \{i_a\}, \{i_b\} \rangle$  are going to be infrequent, since  $s_j^b$  is contained in both these sequences. Thus, we can safely remove the item  $i_b$  from both  $\Sigma_k$  and  $I_k$ . where  $k$  is any node that contains a frequent s-extension of  $s_j$ . This process is called S-step Pruning.

In the second case, consider that we have a node  $j$  containing a sequence  $s_j = \langle s', \{i_1, \dots, i_n\} \rangle$ . Now, suppose  $s_j$  has two possible i-extensions:  $s_j^a = \langle s', \{i_1, \dots, i_n, i_a\} \rangle$  and  $s_j^b = \langle s', \{i_1, \dots, i_n, i_b\} \rangle$ , and that  $s_j^a$  is frequent but  $s_j^b$  is not frequent, and also that  $i_a \prec i_b$  following the lexicographical order. Then, we can assume that, by the Apriori principle,  $\langle s', \{i_1, \dots, i_n, i_a, i_b\} \rangle$  cannot be frequent. Thus, we can safely remove the item  $i_b$  from  $I_k$  where  $k$  is any node that contains a frequent i-extension of  $s_j$ . Additionally, if  $\langle s_j, \{i_c\} \rangle$  is infrequent, then we can assume that  $\langle s_j^a, \{i_c\} \rangle$  is also infrequent given that  $s_j^a$  is any frequent i-extension of  $s_j$ . Thus, we can also remove  $i_c$  from  $\Sigma_k$  where  $k$  is any node that contains a frequent i-extension of  $s_j$ . Note that  $i_c$  would be the same item that was removed during the S-step pruning, so there is no need to re-verify the frequency of this sequence. This process is called I-step Pruning.

### 2.5.1.1 Vertical Bitmaps

SPAM converts the sequences into a vertical bitmap representation to efficiently traverse through the tree. To illustrate this conversion process, consider the example given in Table 4. The verticalization of this SDB consists in constructing a *Bitmap* that maps the positions in which the patterns are present for each sequence in the database.

Table 4 – Another Sequence Database example

SID	Sequence
1	$\langle \{a, b, d\}, \{b, c, d\}, \{b, c, d\} \rangle$
2	$\langle \{b\}, \{a, b, c\} \rangle$
3	$\langle \{a, b\}, \{b, c, d\} \rangle$

Source: adapted from (AYRES et al., 2002)

The result of the verticalization of Table 4 is shown in Table 5. We can see that for the Sequence with ID 1, the pattern  $\langle \{a\} \rangle$  is present in the first position, and is not present in the second and third position of the sequence. Note that the Sequence with ID 1 contains 3 featuresets, so its Bitmaps have a length of 3; while Sequences 2 and 3 contain 2 featuresets, and their bitmaps have a length of 2.

Table 5 – Verticalized SDB example

SID	$\langle\{a\}\rangle$	$\langle\{b\}\rangle$	$\langle\{c\}\rangle$	$\langle\{d\}\rangle$
<b>1</b>	1	1	0	1
	0	1	1	1
	0	1	1	1
<b>2</b>	0	1	0	0
	1	1	1	0
<b>3</b>	1	1	0	0
	0	1	1	1

Source: adapted from (AYRES et al., 2002)

The verticalization of the SDB is efficient because it needs to be done only once, and only for the *canonical patterns* (i.e., the patterns that contain only one featureset, which, in turn, contain only one feature). Then, we apply the s-extension and i-extension operations, which uses only fast bitwise operations, discarding the need to go through the whole dataset again.

To perform an i-extension on the Vertical Bitmaps, we simply perform a vertical bitwise-AND operation over two bitmaps. For example, taking Table 5 as a starting point, if we want to perform an i-extension on the pattern  $\langle\{a\}\rangle$ , adding the feature  $b$  to it, we would bitwise-AND the two bitmaps corresponding to the patterns  $\langle\{a\}\rangle$  and  $\langle\{b\}\rangle$ . The result of this operation is illustrated by Table 6.

Table 6 – Result of i-extension operation through vertical bitmaps

SID	$\langle\{a\}\rangle$	$\langle\{b\}\rangle$	$\langle\{a, b\}\rangle$
<b>1</b>	1	1	1
	0	1	0
	0	1	0
<b>2</b>	0	1	0
	1	1	1
<b>3</b>	1	1	1
	0	1	0

Source: adapted from (AYRES et al., 2002)

To perform an s-extension operation on vertical bitmaps, we first have to construct an auxiliary bitmap from the existing pattern bitmap, and then we bitwise-AND this auxiliary bitmap with the bitmap from the feature being appended to the pattern. This auxiliary bitmap is constructed by first finding out the index  $k_{SID}$  that corresponds to the position of the first occurrence of the pattern in the bitmap (i.e., the index of the first element that is set to 1 on the bitmap), for each sequence in the SDB. Then, for each sequence in the SDB, we have a bitmap in which all indexes bigger than  $k_{SID}$  is set to 1, while all indexes less than or equal to  $k_{SID}$  is set to 0. Take Table 7 as an example. We want to perform a s-extension of pattern  $\langle\{a\}\rangle$  with feature  $\langle\{b\}\rangle$ . To do so, we first

construct the auxiliary bitmap  $\langle\{a\}\rangle_s$ , and then we bitwise-AND the auxiliary bitmap  $\langle\{a\}\rangle_s$  with the bitmap for  $\langle\{b\}\rangle$ .

Table 7 – Result of s-extension operation through vertical bitmaps

<b>SID</b>	$\langle\{a\}\rangle$	$\langle\{a\}\rangle_s$	$\langle\{b\}\rangle$	$\langle\{a\}, \{b\}\rangle$
<b>1</b>	1	0	1	0
	0	1	1	1
	0	1	1	1
<b>2</b>	0	0	1	0
	1	0	1	0
<b>3</b>	1	0	1	0
	0	1	1	1

**Source:** adapted from (AYRES et al., 2002)

Note that the resulting bitmap of the s-extension does not store the occurrence of every featureset of the pattern - it only stores the occurrences of the last featureset of the pattern (given that the anterior featuresets have already occurred in the sequence). This is an important limitation of this algorithm: we output just whether a pattern occurs in a sequence, and the positions of the last featureset of the pattern that appeared after all other featuresets of the pattern. But we do not output the positions of occurrences of the previous featuresets of the pattern. In the example shown in Table 7, the bitmap for  $\langle\{a\}, \{b\}\rangle$  outputs the positions of every occurrence of  $\{b\}$  that appears after an  $\{a\}$  has occurred, but we do not know the positions of any occurrences of  $\{a\}$ .

## 3 Tools and Methods

### 3.1 Computational Musicology Tools

#### 3.1.1 music21

The music21 toolkit for Python that offers a set of tools aiming to aid musicologists in analyzing, transforming, and searching for patterns in symbolic music. It is shipped with a large corpus of Bach Chorales, which might be used to test and benchmark algorithms. [Cuthbert and Ariza \(2010\)](#) introduces the toolkit, along with some examples of practical applications in which music21 can be used.

music21 has a built-in feature extraction module, which includes some features extracted with the jSymbolic toolkit, and some features that are native to music21. It is also possible to write custom feature extractors and append them to the music21 toolkit ([CUTHBERT; ARIZA; FRIEDLAND, 2011](#)).

[Phon-Amnuaisuk \(2019\)](#) utilized music21 to analyze the tendency of chord progressions and cadences in Bach Chorales. [Sampaio et al. \(2013\)](#) extends music21 with a Countour Analysis algorithm, which was used to extract, describe, and compare the melodic contours of the different voices of Bach Chorales. [Garfinkle et al. \(2017\)](#) utilized music21 to make a Content-Based Music Retrieval system for polyphonic symbolic music. [Liang et al. \(2017\)](#) utilizes music21 as part of their BachBot system, which automatically generates Bach Chorales using Deep Long Short-Term Memory (LSTM) Recurrent Neural Networks (RNN).

### 3.2 Inter-Opus Musical Motif Discovery

The task of Inter-Opus Musical Motif Discovery in Symbolic Music Corpus consists in automatically finding the repeating patterns and their variations in a sequence of notes from multiple pieces of music represented in a symbolic format (e.g., music sheet) ([CONKLIN, 2010a](#)). In this section we will define some particularities of this task.

#### 3.2.1 Sequence Clustering VS Sequential Pattern Mining

One of the ways to formally define the problem of Inter-Opus Musical Motif Discovery could be: given a set  $C$  containing  $n$  sequences  $(S_i)_{i \in n} \subset C$ , identify  $k$  clusters, such that each cluster contains subsequences  $(s_{ij})_{j \in m_i} \subseteq S_i$  that are sufficiently similar to the other subsequences in the same cluster, where  $m_i$  is the number of elements in  $(S_i)$ .



This definition of the problem falls into the Sequence Clustering problem definition. However, there are some particularities of this Sequence Clustering problem when considering the application domain of Musical Motif Discovery, which raises these Challenges:

1. We have to consider the possibility of slight variations in rhythm, melodic contour, harmony, etc. when clustering the musical motifs;
2. Each subsequence ( $s_{ij}$ ) might have different lengths, which complicates the similarity measuring;
3. We don't know the length of the subsequences ( $s_{ij}$ ) in advance, meaning one solution might correctly identify the presence of similar musical motifs, but present them shortened or elongated. This might complicate the solution's overall efficacy measure, and the similarity measuring;
4. For a sequence ( $S_i$ ) of length  $m_i$  we have  $\frac{m_i(m_i+1)}{2}$  possible subsequences. This means we would need to spend a very high quantity of computational resources just to find and store all possible subsequences of a sequence -  $O(n^2)$ ;
5. Each sequence ( $S_i$ ) might have different lengths  $m_i$ , which complicates the search and comparison of similar subsequences contained in them;
6. Similar subsequences might be in different positions. This means we would need to spend a very high quantity of computational resources just to compare the similarity of a subsequence with all the other subsequences in another sequence -  $O(n^2)$ ;
7. The number of clusters  $k$  is not known in advance.

To find a solution for the Inter-Opus Musical Motif Discovery task following the Sequence Clustering approach, we might split the problem into four Sub-Tasks, while keeping in mind the aforementioned particularities of the application domain, and trying to keep the problem solvable in feasible time:

1. Define an approach to model the symbolic music into a format acceptable to the subsequent steps;
2. Define a similarity measure method to compare the musical motifs, considering all the possible musical variations;
3. Define a fitting Sequence Clustering algorithm;
4. Define an evaluation method to measure the solution's efficacy.

The same Inter-Opus Musical Motif Discovery task might also be tackled through a Sequential Pattern Mining approach, which consists in finding frequent and relevant subsequences  $s_{ij}$  in the sequence database  $C$  (see [Section 2.5](#)). In this case, we might be able to use popular algorithms such as GSP ([SRIKANT; AGRAWAL, 1996](#)), SPADE ([ZAKI, 2001](#)), SPAM ([AYRES et al., 2002](#)), and others - or a variation of them.

Since finding a similarity measure is not required for the Sequential Pattern Mining approach, we could substitute Sub-Tasks 2 and 3 with one single step, consisting in defining a fitting Sequential Pattern Mining algorithm. However, discarding the similarity measure Sub-Task means that the Sequential Pattern Mining approach will not be able to directly tackle Challenge 1 unless we either model the symbolic music into a musical variation invariant (sic) format, or we embed the musical variation invariance into a Sequential Pattern Mining algorithm, somehow, or we consider only very short musical motifs.

As an advantage, the Sequential Pattern Mining approach would be able to point specifically what are the common patterns between each subsequence, while the Sequence Clustering approach would only be able to point that a subsequence is similar to another subsequence, without pointing exactly why that is the case - unless the similarity measure method chosen is somewhat interpretable. [Karsdorp, Kranenburg and Manjavacas \(2019\)](#) investigated the use of Siamese Recurrent Neural Networks for an automatic melodic similarity measure learning. However, even though it was successful in its task, due to the black-box nature of the deep learning algorithms used, we have no insights into what musical features the RNN takes into consideration for its output. As mentioned in [Section 2.1](#), the Computational Musicology field has the goal of understanding music. For this reason, interpretability is highly valued in this project. Even if we were to use some kind of Machine Learning Explanation algorithm similar to SHAP ([LUNDBERG; LEE, 2017](#)), we might get to faulty conclusions ([RUDIN, 2019](#)), confusing the knowledge that we already have, and hindering our goal of a better understanding of music. Thus, we chose to follow the Sequential Pattern Mining approach instead of the Sequence Clustering one.

### 3.2.2 Related Work

[Mongeau and Sankoff \(1990\)](#) developed a method for localizing similar portions of two scores. They model the pieces as a two-dimensional array - consisting in pitch and duration - and use a dynamic programming approach to compute a similarity measure between each piece's portions. It can align two monophonic melodies, even with small variations in pitch and rhythm. This method could be extended to a corpus containing more than 2 scores; however, it might not be computationally feasible to use it with a large corpus, and it is not ideal for finding novel, musically interesting patterns ([CONKLIN; ANAGNOSTOPOULOU, 2001](#)).

[Conklin and Anagnostopoulou \(2001\)](#) implemented a faster method for discovering

musical patterns in a large corpus of music by employing a search-and-count through a Suffix Tree Data Structure. A method to define if a musical pattern is relevant was also developed: sequences are deemed relevant if there is a statistically significant difference between the frequency they occur in the corpus and the frequency in which they would occur following a Markov Model baseline. However, since the Markov Model represents just a simulation of actual music, using it as a baseline might lead to the presentation of many musically uninteresting patterns (CONKLIN, 2010a).

Conklin and Bergeron (2008) changed the pattern search algorithm from the Suffix Tree Data Structure to a Lexicographical Sequence Tree method, based on the SPAM algorithm (AYRES et al., 2002). However, they mention that they use an *Instance Map* that tracks the position in which the patterns occur (and do not mention Bitmaps), although not much details are given regarding this matter. It is not clear whether they use a method to track the indices of all musical events (i.e., notes/rest/chords) that compose a pattern, or if it tracks only the index of the end of the pattern, just like the Bitmap do. They also use a Subsumption Tree to avoid visiting patterns in the search space that do not exist on the dataset, and to avoid visiting the same pattern in the search space more than once. They compare a complete search space method with an heuristic method. They do mention a limitation of their algorithm: “the pattern discovery method developed here does not permit insertions or deletions of events while computing pattern occurrences”.

Conklin (2010a) introduces the Maximally General Distinctive Patterns (MGDP) for music, and also replaces the Markov Model with an anticorpus for determining the musical relevance of the discovered patterns. The anticorpus might be defined as a set of negative samples; e.g., if one is trying to find patterns of folk music, the anticorpus would consist in non-folk music. A pattern is considered distinctive if it has a support (frequency ratio) in the corpus that is sufficiently higher than its support in the anticorpus. The distinctiveness of a pattern is the ratio of supports from the corpus and anticorpus. A distinctive pattern  $(s_{ij})$  is considered a MGDP if there is not any other distinctive pattern in the corpus that is a subsequence of  $(s_{ij})$ . To discover the MGDPs of a corpus, a variation of the SPAM algorithm (AYRES et al., 2002) was employed, following the work from Conklin and Bergeron (2008).

Conklin (2013) reutilizes the concept of MGDP and the anticorpus introduced by Conklin (2010a) to find the antipatterns of a corpus - i.e., patterns that are absent or have a surprisingly low frequency in a corpus. Antipatterns enlighten patterns that should be absent in a piece of music to match the corpus' class - e.g., medieval Gregorian chants should have few or none tritones. To find the antipatterns, they find the MGDP of the anticorpus in respect to the corpus. They apply this method to find patterns and antipatterns in 1561 Basque folk music, labeled by genre. They also introduce the idea of the one-vs-all for a multi-class comparative analysis. To define if a pattern or antipattern

is indeed associated with a given a class, a Fisher statistical significance test is performed.

Meredith (2013) introduces new sequence pattern discovery algorithms, called COSIATEC and SIATECCompress. The proposed algorithms were tested in the JKU Patterns Development Database, which contains 5 samples in symbolic format (MIDI) and intra-opus musical motif annotations from reference books in the music theory field. The algorithms are proposed for intra-opus pattern discovery. Although the algorithms seem to perform well when comparing with the manual annotations, the author questions the use of manually annotated musical motifs as the ground-truth for assessing the performance of musical motif discovery algorithms. They argue that the claim of “interestingness” or importance of a pattern by a single analyst does not mean much without a thorough explanation of why a pattern is interesting and another is not.

Neubarth and Conklin (2016) extends the method from Conklin (2013), that finds patterns that contrast multiple groups using the one-vs-all strategy. Three case-studies are presented, demonstrating different strategies to model music, different pattern mining methods, and use cases with different datasets. They also show algorithms for discovering both sequential patterns (i.e., musical motifs/themes), and global patterns (i.e., instead of focusing in sequential patterns of musical events, the focus is in features that describe the piece as a whole, such as time signature, tonality, etc.).

Kranenburg and Conklin (2016) reassess the performance of the method described in Neubarth and Conklin (2016) by applying it to the Meertens Tune Collection (MTC-ANN), which is a dataset of 360 Dutch Folk melodies with intra-opus annotations. Similar to Meredith (2013), they also question the use of manually annotated musical motifs to evaluate the musical pattern discovery algorithms, suggesting that the ground-truth would be too subjective. Nonetheless, they obtained a high precision, demonstrating that it can match the human-annotated musical motifs.

Nuttall et al. (2019) proposes the use of the TF-IDF statistic to find inter-opus distinctive patterns. It is used to find melodic patterns in collections of pieces of music (called *nawba*), from the Moroccan Arab-Andalusian region. Scores are represented as a bag-of-patterns in a n-gram fashion, containing only the pitch of each note and discarding the n-grams that contain rests. The TF-IDF is calculated for each n-gram of each score, and then they are averaged for each *nawba*, giving a measure of importance of each pattern for each *nawba*. Then, they apply some filtering rules to discard the most musically irrelevant patterns. They evaluate their algorithm objectively by comparing the algorithm’s outputs with the patterns annotated by one specialist, through the comparison of logistic regression classifiers that are meant to classify each *nawba* using either the specialist’s patterns or the discovered patterns. The mean accuracy of the classifier that used the patterns discovered by the algorithm slightly outperformed the one with the specialist annotated patterns, although no statistical significance test was reported.

Nuttall et al. (2021) implements a system to discover distinctive melodic patterns for each melodic mode from 145 Arab-Andalusian songs. It utilizes the MGD algorithm (CONKLIN, 2010a) alongside the TF-IDF-based algorithm (NUTTALL et al., 2019), and a variation of the SIA algorithm (MEREDITH; LEMSTRÖM; WIGGINS, 2002) that enables SIA to find inter-opus patterns instead of just intra-opus, as originally proposed. Then, a pattern selection from the pool of candidate patterns of all three algorithms is proposed and performed to try and prune the most musically irrelevant patterns. They evaluate the algorithm objectively by comparing the results with the annotated patterns made by one specialist, and reporting the Precision, Recall and F1-Score for each melodic mode.

The Inter-opus Musical Motif Discovery task currently lacks a well-defined state-of-the-art algorithm (NUTTALL et al., 2021). This is probably due to the difficulty in evaluating the algorithms efficacy, apart from time complexity. We saw that a lot of papers use expert annotations, while other state that expert annotations are too subjective (MEREDITH, 2013; KRANENBURG; CONKLIN, 2016), since even experts might disagree with themselves in what are relevant patterns and what are not. One approach that might deal with this evaluation issue is to compare automatic classifier performances using the discovered patterns from each algorithm, similar to what has been done by Nuttall et al. (2019). Also, we should have in mind that different algorithms might be used for different purposes; e.g., MGD might be more useful in inter-opus contexts, while closed-sequence discovery algorithms could be more adequate in the intra-opus context (CONKLIN, 2021).

### 3.2.3 Viewpoint Data Representation

Conklin and Witten (1995) introduces a new data representation for music modeling. They define a *feature set* consisting in a set of key-value pairs for each sequential musical *event* (i.e., note, rest, or chord). Each key-value pair is called a *feature*. The key is the name of a function related to one event, and is called a *viewpoint*, while the value is the result of that function. The process of converting each event to feature sets is called *saturation*. Table 8 shows an example of some viewpoints that are possible to extract from each note and Figure 12 depicts the result from a transformation of a music excerpt using the viewpoints defined in Table 8.

Each feature can be seen as an item, and feature sets can be seen as unordered sets (or itemsets), as described in Section 2.5. In this way, we can use the viewpoint representation to model music scores as sequences, enabling the use of variations of standard Sequential Pattern Mining algorithms to automatically discover musical patterns (CONKLIN; BERGERON, 2008).

One crucial difference of the viewpoint sequences from traditional itemset sequences

Table 8 – Examples of some possible viewpoints

Viewpoint	Description	Range set
<i>pitch</i>	pitch of event as MIDI number	$\{0, \dots, 127\}$
<i>spell</i>	pitch class name	$\{A, A\#, Bb, \dots, G, G\#\}$
<i>dur</i>	duration of event (in sixteenth notes)	$\mathbb{R}_+^*$
<i>onset</i>	time of onset of event (in sixteenth notes)	$\mathbb{R}_+^*$
<i>contour(pitch)</i>	direction of melodic interval from the last event to the current	$\{\perp, =, +, -\}$
<i>contour(dur)</i>	direction of duration ratio between the last and the current events	$\{\perp, =, +, -\}$
<i>rest</i>	event is preceded by a rest	$\{0, 1\}$
<i>pcint</i>	pitch class melodic interval (independent of melodic direction)	$\{\perp, \mathbb{Z}_+\}$
<i>diaintc</i>	diatonic melodic interval (independent of melodic direction)	$\{P1, m2, M2, m3, M3, \dots\}$

Source: Adapted from (CONKLIN; BERGERON, 2008) and (CONKLIN, 2010a)

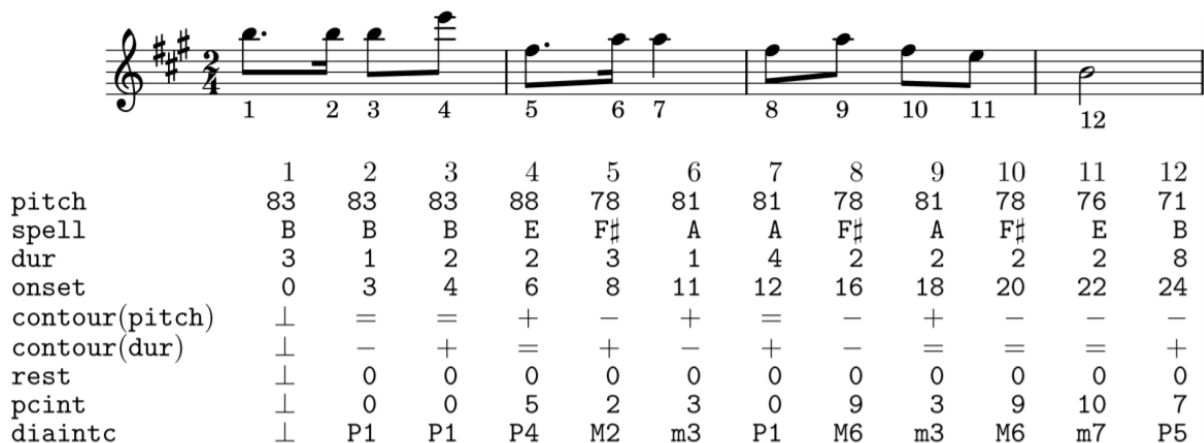


Figure 12 – Viewpoint representation of a music piece excerpt.  $\perp$  represents an undefined value.

Source: (CONKLIN, 2010a)

is that the latter have only boolean attributes, indicating only that a given item is present or not in an itemset; whereas the former might contain other data types for the value of the viewpoint (e.g., integers to represent the pitch of a note). One way to deal with this difference is to consider each key-value pair as a different item during i-extensions and s-extensions, but not allowing more than one of the same viewpoint in the same unordered set.

### 3.2.4 MGDP sequences

To understand what are the requirements for a sequence to be considered a Maximally General Distinctive Pattern (MGDP) according to Conklin (2010a), first we need to introduce the concept of the anticorpus. An anticorpus is a subset from the dataset of musical pieces that are different in some way to the corpus being analyzed. As an example, if we want to analyze a corpus consisting of opera Arias that are meant to convey the emotion sadness, then the anticorpus might consist in other opera Arias that are meant to convey any other emotions that are not sadness. If there are more than two emotions present in the dataset, and all emotions of the dataset should be analyzed, then an approach similar to a one-vs-all approach for multi-class classification can be followed, where each emotion is selected as the corpus and the rest as the anticorpus in each iteration of the analysis, until all emotions are analyzed. Note that the use of an anticorpus implies that the dataset should be labeled.

A pattern  $s_j$  is considered to be distinctive if it is found significantly more frequently in the corpus than in the anticorpus (CONKLIN, 2010a). Figure 13 illustrates this definition of distinctive patterns. If we consider that Label 1 is the corpus, then Label 2 and Label 3 are the anticorpus. The shaded (gray) area represents the patterns that are specific to Label 1, and only to Label 1 - so the gray shaded area represents the distinctive patterns for Label 1.

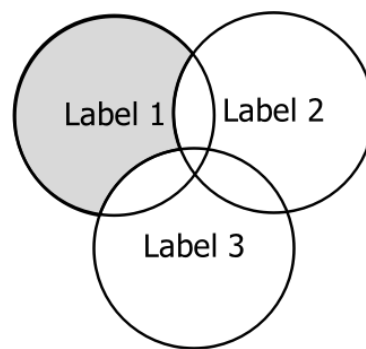


Figure 13 – Depiction of the definition of distinctive patterns.

A measure for the Raw Distinctiveness  $\Delta$  of a pattern can be calculated as

$$\Delta(s_j, SDB^\oplus) \equiv \frac{relSup(s_j, SDB^\oplus)}{relSup(s_j, SDB^\ominus)} \quad (4)$$

where  $relSup$  is defined by Equation 2,  $SDB^\oplus$  is the corpus being analyzed, and  $SDB^\ominus$  is the anticorpus of  $SDB^\oplus$ . Then, a threshold  $\theta$  might be empirically defined. If  $\Delta(s_j, SDB^\oplus) > \theta | \theta > 1$ , then the pattern  $s_j$  is considered a distinctive pattern in the corpus  $SDB^\oplus$ . If  $relSup(s_j, SDB^\ominus) = 0$ , then  $\Delta(s_j, SDB^\oplus) = \infty$  (CONKLIN, 2010b).

Another way to verify if a pattern is distinctive is by performing a Fisher's Exact Test on the contingency table formed by the presence of the pattern in the corpus and in

the anticorpus (CONKLIN, 2013). For this task, a maximum p-value is set. Then, if the result for the Fisher’s Exact Test results in a p-value less than the p-value threshold, the pattern can be considered distinctive.

A pattern  $s_j$  is said to be more general than (subsume) a pattern  $s_k$  if  $s_j \sqsubset s_k$ . Didactically, we can assume that the more general pattern  $s_j$  has fewer elements than the more specific pattern  $s_k$ , given that  $s_j \sqsubset s_k$ . In other words,  $s_j$  is more general than  $s_k$  if  $s_j$  is a subsequence of  $s_k$ .

A pattern  $s_j$  is a Maximally General Distinctive Pattern (MGDP) if  $s_j$  is distinctive, and if there are no other distinctive patterns  $s_k$  that are more general than  $s_j$ . Figure 14 depicts a Sequence Tree with MGDPs highlighted. The MGDPs can be seen as the distinctive patterns that are closer to the tree’s root node.

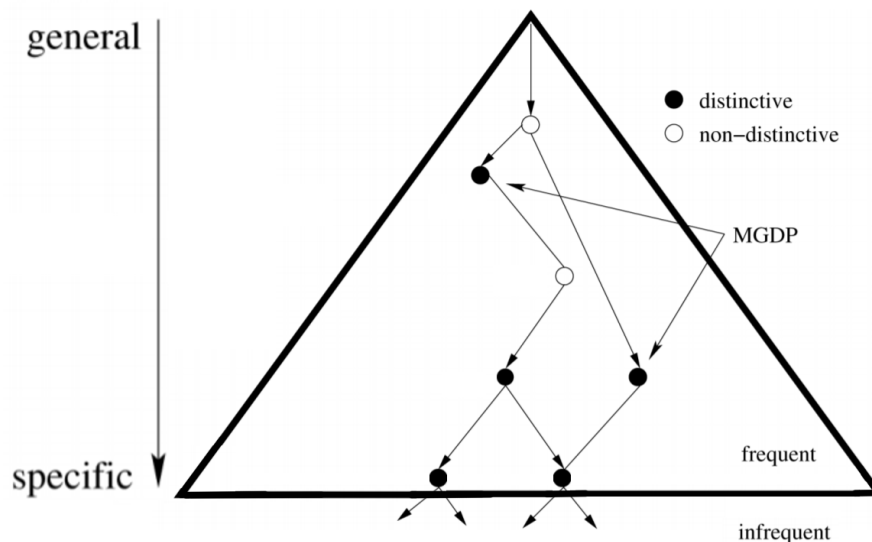


Figure 14 – Depiction of MGDPs in a Sequence Tree

Source: Adapted from (CONKLIN, 2010a)

### 3.2.5 MGDP Algorithm

The MGDP algorithm for inter-opus musical pattern discovery (CONKLIN, 2010a) involves 5 main steps. The overall algorithm is depicted as a simplified block diagram in Figure 15.

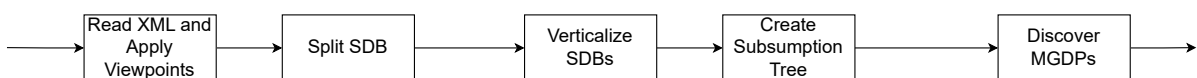


Figure 15 – Overall MGDP algorithm as a simplified block diagram



### 3.2.5.1 Read MusicXML and Apply Viewpoints

The first step involves reading the MusicXML files and applying the defined viewpoints through saturation, as described in [Section 3.2.3](#). This step outputs the Sequential Database to be used in the following steps. It is possible to use the music21 package ([Section 3.1.1](#)) to read the MusicXML files and implement the viewpoints. This step needs to be done only once, and can be done in parallel for each MusicXML file. [Figure 16](#) depicts the inputs and output of this step.

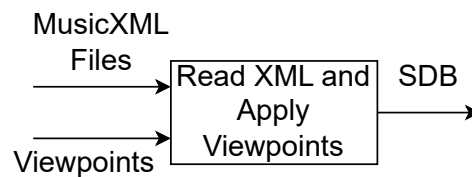


Figure 16 – First step of the MGD algorithm

### 3.2.5.2 Split SDB

The second step involves splitting the SDB into a corpus and an anticorpus, following the one-vs-all method described in [Section 3.2.4](#). This step and the subsequent ones are done once for each label that we want to find the MGDs for. [Figure 17](#) illustrates the inputs and outputs of this step.

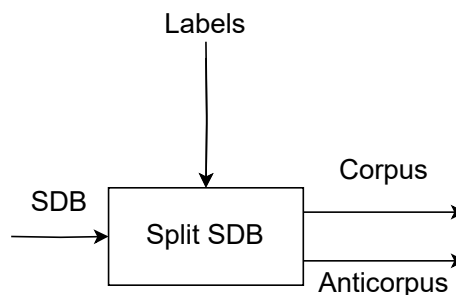


Figure 17 – Second step of the MGD algorithm

### 3.2.5.3 Verticalize SDB

The third step of the MGD algorithm involves verticalizing the SDB as mentioned in [Section 2.5.1.1](#). This step is a requirement of the last step, that involves using the tree search algorithm based on the SPAM method ([AYRES et al., 2002](#)) to actually discover the MGDs. It is necessary to perform the verticalization of both the corpus and the anticorpus, outputting a vertical corpus and a vertical anticorpus, as depicted in [Figure 18](#).

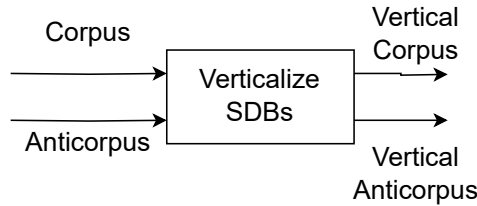


Figure 18 – Third step of the MGDP algorithm

#### 3.2.5.4 Create Subsumption Tree

The fourth step regards the creation of the Subsumption Tree. Figure 19 depicts the inputs and output of this fourth step of the MGDP algorithm. Conklin and Bergeron (2008) reports that a description logic classification algorithm is used to generate a Subsumption Tree from the featuresets that were found to be frequent in the *SDB*, using a method from Brachman and Levesque (2004), which we assume is the algorithm reported in Section 9.5.2 of Brachman and Levesque (2004). Next, we will give details of our interpretation on how this algorithm was used.

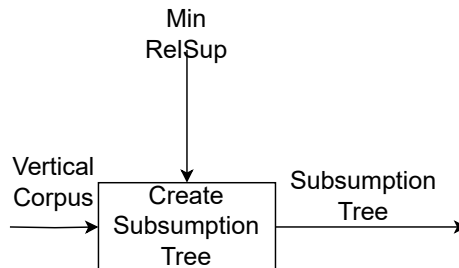


Figure 19 – Fourth step of the MGDP algorithm

A Subsumption Graph (a.k.a. Subsumption Network, Subsumption Taxonomy) might be defined as a Directed Acyclic Graph (DAG) with a single source node consisting in an empty set, where each node  $s_k$  is connected to the nodes  $s_j$ , given that  $s_j \sqsubset s_k$ . It is possible to obtain a Subsumption Graph of the featuresets of a *SDB* by making one pass through the whole *SDB*, obtaining all the unique featuresets present in it. These unique featuresets will be the leaf-nodes of our graph. Then, for each node, we recursively remove one feature contained in the featureset and make a new node with this reduced featureset, making this new, more general node a parent node of the more specific node. We repeat this process until we reach the empty featureset, and do this for all features of all nodes. Figure 20 depicts an example of a Subsumption Graph of featuresets. In this example, we would start from the unique featuresets  $\{a, c\}$ ,  $\{a, b\}$ , and  $\{b, c\}$  as leaf-nodes, then we would apply the algorithm to get the nodes  $\{a\}$ ,  $\{b\}$ , and  $\{c\}$ , and the empty featureset.

Conklin and Bergeron (2008) prunes the nodes from the Subsumption Graph that are infrequent. For example, if it is found that the node that contains the featureset  $\{a, b\}$

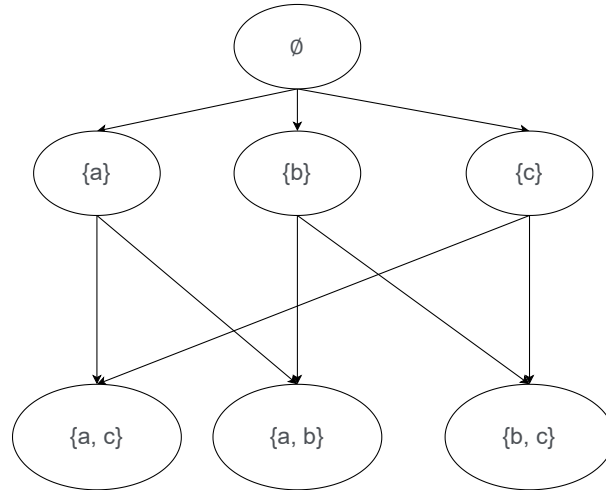


Figure 20 – Example of a Subsumption Graph of featuresets

Source: Adapted from (CONKLIN; BERGERON, 2008)

from Figure 20 is infrequent in the corpus (i.e., has a support lower than the minimum support threshold  $minSup$ ), then we can prune that node from the Subsumption Graph. And, because of the Apriori principle (AGRAWAL; SRIKANT, 1994), we can prune all of its children from the Subsumption Graph as well, resulting in the Subsumption Graph depicted in Figure 21.

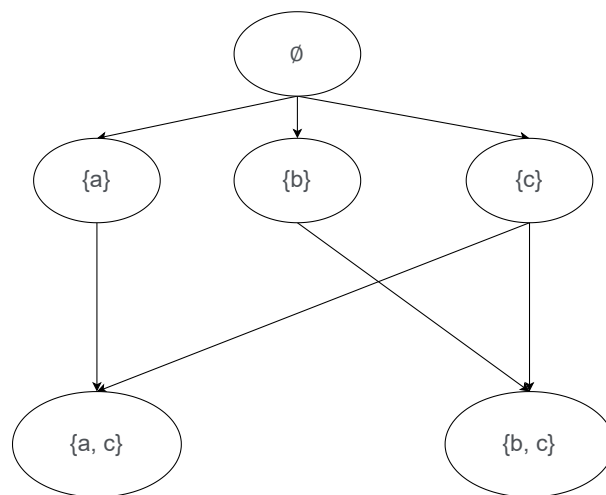


Figure 21 – Subsumption Graph from Figure 20 after pruning the infrequent nodes

A Subsumption Graph is different from a Subsumption Tree in the sense that the former might contain multiple connections to the same, more specific (child) node, while the latter will contain only child nodes with one parent node. One can obtain a Subsumption Tree from a Subsumption Graph by pruning the redundant connections, making sure that the in-degree of every node in the directed graph is not greater than 1. For example, in Figure 21, the node  $\{a, c\}$  has an in-degree of 2 (there are two nodes that

are parents of  $\{a, c\}$ : the node  $\{a\}$  and the node  $\{c\}$ ). The same is true for the node  $\{b, c\}$ . We want to prune the redundant edges that lead to these nodes until there is only one edge connecting to them. Note that there are multiple possible results from this conversion, as depicted in Figure 22. Also, note that in Figure 22c, we would have the node  $\{c\}$  with an out-degree equal to 2, but this is not a problem, since there are no nodes with an in-degree greater than 1.

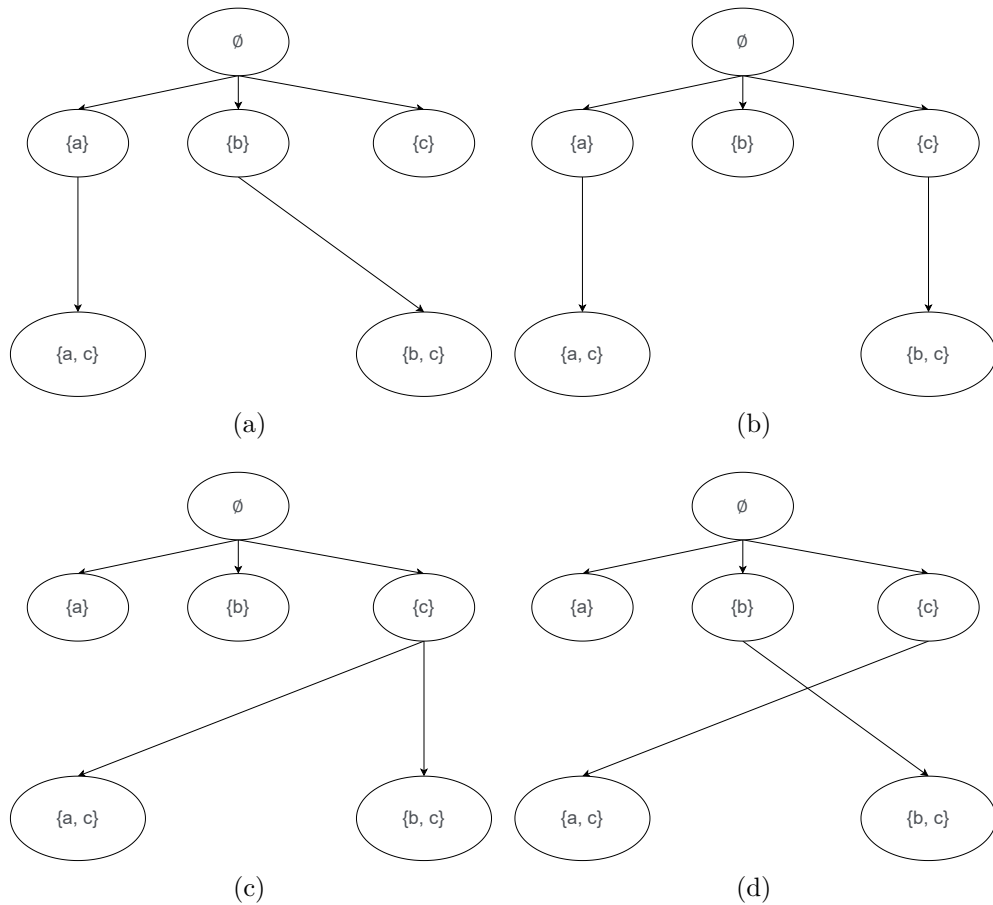


Figure 22 – Examples of possible results from converting the Subsumption Graph of Figure 21 to a Subsumption Tree

### 3.2.5.5 MGD<sub>P</sub> Discovery

The fifth and last step of the MGD<sub>P</sub> algorithm involves mining/searching for the MGD<sub>P</sub>s. For this, an algorithm based on the Sequential PAttern Mining (SPAM) (AYRES et al., 2002) is used (see Section 2.5.1). A tree is generated through i-extensions and s-extensions of feature sets in Viewpoint representation, through a depth-first search. Figure 23 presents the inputs and outputs of the MGD<sub>P</sub> Discovery step. Next, we will give details on the differences between the SPAM algorithm and the MGD<sub>P</sub> Discovery algorithm.

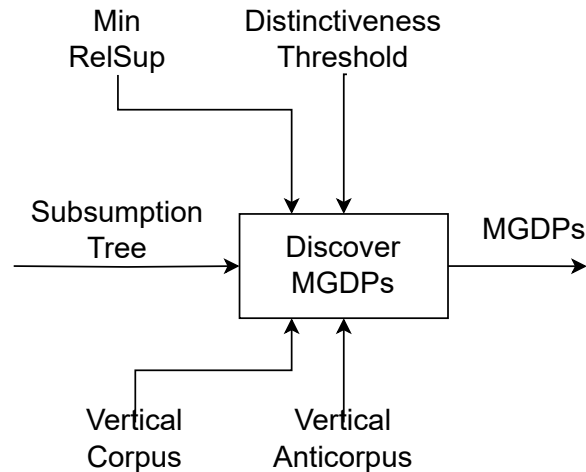


Figure 23 – Fifth step of the MGDP algorithm

The first difference is that [Conklin and Bergeron \(2008\)](#) discard the Lexicographical constraint of the Sequence Tree when expanding the search space. Giving more context, a Subsumption Tree is different from a Sequence Tree (as in a Lexicographical Sequence Tree) in the sense that the former’s nodes contains only (unordered) featuresets; while the latter’s nodes contains (ordered) sequences of featuresets (compare [Figure 22a](#) with [Figure 11](#), for example).

The Subsumption Tree is used in the MGDP Discovery step to populate the set  $I_j$  of all possible i-extensions of the node  $s_j$ , for all nodes  $s_j$  in the Sequence Tree. With this, there is no need to use a Lexicographical constraint, since the Subsumption Tree already guarantees that there won’t be any types of improper i-extensions such as multiple features with the same viewpoint in one featureset. Also, since there are no nodes with an in-degree greater than 1 in the Subsumption Tree, we can safely assume that no pattern will be visited more than once, avoiding unnecessary calculations during the MGDP Discovery step. Finally, by using the Subsumption Tree as reference for the i-extensions, since it is populated based on only the existing unique featuresets of the SDB, the search space does not contain any impossible combinations of features.

While [Ayres et al. \(2002\)](#) uses a Bitmap data structure to perform the i-extensions and the s-extensions operations, [Conklin and Bergeron \(2008\)](#) mention that Instance Maps are used, and [Conklin \(2010a\)](#) mention that Instance Lists are used. We presume that both Instance Maps and Instance Lists refer to the same structure as the Bitmaps that are used by [Ayres et al. \(2002\)](#), since no more details are given besides that these data structures they are implemented as associative arrays ([CONKLIN; BERGERON, 2008](#)) or hash maps ([CONKLIN, 2010a](#)) that maps each pattern and song to the positions of occurrence of the patterns using boolean values. This means that, just as the Bitmaps from [Ayres et al. \(2002\)](#), only the positions of the end of the sequential patterns are stored, and not the positions of the individual events that compose the pattern.

Another crucial difference - one that brings a major limitation to the original MGDP algorithm - is that Conklin and Bergeron (2008) performs all s-extensions by appending an empty featureset to the end of the pattern. As an example, supposing we have a subsequence in viewpoint representation  $s_j = \langle \{pcint : 3\} \rangle$ , an i-extension of  $s_j$  could be  $s_j^a = \langle \{pcint : 3, dur : 1\} \rangle$ , while an s-extension of  $s_j$  would be  $s_j^b = \langle \{pcint : 3\}, \{\} \rangle$ . The empty featureset might be present in the middle of the final MGDPs discovered and outputted by the algorithm. When this occurs, it means that any event (i.e., note, chord, or rest) would fit in that position of the pattern. But it also means that there must be an event in that position of the pattern - while the original SPAM algorithm (AYRES et al., 2002) is capable of finding patterns that are spaced by any number events between them. Multiple empty featuresets could also be chained together, but it is also possible to avoid this by hard-limiting the number of empty featuresets that might be present in the patterns (CONKLIN, 2010a). One might also prohibit any empty featureset as a component of the patterns (CONKLIN, 2010a). To perform the s-extension operation on the bitmaps, Conklin and Bergeron (2008) apply an offset to the bitmap of the pattern.

One more difference of the MGDP Discovery from the original SPAM algorithm is the measure of distinctiveness (CONKLIN, 2010a). After calculating the support of a pattern in the corpus, if it is frequent it is also necessary to calculate the support of the pattern in the anticorpus, to then calculate the distinctiveness of the pattern (using either the Equation 4, or the Fisher's Exact Test as mentioned in Section 3.2.4). Since only MGDPs are sought, the search in a branch of the tree can be stopped as soon as a distinctive pattern is found in that branch, since the first distinctive pattern found will be also the most general pattern in the branch, as stated in Section 3.2.4, making it, by definition, a MGDP.

## 4 Proposed Algorithms

### 4.1 MGDP Algorithms Implementations

Since there were no publicly available implementations of the original MGDP algorithm used in (CONKLIN; BERGERON, 2008), (CONKLIN, 2010a), and (CONKLIN, 2013), we implemented 4 different algorithms from scratch and compared their performances. In our implementations we modified the original MGDP algorithm slightly, to address two opportunities of improvement:

1. If we want to highlight each note that composes the pattern in a song, we would have to use a sequential search algorithm, after discovering the pattern.
2. If one song shares a pattern with other songs, but it has an additional note in-between the pattern, then the original MGDP discovery algorithm would fail to recognize it on this song. The original MGDP algorithm is able to find patterns with slight variations using the “empty featureset” strategy, but even so, it only recognizes patterns in sequences with same length. As an example, the original MGDP algorithm would not recognize the pattern  $\hat{5}-\hat{1}-\hat{5}-\hat{3}$  in the second half of the SMW Swimming excerpt (Figure 2b from Section 2.2), since there is an additional P1 in the middle of the pattern.

To address the first opportunity of improvement, we designed a modification in the base data structure of the algorithm. In 2 out of these 4 algorithms, we used the same data structure as in (CONKLIN; BERGERON, 2008) and in (CONKLIN, 2010a), which we believe is the same data structure that Ayres et al. (2002) used - the Bitmap, which is described in more details in Section 2.5.1. In the other two algorithms, we used a data structure that, to our best knowledge, was not previously reported as being used for the Sequential Pattern Mining task. We call this data structure an **InstanceMap** (not to be confused with the Instance Map from (CONKLIN; BERGERON, 2008), nor an Instance List from (CONKLIN, 2010a). As mentioned in Section 3.2.5.5, we consider that both Instance Maps and Instance Lists are the same as Bitmaps). In summary, instead of storing whether a pattern occurs or not in a given position as a boolean, we store all the positions in which each of the featuresets in the pattern occur, as a list of integers. We give more details about the InstanceMap in Section 4.1.3.1.

To address the second opportunity of improvement, all 4 of our implementations have a different tree traversal algorithm from the original MGDP discovery algorithm. Instead of appending an empty featureset in every s-extension, we directly append a

featureset with one feature (a viewpoint and its value) to the end of the sequence. The set  $\Sigma$  of possible features to append to the end of the sequence is a constant given by the successors of the root node of the Subsumption Graph. With this, we can use the very same s-extension and i-extension algorithms as described in (AYRES et al., 2002). This means that our algorithm will be able to consider that sequences might have any number of events/featuresets in-between them, but still belonging to the same pattern, given that the sequence fits all the pattern’s constraints. We implemented this method with the Bitmap data structure, and applied modifications to work with the InstanceMap data structure as well.

We also designed and implemented two alternate s-extension algorithms (one for the Bitmaps and one for the InstanceMaps) that limits the maximum number of events that might be present in-between the pattern. Differently from the original MGDP algorithm, our alternate s-extension algorithms can take into consideration that sequences with different lengths might still share the same pattern, even when limiting the number of events in-between the pattern.

Table 9 summarizes the differences between the 4 algorithms that we used. All these algorithms share a lot of functionality. Thus, to avoid code duplication, we followed Software Engineering best practices, using a combination of the Strategy Design Pattern and the Factory Design Pattern (GAMMA et al., 1996). We detail these 4 algorithms in the following sub-sections. Apart from the differences that will be pointed in the next sub-sections, the algorithm follows what is described in Section 3.2.5.

Table 9 – MGDP discovery algorithm combination of strategies

MGDP Discovery Algorithm Name	Data Structure	s-extension strategy
MGDP_Bitmap()	Bitmap	Any number of events in-between the pattern
MGDP_Bitmap_limitMaxJumpSize(maxJumpSize)	Bitmap	Limit maximum number of events in-between the pattern
MGDP_InstanceMap()	InstanceMap	Any number of events in-between the pattern
MGDP_InstanceMap_limitMaxJumpSize(maxJumpSize)	InstanceMap	Limit maximum number of events in-between the pattern

#### 4.1.1 MGDP\_Bitmap()

This algorithm uses the Bitmap data structure and the i-extension and s-extension methods exactly as described in (AYRES et al., 2002). This means that we do not follow the s-extension strategy as done by Conklin and Bergeron (2008) and described in Section 3.2.5.5 (i.e., append one empty featureset to the pattern during the s-extension, and shift the bitmap to the right, to reflect the new end of the pattern). We perform the bitwise-AND operation between the auxiliary bitmap of the pattern and the bitmap of



the feature being appended to the end of the pattern. For more details, see [Section 2.5.1.1](#) or ([AYRES et al., 2002](#)).

By following this s-extension strategy, we address the second opportunity of improvement mentioned in [Section 4.1](#). This implementation of the `MGDP_Bitmap()` algorithm is capable of considering that sequences of different lengths might belong to the same pattern, given that the sequences fills the requirements of the pattern, independently of the number of events that might be between the events that compose the featuresets of the pattern.

As an example, one MGDP that this algorithm was able to find is the pattern consisting of diatonic intervals without quality  $\hat{7}-\hat{1}-\hat{2}-\hat{6}-\hat{8}$  for the genre Femmes. This pattern was detected in more than 84% of the songs of the genre, and was found to occur 26 times more often in the Femmes genre than in the combination of all other genres (following [Equation 4](#)). We manually annotated this pattern for two songs in which this pattern occurs. The annotated score is presented in [Figure 24](#). Note the different number of events (notes, rests, or chords) in-between the  $\hat{2}$  and the  $\hat{6}$  in both songs - in [Figure 24a](#) we have 3 notes, while in [Figure 24b](#) we have only 2 notes. Also note that the first note of the pattern is very far from the second note of the pattern, in both cases.

The `MGDP_Bitmap()` algorithm does not address the first opportunity of improvement, since its base data structure is the Bitmap (see [Section 2.5.1.1](#)). This means that to enable the pattern visualization of the example in [Figure 24](#), we actually had to choose two scores in which the pattern occurred, then manually scan through these two scores, looking for each event of the pattern, and manually highlighting each one of the relevant events. This had to be done after the algorithm finished running and outputted the found MGDPs and the songs in which they occur.

#### 4.1.2 `MGDP_Bitmap_limitMaxJumpSize(maxJumpSize)`

There might be cases when having big jumps between the occurrences of events of a pattern is undesirable. For example, in [Figure 24](#), we have a jump with the size of 10 events between the occurrence of the event  $\hat{7}$  and the next occurrence of the event  $\hat{1}$ , in both songs. Some musicologists might find this pattern uninteresting because of this big space between the notes of the pattern, arguing that motifs and themes should be made of events that are close to each other. To address this need, we modify the Bitmap s-extension algorithm slightly, to set a limit of the number of events that might appear in-between the occurrences events of the pattern (i.e., limit the *maxJumpSize* of a pattern).

When setting *maxJumpSize* = 0, the algorithm will only look for occurrences of the pattern with continuous events - the pattern must be identical in the pieces. When setting *maxJumpSize* = 1, the algorithm considers that there might be 0 or 1 events

## NLB070526\_01

Ko - lijn een bra - ve boe - ren zoon Het puik - je van de dor - pe -  
 lin - gen Ar - beid - zaam wel - ge - maakt en schoon Dorst naar Li - set - tes hand te  
 din - gen Dorst naar Li - set - tes hand te din - gen

(a)

## NLB075906\_01

Mijn vrien - den, staat een wei - nig stil Hoort wat de cou - ran - ten ons ver -  
 mel - den. Nie - mand op aar - de naar Gods wil  
 Hier en hier - na - maals zal doen gel - den. Hier en hier - na - maals zal doen  
 gel - den.

(b)

Figure 24 – Pattern  $\hat{7}\text{-}\hat{1}\text{-}\hat{2}\text{-}\hat{6}\text{-}\hat{8}$ , found by the `MGDP_Bitmap()` algorithm, manually annotated in the two scores. Green notes indicate the first event of the pattern, red notes indicate the last event of the pattern, and blue notes indicate the rest of the occurrences of the events of the pattern.

in-between any occurrence of the events of a pattern. Figure 25 demonstrates a pattern found with  $maxJumpSize = 1$ . Note that both excerpts start from measure number 7. Also, note that on Figure 25a, the pattern has a jump between the event with  $\hat{5}$  and  $\hat{6}$ , while on Figure 25b, there is no such jump. However, our algorithm considers that both pieces share the same pattern - which was not the case with the original MGDP algorithm (CONKLIN; BERGERON, 2008; CONKLIN, 2010a; CONKLIN, 2013).

## NLB070526\_01

Figure 25a shows a musical score excerpt in 4/4 time, starting at measure 7. The melody consists of quarter notes: G4, A4, B4, C5, B4, A4, G4. The notes are annotated with numbers 2 through 8. Note 2 (G4) is green, note 3 (A4) is blue, note 5 (B4) is blue, note 6 (C5) is blue, and note 8 (G4) is red. The lyrics are: "Dorst naar Li - set - tes hand te din - gen Dorst naar Li - set - tes hand te".

## NLB075906\_01

Figure 25b shows a musical score excerpt in 4/4 time, starting at measure 7. The melody consists of quarter notes: G4, A4, B4, C5, B4, A4, G4. The notes are annotated with numbers 2 through 8. Note 2 (G4) is green, note 3 (A4) is blue, note 5 (B4) is blue, note 6 (C5) is blue, and note 8 (G4) is red. The lyrics are: "Hier en hier - na - maals zal doen gel - den. Hier en hier - na - maals zal doen".

Figure 25 – Pattern found by the `MGDP_Bitmap_limitMaxJumpSize(maxJumpSize)` algorithm with  $maxJumpSize = 1$ , manually annotated in the two scores' excerpts. Depicts the pattern  $\hat{2}\text{-}\hat{3}\text{-}\hat{5}\text{-}\hat{6}\text{-}\hat{8}$ , Green notes indicate the first event of the pattern, red notes indicate the last event of the pattern, and blue notes indicate the rest of the occurrences of the events of the pattern.

### 4.1.2.1 Bitmap s-extension with `maxJumpSize` - the Bitmatrix

We designed a novel algorithm capable of limiting the jump size during the s-extension, while using Bitmaps, and still retaining the property to address opportunity of improvement number 2. This algorithm uses what we called a Bitmatrix, which is a stack of Bitmaps, combined row-wise (i.e., each row of the Bitmatrix corresponds to one Bitmap). All Bitmaps in a Bitmatrix derive from the same root Bitmap, but each of them is shifted to the right by  $i + 1$  bits, being  $i$  the zero-based row index of the corresponding Bitmap in the Bitmatrix. The number of rows in the Bitmatrix is equal to  $maxJumpSize + 1$ . For example, suppose we have a root Bitmap  $[0, 1, 0, 0, 1]$ , and that we have  $maxJumpSize = 1$ ; the corresponding Bitmatrix would be  $\begin{bmatrix} 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix}$ .

The s-extension algorithm is very similar to the one used originally by (AYRES et al., 2002) (described in Section 2.5.1.1). The key difference is in how we derive the auxiliary bitmap. The Bitmatrix corresponding to the existing pattern is converted into our new auxiliary bitmap. To do so, we perform a row-by-row bitwise-OR operation with all rows of the Bitmatrix. In our example, the resulting auxiliary bitmap would be  $[0, 0, 1, 1, 0]$ . Finally, we perform a bitwise-AND operation of the resulting auxiliary bitmap with the feature bitmap that is being appended through the s-extension, just like the original SPAM algorithm does.

### 4.1.3 MGDP\_InstanceMap()

Both of the implementations using the Bitmap data structure fail to address the first opportunity of improvement mentioned in Section 4.1. We designed a new data structure that can be used as a replacement for the Bitmap that solves the first opportunity of improvement. We call this new data structure an InstanceMap. As the name suggests, the MGDP\_InstanceMap() algorithm is based on the InstanceMap, which is described next.

#### 4.1.3.1 The InstanceMap

We define an InstanceMap as a **list of lists** of integers. E.g.,  $[[0, 3, 5], [2, 3, 5], [6, 8, 12]]$  could be an InstanceMap. One InstanceMap refers to all the occurrences of a sequential pattern in a sequence. Each inner list refers to an occurrence of the pattern. And each integer in the inner lists refers to an event of the pattern. The integers map the position of the events related to the pattern.

As an example, suppose we have a sequence  $S = \langle \{a, b\}, \{c\}, \{f, g\}, \{g\}, \{d\} \rangle$  and we want to make an InstanceMap of the pattern  $s_1 = \langle \{c\}, \{g\}, \{d\} \rangle$ . If we use zero-based indexing, the featureset  $\{c\}$  would be present in the position 1 of  $S$ ,  $\{g\}$  would be present both in positions 2 and 3, and  $\{d\}$  would be present in the position 4. This means that we would have two occurrences of the pattern, and the InstanceMap of the pattern  $s_1$  would be  $[[1, 2, 4], [1, 3, 4]]$ . The featureset  $\{g\}$  appears twice in the sequence, and both times it appears after the preceding featuresets of the pattern ( $\{c\}$ ), so we have two inner lists in our InstanceMap - each one corresponding to one occurrence of the pattern.

Note that the number of elements in the inner lists of the InstanceMap correspond to the number of featuresets in the pattern. E.g., if  $s_2 = \langle \{a, b\}, \{f\} \rangle$ , then the InstanceMap of pattern  $s_2$  would be  $[[0, 2]]$ . Lastly, if a pattern does not occur in the sequence, then the InstanceMap corresponds to an empty list. E.g., the InstanceMap of pattern  $s_3 = \langle \{g\}, \{a\} \rangle$  would be  $[\ ]$ .

### 4.1.3.2 InstanceMap i-extension

Since we are now using InstanceMaps instead of Bitmaps, we need a different strategy to perform the i-extension operation. Recall that the i-extension objective is to explore the occurrence of a new pattern, by appending a feature to the last featureset of an already explored pattern. For example, if we want to apply an i-extension to the pattern  $s_1 = \langle \{a, b\}, \{c\} \rangle$  with the feature  $f_1 = a$ , then the result would be the pattern  $s_2 = \langle \{a, b\}, \{c, a\} \rangle$ . With the InstanceMap of the feature  $f_1$  and the InstanceMap of the already visited pattern  $s_1$ , we should be able to infer a new InstanceMap for the new pattern  $s_2$  without the need of scanning through the whole Sequence again.

The i-extension with InstanceMaps algorithm's pseudocode is presented in [Algorithm 1](#). In summary, our algorithm to perform an i-extension with InstanceMaps works as follows: given a list of positions of occurrences of a feature, check if any of these positions coincides with the position of occurrence of the last featureset of the pattern. We make this check for every occurrence of the pattern, and for every occurrence of the feature. An implementation<sup>1</sup> of the algorithm is presented in [Code 1](#). Note that the input parameter `flattened_feature_instanceMap` is actually a list of integers, instead of being a list of lists of integers like the `pattern_instanceMap`. The InstanceMap of a feature will always contain inner lists with only one integer, (e.g.,  $[[0], [2], [6], [8]]$ ). Thus, to facilitate the computation of the algorithm, we can first flatten the feature InstanceMap to be just a list of integers (e.g.,  $[0, 2, 6, 8]$ ).

---

#### Algorithm 1 The i-extension algorithm for InstanceMaps

---

**Input:** `pattern_instanceMap` ▷ List of list of integers  
**Input:** `flattened_feature_instanceMap` ▷ List of integers  
**Output:** `new_pattern_instanceMap` ▷ List of list of integers

```

function IEXTENSION(pattern_instanceMap, flattened_feature_instanceMap)
  new_pattern_instanceMap ← NEWLIST()
  for each pattern_occurrence in pattern_instanceMap do
    if Last item of pattern_occurrence exists in flattened_feature_instanceMap then
      ADDITEM(new_pattern_instanceMap, pattern_occurrence)
    end if
  end for

  return new_pattern_instanceMap
end function

```

---

As an example, suppose we have a sequence  $S = \langle \{a, b\}, \{c\}, \{f, g\}, \{g\}, \{d\} \rangle$ , an already visited pattern  $s_0 = \langle \{c\}, \{g\} \rangle$ , and we want to perform an i-extension of  $s_0$  with the feature  $f$ . The resulting pattern will be  $s_1 = \langle \{c\}, \{g, f\} \rangle$ . Now we need to compute the new pattern InstanceMap. We have `pattern_instanceMap` =  $[[1, 2], [1, 3]]$  and

<sup>1</sup> Our real implementation actually uses Python-specific concepts (such as List Comprehensions and frozensets), and the function is actually implemented as a class method. However, to ease the understanding of the algorithm, we present a simplified implementation

Code 1 – Python implementation of the i-extension operation with InstanceMaps

```

1 def get_i_extension_instanceMap(
2     pattern_instanceMap: list[list[int]],
3     flattened_feature_instanceMap: list[int]
4 ) -> list[list[int]]:
5
6     new_pattern_instanceMap = []
7     for pattern_occurrence in pattern_instanceMap:
8         position_of_last_featureset_of_occurrence = pattern_occurrence[-1]
9         if position_of_last_featureset_of_occurrence in flattened_feature_instanceMap:
10             new_pattern_instanceMap.append(pattern_occurrences)
11
12     return new_pattern_instanceMap

```

`flattened_feature_instanceMap = [2]`. For each inner list of integers in `pattern_instanceMap` we check if its last element is present in the list given by `flattened_feature_instanceMap`. The last element in the first inner list  $[1, 2]$  is 2, and is present in `flattened_feature_instanceMap`, so we add the inner list to the `new_pattern_instanceMap`. For the second inner list  $[1, 3]$ , its last element (3) is not present in `flattened_feature_instanceMap`, so this inner list is not appended to the new pattern InstanceMap. As a result, the new pattern InstanceMap will be `new_pattern_instanceMap = [[1, 2]]`. Observe that the result is consistent with what was expected: the InstanceMap of the new pattern  $s_1 = \langle \{c\}, \{g, f\} \rangle$  indicates that the featureset  $\{c\}$  occurs in position 1 of  $S$ , and that the featureset  $\{g, f\}$  occurs in position 2 of  $S$  (recall that a featureset is unordered). Each inner list corresponds to one pattern occurrence, so we also know there is one and only one occurrence of the sequential pattern  $s_1$  in  $S$ .

#### 4.1.3.3 InstanceMap s-extension

To perform a s-extension with InstanceMaps we need to compare the positions of occurrence of the feature with the position of the last featureset of the pattern occurrences. If the position of occurrence of the feature is greater than the position of the last featureset of the pattern occurrence, we append that position of the feature to the inner list corresponding to the pattern occurrence. For example, suppose we have a sequence  $S = \langle \{a, b\}, \{c\}, \{f, g\}, \{g\}, \{d\} \rangle$ , an already visited pattern  $s_0 = \langle \{c\}, \{g\} \rangle$ , and we want to perform a s-extension of  $s_0$  with the feature  $d$ . Our new pattern will be  $s_0 = \langle \{c\}, \{g\}, \{d\} \rangle$ . We have `pattern_instanceMap = [[1, 2], [1, 3]]` and `flattened_feature_instanceMap = [4]`. The position of the last featureset of the first pattern occurrence is 2, and 4 is bigger than 2, so we append 4 to the inner list  $[1, 2]$ , resulting in the list  $[1, 2, 4]$ . The same is true for the second pattern occurrence, resulting in the list  $[1, 3, 4]$ . These two new inner lists compose the resulting InstanceMap of the new pattern: `new_pattern_instanceMap = [[1, 2, 4], [1, 3, 4]]`.

[Algorithm 2](#) presents the pseudocode for the s-extension with InstanceMaps, and

Code 2 – Python implementation of the s-extension operation with InstanceMaps

```

1  def get_s_extension_instanceMap(
2      pattern_instanceMap: list[list[int]],
3      flattened_feature_instanceMap: list[int]
4  ) -> list[list[int]]:
5      new_pattern_instanceMap = []
6
7      for position_of_feature_occurrence in flattened_feature_instanceMap:
8          for pattern_occurrence in pattern_instanceMap:
9              position_of_last_featureset_of_occurrence = pattern_occurrence[-1]
10             if (
11                 position_of_feature_occurrence >
12                 position_of_last_featureset_of_occurrence
13             ):
14                 new_pattern_occurrence = pattern_occurrence.copy()
15                 new_pattern_occurrence.append(position_of_feature_occurrence)
16
17                 new_pattern_instanceMap.append(new_pattern_occurrence)
18     return new_pattern_instanceMap

```

Code 2 presents one possible implementation for this algorithm. Observe that we apply a complete scan through the two lists - the pattern InstanceMap and the feature InstanceMap. This algorithm could probably be optimized by leveraging the fact that the InstanceMaps are semi-ordered. Nonetheless, for simplicity, we used the algorithm as it is presented.

---

### Algorithm 2 The s-extension algorithm for InstanceMaps

---

**Input:** *pattern\_instanceMap* ▷ List of list of integers  
**Input:** *flattened\_feature\_instanceMap* ▷ List of integers  
**Output:** *new\_pattern\_instanceMap* ▷ List of list of integers

```

function SEXTENSION(pattern_instanceMap, flattened_feature_instanceMap)
    new_pattern_instanceMap ← NEWLIST()
    for each position_of_feature_occurrence in flattened_feature_instanceMap do
        for each pattern_occurrence in pattern_instanceMap do
            position_of_last_featureset_of_occurrence ← Last item of pattern_occurrence
            if position_of_feature_occurrence is less than position_of_last_featureset_of_occurrence then
                new_pattern_occurrence ← position_of_feature_occurrence
                ADDITEM(new_pattern_occurrence, pattern_occurrence)
                ADDITEM(new_pattern_instanceMap, new_pattern_occurrence)
            end if
        end for
    end for

    return new_pattern_instanceMap
end function

```

---

#### 4.1.3.4 MGD<sub>P</sub>\_InstanceMap() output

The MGD<sub>P</sub>\_InstanceMap() algorithm outputs exactly the same MGD<sub>P</sub>s as the MGD<sub>P</sub>\_Bitmap() algorithm, as expected. However, as mentioned in the beginning of Section 4.1, the InstanceMap approach enables the MGD<sub>P</sub> algorithm to output the position of the events of the pattern, enabling us to automatically highlight the notes in the score, addressing the opportunity of improvement number 1.

We implemented the automatic generation of annotated MusicXML files with excerpts of the pieces of music in which the patterns occur, presenting not only what is the pattern itself, but also an example of this pattern, highlighting each event of the pattern and presenting the corresponding viewpoints and their values - without any human intervention. Figure 26 shows an example of a pattern found by the `MGDP_InstanceMap()` algorithm, which was automatically generated as a MusicXML file, and visualized in the MuseScore software. Note that, by design, the algorithm only outputs the measures relevant to the pattern occurrence, trimming the measures that appear before the beginning or after the end of the pattern occurrence. The note heads' colors and the textboxes with the viewpoints and their values are automatically generated and positioned. The textboxes' x-axis' centers are aligned with the end of the measure of the events that they correspond to.

## NLB072450\_01

toen het mon-ster werd ge-waar Dat ha-re doch-ter zou - de

diatonicIntervalNoQuality 4  
contour\_duration -

4  
trou - wen Ver - tel - de zij het aan haar

diatonicIntervalNoQuality 1  
contour\_duration +

contour\_duration =  
diatonicIntervalNoQuality 8

Figure 26 – Automatically annotated score output from the `MGDP_InstanceMap()` algorithm, visualized in MuseScore 4. Green notes indicate the first event of the pattern, red notes indicate the last event of the pattern, and blue notes indicate the rest of the occurrences of the events of the pattern.

### 4.1.4 `MGDP_InstanceMap_limitMaxJumpSize(maxJumpSize)`

Similar to the `MGDP_Bitmap_limitMaxJumpSize(maxJumpSize)` algorithm (Section 4.1.2), we implemented an `InstanceMap` variation to limit the number of events that might appear in-between the occurrences of events of a pattern. The MGDPs found with this new `MGDP_InstanceMap_limitMaxJumpSize(maxJumpSize)` algorithm are the same



as the ones found by the `M GDP_ Bitmap_ limitMaxJumpSize(maxJumpSize)` algorithm, but it has the advantage of addressing opportunity of improvement number 1 as well. Figure 27 presents an output of the algorithm, which corresponds to exactly the same M GDP depicted in Figure 25, but which was generated automatically by the algorithm, without any human intervention.

### NLB070526\_01

Dorst naar Li - set - tes hand te din - gen

diatonicIntervalNoQuality 2

diatonicIntervalNoQuality 3

diatonicIntervalNoQuality 5

3  
Dorst naar Li - set - tes hand te

diatonicIntervalNoQuality 6

diatonicIntervalNoQuality 8

Figure 27 – Output from the `M GDP_ InstanceMap_ limitMaxJumpSize(maxJumpSize)` algorithm, with `maxJumpSize=1`, visualized in MuseScore 3. Depicts the automatically annotated score. Each box corresponds to one event. Boxes are stacked up vertically from top down, corresponding to the order in which their associated events appear. Green notes indicate the first event of the pattern, red notes indicate the last event of the pattern, and blue notes indicate the rest of the occurrences of the events of the pattern.

To make this modification, we need to design a new s-extension algorithm using the InstanceMaps. This algorithm is presented in Algorithm 3. In this new algorithm, instead of just checking whether the position of occurrence of the feature is greater than the position of the last featureset of the pattern occurrence, we need to also check if the position of the feature is not greater than the maximum position stipulated by the `maxJumpSize`. Compare the if-statement in lines 10-13 of Code 2 with the if-statement in lines 11-15 of Code 3. In Code 3 we have added the check `position_of_feature_occurrence <= (position_of_last_featureset_of_occurrence+1+maxJumpSize)`. Notice that apart from the new parameter `maxJumpSize`, the algorithm is the same in both cases, apart from the if-statements.

As an example, suppose we have a sequence  $S = \langle \{a, b\}, \{c\}, \{f, g\}, \{g\}, \{d\} \rangle$ , an already visited pattern  $s_0 = \langle \{a, b\}, \{c\} \rangle$ , and we want to perform a s-extension on  $s_0$  adding the feature  $d$ . The resulting new pattern would be  $s_1 = \langle \{a, b\}, \{c\}, \{d\} \rangle$ . We have

**Algorithm 3** The s-extension algorithm for InstanceMaps, limiting the maxJumpSize

---

**Input:** *pattern\_instanceMap* ▷ List of list of integers  
**Input:** *flattened\_feature\_instanceMap* ▷ List of integers  
**Input:** *maxJumpSize* ▷ Integer  
**Output:** *new\_pattern\_instanceMap* ▷ List of list of integers

```

function SEXTENSION(pattern_instanceMap, flattened_feature_instanceMap, maxJumpSize)
  new_pattern_instanceMap ← NEWLIST()
  for each position_of_feature_occurrence in flattened_feature_instanceMap do
    for each pattern_occurrence in pattern_instanceMap do
      position_of_last_featureset_of_occurrence ← Last item of pattern_occurrence
      if (position_of_feature_occurrence is less than position_of_last_featureset_of_occurrence) and
(position_of_feature_occurrence is greater than or equal to (position_of_last_featureset_of_occurrence + 1
+ maxJumpSize)) then
        new_pattern_occurrence ← position_of_feature_occurrence
        ADDITEM(new_pattern_occurrence, pattern_occurrence)
        ADDITEM(new_pattern_instanceMap, new_pattern_occurrence)
      end if
    end for
  end for

  return new_pattern_instanceMap
end function

```

---

Code 3 – Python implementation of the s-extension operation with InstanceMaps, limiting the maxJumpSize

```

1  def get_s_extension_instanceMap(
2      pattern_instanceMap: list[list[int]],
3      flattened_feature_instanceMap: list[int],
4      maxJumpSize: int
5  ) -> list[list[int]]:
6      new_pattern_instanceMap = []
7
8      for position_of_feature_occurrence in flattened_feature_instanceMap:
9          for pattern_occurrence in pattern_instanceMap:
10             position_of_last_featureset_of_occurrence = pattern_occurrence[-1]
11             if (
12                 position_of_last_featureset_of_occurrence <
13                 position_of_feature_occurrence <=
14                 (position_of_last_featureset_of_occurrence + 1 + maxJumpSize)
15             ):
16                 new_pattern_occurrence = pattern_occurrence.copy()
17                 new_pattern_occurrence.append(position_of_feature_occurrence)
18
19                 new_pattern_instanceMap.append(new_pattern_occurrence)
20     return new_pattern_instanceMap

```

`pattern_instanceMap` =  $\llbracket [0, 1] \rrbracket$ , and `flattened_feature_instanceMap` =  $\llbracket 4 \rrbracket$ . Suppose we also set `maxJumpSize` = 1 (meaning that there might exist at most 1 event in-between the events that occur in a pattern). Thus,  $(\text{position\_of\_last\_featureset\_of\_occurrence} + 1 + \text{maxJumpSize}) = 3$ . Since `position_of_feature_occurrence` = 4, we have  $4 > 3$ , and thus the new pattern's InstanceMap will be the empty list  $\llbracket \rrbracket$ . This makes sense, since we have two events between the event  $\{c\}$  and the event  $\{d\}$  in  $S$  (the events  $\{f, g\}$  and  $\{g\}$ ), so the jump between the events  $\{c\}$  and the event  $\{d\}$  is  $2 > \text{maxJumpSize} = 1$ , making this occurrence of the pattern to be discarded during the computation of the new InstanceMap.

## 5 Experimental Setup

### 5.1 Dataset

During the development of this work, we could not gather a dataset of digitized sheet music that were reliably labeled by emotion. However, an algorithm that is capable of mining patterns in any dataset of labeled music sheets is also capable of mining patterns in a dataset labeled by emotion. Thus, to conduct this research we used a public dataset named MTC-ANN-2.0.1 (KRAMENBURG; JANSSEN; VOLK, 2016) - hereinafter called MTC.

The MTC dataset consists of a collection of 360 dutch folk songs in MusicXML format, labeled by their genre<sup>1</sup>, with 26 genres available. Table 10 presents all 26 genres, the abbreviations used hereinafter, and the number of samples of each genre in the dataset.

Table 10 – MTC dataset’s genres, genres abbreviations, number of songs in each genre, and average( $\pm$ standard deviation) number of events per song, grouped by genre

Genre	Genre abbreviation	#Samples	Avg. #events ( $\pm$ std)
Daar_ging_een_heer_1	Heer	16	42.9( $\pm$ 2.9)
Daar_reed_een_jonkheer_1	Jonkheer	12	39.9( $\pm$ 8.0)
Daar_was_laastmaal_een_ruiter_2	Ruiter2	17	54.5( $\pm$ 12.5)
Daar_zou_er_een_maagdje_vroeg_opstaan_2	Maagdje	10	55.8( $\pm$ 10.0)
Een_Soudaan_had_een_dochtertje_1	Dochtertje	13	45.5( $\pm$ 10.2)
Een_lindeboom_stond_in_het_dal_1	Lindeboom	9	33.2( $\pm$ 1.3)
En_er_waren_eens_twee_zoeteliefjes	Zoeteliefjes	16	43.6( $\pm$ 4.8)
Er_reed_er_eens_een_ruiter_1	Ruiter1	27	49.8( $\pm$ 14.5)
Er_was_een_herderinnetje_1	Herderinnetje	11	86.3( $\pm$ 2.0)
Er_was_een_koopman_rijk_en_machtig	Koopman	17	55.9( $\pm$ 3.6)
Er_was_een_meisje_van_zestien_jaren_1	Meisje	15	34.2( $\pm$ 6.8)
Er_woonde_een_vrouwtje_al_over_het_bos	Vrouwtje	12	68.5( $\pm$ 14.0)
Femmes_voulez_vous_eprouver	Femmes	13	66.9( $\pm$ 21.4)
Heer_Halewijn_2	Halewijn2	11	35.6( $\pm$ 2.1)
Heer_Halewijn_4	Halewijn4	11	32.9( $\pm$ 4.2)
Het_vrouwtje_van_Stavoren_1	Stavoren	8	64.0( $\pm$ 8.6)
Het_was_laast_op_een_zomerdag	Zomerdag	17	39.8( $\pm$ 2.1)
Het_was_op_een_driekoningenavond_1	Driekoningenavond	12	45.8( $\pm$ 9.9)
Ik_kwam_laast_eens_in_de_stad	Stad	18	37.2( $\pm$ 1.8)
Kom_laat_ons_nu_zo_stil_niet_zijn_1	Stil	11	74.0( $\pm$ 9.5)
Lieve_schipper_vaar_me_over_1	Schipper	15	40.1( $\pm$ 9.1)
O_God_ik_leef_in_nood	Nood	8	80.1( $\pm$ 20.2)
Soldaat_kwam_uit_de_oorlog	Soldaat	17	56.7( $\pm$ 10.2)
Vaarwel_bruidje_schoon	Bruidje	11	58.2( $\pm$ 8.7)
Wat_zag_ik_daar_van_verre_1	Verre	15	42.7( $\pm$ 9.0)
Zolang_de_boom_zal_bloeien_1	Boom	18	45.3( $\pm$ 9.4)

Source: adapted from (KRAMENBURG; JANSSEN; VOLK, 2016)

<sup>1</sup> Musicologists believe that a lot of dutch folk songs share the same base song, but due to oral tradition the base songs went through successive modifications, until they became new songs altogether. The songs in the MTC dataset are clustered according to what the musicologists believe are songs that share the same base song, and these clusters are called tune families. Thus, the genre is actually the tune family of the song. (KRAMENBURG; WIERING; VOLK, 2013)

The MTC dataset is distributed in the Humdrum `**kern` format. Thus, we made a simple Python script that converts the songs into MusicXML format using the converter included in the Music21 toolkit. This script ran only once, and converted the files in parallel using concurrent programming, taking just a few minutes to complete.

The MTC dataset also comes with a set of pattern annotations, discovered manually by humans. However, these annotations are found as intra-opus patterns instead of inter-opus. So, as also noted by (KRAMENBURG; CONKLIN, 2016), these annotations are not ideal for usage as a ground-truth for inter-opus pattern discovery algorithms. Thus, in our work, we disregard these annotations.

All songs contain only the vocal part and the lyrics. Table 10 shows the distribution of musical events (i.e., notes, rest, or chords) per song, grouped by each genre. The songs are relatively short (avg.  $49.9 \pm 16.4$  events).

## 5.2 Implemented Viewpoints

We implemented some viewpoints for the experiments. We have not exhausted the list of possible viewpoint implementations, as finding the optimal viewpoints is not in the scope of this project. The implemented viewpoints are described as follows:

- *isRest*: indicates whether the event is a rest.
- *durationQuarterLength*: the duration of the event as a ratio of quarter notes. For example, a quarter note has *durationQuarterLength* = 1. An eighth note has *durationQuarterLength* = 0.5. A half rest has *durationQuarterLength* = 2.
- *relativeDuration*: the *durationQuarterLength* divided by the value of the denominator of the time signature in quarter lengths.
- *diatonicInterval*: the interval between the tonic of the piece and the event. E.g., if the tonic is a C, and the event is a D $\sharp$ , then the *diatonicIntervalNoQuality* = M2.
- *diatonicIntervalNoQuality*: the interval between the tonic of the piece and the event, but with the quality of the interval discarded. E.g., if the tonic is a C, and the event is a D $\sharp$ , then the *diatonicIntervalNoQualit* = 2.
- *contour\_pitch*: the sign of the difference in pitch between the previous note and the current note event. E.g., if the previous note has a MIDI pitch value of 52, and the current note event has a MIDI pitch value of 49, then *contour\_pitch* = -, since the current pitch is smaller than the previous pitch. If both notes are the same, then *contour\_pitch* = =.

- *contour\_duration*: the sign of the difference in duration (in quarter lengths) between the previous event and the current event. E.g., if the previous event was a quarter rest, and the current event is a half note, then the duration increased, so *contour\_duration* = +.

We have not found any descriptions in the literature regarding the use of something as the *relativeDuration* as a feature to analyze rhythmic patterns. Figure 28 gives an example of where the *relativeDuration* viewpoint could be useful: in Figure 28a we have a 4/4 time signature and a 260bpm tempo, while in Figure 28b we have a 4/8 time signature and a 130bpm. Although these two scores are written differently, they sound exactly the same. The first three notes in Figure 28a are quarter notes, so their *durationQuarterLength* = 1, and the time signature’s quarter length value is also 1, so their *relativeDuration* =  $\frac{1}{1} = 1$ . In Figure 28b, the first three notes are eighth notes, so their *durationQuarterLength* = 0.5, and the time signature’s quarter length value is also 0.5, so their *relativeDuration* =  $\frac{0.5}{0.5} = 1$ . The whole note in Figure 28a has *relativeDuration* =  $\frac{4}{1} = 4$ , while the half note in Figure 28b has *relativeDuration* =  $\frac{2}{0.5} = 4$ . In conclusion, by dividing the quarter length value of each event by the time signature’s denominator we have a way to compare both song’s rhythmic structure in a fair way: even with different duration values, both songs have the same *relativeDuration* for all events.

The advantage of this approach over the *dr* (duration ratio) from Conklin and Bergeron (2008) is that, by using the time signature’s denominator as an anchor, we have a way to compare one song with another even if they have different time signatures, without depending on the previous event value. The property of rightfully comparing one song with another even if they are written in different time signatures (or key signatures) is of utmost importance, since we are dealing with the inter-opus pattern discovery task.

Regarding the *diatonicInterval* and the *diatonicIntervalNoQuality*, both of them use the tonic (a.k.a. tonal center) of the piece in their computation. These viewpoints are similar to the *intfref* viewpoint from (CONKLIN; WITTEN, 1995), but with a key difference in how they compute the pedal/root/tonic/bass drone note. The *intfref* viewpoint by (CONKLIN; WITTEN, 1995) use the key signature to determine what is the tonic of the piece (which is called the *referent* by (CONKLIN; WITTEN, 1995)). However, we realized that songs that are converted from MIDI to MusicXML are usually annotated with an atonal key signature (which would be interpreted as C Major). Also, humans might annotate keys wrongly, especially when transcribing by ear. Lastly, the key signature might be selected due to the easiness in reading the score, minimizing the number of accidentals; however, there might be cases in which the songs are in modes different than Ionian or Aeolian, which would cause confusion in the determination of the root note only by the key signature. Thus, we decided to use the Bellman-Budge’s Automatic Key Analysis (BELLMANN, 2005) method to automatically determine the key

## Beethoven's 5th Symphony Riff in 4/4



(a)

## Beethoven's 5th Symphony Riff in 4/8



(b)

Figure 28 – Introduction riff from Beethoven’s 5th Symphony written with two different time signatures, but sounding exactly the same

signature from the pieces, using music21’s implementation. It is noteworthy reinforcing that our objective by using the Bellman-Budge’s algorithm is not to determine the key of the music pieces, but only their tonic, so that we can calculate the interval between the tonic and the current event. The Bellman-Budge’s algorithm method to find keys is to find the most prominent notes in the song, so this algorithm is fit for the task of finding the tonic.

### 5.3 Experiments

We setup our experiments to answer the following Research Questions:

- **RQ1:** Does limiting the maximum number of jumps between events of a pattern results in more interesting patterns in the musicological aspect?
- **RQ2:** Which data structure (Bitmap or InstanceMap) is the most efficient in terms of memory consumption and processing time?

To answer these research questions we implemented the algorithms described in [Section 4.1](#), based on the MGDGP algorithm described in [Section 3.2.5](#), and applied them to the dataset described in [Section 5.1](#). We permuted the input hyper-parameters of the MGDGP algorithm (indicated in [Figure 29](#)). Note that we added two hyper-parameters that were not previously implemented by ([CONKLIN, 2010a](#)) - the Max Jump Length (equivalent to the *maxJumpSize* described in [Section 4.1.2](#)), and the Maximum Pattern

Length (to limit the maximum number of events that can compose a pattern - a way to avoid very long processing times).

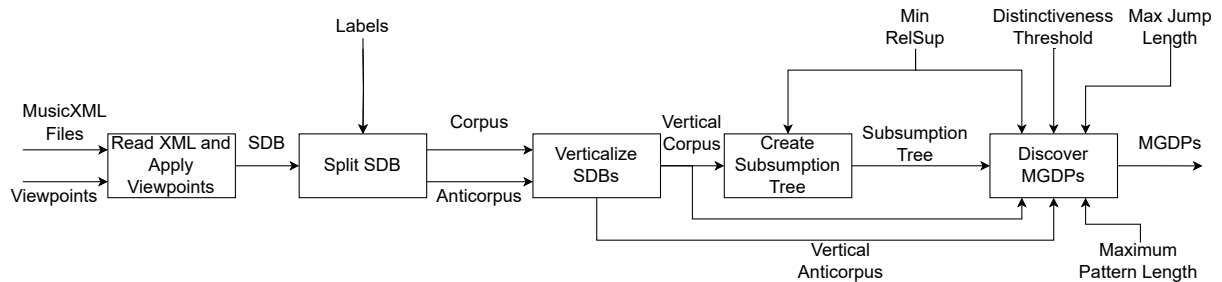


Figure 29 – Complete block diagram representing the implemented MGDG algorithm. Blocks are steps of the algorithm, and arrows indicate the input hyper-parameters of the algorithm and the outputs from each step

We did not implement the I-Step Pruning nor the S-Step Pruning from the SPAM algorithm (AYRES et al., 2002). This does not affect the results, though. Conklin and Bergeron (2008) also do not mention using this feature from the original SPAM algorithm in their implementation.

We also did not prune the Subsumption Graph into a Subsumption Tree (as done by Conklin and Bergeron (2008) and described in Section 3.2.5.4). We did implement the pruning algorithm, and the tree was correctly generated, but for some unknown reason, the number of MGDGs found were not deterministic when we followed this approach. This became evident when dealing with many viewpoints and long patterns. Thus, we decided to use the unpruned Subsumption Graph as the source of the set  $I_j$  of possible i-extensions of the nodes. To avoid visiting the same pattern more than once, after generating the extended pattern, and before calculating the new pattern’s Bitmap/InstanceMap, we instead do a check if the pattern already exists in our Sequence Tree. Since the Sequence Tree is actually implemented as a hash map, looking for the existence of a pattern is  $O(1)$ , so it does not hinder the performance of the algorithm too much. Further investigation needs to be done to determine if this results’ stochasticity was produced by an implementation bug, or by a flaw in the algorithm.

## 5.4 Hardware and Software

To perform our experiments, we setup a containerized development environment using the Docker platform, with a PySpark image<sup>2</sup>. Most of our experiments were done on a single-node machine with 32GB RAM and 4 CPU cores, but some experiments were also done on another single-node machine with 8GB RAM and 8 CPU cores. We use the PySpark framework only for the 4 initial steps of the algorithm - which do not take many

<sup>2</sup> <https://hub.docker.com/r/jupyter/pyspark-notebook>



computational resources when compared to the actual MGD Discovery step - so we refrain ourselves of doing any analysis regarding the number of CPU cores in this work.

Our algorithm was implemented using various Python third-party libraries, including:

- NetworkX<sup>3</sup>, for the Subsumption Graph and Subsumption Tree creation;
- NumPy<sup>4</sup>, for dealing with Bitmaps and Bitmatrices;
- Music21<sup>5</sup>, for reading the MusicXML dataset, applying the viewpoints, and outputting the patterns as highlighted MusicXML scores;
- SciPy<sup>6</sup>, for performing the Fisher's Exact Test as a measure of distinctiveness.

We also used several other internal Python libraries (e.g., `functools`, `dataclasses`, `abc`, `math`) for other tasks.

To perform our results' analyses, we used the Pandas<sup>7</sup> framework and the Plotly<sup>8</sup> library for the exploratory data analysis and the data visualization. We also used the SciPy library to perform statistical analyses on the results.

---

<sup>3</sup> <https://networkx.org/>

<sup>4</sup> <https://numpy.org/>

<sup>5</sup> <http://web.mit.edu/music21/>

<sup>6</sup> <https://scipy.org/>

<sup>7</sup> <https://pandas.pydata.org/>

<sup>8</sup> <https://plotly.com/python/plotly-express/>

## 6 Results and Discussion

In this section we will present the results from our analyses, aiming to answer the Research Questions raised in [Section 5.3](#).

### 6.1 RQ1

To answer RQ1 (i.e., “Does limiting the maximum number of jumps between events of a pattern results in more interesting patterns in the musicological aspect?”) we sent some of the patterns discovered to a musicologist, so that they could analyze them. One of the patterns that the musicologist analyzed was the one depicted in [Figure 30](#), which was discovered by using the *MGDP\_InstanceMap()* algorithm (i.e., without setting a *maxJumpSize*).

The feedback we received was that there was not much “musical logic” in the patterns discovered, and that the patterns presented “quite random musical fragments with elements that could appear in any musical style”. The musicologist understood what the algorithm was capable to do, but they mentioned that they did not understand what they could do with the patterns discovered.

We believe that this feedback could be due to two main reasons:

1. The combination of various viewpoints in one pattern. E.g., *isRest*, *contour\_pitch* and *contour\_duration* in [Figure 30](#);
2. The big jumps between notes that compose the pattern. E.g., the many events between the first (blue) rest and the second (green) rest in [Figure 30a](#) and [Figure 30b](#).

These two characteristics of the patterns sent to the musicologist could mean that the patterns discovered are not “hearable”. It is indeed hard to imagine what the pattern  $\{\{isRest\ True\}, \{isRest\ True\}, \{contour\_pitch\ =, contour\_duration\ =\}, \{contour\_duration\ =, contour\_pitch\ -\}\}$  sounds like, especially when this pattern can occur with any number of events in between them. In fact, even by playing these piece’s excerpts one after another, and visually following the pattern’s notes, this pattern is still not obvious to the listener.

To test the hypothesis that these two characteristics are what deems the pattern as musically irrelevant, we analyzed patterns discovered by the *MGDP\_InstanceMap\_limitMaxJumps()* algorithm, with *maxJumpSize* = 1. The first pattern we analyzed was the one depicted in [Figure 31](#). The pattern was present in more than 92% of the pieces of the genre *Femme*, but in [Figure 31](#) we show it in only 3 pieces. It was also found to be

## NLB111478\_01

maakt en schoon, dorst naar Li - set-tes hand te din - gen. Het meis-je,

7  
schoon ook een boe - rin, had ech - ter

isRest True

isRest True

contour\_pitch =  
contour\_duration =

contour\_duration =  
contour\_pitch -

(a)

## NLB074390\_01

in het bos in't rond wat von-den zij wie kan dat ra - den

5  
Wat von - den zij wie kan dat

isRest True

isRest True

contour\_pitch =  
contour\_duration =

contour\_duration =  
contour\_pitch -

(b)

## NLB146731\_01

lijn, een bra - ve boe - ren - zoon, Het puik - je van de dor-pe

isRest True

isRest True

contour\_pitch =  
contour\_duration =

contour\_duration =  
contour\_pitch -

(c)

Figure 30 – One of the patterns sent to be analyzed by the musicologist

more than 45-fold over-represented in the genre *Femme* than in the combination of all other genres. This pattern contains only one viewpoint (*diatonicIntervalNoQuality*),

and has  $maxJumpSize = 1$ . We have set the algorithm's hyper-parameters as follows:  $MIN\_REL\_SUP = 0.9$ ,  $DISTINCTIVENESS\_THRESHOLD = 29$  (using the Raw Distinctiveness), and  $MAX\_PATTERN\_LENGTH = 7$ . This pattern is much easier to visualize and understand than the one depicted in Figure 30. In fact, by hearing this pattern, it becomes obvious how they are very similar in the 3 pieces depicted (and in 9 other pieces of the genre *Femme*, in which this pattern was also present). Note that even by having songs with different keys, the same pattern applies, and it is still easy to recognize the pattern by visualizing it in the score and by hearing it - Figure 31c is on a  $E_b$  Major key, while the other two are on a G Major key.

To further confirm our hypothesis, we analyzed the pattern depicted in Figure 32, which was discovered using the same algorithm with  $maxJumpSize = 1$ , but with 3 viewpoints instead of 1. The algorithm's parameters were also the same, apart from the Maximum Pattern Length, which was set to 4. This pattern was also present in over 92% pieces of the genre *Femme*, and was 29-fold over-represented in the corpus than in the anticorpus. Although this pattern is more recognizable than the pattern from Figure 30, we find that it presents more cognitive complexity than the one from Figure 31, which might limit its usefulness.

All patterns from Figures 30, 31 and 32 share similar measures of  $relSup$ , Raw Distinctiveness ( $\Delta$ ), and number of events (4 and 5), but they have varying degrees of complexity. Unfortunately, we did not find a way to objectively measure which of these patterns is the most recognizable and interesting in the musicological aspect. However, to answer RQ1, after analyzing these patterns we do believe that, in general, limiting the number of jumps between events of a pattern results in more interesting patterns. This answer also depends on what is the purpose of the algorithm: if the objective is finding motifs, recurring phrases, or themes, then limiting the number of patterns might indeed be interesting; however, if the objective is to find patterns in song structure, then limiting the number of jumps between events of a pattern might not be ideal.

## 6.2 RQ2

We will break RQ2 (i.e., "Which data structure (Bitmap or InstanceMap) is the most efficient in terms of memory consumption and processing time?") into two sub-questions, which will be answered in the following subsections:

1. Which Data Structure is the most efficient when setting the  $maxJumpSize$ ?
2. Which Data Structure is the most efficient when using the algorithms that do not set  $maxJumpSize$ ?

## NLB074286\_01

mij ach - ter de deur ver - ste - ken 'k Gaat mij ach -

ter de deur ver

diatonicIntervalNoQuality 2

diatonicIntervalNoQuality 3

diatonicIntervalNoQuality 5

diatonicIntervalNoQuality 6

diatonicIntervalNoQuality 8

(a)

## NLB074470\_01

Komt men tot vre-se - lij - ke din - gen komt men tot vre-se - lij - ke

gen komt

diatonicIntervalNoQuality 2

diatonicIntervalNoQuality 3

diatonicIntervalNoQuality 5

diatonicIntervalNoQuality 6

diatonicIntervalNoQuality 8

(b)

## NLB146731\_01

Van veld en kud-de af-ge-we - ken, Van veld en

kud - de af - ge - we

diatonicIntervalNoQuality 2

diatonicIntervalNoQuality 3

diatonicIntervalNoQuality 5

diatonicIntervalNoQuality 6

diatonicIntervalNoQuality 8

(c)

Figure 31 – Pattern outputted by the `MGDP_InstanceMap_limitMaxJumps()` algorithm, with `maxJumpSize = 1`, with only one viewpoint

### 6.2.1 Setting maxJumpSize

We ran the `MGDP_Bitmap_limitMaxJumpSize()` and the `MGDP_InstanceMap_limitMaxJumpSize()` algorithms pairwise, varying the hyper-parameters each time, and pairing the runs with

## NLB072450\_01



zoon En zij ging - en haar plan ont

contour\_pitch =  
diatonicIntervalNoQuality 5

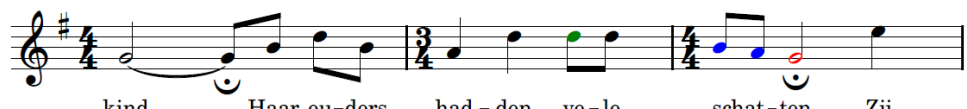
relativeDuration 0.125  
diatonicIntervalNoQuality 3

relativeDuration 0.125  
contour\_pitch -

contour\_pitch -

(a)

## NLB076118\_01



kind. Haar ou-ders had - den ve - le schat - ten. Zij

contour\_pitch =  
diatonicIntervalNoQuality 5


relativeDuration 0.125  
diatonicIntervalNoQuality 3

relativeDuration 0.125  
contour\_pitch -

contour\_pitch -

(b)

## NLB111478\_01



dorst naar Li - set - tes hand te din - gen. Het meis - je,

contour\_pitch =  
diatonicIntervalNoQuality 5

contour\_pitch -

relativeDuration 0.125  
diatonicIntervalNoQuality 3

relativeDuration 0.125  
contour\_pitch -

(c)

Figure 32 – Pattern outputted by the *MGDP\_InstanceMap\_limitMaxJumps()* algorithm, with *maxJumpSize* = 1, but with 3 viewpoints

the same hyper-parameters settings from each Data Structure, to analyze them using the Wilcoxon's non-parametric statistical significance test. In total, we ran 20 pairwise comparisons (summing up to 40 MGD Discovery runs).

Figure 33 depicts the box plot distributions of the execution time of the algorithms' runs. The two-tailed paired Wilcoxon's tests resulted in  $p = 0.000002 < 0.05$  for the Execution Time comparison, with an average of 319s( $\pm 697$ s) for the InstanceMap algorithm, and an average of 481s( $\pm 910$ s) for the Bitmap algorithm. This indicates that the InstanceMap algorithm is significantly faster to run than the Bitmap algorithm when using the versions of the algorithms that limits the maximum jump size.

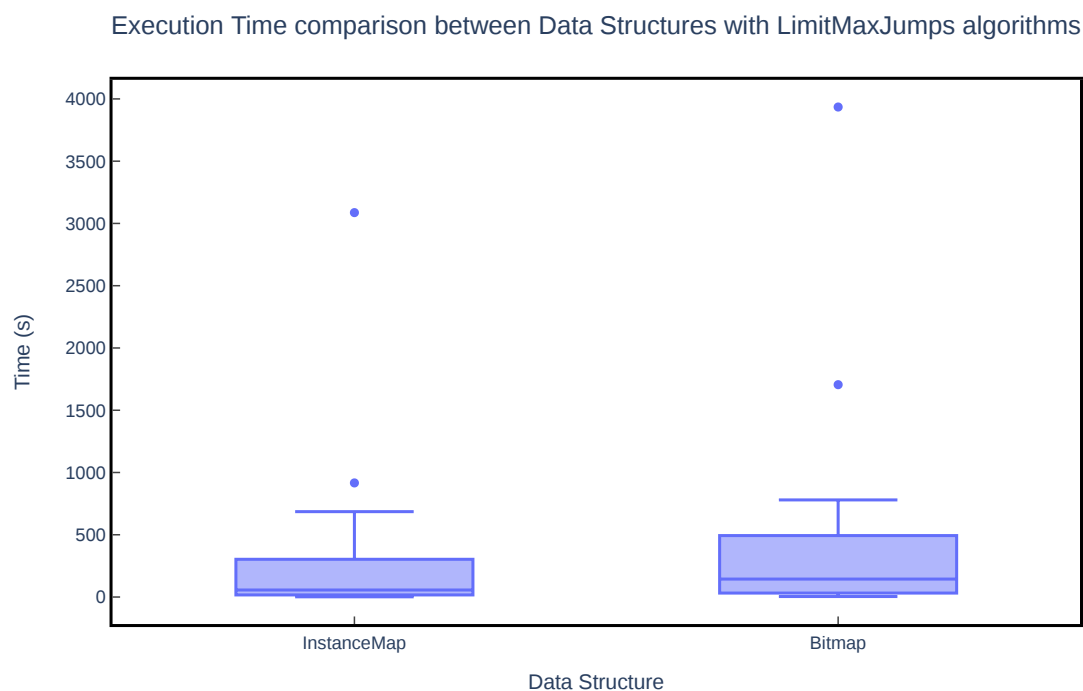


Figure 33 – Box plot distributions of the execution time of Bitmaps and InstanceMaps algorithms' runs, when setting *maxJumpSize* (lower is better)

The box plot distributions of the memory consumption of the algorithms' runs are depicted in Figure 34. The two-tailed paired Wilcoxon's tests resulted in  $p = 0.189348 \geq 0.05$  for the Memory Consumption comparison, which indicates that there is not a statistically significant difference in Memory Consumption between the Data Structures used in the algorithms, when setting the limit *maxJumpSize*.

## 6.2.2 Without setting maxJumpSize

Similar to what has been done in Section 6.2.1, we ran the `MGDP_Bitmap()` and the `MGDP_InstanceMap()` algorithms pairwise, varying the hyper-parameters each time, and pairing the runs with the same hyper-parameters settings from each Data Structure, to analyze

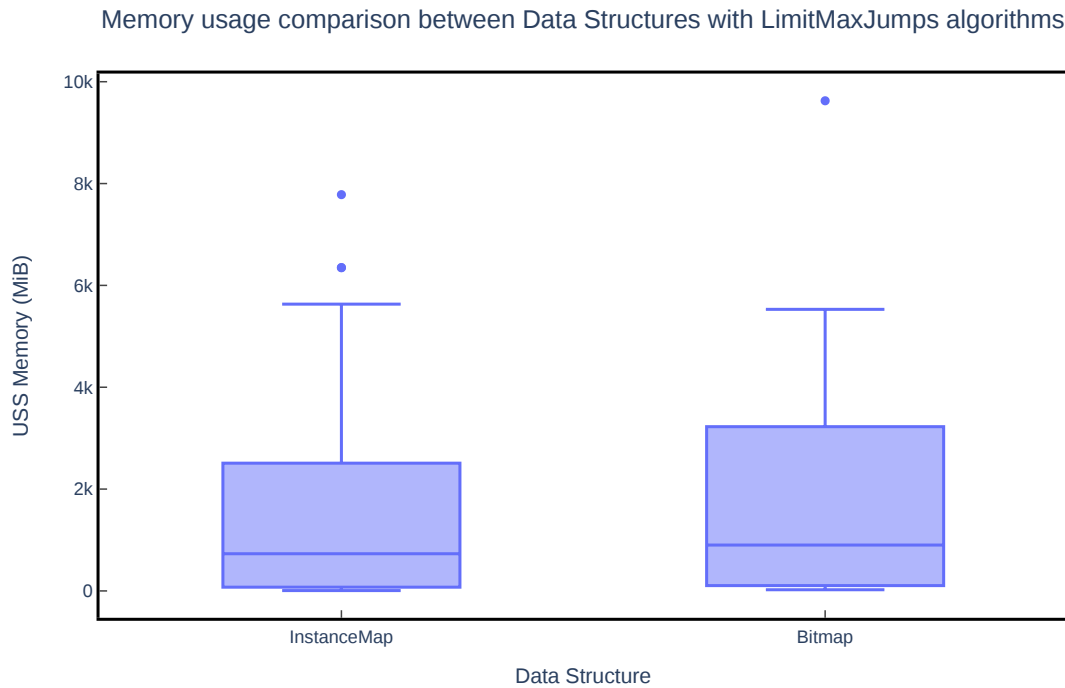


Figure 34 – Box plot distributions of memory consumption of Bitmaps and InstanceMaps algorithms’ runs, when setting *maxJumpSize* (lower is better)

them using the Wilcoxon’s non-parametric statistical significance test. We ran 6 pairwise comparisons (12 runs in total).

Figure 35 depicts the box plot distributions of the memory consumption of the algorithms’ runs. The two-tailed paired Wilcoxon’s tests resulted in  $p = 0.031250 < 0.05$  for the Memory Consumption comparison, with an average of 17715MB( $\pm 1144$ MB) for the InstanceMap algorithm, and an average of 240.8MB( $\pm 321.4$ MB) for the Bitmap algorithm. This indicates that the InstanceMap algorithm consumes significantly more memory to run than the Bitmap algorithm when using the versions of the algorithms that do not limit the maximum jump size.

The box plot distributions of the execution time of the algorithms’ runs are depicted in Figure 36. The two-tailed paired Wilcoxon’s tests resulted in  $p = 0.031250 < 0.05$  for the Execution Time comparison, with an average of 254.9s( $\pm 34.27$ s) for the InstanceMap runs, and an average of 35.21s( $\pm 53.77$ s) for the Bitmap runs. This indicates that the InstanceMap algorithm takes significantly more time to run than the Bitmap algorithm when using the versions of the algorithms that do not limit the maximum jump size.

The results were similar for all 6 pairwise runs of the algorithms without setting a limit with *maxJumpSize*: the runs using the Bitmap took much less time and consumed much less memory than the runs using the InstanceMap. Table 11 presents the memory consumption of the 6 pairwise comparisons. It is noticeable how the InstanceMap is



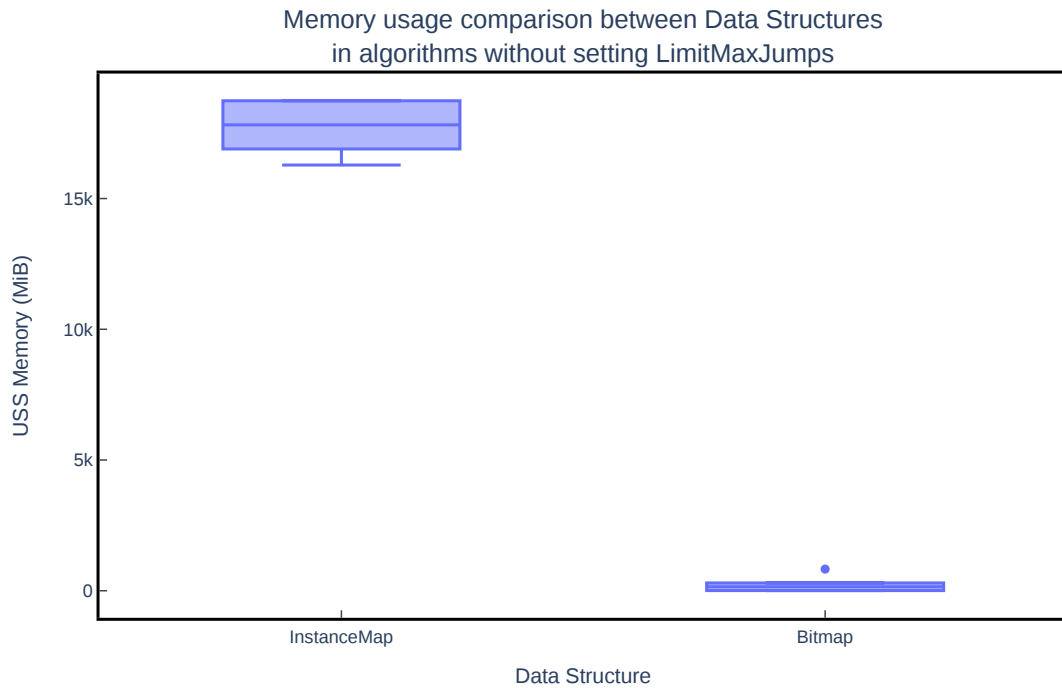


Figure 35 – Box plot distributions of memory consumption of Bitmaps and InstanceMaps algorithms' runs, when NOT setting *maxJumpSize* (lower is better)

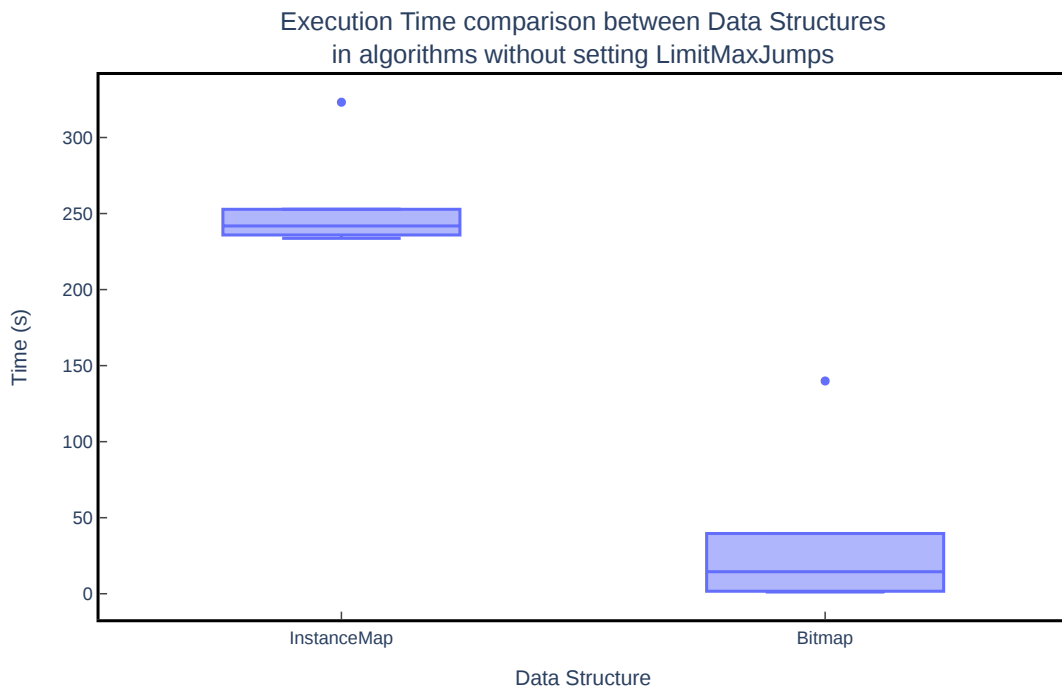


Figure 36 – Box plot distributions of execution time of Bitmaps and InstanceMaps algorithms' runs, when NOT setting *maxJumpSize* (lower is better)

Table 11 – Pairwise memory consumption comparison (in MB) between algorithms using different data structures, when *maxJumpSize* is not set. Values in the same row presents runs with the same hyper-parameter configuration (apart from the Data Structure). Lower is better

<b>InstanceMap</b>	<b>Bitmap</b>
16896.0	298.1
16281.6	828.4
16896.0	298.4
18739.2	6.8
18739.2	6.6
18739.2	6.6

consistently using much more memory than the Bitmap. In fact, most InstanceMaps runs from our experiments crashed because they reached the maximum RAM memory available on the Docker container (which was set to 27GB), while the Bitmap runs with the same hyper-parameters finished relatively fast and without consuming much memory. We discarded these crashed runs for our statistical tests.

In summary, to answer RQ2, the `MGDP_InstanceMap_limitMaxJumpSize(maxJumpSize)` algorithm is slightly more efficient than the `MGDP_Bitmap_limitMaxJumpSize(maxJumpSize)` algorithm, whereas the `MGDP_Bitmap()` algorithm is much more efficient than the `MGDP_InstanceMap()` algorithm.

## 7 Conclusion

In the previous chapters we summarized information that were deemed necessary to complete the given task of Inter-Opus Musical Motif Discovery in Symbolic Music. We presented the motivation for the project; a review of basic theoretical concepts required to understand this work; the particularities and requirements of the system; a literature review on existing systems for the given task; a description of some computational tools and methods that could be used to tackle the given problem; a thorough description of novel algorithms to bring new capabilities to the current state-of-the-art methods; a statistical analysis of the performance of these new algorithms; and a dialectical analysis of the musical patterns found.

This work has two main contributions:

1. Novel computational methods to allow mining of musical patterns with a variable limit on the maximum number of events in-between the events of a pattern
2. A method to automatically create a visualization of the found sequential musical patterns without the need of re-scanning the pieces;

Our results showed that the approaches using the InstanceMap data structure proposed in this work have the benefit of enabling the automatic pattern visualization without the need of a re-scan of the score after the pattern was found. The InstanceMap is also more efficient in Execution Time and without a significant difference in Memory Consumption when comparing to the Bitmap in a setup when there is a limit in the number of events that might be in-between the events of the pattern - which is a better fit for finding musical motifs and recurring themes across multiple pieces. The algorithms with variable limit on the maximum number of events in-between the events of a pattern are able to find patterns that would be discarded by the original MGDP algorithm([CONKLIN, 2010a](#)).

When we use the variants of the algorithms that do not set a limit for the number of events that might be in-between the events of the pattern, we see that the Bitmap method outperforms the InstanceMap by a large margin; thus, when trying to find patterns related to the song structure, using the Bitmap and then, for the visualization, performing a re-scan of the piece looking for the discovered patterns might be more efficient than trying to store all occurrences of the events of the pattern in all pieces (which is what the InstanceMap does).

The main limitation of this work is the lack of objective metrics to quantify the quality of the patterns found. It is expected that the algorithm finds a reasonable

number of patterns to be further analyzed by human musicologists, and that these patterns are interesting and useful. However, apart from the distinctiveness metrics and the Corpus Relative Support, we cannot objectively measure whether the algorithm is finding interesting patterns or not.

Another limitation of this work is that we performed our tests on only one dataset in this work. The algorithms' performances might be dependent on the length of the pieces, so it would be interesting to analyze the results with other datasets as well.

Some possible future works that stem out from this one are:

1. Trying to optimize the algorithms using extensions written in efficient languages (e.g., implementing a C++ or Rust extension module to deal with the InstanceMap and Bitmap calculations);
2. Implement these algorithms as a plugin for one of the popular music score visualization software (e.g., MuseScore), possibly enhancing the pattern visualization.
3. Implement a distributed version of the algorithm (e.g., using the SPAMC-UDLT (CHEN; SHUAI; CHEN, 2017), which could also mitigate the high memory consumption problem of the InstanceMap algorithm);
4. Analyze the difference in cognitive load of the patterns found when varying the Viewpoints (e.g., is analyzing two separate patterns with two different viewpoints more interesting than analyzing one pattern found with the combination of the two viewpoints?);
5. Investigate the impacts of varying the maximum number of events that might appear in-between the events of a pattern (both musicologically and computationally);

We believe that, with this work, we took another step into unveiling the mysteries behind the sequential patterns of musical events that lead to the conveying of specific emotions. We hope that this work will be useful in this endeavor, by applying the proposed algorithms to a dataset that is labeled by emotions, in the future.

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001

## References

- AGRAWAL, Rakesh; SRIKANT, Ramakrishnan. Fast algorithms for mining association rules. In: *Proc. of 20th International Conference on Very Large Data Bases, {VLDB'94}*. [S.l.: s.n.], 1994. Cited 2 times, on pages 35 and 50.
- ANDERS, Torsten. A model of musical motifs. In: *International Conference on Mathematics and Computation in Music*. [S.l.: s.n.], 2007. ISSN 18650929. Cited on page 23.
- ARJMAND, Hussain Abdulah; HOHAGEN, Jesper; PATON, Bryan; RICKARD, Nikki S. Emotional responses to music: Shifts in frontal brain asymmetry mark periods of musical change. *Frontiers in Psychology*, v. 8, 2017. ISSN 16641078. Cited on page 14.
- ASSOCIATION, MIDI Manufacturers; OTHERS. *The complete MIDI 1.0 detailed specification*. [S.l.]: Los Angeles, CA, The MIDI Manufacturers Association, 1996. Cited on page 27.
- AYRES, Jay; FLANNICK, Jason; GEHRKE, Johannes; YIU, Tomi. Sequential pattern mining using a bitmap representation. In: *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. [S.l.: s.n.], 2002. Cited 17 times, on pages 16, 34, 35, 36, 37, 38, 41, 42, 48, 51, 52, 53, 54, 55, 56, 59, and 71.
- BAINBRIDGE, David; BELL, Tim. The challenge of optical music recognition. *Computers and the Humanities*, v. 35, n. 2, 2001. ISSN 00104817. Cited on page 20.
- BAUMGARTNER, Thomas; LUTZ, Kai; SCHMIDT, Conny F.; JÄNCKE, Lutz. The emotional power of music: How music enhances the feeling of affective pictures. *Brain Research*, v. 1075, 2006. ISSN 00068993. Cited on page 14.
- BELLMANN, Hector. About the determination of key of a musical excerpt. In: . [S.l.: s.n.], 2005. p. 76–91. ISSN 03029743. Cited on page 69.
- BERRY, David M. The computational turn: Thinking about the digital humanities. *Culture Machine*, v. 12, p. 1–22, 2011. ISSN ISSN 1465-4121. Cited on page 19.
- BEZDEK, Matthew A.; GERRIG, Richard J. *Musical emotions in the context of narrative film*. 2008. Cited on page 14.
- BRACHMAN, Ronald J.; LEVESQUE, Hector J. *Knowledge Representation and Reasoning*. [S.l.: s.n.], 2004. Cited on page 49.
- BRINK, David Van. *The MIDI Specification*. 1995. Available at: <<https://www.cs.cmu.edu/~music/cmsip/readings/davids-midi-spec.htm>>. Accessed on: 22 mar 2021. Cited on page 27.
- CALVO-ZARAGOZA, Jorge; HAJIČ, Jan; PACHA, Alexander. *Understanding optical music recognition*. 2019. Cited on page 20.

- CHEN, Chun Chieh; SHUAI, Hong Han; CHEN, Ming Syan. Distributed and scalable sequential pattern mining through stream processing. *Knowledge and Information Systems*, v. 53, 2017. ISSN 02193116. Cited on page 83.
- CONKLIN, Darrell. Discovery of distinctive patterns in music. *Intelligent Data Analysis*, v. 14, 2010. ISSN 1088467X. Cited 14 times, on pages 16, 34, 39, 42, 44, 45, 46, 47, 52, 53, 54, 58, 70, and 82.
- CONKLIN, Darrell. Distinctive patterns in the first movement of Brahms' String Quartet in C minor. *Journal of Mathematics and Music*, Taylor & Francis, v. 4, n. 2, p. 85–92, 2010. Cited on page 46.
- CONKLIN, Darrell. Antipattern discovery in folk tunes. *Journal of New Music Research*, v. 42, 2013. ISSN 09298215. Cited 5 times, on pages 42, 43, 47, 54, and 58.
- CONKLIN, Darrell. Mining contour sequences for significant closed patterns. *Journal of Mathematics and Music*, 2021. ISSN 1745-9737. Cited on page 44.
- CONKLIN, Darrell; ANAGNOSTOPOULOU, Christina. Representation and discovery of multiple viewpoint patterns. *International Computer Music Conference*, 2001. Cited on page 41.
- CONKLIN, Darrell; BERGERON, Mathieu. Feature set patterns in music. *Computer Music Journal*, v. 32, 2008. ISSN 01489267. Cited 12 times, on pages 42, 44, 45, 49, 50, 52, 53, 54, 55, 58, 69, and 71.
- CONKLIN, Darrell; WITTEN, Ian H. Multiple viewpoint systems for music prediction. *Journal of New Music Research*, v. 24, 1995. ISSN 17445027. Cited 2 times, on pages 44 and 69.
- CUTHBERT, Michael Scott; ARIZA, Christopher. Music21: A toolkit for computer-aided musicology and symbolic music data. In: *Proceedings of the 11th International Society for Music Information Retrieval Conference, ISMIR 2010*. [S.l.: s.n.], 2010. Cited on page 39.
- CUTHBERT, Michael Scott; ARIZA, Christopher; FRIEDLAND, Lisa. Feature extraction and machine learning on symbolic music using the music21 toolkit. In: *Proceedings of the 12th International Society for Music Information Retrieval Conference, ISMIR 2011*. [S.l.: s.n.], 2011. Cited on page 39.
- DOUEK, Joel. Music and emotion -a composer's perspective. *Frontiers in Systems Neuroscience*, v. 7, 2013. ISSN 16625137. Cited on page 15.
- DOWNIE, J. Stephen. Music information retrieval. *Annual Review of Information Science and Technology*, v. 37, 2003. ISSN 00664200. Cited on page 20.
- DRABKIN, William. *Motif*. Oxford Music Online, 2008. Available at: <<https://www.oxfordmusiconline.com/grovemusic/view/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-0000019221>>. Accessed on: 07 may 2021. Cited on page 21.
- ENGELMAN, Shelly; MAGERKO, Brian; MCKLIN, Tom; MILLER, Morgan; EDWARDS, Doug; FREEMAN, Jason. Creativity in authentic steam education with earsketch. In: *Proceedings of the Conference on Integrating Technology into Computer Science Education, ITiCSE*. [S.l.: s.n.], 2017. Cited on page 14.

- FOURNIER-VIGER, Philippe; CHUN, Jerry; LIN, Wei; KIRAN, Rage Uday; KOH, Yun Sing; THOMAS, Rincy. A survey of sequential pattern mining. *Data Science and Pattern Recognition*, v. 1, 2017. Cited 3 times, on pages 32, 33, and 34.
- FRIELER, Klaus. Miles Vs. Trane. Computational and Statistical Comparison of the Improvisatory Styles of Miles Davis and John Coltrane. *Jazz Perspectives*, Taylor & Francis, v. 12, n. 1, p. 123–145, 2020. ISSN 17494079. Cited on page 20.
- FU, Jiulin Zhang Xiaoqing. The influence of background music of video games on immersion. *Journal of Psychology & Psychotherapy*, 2015. Cited on page 14.
- GAMMA, E; HELM, R; JOHNSON, R; VLISSIDES, J. Design patterns: Elements of reusable software. *Addison-Wesley Professional Computing Series*, 1996. ISSN ISBN: 0-201-63361-2. Cited on page 55.
- GARFINKLE, David; ARTHUR, Claire; SCHUBERT, Peter; CUMMING, Julie; FUJINAGA, Ichiro. PatternFinder: Content-Based Music Retrieval with music21. In: *ACM International Conference Proceeding Series*. [S.l.: s.n.], 2017. Cited on page 39.
- GOOD, Michael. MusicXML: An internet-friendly format for sheet music. *XML Conference and Expo*, 2001. Cited 2 times, on pages 28 and 31.
- HAJIC, Jan; PECINA, Pavel. The MUSCIMA++ Dataset for Handwritten Optical Music Recognition. In: *Proceedings of the International Conference on Document Analysis and Recognition, ICDAR*. [S.l.: s.n.], 2017. ISBN 9781538635865. ISSN 15205363. Cited on page 20.
- HUANG, Rong Hwa; SHIH, Yi Nuo. Effects of background music on concentration of workers. *Work*, v. 38, 2011. ISSN 10519815. Cited on page 14.
- HUNTER, Patrick G.; SCHELLENBERG, E. Glenn; SCHIMMACK, Ulrich. Feelings and perceptions of happiness and sadness induced by music: Similarities, differences, and mixed emotions. *Psychology of Aesthetics, Creativity, and the Arts*, v. 4, 2010. ISSN 19313896. Cited on page 14.
- JUSLIN, Patrik N.; VÄSTFJÄLL, Daniel. Emotional responses to music: The need to consider underlying mechanisms. *Behavioral and Brain Sciences*, v. 31, 2008. ISSN 0140525X. Cited 2 times, on pages 14 and 15.
- KARSDORP, Folgert; KRANENBURG, Peter Van; MANJAVACAS, Enrique. Learning similarity metrics for melody retrieval. In: *Proceedings of the 20th International Society for Music Information Retrieval Conference, ISMIR 2019*. [S.l.: s.n.], 2019. ISBN 9781732729919. Applies Deep Learning to learn the Melodic Similarity Metric using the MTC dataset. Cited on page 41.
- KEPPER, Johannes. XML-basierte Codierung musikwissenschaftlicher Daten — Zu den Voraussetzungen einer digitalen Musikedition. *it - Information Technology*, v. 51, n. 4, 2009. ISSN 1611-2776. Cited on page 32.
- KRANENBURG, Peter Van; CONKLIN, Darrell. A pattern mining approach to study a collection of dutch folk-songs. In: *Proceedings of the 5th International Workshop on Folk Music Analysis (FMA 2016)*. [s.n.], 2016. p. 71–73. Available at: <<http://www.ehu.eus/cs-ikerbasque/conklin/papers/fma2016.pdf>>. Cited 3 times, on pages 43, 44, and 68.

- KRANENBURG, Peter Van; WIERING, Frans; VOLK, Anja. On operationalizing the musicological concept of tune-family for computational modeling. In: *Proceedings of the Third International Workshop on Folk Music Analysis*. [S.l.: s.n.], 2013. Cited on page 67.
- KRANENBURG, Peter van; JANSSEN, Berit; VOLK, Anja. The meertens tune collections: The annotated corpus (mtc-ann) versions 1.1. and 2.0.1. In: *Meertens Online Reports*. [S.l.: s.n.], 2016. Cited on page 67.
- LESIUK, Teresa. The effect of music listening on work performance. *Psychology of Music*, v. 33, 2005. ISSN 17413087. Cited on page 14.
- LI, Chia Wei; CHENG, Tzu Han; TSAI, Chen Gia. Music enhances activity in the hypothalamus, brainstem, and anterior cerebellum during script-driven imagery of affective scenes. *Neuropsychologia*, v. 133, 2019. ISSN 18733514. Cited on page 14.
- LIANG, Feynman; GOTHAM, Mark; JOHNSON, Matthew; SHOTTON, Jamie. Automatic stylistic composition of bach chorales with deep LSTM. In: *Proceedings of the 18th International Society for Music Information Retrieval Conference, ISMIR 2017*. [S.l.: s.n.], 2017. ISBN 9789811151798. Cited on page 39.
- LOY, Gareth. Musicians Make a Standard: The MIDI Phenomenon. *Computer Music Journal*, 1985. ISSN 01489267. Cited on page 27.
- LUNDBERG, Scott M.; LEE, Su In. A unified approach to interpreting model predictions. In: *Advances in Neural Information Processing Systems*. [S.l.: s.n.], 2017. v. 2017-December. ISSN 10495258. Cited on page 41.
- LUNDQVIST, Lars Olov; CARLSSON, Fredrik; HILMERSSON, Per; JUSLIN, Patrik N. Emotional responses to music: Experience, expression, and physiology. *Psychology of Music*, v. 37, 2009. ISSN 17413087. Cited on page 14.
- MEREDITH, David. Cosiatec and siateccompress: Pattern discovery by geometric compression. In: *COSIATEC and SIATECCompress: Pattern discovery by geometric compression*. [S.l.: s.n.], 2013. Cited 2 times, on pages 43 and 44.
- MEREDITH, David; LEMSTRÖM, Kjell; WIGGINS, Geraint A. Algorithms for discovering repeated patterns in multidimensional representations of polyphonic music. *International Journal of Phytoremediation*, v. 21, 2002. ISSN 15497879. Cited on page 44.
- MONGEAU, Marcel; SANKOFF, David. Comparison of musical sequences. *Computers and the Humanities*, v. 24, 1990. ISSN 00104817. Cited on page 41.
- MOR, Bhavya; GARHWAL, Sunita; KUMAR, Ajay. A Systematic Literature Review on Computational Musicology. *Archives of Computational Methods in Engineering*, 2020. ISSN 18861784. Cited on page 20.
- MOREIRA, Shirlene Vianna; JUSTI, Francis Ricardo dos Reis; MOREIRA, Marcos. Can musical intervention improve memory in alzheimer's patients? evidence from a systematic review. *Dementia e Neuropsychologia*, v. 12, 2018. ISSN 19805764. Cited on page 14.
- MORI, Kazuma; IWANAGA, Makoto. Two types of peak emotional responses to music: The psychophysiology of chills and tears. *Scientific Reports*, v. 7, 2017. ISSN 20452322. Cited on page 14.



- MUÑOZ-LAGO, Paula; USULA, Nicola; PARADA-CABALEIRO, Emilia; TORRENTE Álvaro. Visualising the structure of 18th century operas: A multidisciplinary data science approach. In: *24th International Conference Information Visualisation*. [s.n.], 2020. p. 527–533. ISBN 9781728191348. Available at: <<https://didone.eu/wp-content/uploads/2020/10/913400a527.pdf>>. Cited 2 times, on pages 24 and 26.
- NEUBARTH, Kerstin; CONKLIN, Darrell. Contrast pattern mining in folk music analysis. In: MEREDITH, David (Ed.). *Computational Music Analysis*. [S.l.]: Springer International Publishing Switzerland 2016, 2016. cap. 15, p. 393–424. Cited on page 43.
- NUTTALL, Thomas; CASADO, Miguel G.; FERRARO, Andres; CONKLIN, Darrell; REPETTO, Rafael Caro. A computational exploration of melodic patterns in arab-andalusian music. *Journal of Mathematics and Music*, 2021. ISSN 1745-9737. Cited on page 44.
- NUTTALL, Thomas; CASADO, Miguel García; TARIFA, Víctor Núñez; REPETTO, Rafael Caro; SERRA, Xavier. Contributing to new musicological theories with computational methods: The case of centonization in arab-andalusian music. In: *Proceedings of the 20th International Society for Music Information Retrieval Conference, ISMIR 2019*. [S.l.: s.n.], 2019. Cited 2 times, on pages 43 and 44.
- PARADA-CABALEIRO, Emilia; TORRENTE, Álvaro. Preventing Conversion Failure across Encoding Formats: A Transcription Protocol and Representation Scheme Considerations. In: *Music Encoding Conference 2020*. [s.n.], 2020. p. 105–107. Available at: <<http://dx.doi.org/10.17613/etwb-m434>>. Cited on page 32.
- PHON-AMNUAISUK, Somnuk. Exploring music21 and gensim for music data analysis and visualization. In: *Communications in Computer and Information Science*. [S.l.: s.n.], 2019. v. 1071. ISSN 18650937. Cited on page 39.
- RAMOS, D.; BUENO, J. L.O.; BIGAND, E. Manipulating greek musical modes and tempo affects perceived musical emotion in musicians and nonmusicians. *Brazilian Journal of Medical and Biological Research*, 2011. ISSN 0100879X. Modes and emotions correlation. Cited on page 14.
- REBELO, Ana; FUJINAGA, Ichiro; PASZKIEWICZ, Filipe; MARCAL, Andre R.S.; GUEDES, Carlos; CARDOSO, Jaime S. Optical music recognition: state-of-the-art and open issues. *International Journal of Multimedia Information Retrieval*, 2012. ISSN 2192662X. Cited on page 20.
- ROLAND, Perry. The music encoding initiative (mei). In: *MAX2002. Proceedings of the First International Conference on Musical Application using XML*. [s.n.], 2002. p. 55–59. Available at: <<http://xml.coverpages.org/MAX2002-PRoland.pdf>>. Accessed on: 26 may 2021. Cited on page 31.
- ROLAND, Perry; HANKINSON, Andrew; PUGIN, Laurent. Early music and the Music Encoding Initiative. *Early Music*, v. 42, n. 4, 2014. ISSN 03061078. Cited on page 32.
- RUDIN, Cynthia. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature Machine Intelligence*, v. 1, 2019. ISSN 25225839. Cited on page 41.

- SAMPAIO, Marcos da Silva; KROGER, Pedro; MENEZES, Mara Pinheiro; ROCHA, Jean Menezes da; OURIVES, Natanael; CARVALHO, Dennis Queiroz de. The Implementation of a Contour Module for Music21. *ART Music Review*, v. 24, 2013. Available at: <<http://www.revista-art.com/the-implementation-of-a-contour-module-for-music21>>. Accessed on: 08 may 2021. Cited on page 39.
- SCHELLENBERG, E. Glenn. Long-term positive associations between music lessons and iq. *Journal of Educational Psychology*, v. 98, 2006. ISSN 00220663. Cited on page 14.
- SCHELLENBERG, E. Glenn; KRYSCIAC, Ania M.; CAMPBELL, R. Jane. Perceiving emotion in melody: interactive effects of pitch and rhythm. *Music Perception*, v. 18, 2000. ISSN 07307829. Cited on page 14.
- SCHELLENBERG, E. Glenn; WEISS, Michael W. *Music and Cognitive Abilities*. 2013. Cited on page 14.
- SHIH, Yi Nuo; HUANG, Rong Hwa; CHIANG, Hsin Yu. Background music: Effects on attention performance. *Work*, v. 42, 2012. ISSN 10519815. Cited on page 14.
- SRIKANT, Ramakrishnan; AGRAWAL, Rakesh. Mining sequential patterns: Generalizations and performance improvements. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*. [S.l.: s.n.], 1996. v. 1057 LNCS. ISSN 16113349. Cited on page 41.
- TORRENTE, Álvaro; LLORENS, Ana. The musicology lab: Teamwork and the musicological toolbox. In: *Music Encoding Conference Proceedings 2021*. [S.l.: s.n.], 2022. v. 2021. Cited 2 times, on pages 16 and 25.
- VANDENNEUCKER, Dominique. *MIDI Tutorial*. 2012. Available at: <<http://www.music-software-development.com/midi-tutorial.html>>. Accessed on: 22 mar 2021. Cited on page 27.
- VOLK, Anja; WIERING, Frans; KRANENBURG, Peter Van. Unfolding the potential of computational musicology. In: *ICISO 2011 - Proceedings of the 13th International Conference on Informatics and Semiotics in Organisations: Problems and Possibilities of Computational Humanities, co-located with IWRA 2011 IFIP WG8.1 Working Conference*. [S.l.: s.n.], 2011. ISBN 9789490719005. Cited 2 times, on pages 19 and 20.
- WHITTALL, Arnold. *Leitmotif*. Oxford Music Online, 2001. Available at: <<https://www.oxfordmusiconline.com/grovemusic/view/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-0000016360?rskey=wR4S1j&result=1>>. Accessed on: 07 may 2021. Cited on page 21.
- ZAKI, Mohammed J. Spade: An efficient algorithm for mining frequent sequences. *Machine Learning*, v. 42, 2001. ISSN 08856125. Cited on page 41.