Department of Computer Science and Engineering,
The Chinese University of Hong Kong

Final Year Project - KY1601

Final Report

Automatic Piano Reduction (Backend): Chord Identification

By
LIN Hang Yu 1155048547
with teammate
Au Ho Wing 1155048142

Supervisor: Prof. Kevin Yip
Co-supervisor: Prof. Lucas Wong

Table of Contents

1 Introduction

1.1 Background

The goal of the whole project is to develop an automatic piano reduction software tool. This year, we focus on chord identifications and music harmonic analysis because they are crucial to piano reduction.

This project is co-supervised by Professor Lucas Wong at Soochow University School of Music. Professor Wong has assisted us at understanding music theory and given advice on music matters.
Although the piano reduction tool is developed by student NG Chiu Yuen in final year project KY1302 in 2013, the accuracy and correctness of software generated musical piece do not fulfill the standard in terms of music. To produce a better result, the issue about music harmony should be studied and put into consideration. Therefore, the focus of this year is chord identification and harmonic analysis so that the results can be enhanced. The purpose of this report is to explain the work we have done in this year.

In the report, the usage of chords and music keys is explained clearly on how they interact together to form meaningful music pieces. Then, chord identification algorithms are explained in details. After that, we look into harmonic analysis and explain algorithms that can recognize music keys. Last but not last, the integration of chord identifications back to piano reduction tool is described.

1.2 Piano Reduction

Piano reduction is an arrangement of scores with multiple instruments into a simpler score to fit on a two-line staff and be playable on the piano by both hands. The timbre and thickness of multi-staff scores should largely be ignored as piano reduction is analog to the viewing of a sculpture from only one viewpoint (Schoenberg, 1975). Piano reduction is subjective because different musicians have their own perception of harmony. How many notes should be kept from reduction? How playable is the reduced score?

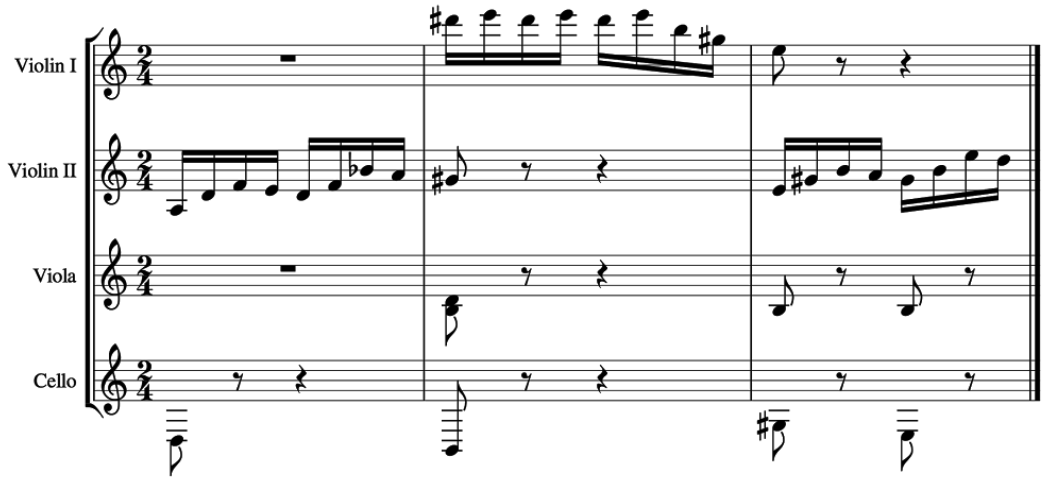


Figure 1.1: Excerpt of String Quartet provided by Prof. Lucas Wong (SQ-Original.xml)

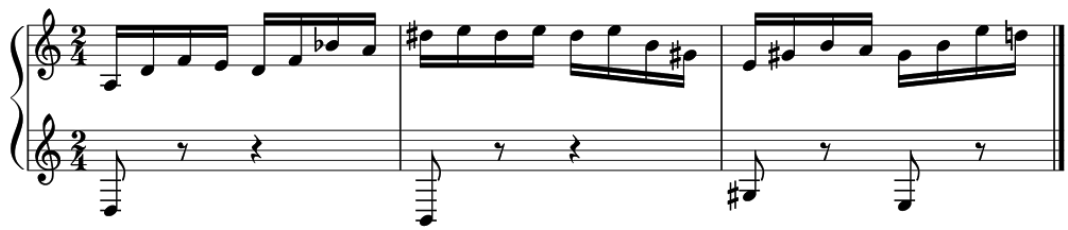


Figure 1.2: Excerpt of piano-reduced string quartet provided by Prof. Lucas Wong (SQ-Important entrances plus bass line 2 (more playable).xml)

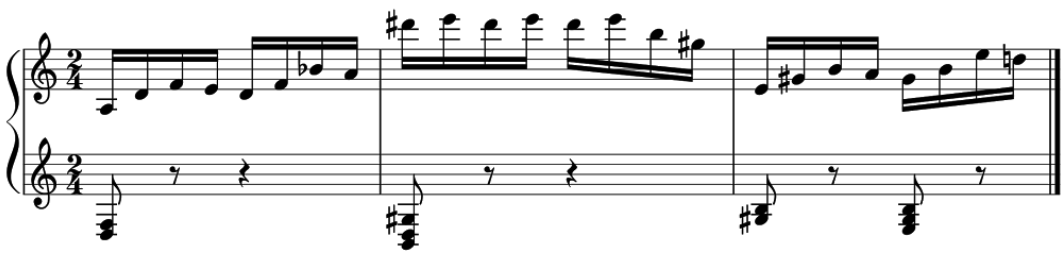


Figure 1.3: Another excerpt of piano-reduced string quartet provided by Prof. Lucas Wong with more harmonic composition.
(SQ-Important entrances plus bass line and harmonic skeleton 1.xml)

Figure 1.2 and 1.3 are similar in terms of melody considering the right hand. However, figure 1.3 sound fuller as musicians need to play more notes with their left hands, delivering a fuller sound.

1.3 Objectives

There are several objectives in this semester:

1. Improve chord identification result
2. Apply harmonic analysis to identify music keys, modulations and chord progression
3. Add a new reduction algorithm - dissonance
4. Present results by embedding labels and marking dissonances in output MusicXML scores

1.4 Music Characteristics

1.4.1 Pitch Class

The most common tuning system remains as twelve tone equal temperament for classical music and western music, which means music can be classified as twelve different classes of pitch. See the figure below for reference

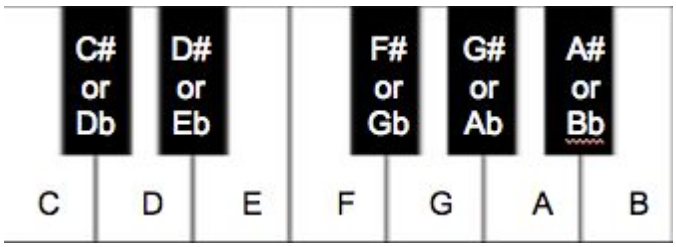


Figure 1.1: Piano keys and pitch classes of an octave

Mathematically, a single pitch class can be represented as the following.

$$C_n \text{ where } n \text{ is an integer}$$

C_n is a set containing all pitches with different octave but sharing the same quality as C. Other pitch classes can be derived similarly as C.

In modern piano, there are 88 keys with 52 being white and 36 being black. Piano is one of the most instruments with large designated pitch range. The lowest pitch is A_0 while the highest pitch is C_8. Therefore, using the pitch range of piano is sufficient for music analysis because the pitch ranges of most of the instruments fall within that of the piano. For example, a typical string family from the violin, the highest one in terms of pitch range to the double bass, the lowest one still can be covered by piano.

1.4.2 Chords

For music, it mostly involves multiple voices or instruments playing together, the simultaneous sound of two or more notes constitutes a musical harmony which is called chords. Different compositions of notes spawn different chords with unique acoustic effects. For technical details on how to form chords with algorithms, see section 2.3.

1.4.3 Chord Inversion

Chord inversion inverts chords, rearrange the pitch order of how chords are played. For example, a C major chord is composed of C,E,G. If E becomes the bass note (E,G,C), the chord is in first inversion. If G becomes the bass note (G,C,E), the chord is in second inversion.

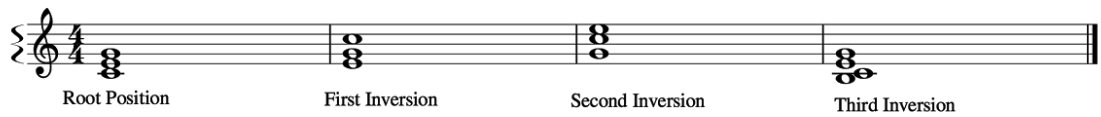


Figure 1.2: the root position and three inversions of C major/major 7th chord.

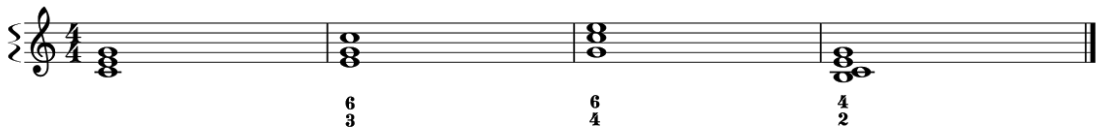


Figure 1.3: The figured bass notation of inversions.
Since full English description is clumsy in music score, inversions are displayed in the form of figured bass for better readability to musicians.

1.4.4 Chord Voicing

Chord voicing is the way of how one distributes music notes among the various instruments. To put simply, chord voicing concerns about whether notes are being played in the same octave or in different octave. Voicing typically has two position: Close Position and Open Position.

For instance, C major chord is composed of C,E,G. In close position, C major is played as C4E4G4, which all notes belong to the same octave. While in open position, C major can be played as C4E5G4, C5E4G6 and so on given that they do not belong to the same octave.

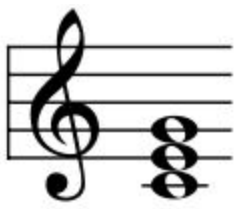


Figure 1.4
C major chord, close position

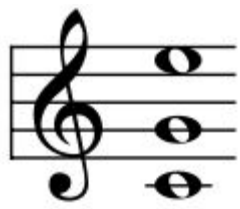


Figure 1.5
C major chord, open position

1.4.5 Roman Numeral System

Roman numeral system bestows chords the “meaning” by using roman number (I,II,III,IV). It is also referred as scale degree in music. The key idea is to denotes a series of chords that are related to a particular music key. It is important because if we want to have a better picture of a music score, their music key must be known.

I (major triad)	C E G (C major)
II (minor triad)	D A F (D minor)
III (minor triad)	E G B (E minor)
IV (major triad)	F A C (F major)
V (major triad)	G B D (G major)
VI (minor triad)	A C E (A minor)
VII (diminished triad)	B D F (B diminished)

Table 1.6: Simplified roman numeral table in C major for easy understanding.

By relating chords with roman number, they become a good analyzing tools because they can be categorized into the following functions for both composers and music analyzers.

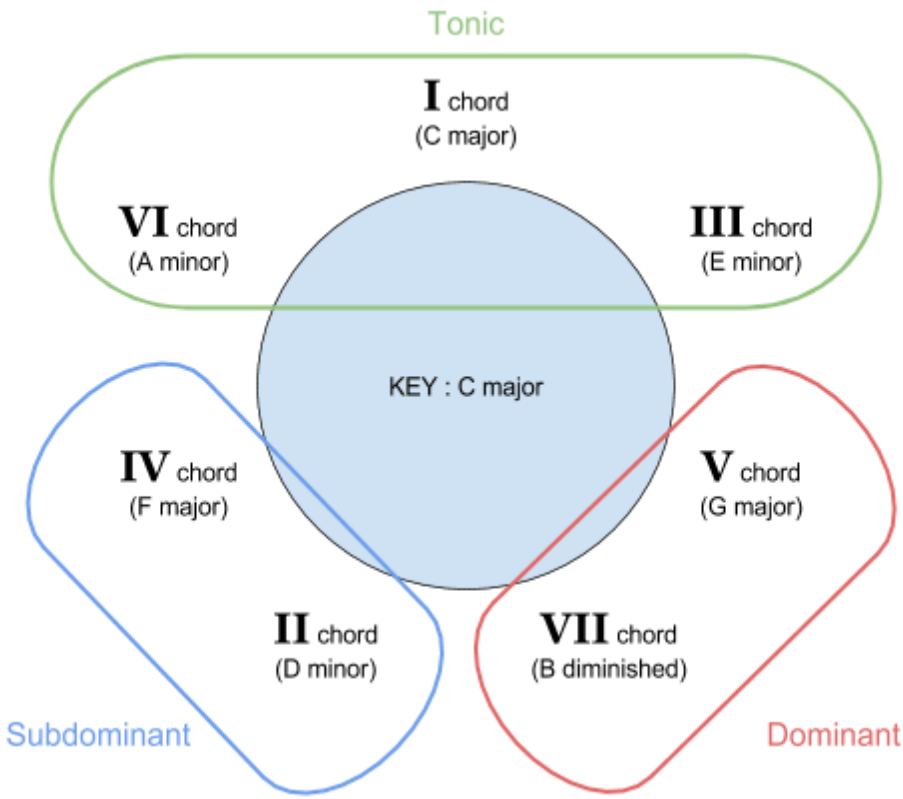


Figure 1.7: The categorization of roman numbers

From figure 1.7, there are three main categories with three different functionalities.
1. Tonic (most important): Chords belong to this category deliver most stable tone as they have the "homecoming" characteristics when played. In most cases, they serve as the ending chord of piece.

I chords are tonic as it is the root chord and its scale degree is also called tonic.
VI chords are tonic as well since their inversion counterpart shares same two notes in C major chord.

I (**C, E, G**) VI6 (**C, E**, A)

Although III chord is less similar because it does not contain C, it is still considered as tonic category.

I (**C, E, G**) III (**E, G**, B)

2. Dominant (secondly important): Chords in Dominant category deliver most unstable tone. These chords are to create "tension" in music and they often require tonic chords to resolve.
V and VII chords belong to Dominant group as they share two same notes.

V (G, **B**, **D**) VII (**B**, **D**, F)

3. Subdominant (less important): Chords in Subdominant create less "tension" than dominants. Therefore, it mostly serves as passing chords between other chords.
II and IV chords are subdominant because of its likeliness.

II6 (**F, A**, D) IV(**F, A**, C)

1.4.6 Modulations

Modulation means the process of changing a key. There are several types of modulations. However, for easy understanding, only the most general type of modulation - **Common-chord Modulation** is introduced.
Intuitively, modulation by common-chord means changing key across a common chord that both keys share.

Key: D	I D major	ii E minor	ii F# minor	IV G major	V A major	vi B minor	vii C# dim
Key: G	V D major	vi E minor	vii F# dim	I G major	ii A minor	iii B minor	IV C major

Table 1.8: Table showing the common chords of two music keys

From table 1.8, there are four chords (highlighted in yellow) that can be acted as "pivots" for modulation.
For example, consider the chord progression below,

	D	A	D	A	D	G
D Major	I	V	I	V	I	
G Major					V	I

Table 1.9: Example of a modulation from D major to G major

From table 1.9, The first four chords constitute D major key. For the fifth chord, it is a ‘pivot’ of D major and G major chord. It can be I chord in D major or V chord in G major. For the sixth chord, it modulates into G major, continuing as I chord in G major.

Modulations are very common in classical music because it allows many possibilities in terms of music quality, introducing different moods and tension. The music would sound boring and dull if there is only one music key throughout the music.

1.5 Computer Technologies and Tools Used

1. Python 3.4.4: main computer language for programming.
2. music21: a toolkit for computer-aided musicology. It is used for parsing and retrieving music information of files in MusicXML format.
3. pybrain: machine learning library for Python. It is used in neural network component.
4. Flask: a microframework for web development. It is used to implement backend APIs
5. MusicXML: a standard format for exchanging digital sheet music.
6. Weka: open source data mining software. It is used to train a weight table of chord flows for progression and modulation recognition.
7. MuseScore: open source music notation software for viewing and editing MusicXML files.

2 Previous Works

2.1 Works from Previous Years

2.1.1 User-Adjustable Automatic Piano Reduction (KY1302)

In final year project, KY1302 User-adjustable automatic piano reduction by then-undergraduate student NG Chiu Yuen, she has successfully created a piano reduction tool with machine learning component.
Program pipeline:

1. Using eleven reduction algorithms to perform marking of music notes both in training input samples and training output samples.
2. Perform training on multi-layer neural network.
3. Output reduced score in MusicXML format.

The eleven reduction algorithms are modularized code that approximate how a composer percepts a music score when doing music reduction.
Further Details on automatic piano reduction are illustrated in Section 5.

2.2 Works by Us

2.2.1 Chord Identification - Previous Semester

In previous semester, we have done a basic chord identification program. Report in the previous semester has explained the principles of chord identification and key identification and how to represent them in appropriate data structures.
The summary is as follows:

1. Exact chord matching of chord identification: The program can identify the right chord when the measure has explicitly ring that chord. For example, if a measure has an obvious sounding of a C major chord (CEG), the chance of getting identified is very high due to the exact chord matching algorithm. However, for measures without an obvious harmony, the program cannot identify as partial matching for chords is yet implemented.
2. Key identification using statistical approach: Deduce a key by counting the occurrences of chords belong to the same key. However, it is problematic in the sense of music as there is no direct relation between extensive use of some particular chords and their keys. For example, in the key of D minor, it does not necessarily mean there must be an extensive use of D minor chord.

2.2.2 Music Rules Library (lib.py) - Previous Semester

Music Rules Library is a python package for any low-level music computations. It is designed to minimizing time complexity on performance because of its high usage. There are two components in the library. “Interval” is to handle music pitch comparisons, define music intervals and define chords with music intervals. “MusicManager” is for defining music chords, chord progressions and its methods for building chord database.

2.2.2.1 Interval

2.2.2.1.1 Constants

2.2.2.1.1.1 Music Interval

All music intervals are defined as constants.
<interval_name> = [<natural_steps>,<half_steps>]
natural_steps are interval steps on seven tones, i.e. all white keys on piano.
half_steps are interval steps between two notes on twelve tones, i.e. considering all pitch classes including accidentals.

perfect_unison = [0, 0] minor_second = [1, 1] major_second = [1, 2] minor_third = [2, 3] major_third = [2, 4] perfect_fourth = [3, 5] augmented_fourth = [3, 6] diminished_fifth = [4, 6]	perfect_fifth = [4, 7] minor_sixth = [5, 8] major_sixth = [5, 9] augmented_sixth = [5, 10] diminished_seventh = [6, 9] minor_seventh = [6, 10] major_seventh = [6, 11] perfect_octave = [7, 12]

Table 2.1: Music intervals defined in library

For example, from table 2.1, in minor_third = [2, 3] <= 2 natural steps, 3 half-steps
By counting the half-steps from C, the pitch is either D# or Eb, thus raising the enharmonic equivalent problems. Natural steps here come into play.
The usage of natural_steps is to find the right white key.
Therefore, by counting natural steps (or the white keys on piano) from C, it is E and eliminate D# as the pitch. As a result, the minor_third of C must be Eb, but not D#.

2.2.2.1.1.2 Construction of Chords

Music Interval constants described in 2.3.1.1.1 have solved enharmonic equivalent issues and chords can be built upon several music intervals.

Major Triad	perfect_unison	major_third	perfect_fifth	
Minor Triad	perfect_unison	minor_third	perfect_fifth	
Major 7th	perfect_unison	major_third	perfect_fifth	major_seventh
Minor 7th	perfect_unison	minor_third	perfect_fifth	minor_seventh
Dominant 7th	perfect_unison	major_third	perfect_fifth	minor_seventh
German 6th	perfect_unison	major_third	perfect_fifth	augmented_sixth
French 6th	perfect_unison	major_third	augmented_fourth	augmented_sixth
Italian 6th	perfect_unison	major_third	augmented_sixth	
Minor Triad b5	perfect_unison	minor_third	diminished_fifth	
Half Diminished 7th	perfect_unison	minor_third	diminished_fifth	minor_seventh
Diminished 7th	perfect_unison	minor_third	diminished_fifth	diminished_seventh

Table 2.2: Chords defined in library

If there are more chords needed in the future, new chords can be defined easily with a composition of several music intervals.

2.2.2.1.2 Methods

This component has two methods: **LetterToPitchNumber**() and **GetIntervalNote**().

method **LetterToPitchNumber**() takes a pitch class as input and output a number. Pitch classes are not suitable for comparisons because of their enharmonic qualities. Therefore, by converting into a number, we can do comparisons just by their numeric differences.

```
Procedure LetterToPitchNumber(String pitchClass)
        return pitchNumberDict[pitchClass]
```

method **GetIntervalNote**() returns a pitch class encapsulated with music21.note.Note object based on music interval. It takes music21.note.Note object as input.

```
Procedure GetIntervalNote(music21.note.Note note, int[] interval)
        targetPitchClassNoAccidentals = get the target note without accidentals by adding interval[0] (neutral) steps
        // Get the pitch number of target note without accidentals
        targetPitchNumNoAcc = LetterToPitchNumber(targetPitchClassNoAccidentals)
        // Get the pitch number of target note
        targetPitchNum = (note.pitch.pitchClass + interval[1]) % 12
        intervalDifference = targetPitchNum - targetPitchNumNoAcc

        accidentals can be assigned to target note based on the difference.
        {-2 : 'bb', -1 : 'b', 1 : '#', 2 : '##'}

        targetNote = music21.note.Note(targetPitchClass)
        return targetNote
```

2.2.2.2 MusicManager

MusicManager is a singleton focusing on doing batch operations on building music database.

2.2.2.2.1 Constants

To build a chord database, chord progression rules must be defined so that only right chords are constructed with a particular music key for the database.

roman numbers	music interval (from Interval)	chord (from Interval)
I	perfect_unison	Major Triad
bII	minor_second	Major Triad
II	major_second	Minor Triad
II7	major_second	Minor 7th
III	major_third	Minor Triad
IV	perfect_fourth	Major Triad
V	perfect_fifth	Major Triad
V7	perfect_fifth	Dominant 7th
bVI	minor_sixth	Major Triad
gerVI	minor_sixth	German 6th
freVI	minor_sixth	French 6th
itaVI	minor_sixth	Italian 6th
VI	major_sixth	Minor Triad
VII	major_seventh	Minor Triad b5
dimVII7	major_seventh	Diminished 7th

Table 2.3: major_chord_progressions: dictionary mapping of roman numbers with music intervals and chords in major key

For example, in the key of C, if the target roman chord is V7. First, get the target pitch class by intervals. Since it is perfect_fifth, the pitch class is G. According to table 2.3, the chord of V7 is dominant 7th chord, which according to table 2.2, goes with intervals perfect_unison, major_third, perfect_fifth, minor_seventh.
By comparing these intervals using methods in Interval (Section 2.3.1.2), a G dominant 7th chord (GBDA) can be obtained.

2.2.2.2.2 Methods

1) **makeTable**(String pitchClass, bool isMajor=true)

This method returns a dictionary with a list of pitch classes as dictionary key and roman numerals as value.

```
Procedure makeTable(String pitchClass, bool isMajor=true)
        // init a dictionary
        chord_table = {}
        // create music21.note.Note object for the pitchClass
        noteObj = music21.note.Note(pitchClass)

        if (isMajor) {
                for each romanNumber in major_chord_progressions
                        chord = Find the romanNumber corresponding notes and build chord
                        chord_table[chord] = romanNumber

        } else {
                for each romanNumber in minor_chord_progressions
                        chord = Find the romanNumber corresponding notes and build chord
                        chord_table[chord] = romanNumber
        }
```

2) **makeTableFull**()

This method is just an extension of **makeTable**(). Instead of taking a ***pitchClass*** as input, it takes all ***pitchClasses*** and return a list of dictionary for each pitch classes.

3 Overall System

The system is a combination of the work by student NG Chiu Yuen in final year project KY1302 in 2013 and the work in this year. It is developed to generate piano reduction automatically. It is composed of three individual components.

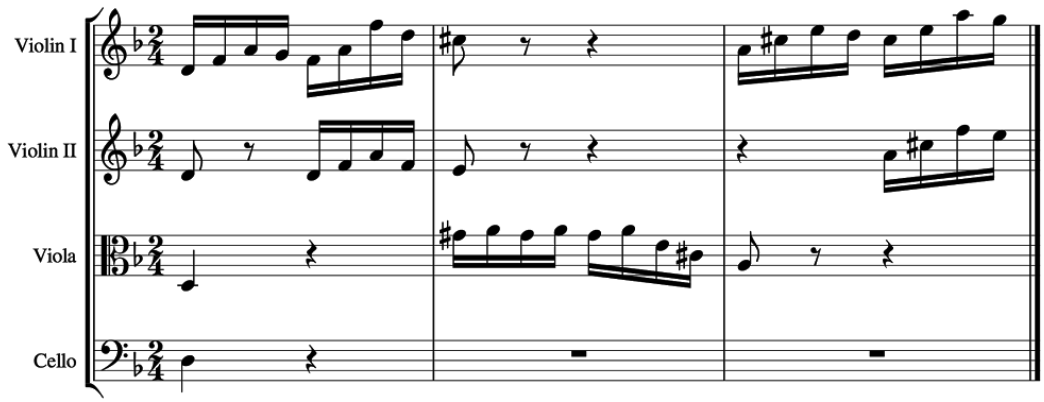


Figure 3.1: three measures of a string quartet music

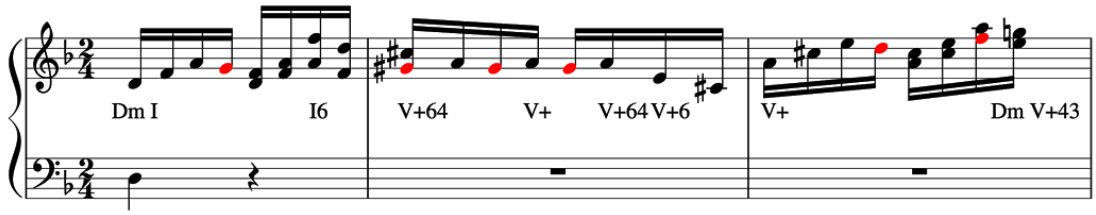


Figure 3.2: three measures of a piano-reduced string quartet music

An example usage of the system is described as follows:

1. Figure 3.1 shows an excerpt with four string instruments. When it is input to the system, the score is analyzed first.
2. Note information is extracted and recognized by **Chord Identification** component (see section 3.1.1 for introduction and section 4 for technical details).
3. After chord recognition, the chord data are applied to ***HarmonicAnalyzer*** (see section 6 for technical details) for music key and modulation recognition.
4. The system then marks all dissonance of a piece and put the dissonance data as marking in the **Neural Network** for automatic piano reduction (see section 5 for technical details).
5. After training, a reduced score object is generated.
6. Before output as MusicXML file, modulation data are converted into readable text and embed into the score object. Moreover, all dissonances are marked as red color (see section 4.3.3 for details). Finally, a MusicXML file is generated and shown on user-preferred score reader.

As the system is composed of three modularized components, all components can be used individually. For example, one can just add modulations and roman labels in one's target score without doing piano reduction.

3.1 Individual components

Each individual component provides essential function in the whole project. A brief introduction of usage is illustrated below.

3.1.1 Chord Identification

It is a component used to handle the main objective of this year final year project. It is used to identify chord in both pitch and roman numerals. It consists of ***EventAnalyzer*** and the ***Postprocessor***. The details of this chord identification including its feature, mechanism and implementation are illustrated in Section 4.

3.1.1.1 EventAnalyzer

It is the main component of It is responsible to identify chord, pitch and roman numerals of each event in the score. Its mechanism and implementation details are illustrated in Section 4.3.1 and Section 4.3.2.

3.1.1.2 Postprocessor

It is the side component for the chord identification. It is used to meet the fourth objective, present results by coloring and embedding labels in output .xml scores. Its implementation details are illustrated in Section 4.3.3.

3.1.2 Neural Network

The neural network was developed by then-undergraduate student NG Chiu Yuen in her final year project KY1302 User-adjustable automatic piano reduction. In this year, the neural network has been merged as a component to the whole system.

Based on the developed neural network, we have added a new reduction algorithm - dissonance - which is accomplished by enhancing the dissonance features to this component. The introduction and the implementation of enhanced feature of this neural network are illustrated in Section 5.

3.1.3 HarmonicAnalyzer

The ***HarmonicAnalyzer*** is a side product produced by chord identification. Since the main objective of this year requires identifying the chord with roman numerals, this component is developed to apply harmonic analysis and obtain the roman numeral result. Its feature, mechanism and implementation details are explained in Section 6.

3.2 Relational graph

The following graph is the flow and relation graph of the system. In automatic piano reduction, the input score is inputted to the neural network. It calls the ***EventAnalyzer*** with inputted score and sample scores to obtain the analysis results for training and reduction. The reduction result is passed to the ***Postprocessor***. The ***Postprocessor*** calls the ***EventAnalyzer*** with the input score to obtain the analysis result and use the result to label the reduction result.

In each call of the ***EventAnalyzer***, it calls the ***HarmonicAnalyzer*** for harmonic analysis on the score. However, the ***HarmonicAnalyzer*** only initialize once using the input roman flow weight excel file.

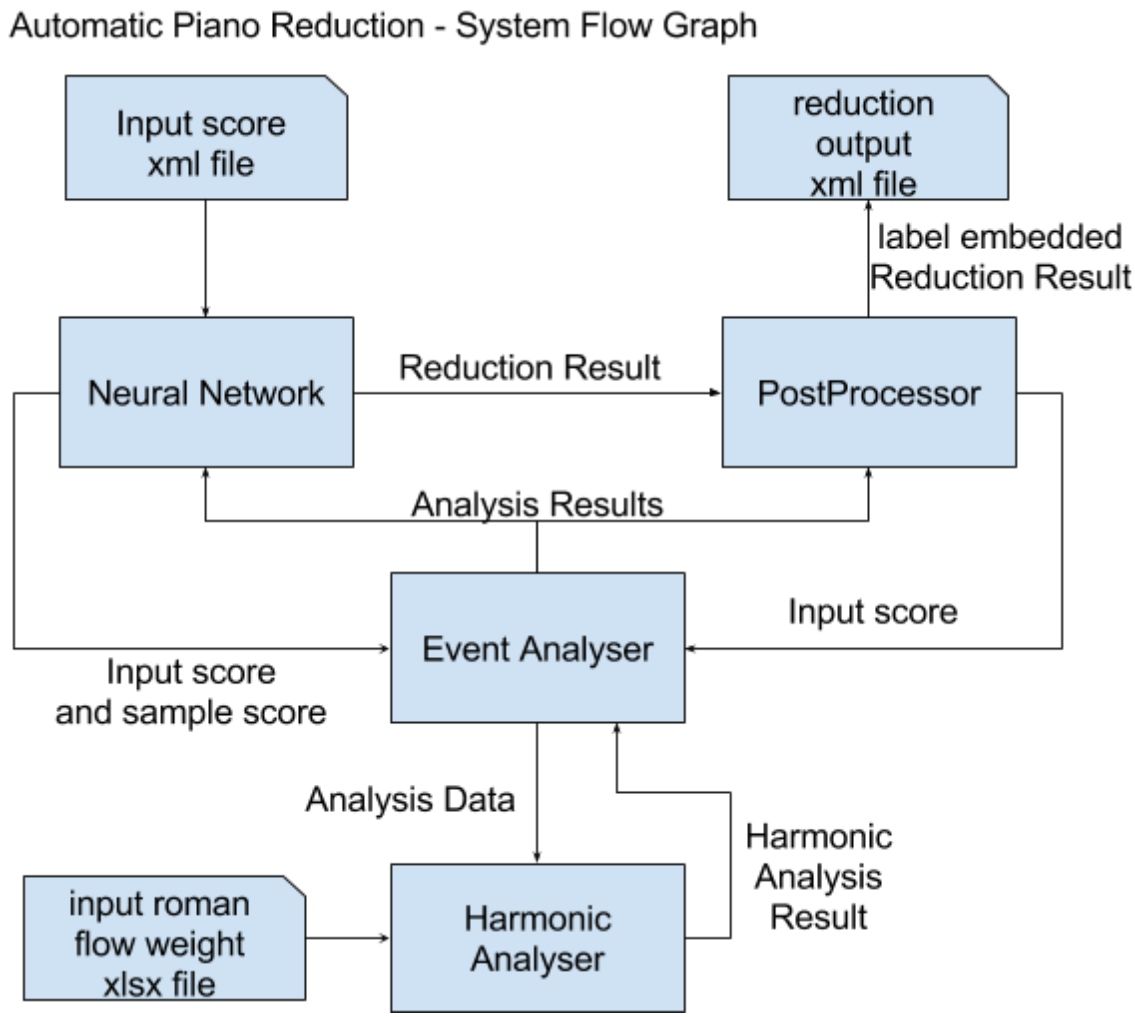


Figure 3.3: Automatic Piano Reduction - System Flow Graph

The simple chord analysis can also be applied in the system with the following flow. The input score is passed to the ***Postprocessor***, which it calls the ***EventAnalyzer*** to retrieve the chord identification results. The ***Postprocessor*** then embeds the label into the score and output to the output xml file.

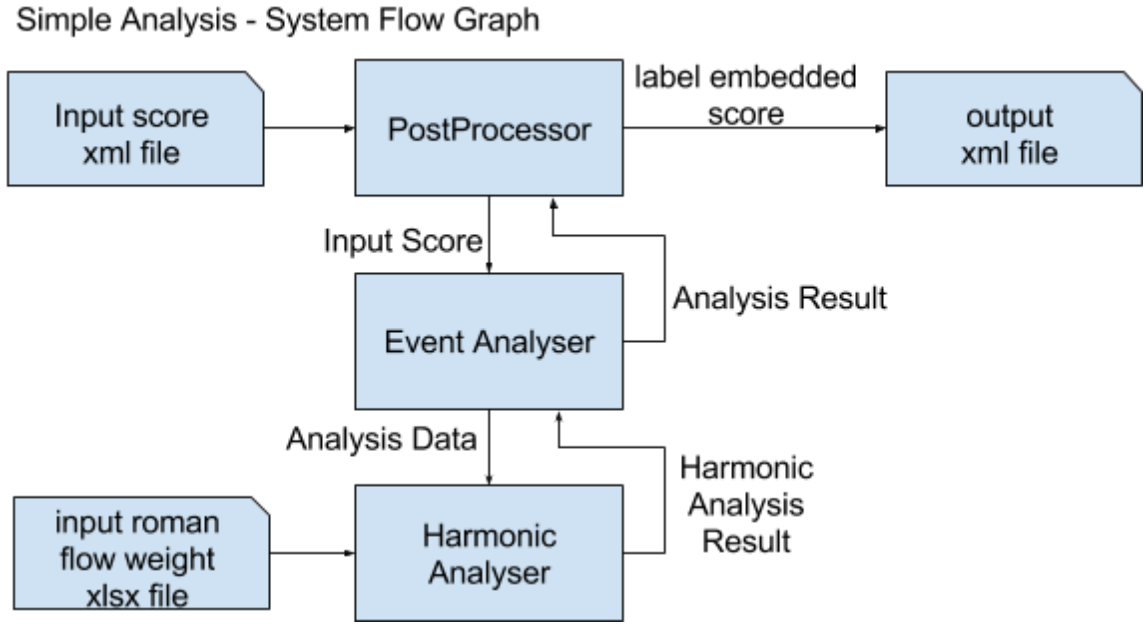


Figure 3.4: Simple Music Analysis - System Flow Graph

4 Chord Identification - EventAnalyzer

The name "***EventAnalyzer***" is about the idea of partitioning a music score into many musical events. Each events contains own pitch data, matched chord and methods to manipulate them from top level which controls how chords are recognized.

4.1 Features

4.1.1 Chord Recognition

Chords recognized are drawn from database in lib.py (see section 2.3 for details). In other words, the recognition results only match with chords belong to the database.

4.1.2 Roman Numerals Recognition and Indication

The Roman numerals are used to denote chords relative to a music key. Therefore, at least one music key must be known before chords can be denoted as roman numerals.
After chords recognized, the program tries to match a music key by a series of chords using chord flow weight generated by ***HarmonicAnalyzer*** (Section 6). The best fit music key with lowest possible modulations are selected as the final results.

The results are then shown in the output score with all music keys and modulations and their respective roman numerals.

4.1.3 Dissonance Recognition and Indication

Dissonance recognition is simple. For each music events, the pitch classes are compared with the matched chord. If there is/are pitch(es) which do(es) not belong to that chord, they are branded as dissonance. In the output score, all dissonance is colored as red.

4.2 Mechanism

This ***EventAnalyzer*** provide chord analysis on the given score. It can be applied to the ***Postprocessor*** for the ***EventAnalyzer*** to display the result on a xml score. It can also be applied to the automatic piano reduction by neural network developed by NG.

The ***EventAnalyzer*** is used on chord recognition. It recognizes chords by limiting the possibility by steps. The mechanism is as follow.

In the first step, the ***EventAnalyzer*** gives a set of possible chords for each event. Then, it asks the ***HarmonicAnalyzer*** to run a progression analysis on these possible chords and return the most possible chord progression for the current score. This chord progression, with two chord, is used to select the chord for each event.

The event selects the possible chord from the ***FromChord*** and ***ToChord*** of the chord progression. It is because it is hard to determine for the current modulation that where the chord change exactly within the progression. The previous chord is also considered as a chord is possibly included from the previous progression.

These results are passed to further resolve the seventh chord, which provide a better solution. Then they are passed to a section that use consecutive dissonance to check whether a chord is collectively recognized. Since in most situation, it is not possible for dissonance appears consequently. The results are also corrected to eliminate these consecutive dissonances.

After that, it requests a progression analysis on the recognized score to obtain the progression data for each event.

4.3 Implementation

The chord identification application ***EventAnalyzer*** can be divided into three important components. The ***EventAnalyzer*** frame, *Chord Recognition* component and ***Postprocessor*** component. These components together with the ***HarmonicAnalyzer***, illustrated in Section 6, provides the features illustrated above. In the following section, the implementation of these three components together with its data structure are explained in details.

4.3.1 EventAnalyzer Frame

4.3.1.1 Implementation and Pseudocode

```
Class EventAnalyzer

EventContainer Score
ProgressionResult ChordProgression

Procedure Analysis(Music21.Score Input, HarmonicAnalyzer Flow)

        //Initialization
        Score = Self.Data_Preparation(Input)

        //PreProcess Part
        For Measure in Score.EventGroups
                Self.Acoustic_Lasting(Measure, Flow)
                ChordRecognition.Match_Possible_Chord(Measure)

        //Progression Analyst Part
        ChordRecognition.Remove_No_Flow_Chord(Flow, Score)
        ChordProgressions = Flow.Progression_Analysis(EventContainer)

        //Chord Identification Part
        For Measure in Score.EventGroups
                ChordRecognition.Chord_Selection_With_Progression(ChordProgressions)
                ChordRecognition.Resolve_Seventh_Chord(Flow)
                ChordRecognition.Resolve_Chord_Based_on_Dissonance(Flow)

        ChordProgressions = Flow.Progression_Analyst(Score)

End of Procedure
```

In the initialization, the ***EventAnalyzer*** with prepare the data from music21 score object into the ***EventContainer*** structure using **Data_Preparation**() and the reference of ***HarmonicAnalyzer***.

Then in the Analysis Procedure, for each Measure, it processes through the **Acoustic_Lasting**() Function. This function prepares corrected pitch class for **Match_Possible_Chord**() in *ChordRecognition* (see section 4.3.2.1) component to obtain the possible chord for each event.

Then, it comes to the progression part. First, it removes chord which have zero possible flow in all music key to the chord in the next events with the **Remove_No_Flow_Chord**() in

ChordRecognition component. Then, it undergoes **Progression_Analysis**() using ***HarmonicAnalyzer***.

Finally, it uses the progression and flow weight table to undergo the chord identification process. For each Measure, it undergoes the process of **Chord_Selection_With_Progression**(), **Resolve_Seventh_Chord**() and **Resolve_Chord_Based_on_Dissonance**() in *ChordRecognition*, which determines the chord and dissonances of each Event.

4.3.1.2 Sub-Components

4.3.1.2.1 Data Preparation (Completed in previous semester)

Data_Preparation() is used to prepare music data for the ***EventAnalyzer***.
The music data is composed of Event object, which is the basic unit of the music data. The whole music score is partitioned into Event object by smallest possible music interval.

The **Data_Preparation**() component takes music21.score object as input to transform its own structure to our custom structure managed by ***EventContainer***, which contains a number of groups of music events. The reasons for changing into a customer structure are 1) The idea of “events” as basic unit is different from music21’s structure. music21 structures the score as a series of offset or music beats. 2) By conforming to our structure, only information that proves useful are extracted, thus reducing initialization time. It is because music21 structures are complicated with many properties and multi-layered. More time is needed to traverse the structure. If a custom structure is used instead, the time to traverse the new structure reduced massively.

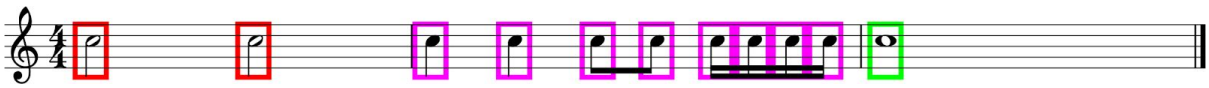


Figure 4.1: Excerpt showing how a score is sliced into events.

Each mark denotes a single event. Events within the same measure form a group. (Highlighted by their different colours) In measure 1 (Red), since there are only two half notes, this group has two events. In measure 2 (purple), there are eight notes and this group has eight events. The last one is a full notes, thus forming a group itself.

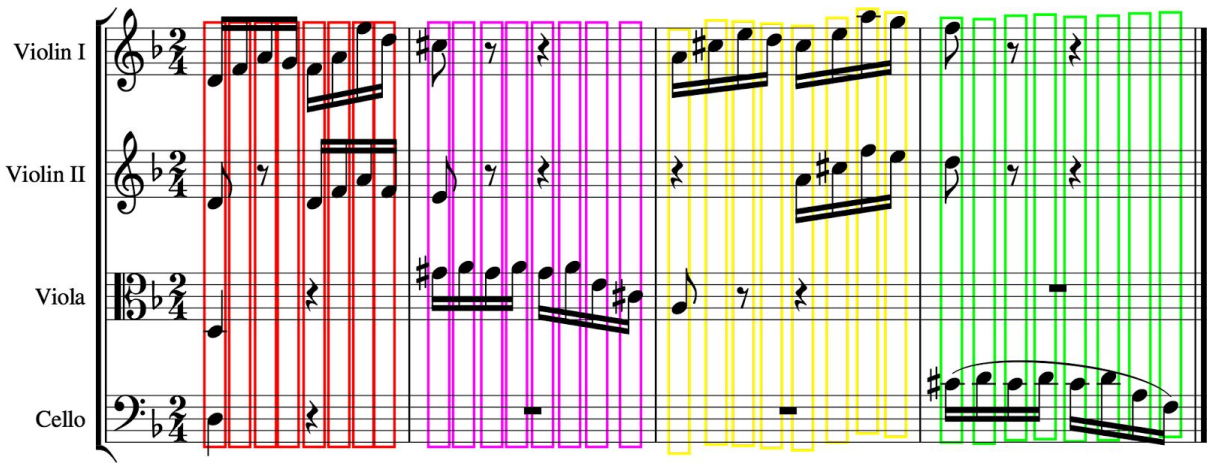


Figure 4.2: An extension example following figure 4.1

There are four measures, meaning four groups of events i.e. four ***EventGroup*** objects. Each group has eight events, denoted by the rectangular marks in figure 2.

The whole structure goes like:

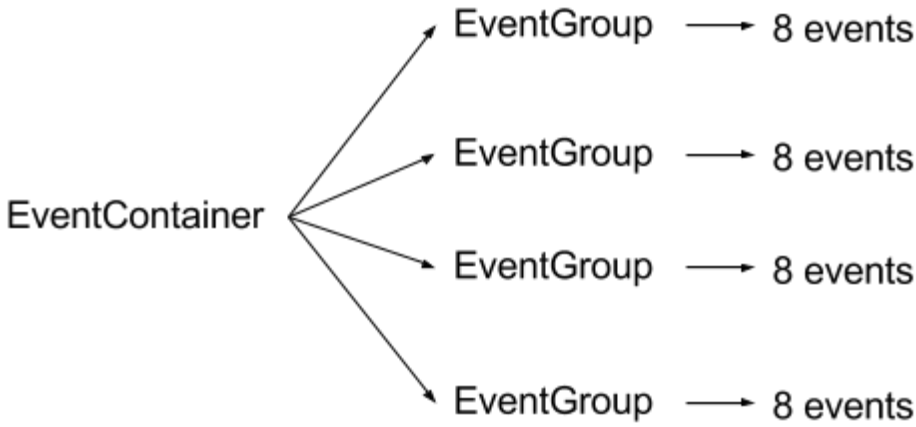


Figure 4.3: Strcuture of score in figure 2

The purpose of ***EventContainer*** object is to contain all event group and events, and provides methods to manipulate events.

For more details of data structure implemented for the EventAnalyzer, see section 3.2.4

4.3.1.2.2 Acoustic Sustaining Effect

Acoustic sustaining simulates how human percepts music.

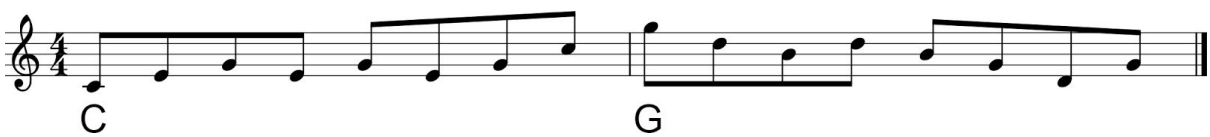


Figure 4.4: An excerpt with two arpeggios chords

The figure shows two measures with two chords broken into a series of notes. Human percepts these notes as a chunk rather than one by one. When we hear the first note and then second note. The memory of the first note still exists while the second note is heard. As this kind of acoustic memory goes on, the sound would eventually form a chord in our mind.

For example, when a musician hears notes in measure 1, the musician would know it is a C major chord, not just a series of notes. Since machine is different from human in terms of understanding music harmony, it is feasible to simulate this kind of acoustic effect for machine to imitate how musician percepts music.

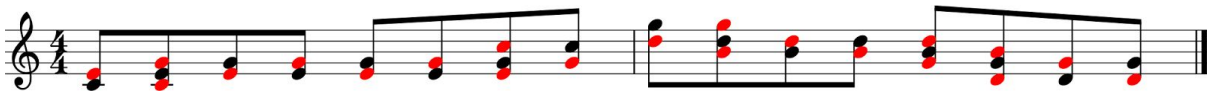


Figure 4.5: Same excerpt of figure 4.4 but with acoustic sustaining effect turned on

Figure 4.5 above shows how computer can imitate human. For each music event, add pitches from previous and next events. This way can strengthen the harmonic structure of a music. The score becomes a series of chunk of notes. Thus, computers can imitate a musician to analyze music.

4.3.2 Chord Recognition Component

4.3.2.1 Match Possible Chord Sub-component

This sub-component is used mainly to provide chord suggestions to measure and events. It provides all chord possibilities using simple rules for other different sub-components to further process, yielding the best results. It used three criteria to match possible chord. From the most restricted rule, **Match_Exact_Chord**() to the less restricted rule, **Match_Chord_With_Peer_Chord**() and the least restricted rule, **Match_Possible_Chord_Share_Part_Of_Notes**() in the Event are used. These rules may not include all the possible chord as wrong chords may exist.

In implementation, it provides chord matching using the hashed index of the Flow Weight Table in the ***HarmonicAnalyzer*** in order to minimize the time complexity of matching a chord.

4.3.2.1.1 Match Exact Chord

As a chord is formed by harmonic, the harmonic formed by the pitch class or by the corrected pitch class with acoustic lasting effect can immediately recognize a chord for the event with a high correctness.

By using this mechanism, it matches the chord using the pitch class and the corrected pitch class in the event if and only if one or both of these notes set can form a chord independently.

To handle inversion of a chord, the chord with different arrangement, it can provide matching on any permutation of these note set. Besides that, in case for the seventh chord, a chord formed by four notes, it is possible that one of the note is dissonance and excluded from the chord. As a result, it also provides all the triad chords as matched chord, that can be included into the seventh chord.

4.3.2.1.2 Match Chord with Peer Chords

It is common that the notes in the current events contribute part of the chord matched in other event in the measure. As a result, the possible chords matched in other event becomes a hint for the current event.

By using this mechanism, it matches the chord for an event if the chords matched in the same measure can provide a high similarity to the corrected pitch class in the measure. All the triad chord recognized in the same measure are added to the candidate list of the measure.

In action, it examines through the candidate list of the measure of the current event. If there is two common note shared by the corrected pitch class and chords in the candidate list, these chords are the possible chords of the current event.

4.3.2.1.3 Match Possible Chord Share Part of Notes

If there is no possible chord for the current event, this type of matching is going to be the last resort for the event. Since a chord is the harmony of the current event, it is obvious that these notes should take the majority in the corrected pitch class.

By using this mechanism, it matches all possible chords sharing two notes with the corrected pitch class. This is the least restricted rule can be figured out to limit the possible chords.

To prevent loss of linking of the chord when exact chord matched between the match. A further Match Chord with Peer Chord action is gone through again for all of the events even some events has got an exact chord matched.

4.3.2.1.4 Matching Flow and Example

The matching step is the following:

1. **Match_Exact_Chord**() for all the events in the measure
2. **Match_Chord_With_Peer_Chord**() using the result of **Match_Exact_Chord**()
3. **Match_Possible_Chord_Share_Part_Of_Notes**() for all the events in the measure
4. **Match_Chord_With_Peer_Chord**() using the result of **Match_Possible_Chord_Share_Part_Of_Notes**()

The example is the following: Step 1 , Step2, Step3, Step4
SQ-Original-Fixed Measure: 15
Event: 102 Pitches: ACD Corrected Pitches: DAC
Event: 103 Pitches: D Corrected Pitches: DG#ABC
Event: 104 Pitches: G#BD Corrected Pitches: DG#ABC
Event: 105 Pitches: ACD Corrected Pitches: G#ABCD

Measure: 15 Possible Chord Result:
Candidate Chord ['G#,B,D', 'A,C,E', 'D,A,F', 'D,A,F#', 'F,C,A', 'F#,C,A']

Event: 102 Possible Chord: ['ACE', 'DAF', 'DAF#', 'ACF', 'ACF#']
Event: 103 Possible Chord: ['DG#B', 'ACE', 'DAF', 'DAF#', 'ACF', 'ACF#']
Event: 104 Possible Chord: ['DG#B', 'ACE', 'DAF', 'DAF#', 'ACF', 'ACF#']
Event: 105 Possible Chord: ['G#BD', 'ACE', 'ADF', 'ADF#', 'ACF', 'ACF#']

The correct chord is ['D,F#,A,C'] for these events. Although it does not exist in the possible chord set, these rules is together successful to include its trial form ['DAF#']. The correct chord can be further fetched.

4.3.2.2 Remove No Flow Chord Sub Component

The chord flow is the transition between one chord to the chord. This chord flow is identified using music key and its roman numerals. In most situation in classical music, there exists a chord flow between chord even with modulation. According to the Co-supervisor, most of the music key changes consist of a common chord which is diatonic to both music keys.

By using this mechanism, it can eliminate the chord that there exists no music keys to be diatonic for the chord and all possible chords in the succeeding event.

In implementation, it compares all combinations of chords between two consecutive events using the flow weight table. It removes any chords in the possible matching chord for each event if these chords are unable to obtain a flow weight between all possible chords in its succeeding event.

4.3.2.3 Chord Selection with the help of chord progression

Chord flow is useful for identification of chords. As there must some harmony rules defined in music theory, by using the ***HarmonicAnalyzer****,* we can limit the search for possible chords using the chord flow progression. Since all of the remaining chords can suit in the Harmonic of the score, these chords are acceptable for chord match for the event.

In action, there are three chords, the matched chord for the previous event, the starting chord of the progression and the ending chord of the progression, provided for selection if the event is within a progression. There are four chords for the events between progression.

These chords are examined first with the possible chords in the event. The event selects the matching chord from the examined chords based on their similarity on the pitch class. If

there are more than one chord having the same similarity, the priority of these chord is applied for selection.

The structure of progression is illustrated with details in Section 6.3.2.3.

The example is the following:
SQ-Original-Fixed Measure: 15
Event: 102 Possible Chord: ['ACE', 'DAF', 'DAF#', 'ACF', 'ACF#']
Event: 103 Possible Chord: ['DG#B', 'ACE', 'DAF', 'DAF#', 'ACF', 'ACF#']
Event: 104 Possible Chord: ['DG#B', 'ACE', 'DAF', 'DAF#', 'ACF', 'ACF#']
Event: 105 Possible Chord: ['G#BD', 'ACE', 'ADF', 'ADF#', 'ACF', 'ACF#']

Modulation:
[14, 1.75, 'G,Major', ['EBG'], ['AEC'], 'VI', 'II', 10.9146] ← Only for Event 102
[15, 0.0, 'G,Major', ['AEC'], ['DF#A'], 'II', 'V', 11.4344]

Chord for Selection after examine and Result:
Event: 102 ['ACE', 'ACE', 'DAF#', 'GBE'] ['DAF#]
Event: 103 ['ACE', 'DAF#', 'DAF#'] ['DAF#]
Event: 104 ['ACE', 'DAF#', 'DAF#'] ['DAF#]
Event: 105 ['ACE', 'DAF#', 'DAF#'] ['DAF#]
These chords are listed with prioritized from right to left. The last element has the highest priority.

Since each chord for selection is examined with the possible chord, it is the subset to possible chords except for the previous chord, indicated by purple. This is to ensure the chord recognized for even if there are no possible chord due to the insufficient of data.

The music key in the progression is also be stored to ease the implementation of resolving seventh.

4.3.2.4 Resolve Seventh Chords

Seventh chord and its inversions are also important to the score and to eliminate dissonance. However, they limit the chord progression as it limits the existence of music keys when the additional note acted as dissonance.

In fact, after the progression analyst, it gives enough information to resolve this problem. It is used to resolve the problem of seventh chord and recreate these chords using the music key stored for the current event in the progression.

In the implementation, it finds all the possible seventh chords for the matching chord of the current event. After that, it checks whether the additional note exists in the progression. If the note exists, it checks whether the seventh chord is allowed in the current progression by examine its existence of roman numeral in the current music key.

4.3.2.5 Resolve Chord Based on Dissonance

Since the ***HarmonicAnalyzer*** is based on data mining and statistic. It is possible that the best progression is not selected with the highest flow weight. As a result, this is used to ensure a chord is correctly recognized and correct the result if it is wrongly recognized. Its mechanism is based on the property of dissonance.

Since dissonance is the quality of sound that seems unstable. It is hardly for a dissonance to remain in consecutive events and not resolved. In most situation, a detection of consecutive dissonance means directly to a wrong recognition.

By using this mechanism, it is possible to correct a chord using these consecutive dissonances. First, it creates a set of dissonances for the current event. Then, it checks the previous event for consecutive dissonance. If these dissonances exist, it uses the candidate list in the measure of the chord to find chords that is possible to includes these dissonance. If there exist more than one chord. It checks its similarity with the pitch class and the its priority.

This function can be further developed based on the mechanism that dissonance must be resolved by step. It can provide a better sureness on correctness of chord recognition. Besides that, the priority list can be further examined to provide a better correcting effect.

4.3.3 Postprocessor Component

The ***Postprocessor*** is responsible for embedding the roman numerals and mark the dissonance on the MusicXML file to display the analysis result. It is also used to start the ***EventAnalyzer***.

4.3.3.1 Implementation and Pseudocode

```
Class Postprocessor

EventContainer Score
HarmonicAnalyzer Flow
ProgressionResult Progression

Procedure StartProcess(Music21.Score Input, HarmonicAnalyzer H_A)
        // Initialization
        Flow = H_A
        Event_Analyzer = New EventAnalyzer(Input, Flow)
        Progression = Event_Analyzer.ChordProgression

        // Post Process
        Self.Embed_Roman_Numeral(Input, Progression)
        Self.Mark_Dissonance(Input, Event_Analyzer)
End of Procedure
```

In the ***Postprocessor***, it first runs the ***EventAnalyzer*** on the Music21 Score, then its output is used for Post Processing. It embeds the roman numeral in the xml as lyrics in the **Embed_Roman_Numeral**() Sub-component. Then it marks the dissonance with different color into the xml file using the **Mark_dissonance()** Sub-component.

4.3.3.2 Sub-component

4.3.3.2.1 Embed Roman Numeral

Modulation data are retrieved from ***EventAnalyzer***. By comparing the current music key with chord table. Roman numbers of that music key can be obtained. When enabling this sub-component, it embed the roman numbers as lyrics in the output MusicXML music score by modifying the lyric property of notes from music21 library (music21.note.Note.lyric)

4.3.3.2.2 Dissonance Marking

Dissonance Marking is very simple. For each event that is analyzed in ***EventAnalyzer***, if its pitches do not belong to the matched chord, these pitches are considered as dissonance. When this sub-component is enabled, it changes the color of notes from default color (black) to red in the output MusicXML score by modifying the color property of notes from music21 library. (music21.note.Note.color())

4.4 Testing Conclusion

The detailed testing mechanism and testing result are illustrated in Section 7.

4.5 Possible Usage in the future

4.5.1 Suggestion of Music Analysis

This component serves as a tool to suggest analysis result to musicians. Since it is able to recognize chords with decent correctness, it can be a good helper for musicians to analyze music. For music beginners, the suggestion of music analysis is useful for them to learn music theory.

4.5.2 Indication of Incorrectly Notated Score

This usage was accidently found when an incorrectly notated music piece is input into the chord identification program. The result shows a score with large and abnormal number of dissonance marked in red color due to incorrect notation.

Incorrect notation means notes are notated in wrong enharmonic equivalents. For example, all C# is notated as Db. As music theory treats enharmonic equivalents differently even though they have same pitches, all wrong notated notes are label as dissonances and shown in red. There, when a score with large amount and abnormal numbers of dissonances, it is probably an incorrectly notated score.

5 Automatic Piano Reduction - Neural Network

5.1 Mechanism

The neural network is developed by student NG Chiu Yuen in final year project KY1302 in 2013. The neural network is used to consider whether to reserve or reduce each pitch. The following paragraphs describe her work on the neural network.

The data set is prepared as supervised data set which contains two field, input and output. The numbers of output is 1 and its value is defined by checking the existence of the note in the sample output score. While the input is the marking of the note in the reduction algorithm

At the beginning of the neural network, before building the network, multiple features implemented as algorithm marks all music notes in the score object with a weight. These **Weight** acted as the input for the neural network.

The number of input for the neural network is depends on the numbers of feature user selected for reduction. These features are implemented as reduction algorithms which mark the note object with weights, as a result, it can be turn on or off easily by not adding the reduction algorithm.

The number of hidden layer is one. The length of the hidden layer is twice of the number of input in default. The hidden layer is using sigmoid function. Each node in the hidden layer is also fully connected to all inputs and the only output.

The length of output is one. It is also using sigmoid function. The output result defines whether to keep the target note or not in the reduction process.

Besides that, a bias note is also be added to the network in default. This bias note is fully connected to all layers to prevent the limitation on output and yield a poor fit. The initial weight of each edge between the input layer, hidden layer and output layer is set by random value. The network is randomized before training.

5.2 Feature Enhanced

These features are implemented this year by us, with aim to enhance the result of piano reduction and setup functions that enable usage as backend server APIs.

5.2.1 Dissonance Feature

In this project, we have successfully included *Dissonance* as a new reduction algorithm into the neural network model.
Dissonances are the notes that are not included as part of the chords. Instead of putting chords as input into the network, marking the dissonances conversely still has the same effect to differentiate chords from dissonances.

The dissonance notes recognized by the *EventAnalyzer* in chord identification component are marked as 1 in the input of neural network while the other notes are 0 in default.

5.2.2 Saving Feature

Besides, saving function is implemented for the neural network model. Users can save the network weight and their reduction algorithm parameters and write to a file. The files are used to load the network and the parameters back to the system.
The network weights are saved as .xml file by using API from pybrain while the parameters are saved as JSON format.

For example, a single reduction saves two files.
/saved_reduction_models/<model_name>.json
/saved_reduction_models/<model_name>.xml

5.2.3 Simple Reduction without Machine Learning

This reduction method is developed by the other FYP teams in the student NG Chiu Yuen in final year project KY1302 in 2013. In this year, we provide optional usage by calling this reduction method.
Besides automatic piano reduction using reduction algorithms with neural network, a simple version that disables the machine learning component can be used. It only requires reduction algorithm parameters and a threshold value as input. The threshold value determines the level of whether a note should be kept in the piano-reduced score.

5.2.4 Backend Server API

A backend server is implemented by using python flask. It implements 3 HTTP post methods.

5.2.4.1 Reduction with Sample Reduction Data

The following HTTP POST method is used to for front end to send a request to the backend for Automatic Piano Reduction using Neural Network. The feature can be selected on or off using the URL Params. The model is stored automatically after the reduction using the modelName or the timestamp as the name.

It returns the reduced score of the targetXml in MusicXML format.

HTTP POST method: /learn

URL Params:
OnsetAfterRest = Int [0 or 1]
StrongBeats = Int [0 or 1]
StrongBeatsDivision = Float [0.25 or 0.5 or 1]
ActiveRhythm = Int [0 or 1]
SustainedRhythm = Int[0 or 1]
RhythmVariety = Int [0 or 1]
VerticalDoubling = Int [0 or 1]
Occurrence = Int [0 or 1]
PitchClassStatistics = Int[0 or 1]
BassLine = Int [0 or 1]
EntranceEffect = Int [0 or 1]
Dissonance = Int [0 or 1] (newly added reduction algorithm)
modelName(optional) = String
targetXml = String
sampleInXml = Array of strings of .xml files
sampleOutXml = Array of strings of .xml files

Successful response: The HTTP response from server is the text of a reduced score in MusicXML format after done training on backend server.

5.2.4.2 Reduction with Saved Network Model

This HTTP POST method is used for front end to send a request to the backend for automatic piano reduction using the model stored. Only the targetXml and the modelFile name is required for URL Params.

It returns the reduced score of the targetXml in MusicXML format.

HTTP POST method: /load
URL Params:
targetXml = String
modelFile = String
Successful response: The backend retrieve saved model and deliver the text of a reduced score in MusicXML format as response.

5.2.4.3 Simple Reduction

This HTTP POST method is used for front end to send a request to the backend for simple reduction without using the neural network. The features can be turned on and off using the URL Params. A threshold is required as URL Params for this simple reduction.

It returns the reduced score of the targetXml in MusicXML format.

HTTP POST method: /simple
URL Params:
OnsetAfterRest = Int [0 or 1]
StrongBeats = Int [0 or 1]
StrongBeatsDivision = Float [0.25 or 0.5 or 1]
ActiveRhythm = Int [0 or 1]
SustainedRhythm = Int[0 or 1]
RhythmVariety = Int [0 or 1]
VerticalDoubling = Int [0 or 1]
Occurrence = Int [0 or 1]
PitchClassStatistics = Int[0 or 1]
BassLine = Int [0 or 1]
EntranceEffect = Int [0 or 1]
Dissonance = Int [0 or 1] (newly added reduction algorithm)
threshold = Int
modelName(optional) = String
targetXml = String
Successful response: The HTTP response is the text of a reduced score in MusicXML format without machine learning component involved.

5.3 Result Improvement

The implementation of dissonance shows no improvement in the current situation. It is possible caused by the insufficient of testing result and the wrong chord recognition by the ***EventAnalyzer***. Further work on testing for this section is suggested.

Besides that, it exists a possible idea to include the modulation and roman numeral as input of the neural network. It can provide more information for the neural network to decide whether to keep the note in the reduction or not.

6 HarmonicAnalyzer

6.1 Feature

6.1.1 Chord Progression Analysis

Chord progression analysis is the first feature to recognize a series of roman numerals from chords. The best fit chord progression is selected.

6.1.2 Modulation Analysis

HarmonicAnalyzer is developed to recognize music keys and modulation given a series of chords. It serves as the second stage of music analysis. In section 4, the principles of chord identification is explained and its usage is to recognize chords in the first stage. For ***HarmonicAnalyzer***, it makes use of the recognized chords and try to analyze the harmony from a series of chord to find the best matching music keys and modulations.

6.1.3 Chord Flow WeightTableGenerator

The chord flow weight table is a generated result of data mining by feeding real world samples of music excerpts. It is a 2D table showing the probability of a chord to another chord. Its usage is to assist the modulation analysis component in the ***HarmonicAnalyzer*** in deciding the most possible flows of modulations.

6.2 Mechanism

The ***HarmonicAnalyzer*** is used by ***EventAnalyzer*** on chord progression analysis. This analysis returns progression data, roman numeral data and modulation data after the analysis. Besides, it provides the most feasible chord progression, roman numeral data and modulation data if there are more than one possible match chord for each event. The ***HarmonicAnalyzer*** mechanism is as follows.

The ***HarmonicAnalyzer*** is based on valid flow weight table as a input. Its result can be greatly different according to the flow weight table. This flow weight table is generated by the major and minor Roman Flow Weight Excel.

Considering each possible chord flow as a path. The progression analysis can provide the path with maximum weight on same music key. For example, it can detect there exist a maximum weight flow from G# C# E to A C# E and then to F# A C# with C# minor key. The length of flow traversed for selection of music keys is defined by user with number of measure. This is used to prevent consecutive changing of music keys and local maximum problem.

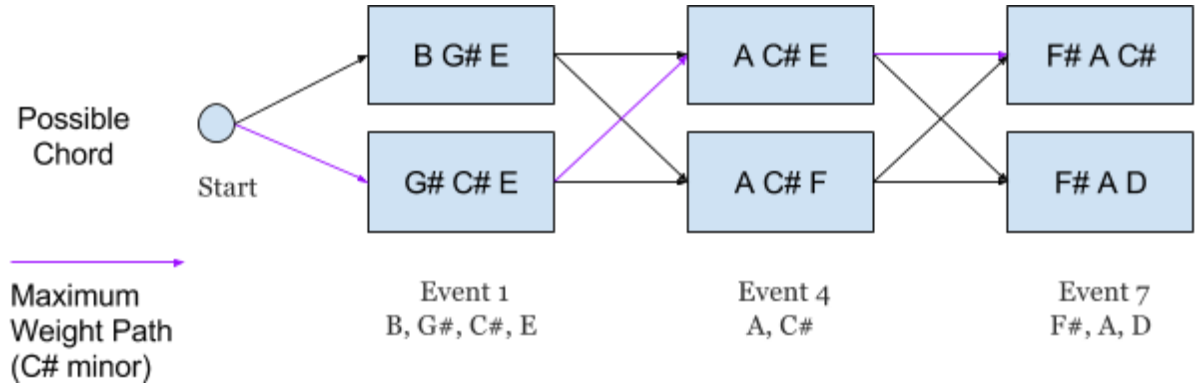


Figure 6.1: Paths of possible modulations 1

It is also possible for modulation, which is the changing of music keys, by setting the tolerance on total weight or if there exist no flow from the current key to the next chord. For example, there are no transition from B G# E to B G E in C# minor key. As a result, it selects B minor as the modulated key.

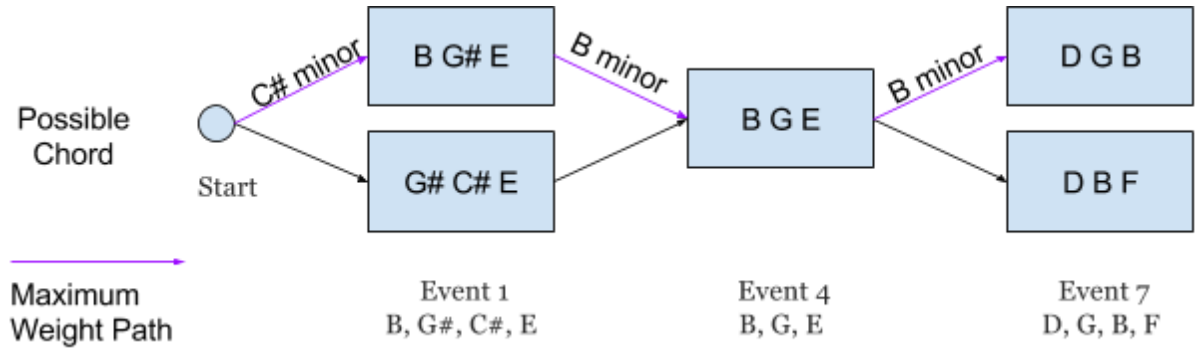


Figure 6.2: Paths of possible modulations 2

Also, it also detects the maximum roman numeral flow if there are numbers of roman numeral selection exist even for the same chord. For example, it can output the chord progression of V to I in C Major and then dimVII to III in D Major based on the flow weight and the tolerance the user input.

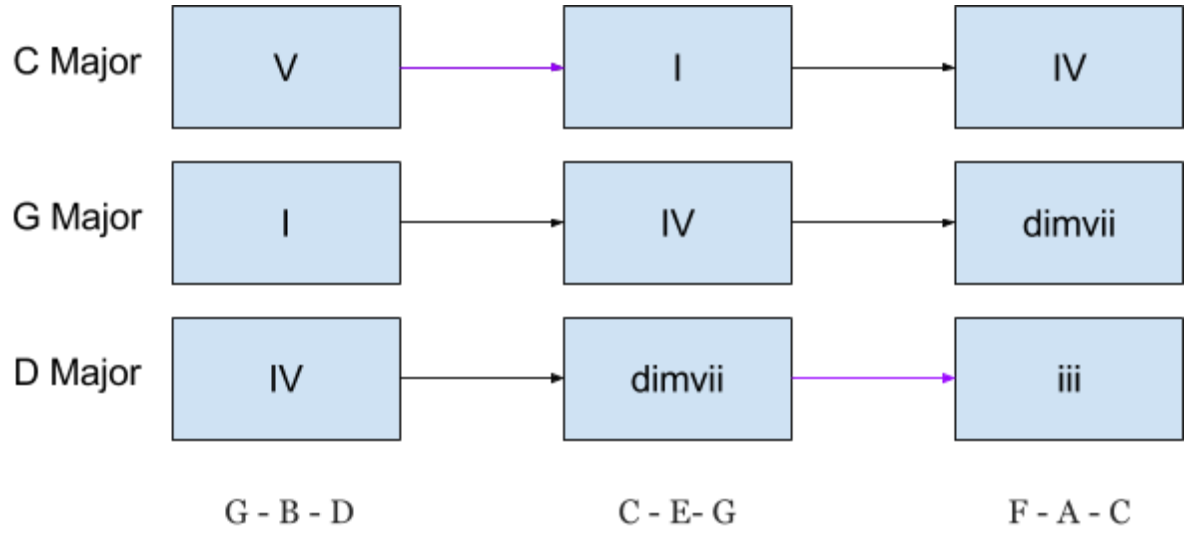


Figure 6.3: three possible keys of chord progressions

6.3 Implementation

The ***HarmonicAnalyzer*** provides harmonic analysis on the target score. It is mainly used in ***EventAnalyzer***. Since it is used to provide harmonic analysis data, the data structure of the ***HarmonicAnalyzer*** is also be provided with details to helps illustrate the performance on time and memory of the ***HarmonicAnalyzer***.

It takes a chord flow weight table in Microsoft Excel format, which defines all possible chord progressions with weights, as an input for harmonic analysis.

The chord flow weight table is generated in ***WeightTableGenerator*** component.
The ***WeightTableGenerator*** component is developed to convert music data (in MIDI) to a human readable text format defined by us, preprocess all text files for data mining tool Weka and output a list of chord flow weights. Finally, the list of chord flow weights are converted into Microsoft Excel format for readability.

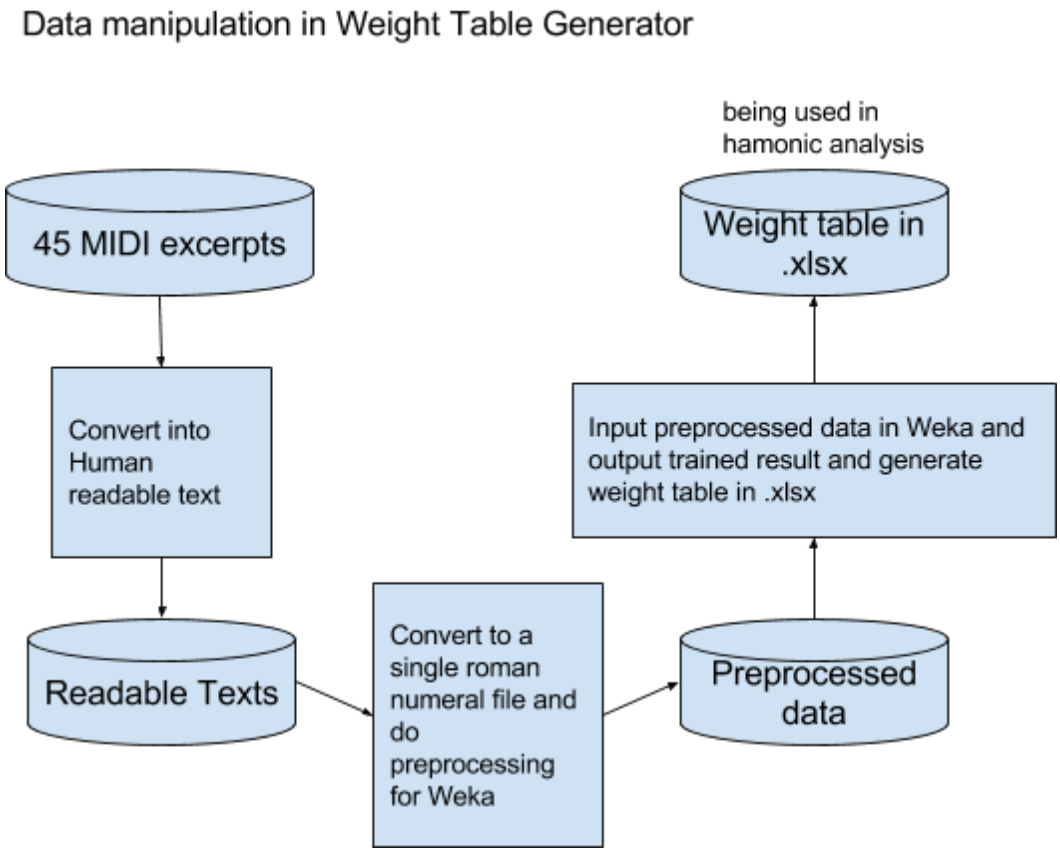


Figure 6.4: Flow graph of data manipulation in WeightTableGenerator component.

6.3.1 HarmonicAnalyzer

The ***HarmonicAnalyzer*** is used mainly on chord progression, roman numeral and modulation analyse together and output a harmonic analysis result. Besides that, it is used also to check for existence of a chord by using the key of the table by the Match Possible Chord Sub-component (Section 4.3.2.1) in ***EventAnalyzer***.

The pseudocode is as follow:

```
import pyexcel_xlsx as XlsxParser // for parsing Microsoft Excel files

Class HarmonicAnalyzer

FlowWeightTable FlowTable

Procedure Initialization(MajorExcelFile MajorFile, MinorExcelFile MinorFile)
        // Data Preparation
        RomanFlowWeightTable MajorTable = XlsxParser.get_data(MajorFile)
        RomanFlowWeightTable MinorTable = XlsxParser.get_data(MinorFile)

        // FlowWeightTable Creation
        Self.Make_Flow_Weight_Table(MajorTable, Major)
        Self.Make_Flow_Weight_Table(MinorTable, Minor)
End of Procedure
```

The ***HarmonicAnalyzer*** first takes two excel file path as input. This two file path is pointed to the xlsx file that contains the roman numeral progression weight for major keys and minor keys. Then the analyzer parses this two file data for data preparation using the python library pyexcel_xlsx.

The data prepared are then passed to the **Make_Flow_Weight_Table**() component to prepare the ***FlowTable***.

6.3.1.1 Make Flow Weight Table

Using our music library, it can generates all the roman chords with a music key.

By using these chords, a map is generated to store all the chord flow by doing two loop and mapping the weight into the map. Since the roman numeral also includes inversion of chord, the permutation is an element to be considered in implementation.

The pseudocode is as follow:

```
Procedure Make_Flow_Weight_Table(RomanFlowWeightTable Table, Quality
quality)
        //Data Preparation
        ChordMap ChordMap = MusicLib.Make_Table_Full(Quality)

        // FlowWeightTable Creation
        For Key in ALL_VARIATION_OF_KEY:
                For FromChord in ChordMap[Key]:
                        For ToChord in ChordMap[Key]:
                                FlowTable.AddIndex(FromChord, ToChord, Table, Key)
End of Procedure
```

First, the list of roman numeral chord of all music keys with the quality is generated to the ***ChordMap*** for data preparation. The chord list can be accessed easily using key.

Then it runs through all variation of key to generate all flow respect to that key. In this process, it runs two loops on ***ChordMap***[Key] to make all the combination of the flow. In the AddIndex function, it looks up the weight of the table and add the index to the ***FlowTable*** with the ***FromChord***, ***ToChord***, ***key*** and the ***weight***.

In implementation of the addIndex, since it is much more efficient to find a flow using hash table, the chord set is converted to a string as hashkey using “,”.join function in python for both the ***FromChord*** and ***ToChord***. This process can be reversed by using String.split(“,”) easily and effectively. To handle the Problem of permutation and inversion. The index with permuted hash key from the same inversion is also added to the ***FlowTable***.

6.3.1.2 Progression Analysis

It acts as a state machine algorithm. The state is the possible chord for each event while the key is the next input. This is used mainly to trace the possible progression to minimize the weight change. Each state has numbers of arrow based on the number of key it is possible to progress to the next state.

The first state and music key are determined by tracing all possible starting state and key. The tracing depth is defined by the user. The state and music key that produce maximum weight among the maximum depth are the output. In each transition, a progression indicated by current chord, next chord, key and weight, is appended to the progression result.

In case for the critical situation that some state has no transition to the other state. Although it have been eliminated by the **Remove_No_Flow_Chord**() Sub Component(Section 4.3.2.2) in ***EventAnalyzer***, as a standalone analyser. It skips the progression and restart it in the next progression.

```
Procedure Progression Analyst(EventContainer Score, Measure=4,
Tolerance=-0.5)
        //Data Preparation
        ProgressionList = DataPreparation(Score, FlowTable)

        // Progression Preprocess:
        Key, NextChord = Get_Max_Progression_Key_Chord(ProgressionList,
                                                       Measure)
        //Progression
        For Progression in ProgressionList:
                If NextChord in Progression:
                        If Key not in Progression[NextChord]:
                        // Modulation Change
                                Weight = 0
                                Key = Get_Max_Progression_Key(ProgressionList,
                                                       Measure,
                                                       NextChord)
                        If Key in Progression[NextChord]:
                        // Progression Appending
                                CurrentProgression = Progression[NextChord][Key]
                                ProgressionResult.Append(NextChord,
                                                CurrentProgression.Chord,
                                                Key,
                                                CurrentProgression.Weight)
                                Weight += CurrentProgression.Weight
                        Else:
                        // Skip Current Progression As No Common Key
                                Key, NextChord =Get_Max_Progression_Key_Chord(
                                                               ProgressionList,
                                                               Measure)
                If Tolerance > Weight:
                        Key = ""

        Return ProgressionResult
End of Procedure
```

In the Progression Analysis, it first take the Score data and ***FlowTable*** and generate the possible Progression in the **DataPreparation**(). It produces the data structure ease the state machine approach. In each ***Progression***, it contains lists with starting chord as index. Then the key is used for these lists to access the weight and the most possible next chord for the current ***Progression***. This helps us trace the most possible next chord and its weight easily for each ***Progression*** using the current chord and key.

Then it passes to the preprocess part. It uses **Get_Max_Progression_Key_Chord**() to trace through all of the possible starting music key and starting chord with a depth in terms of Measure defined by user. The ***Nextchord*** and ***key*** with the maximum weight are selected.

In the progression part. It traces through the current progression by using the ***Nextchord*** and ***key*** from the previous event. There exists a modulation if the music key is not exist for all flow to the current progression or the weight of all flow in the music key is below the tolerance defined by user in the previous progression. It then finds the ***key*** using the ***NextChord*** by **Get_Max_Progression_Key**(), which trace all flow of the current state.

The result is appended to the ***ProgressionResult*** if and only if an progression is found. This information includes measure and offset of the event, the music key and the current and next chord. The weight is cumulated for tolerance.

If there are no music keys to the current progression. It skips the current progression and find the ***key*** and ***NextChord*** to trace in the next progression using the same function to obtain the start key and chord, **Get_Max_Progression_Key_Chord**().

Example:
SQ-Original-Fixed.xml Measure 15
Event 102: ['ACE', 'DAF', 'DAF#', 'ACF', 'ACF#']
Event 103: ['DG#B', 'ACE', 'DAF', 'DAF#', 'ACF', 'ACF#']
Event 104: ['DG#B', 'ACE', 'DAF', 'DAF#', 'ACF', 'ACF#']

the state machine with part of the flow shown:

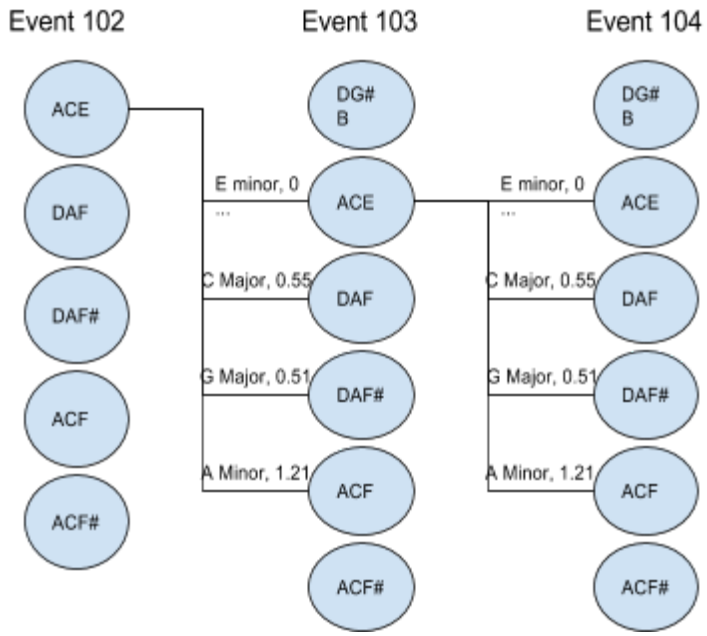


Figure 6.5 :Most probable next chord

6.3.1.3 Chord Existence Check

It is a trivial function to check whether a chord exist using the hashkey of the ***FlowTable***. It is provided for efficient possible chord matching in **Match_Possible_Chord**() Sub-component (Section 4.3.2.1) in ***EventAnalyzer***.

In implementation, it takes a set of chords as input. Then it convert the chord set into a string of hashkey using “,”.join in python. The key is checked by using the dictionary function key in dictionary. This provide an efficient checking on all permutation using the hashkey property of dictionary.

6.3.2 Data Structure

6.3.2.1 FlowTable

Due to the frequently access of flow data and chord existence on ***FlowTable***. The data structure of ***FlowTable*** is implemented to minimising the lookup time complexity.

The Flow Table is a dictionary. It joins the ***FromChord*** into a string as hashkey and use it as the index. This is implemented to minimise the lookup complexity on chord matching.

Then, the data section contains a dictionary again. The string converted by ***ToChord*** with the same converting method is used as the index. This is implemented to minimise the lookup time complexity on reading the flow data.

Then the data section contains a dictionary with ***key*** as the index. Its data section contains the roman numeral and the ***weight*** of the flow. This is implemented to ease the data preparation of the **Progression_Analysis**().

The overall Data Structure is:

FlowTable = Dictionary {
    FromChord : Dictionary{
        ToChord : Dictionary{
            Index ***Key*** : List [***FromChord*** Roman, ***ToChord*** Roman, ***Weight***]
        }
    }
}

6.3.2.2 ProgressionList

The data structure of ***ProgressionList*** is complicated. Each progression includes a combination of the matched chord from the current and next event.

The ***ProgressionList*** is implemented as a list, it contains the measure, offset and the ***ProgressionTable*** for the current progression.

The ***ProgressionTable*** contains the string of ***CurrentChord*** and the dictionary of its maximum possible flow to the ***NextChord***. Although it can be implemented using dictionary instead of list to improve the time complexity, the improvement is not obvious as permutation is required since different permutations conceive different meanings in music theory.

The dictionary with the maximum possible flow is implemented because of the implementation of the inner dictionary of the ***FlowTable***. As a result, the similar structure can helps prepare data from the ***FlowTable*** into the ***ProgressionList***.

The Overall Data Structure is:

ProgressionList = List[
    MeasureNumber, Offset, ***ProgressionTable*** = List[
        CurrentChord Hashkey, ***MaximumPossibleFlow*** = Dictionary{
            Key : ***NextChord*** Hashkey
        }
    ]
]

The hashkey of the chord is used to improve the simplicity and readability on the data.

6.3.2.3 ProgressionResult

The ***ProgressionResult*** is simple and straight-forward. The data structure is provided for a reference.

The Overall Data Structure is:

ProgressionResult = List[
    Progression = List[
        Measure, Offset, ***Key***,
        FromChord hashkey, ***ToChord*** hashkey,
        FromChord Roman, ***ToChord*** Roman,
        Current ***Weight***
    ]
]

6.3.3 WeightTableGenerator

The ***WeightTableGenerator*** is an analyzer that analyze the input score and generate the flow weight table. The user can choose to create a weight table by their own method or by using this ***WeightTableGenerator***.

6.3.3.1 Dataset Format

A text-based format is defined for easy storage and organization. The content is as follows:

```
Opening <filename of the excerpt>.txt:
<filename of the excerpt>      <name of the excerpt>          <composer>
<pitch class of music key>     <'Major' | 'Minor'>      <no. of chords>
<chords_0>
<chords_1>
…
<chords_-1> // from 0 to the last chord
<pitch class of music key>     <'Major' | 'Minor'>      <no. of chords> // modulation, if exists
<chords_0>
<chords_1>
…
<chords_-1> // from 0 to the last chord
```

The presentation of chords goes with a composition of pitch class, chord quality and inversion number.
<{pitch class}_{maj | min | dom7 | dim7 | hdim7 | ger6 | ita6 | fre6}_{0 | 1 | 2 | 3 }>

Example: ex24a.xml
Opening ex24a.xml

```
ex24a.xml	String Quartet Op. 76 No. 6, II, mm. 31-39	Haydn
Bb	Major	12
Bb_maj_0
F_dom7_2
Bb_maj_0
F_dom7_1
Bb_maj_0
Eb_maj7_0
C_min_0
C_dom7_1
F_maj_0
Bb_min_0
F_dom7_2
Bb_min_1
B	Major	5
B_maj_1
A#_dim_2
B_maj_2
F#_dom7_0
B_maj_0
```

6.3.3.2 Dataset Preparation

Since section 6.3.3.1 describes a custom data format designed specifically for data mining mentioned below, raw datasets must be converted to conform the format mentioned before. Currently, a dataset with 45 music excerpts is prepared for data mining. There are 46 MIDI excerpts with encoded chord information are from Tonal Harmony by Stefan Kostka, with one excerpt is corrupted, rendering unusable. They are then converted into readable text format in section 6.3.3.1. The dataset is not limited to these 45 excerpts. When needed, one can add new excerpts by following the format in 6.3.3.1.

6.3.3.3 Preprocessing

The dataset is preprocessed into a single large text file containing only roman numerals that is going to be put into data mining tool Weka.

Since each excerpt in dataset contains modulation and chord information. By using methods from lib.py (see section 2.2.2). These chords can be converted into roman numerals

according to their modulations. Thus, the result preprocessed file is a large chunk of roman numerals regardless of music keys.

For data mining, correct answers and incorrect answers are needed for tool to differentiate and learn.

The single roman numeral file is converted into roman numeral flow by taking the current roman as the first element and the second roman as the second element for each roman numeral except the last one. There is a n-1 flow created for a roman numeral list with n length.

For most of the chord the program can recognize, they are a common chord. These common chord is diatonic to different music keys. As a result, it is required to estimate the bias on the flow recognized belong as different keys.

As a result, for each chord progression, a set of chord progression is expanded if there exist another roman numeral set that possible to be recognized as these chord progressions. These roman numeral set is a wrong recognition while the original one is correct recognition.

For example, the chord progression from I to V in C major key is possible to be recognized into a chord progression from IV to I in G major key and dimVII to IV in D major. As a result, the data set includes I to V as correct recognition while IV to I is incorrect recognition for each chord progression. The example of the data is:

FromChord Roman	***ToChord*** Roman	Chord Progression	Recognized	Quality
I	V	I-to-V	True	Major
IV	I	IV-to-I	False	Major
dimVII	IV	dimVII-to-IV	False	Major

Since the flow weight table is separated as minor keys and major keys due to their different roman numeral presentations. The chord progressions that recognized as major keys and minor keys is then grouped into data set in their respective music quality.

These training data set is stored as .csv file for data mining using Weka.

6.3.3.4 Data Mining

The data processed is ready for data mining. In this situation, weka is chosen as the data mining software to be used. There is no limitation on the choose of the filter algorithm is required.

In our analyzer, unsupervised attribute filter **SortLabel** is used to sort the label and improve the readability on the data mining result. In classification of data, the classifier Stochastic gradient descent **SGD** function on binary class Support Vector Machine **SVM** is selected due to its performance and compatibility on ***FlowTable***.

In the testing situation with 664 test case for the major chord progression, classifiers are examined with 10 folds of cross validation. The analysis result for classifiers are as followed.

Classifier Examined	Average TP Rate	Relative absolute Error
SGD on binary SVM	85.0904 %	30.5086 %
Logistic model tree	84.6386 %	37.2923 %
J48 pruned tree	82.9819 %	40.7775 %
Naive Bayes	84.0361 %	34.5121 %
Logistic Regression	85.3916 %	28.1495 %
SimpleLogistic	84.7892 %	38.6304 %

In total, there are 45 classifiers have been examined. Among these classifiers, the best choice is the Stochastic gradient descent, which provides a relatively positive result among the classifier and easy to implement in the ***FlowTable***.

It is true that the classification can be improved by providing more testing data, examine more classifier and testing with more filter. It can be improved in the future work.

6.3.3.5 Result Parsing

The model result is then parsed to the Roman Flow Weight Excel. These model result is required for user to directly copy to the input file. The SGD on binary SVM is the only parsable model.

However, it is possible that the model result from the other classifier can be parsed in the future with implementation.

7 Testing

7.1 Chord Correctness Test

7.1.1 Test Cases

Seven score excerpts in MusicXML format derived from four musical scores are selected as testing data for the program.

7.1.1.1 Canon in D excerpt 1

Canon in D, mm. 5-8, Pachelbel (canon_in_D_excerpt_1.xml)
The excerpt features only half notes with no dissonances. The purpose of choosing this excerpt is to test the system whether the basic harmonic skeleton can be recognized well.

7.1.1.2 Canon in D excerpt 2

Canon in D, mm. 9-16, Pachelbel (cannon_in_D_excerpt_2.xml)
This excerpt features stable progressions of chords with neat dissonances as passing notes. The purpose of choosing this excerpt is the extension of excerpt 1. By introducing small number of dissonances, we want to see whether the result is still stable.

7.1.1.3 Canon in D excerpt 3

Canon in D, mm. 17-32, Pachelbel (cannon_in_D_excerpt_3.xml)
This excerpt features a general monophonic structure, which means only one hand is responsible for main melody. While the left hand is responsible for harmonic textures, the right hand is responsible for melody line. The purpose of choosing this excerpt is to examine how the fast melody would affect the analysis result. Would it be stay stable or deviate from testing results above.

7.1.1.4 Canon in D excerpt 4

Canon in D, mm. 37-41, Pachelbel (cannon_in_D_excerpt_4.xml)
This excerpt features an inverted monophonic structure. The roles of both hands are exchanged. The left hand is responsible for a melodic bass line whereas the right hand is responsible for harmonic textures. The purpose of choosing this excerpt is to examine how the system reacts to inverted monophonic music. If the result is acceptable, it means the system can handle polyphonic music, which means there are two simultaneous melody lines within a score.

7.1.1.5 Moonlight Sonata First Movement excerpt

Moonlight Sonata, Op. 27 No. 2, I, mm. 1-14, Beethoven (moonlight_sonata_I_excerpt.xml)
This excerpt features a monophonic structure with left hand as the rhythmic part and right hand as the melody and lead part. The purpose is to test a more “real world” problem as Moonlight Sonata is a well-known classic in classical music.

7.1.1.6 Hoppipolla excerpt

Hoppipolla, Sigur Ros (hoppipolla_excerpt.xml)
Hoppipolla is a song by Iceland post rock band Sigur Rós. It features a modern style approach to composition. The style of chord progression leans towards today’s popular and rock style. The purpose is to test the forgiveness of the system against a song with unfamiliar chord progression.

7.1.7 String Quartet excerpt

String Quartet, Op. 18 No. 1, IV, Beethoven (SQ-Original.xml)
String Quartet is the first piano reduction provided by Prof. Lucas Wong. Throughout the year, we consider this music piece as a starting point for our work, trying to produce the most accurate result.

7.1.2 Sample Data Preparation

To test the correctness, comparisons are required between system output and a sample correct data. the sample data are a list of text files labeling the right chord of each events defined in ***EventAnalyzer*** (See section 4).

```
> cat canon_in_D_excerpt_1.txt
        canon_in_D_excerpt_1,16 // <filename>,<number_of_events>
        0,D,F#,A
        1,D,F#,A
        2,A,C#,E
        …
        15,A,C#,E
```

for each chord, the integer represents the index of events with correct chords.

7.1.3 Test Results

7.1.3.1 Terminology

Number of chord events - refer to the total number of chord events given a score excerpt.
Chords recognized - refer to the number of chord events that have a matched chord, regardless of correct or wrong chord.

Exactly correct - of the chords recognized, the number of matched chords that are completely correct
Overly recognized - of the chords recognized, the number of matched chords that are overly recognized e.g. a seventh chord is recognized while the correct result is a triad.
Same functionality - of the chords recognized, the number of matched chords that share the same functionality with the correct result chord. As explained in section 1.3.5, roman numeral chords can be categorized into 3 groups, Tonic, Dominant and Subdominant.
For instance, FAD vs FAC. In the key of C major, FAD is II6 while FAC is IV. Although they are different from each other in terms of roman number, II6 and IV are belong to subdominant chords, which share same functionality as usage.
Wrong bass, same consonance - of the chords recognized, the number of matched chords that have the same consonance but different notes as the bass. E.g. GBD vs BDG. they have the same consonance but different inversion.
Wrong chords - of the chords recognized, the number of matched chords that are totally wrong and unrelated.
Chord correctness - The percentage of correct chords.
(number of correct chords / number of recognized chords) * 100
Overall correctness - The percentage of correct chords against total chord events
(number of correct chords / number of recognized chords) * 100
Correct chords (hard) - it only considers exactly correct chords as correct.
Correct chords (normal) - it considers exactly correct chords, overly recognized chords and chords with same functionality as correct.
Total correct chords (hard) = exactly correct + overly recognized + same functionality
Correct chords (soft) - it considers all chords as correct except wrong chords.
Total correct chords (soft) = chords recognized - wrong chords

7.1.3.2 Global Statistics

Total excerpts: 7
Total chord events: 633
Total chords recognized: 633 (100.0%)
Of the recognized chords:

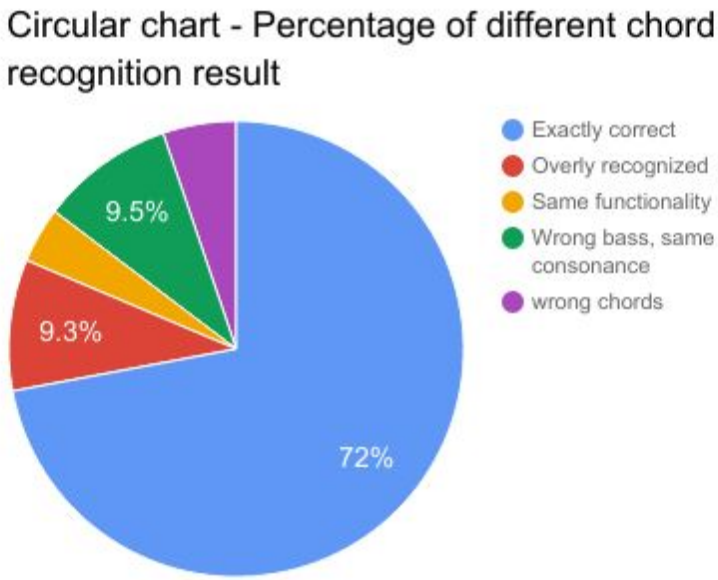


Chart 7.1: Percentage of different chord recognition result

Total exactly correct chords: 456 (72.04%)
Total overly recognized chords: 59 (9.32%)
Total same functionality chords: 25 (3.95%)
Total wrong bass, same consonance chords: 60 (9.48%)
Total wrong chords: 33 (5.21%)

Correctness:
Chord correctness (hard) : 72.04%
Overall correctness (hard) : 72.04%
Chord correctness (normal) : 85.3%
Overall correctness (normal) : 85.3%
Chord correctness (soft) : 94.7%
Overall correctness (soft) : 94.7%

Although the number of exactly correct chords is 456 with only 72%, many cases are still acceptable in terms of music analysis even they are not exactly the same. There are 3 categories for these cases.

The first category is overly recognized chords, constituting the largest group of not exactly correct chords. The reason for quite an amount of overly recognized chords is issue between a triad and a triad with seventh note. In other words, it is an issue about whether the seventh note is a dissonance or it is part of the chord.

For example, suppose a single event with pitch classes D F# A C. It can be recognized as D major or D7. If it is D major, then the event has C as the sole dissonance. If it is D7 instead, the event has no dissonances. The issue is controversial as both ways of recognition meets the music theory. Therefore, it is acceptable to consider overly recognized chords as correct.

For chords with same functionality category, it is also acceptable as correct cases because of same functionality of chords. Chords falling into this category are belong to events that only contain two pitch classes. For a chord with only two notes, one more note is guessed by the system so that a triad chord can be formed and analyzed.

For example, given an event with pitches F and A. To form a meaningful triad. It either becomes F A D or F A C. Assume it is in the key of C, F A D is II6 while F A C is IV. Although they are different in terms of roman number, they share same functionality as subdominants for having the same F and A notes.

The last category is **wrong bass, but same consonances**. Both outputs of a particular chord and sample correct answer have same consonances. However, they have different inversions or the bass note is mismatched. Although they are identical in terms of composition of notes, they are not as acceptable as the two categories mentioned above.

Wrong chords are self-explanatory, meaning that there is no relation between system result and correct answer.

Considering the correctness, the normal level (85.3%) is most suitable to represent how accurate of the system. While it is not as strict as hard level (72.04%), by allowing a draw on the controversy, it still complies with music theory. For soft level (94.7%), even it has high correctness, it loosens the criteria too much.

7.1.3.3 Individual Statistics

7.1.3.3.1 Excerpt 1 of Canon in D

Input: canon_in_D_excerpt_1.xml, canon_in_D_excerpt_1.txt
Number of chord events: 16
Chords recognized: 16 (100.0%)

Exactly correct: 13 (81.25%)
Overly recognized: 0 (0.0%)
Similar function: 0 (0.0%)
Wrong bass, same consonance: 3 (18.75%)
Wrong chords: 0 (0.0%)

Correctness analysis:
Correct chords (hard): 13
Chord correctness (hard): 81.25%
Overall correctness (hard): 81.25%
Correct chords (medium): 13
Chord correctness (medium): 81.25%
Overall correctness (medium): 81.25%
Correct chords (soft): 16
Chord correctness (soft): 100.0%
Overall correctness (soft): 100.0%

It scores 81.25% correctness at normal level but 100% at soft level. It is due to the fact that wrong inversion is recognized instead. The reason for this to happen is the acoustic sustaining effect. It drags bass notes from the previous and next events into consideration so wrong inversion may occur.

7.1.3.3.2 Excerpt 2 of Canon in D

Input: canon_in_D_excerpt_2.xml, canon_in_D_excerpt_2.txt
Number of chord events: 32
Chords recognized: 32 (100.0%)

Exactly correct: 26 (81.25%)
Overly recognized: 0 (0.0%)
Similar function: 0 (0.0%)
Wrong bass, same consonance: 6 (18.75%)
Wrong chords: 0 (0.0%)

Correctness analysis:
Correct chords (hard): 26
Chord correctness (hard): 81.25%
Overall correctness (hard): 81.25%
Correct chords (normal): 26
Chord correctness (normal): 81.25%
Overall correctness (normal): 81.25%
Correct chords (soft): 32
Chord correctness (soft): 100.0%
Overall correctness (soft): 100.0%

It scores 81.25% in normal level correctness. This case is similar to the previous one. The wrong bass or inversion is recognized due to the acoustic sustaining effect.

7.1.3.3.3 Excerpt 3 of Canon in D

Input: canon_in_D_excerpt_3.xml, canon_in_D_excerpt_3.txt
Number of chord events: 174
Chords recognized: 174 (100.0%)

Exactly correct: 118 (67.82%)
Overly recognized: 45 (25.86%)
Similar function: 0 (0.0%)
Wrong bass, same consonance: 10 (5.75%)
Wrong chords: 1 (0.57%)

Correctness analysis:
Correct chords (hard): 118
Chord correctness (hard): 67.82%
Overall correctness (hard): 67.82%
Correct chords (normal): 163
Chord correctness (normal): 93.68%
Overall correctness (normal): 93.68%
Correct chords (soft): 173
Chord correctness (soft): 99.43%
Overall correctness (soft): 99.43%

It attains 93.68% correctness at normal level, meaning that the system yields good recognition result. Although it is only 67.82% at hard level, it is mainly due to a considerable amount of overly recognized chords, which do not mean that they are wrong and unacceptable as explained in section 7.3.2. Therefore, the system did a good job on this excerpt event this excerpt features a rapid melody line, which may mess up the recognition.

7.1.3.3.4 Excerpt 4 of Canon in D

Input: canon_in_D_excerpt_4.xml, canon_in_D_excerpt_4.txt
Number of chord events: 33
Chords recognized: 33 (100.0%)

Exactly correct: 20 (60.61%)
Overly recognized: 0 (0.0%)
Similar function: 0 (0.0%)
Wrong bass, same consonance: 13 (39.39%)
Wrong chords: 0 (0.0%)

Correctness analysis:
Correct chords (hard): 20
Chord correctness (hard): 60.61%
Overall correctness (hard): 60.61%
Correct chords (normal): 20
Chord correctness (normal): 60.61%
Overall correctness (normal): 60.61%
Correct chords (soft): 33
Chord correctness (soft): 100.0%
Overall correctness (soft): 100.0%

This excerpt is inverted homophonic, with left hand part as the melody. The correctness at normal level is lower than previous ones, with only 60.61%. It is due to the rapid change of bassline at the left hand. Coupled with acoustic sustaining effect, the bass is easily messed up, thus yielding a quite amount of “wrong bass but having same consonances” chords.

7.1.3.3.5 Excerpt of Hoppipolla

Input: hoppipolla_excerpt.xml, hoppipolla_excerpt.txt
Number of chord events: 100
Chords recognized: 100 (100.0%)

Exactly correct: 65 (65.0%)
Overly recognized: 2 (2.0%)
Similar function: 15 (15.0%)
Wrong bass, same consonance: 0 (0.0%)
Wrong chords: 18 (18.0%)

Correctness analysis:
Correct chords (hard): 65
Chord correctness (hard): 65.0%
Overall correctness (hard): 65.0%
Correct chords (normal): 82
Chord correctness (normal): 82.0%
Overall correctness (normal): 82.0%
Correct chords (soft): 82
Chord correctness (soft): 82.0%
Overall correctness (soft): 82.0%

Hoppipolla is a modern style piece with chords that is unfamiliar in classical music. It attains 82.0% at normal level with 15 chords with similar functionalities and 18 wrong chords. It is due to the fact that the system can only try matching similar chords as these unfamiliar chords do not exist in our chord database.

7.1.3.3.6 Excerpt of Moonlight Sonata, First Movement

Input: moonlight_sonata_I_excerpt.xml, moonlight_sonata_I_excerpt.txt
Number of chord events: 172
Chords recognized: 172 (100.0%)

Exactly correct: 156 (90.7%)
Overly recognized: 4 (2.33%)
Similar function: 8 (4.65%)
Wrong bass, same consonance: 0 (0.0%)
Wrong chords: 4 (2.33%)

Correctness analysis:
Correct chords (hard): 156
Chord correctness (hard): 90.7%
Overall correctness (hard): 90.7%
Correct chords (normal): 168
Chord correctness (normal): 97.67%
Overall correctness (normal): 97.67%
Correct chords (soft): 168
Chord correctness (soft): 97.67%
Overall correctness (soft): 97.67%

97.67% correctness at normal level is encouraging. By having a good correctness of a well-known classic, it serves a good starting point for an accurate music analysis on all classical music. Strictly speaking, a 90.7% hard correctness is good news as it successfully matched over ninety percent identical chords.

7.1.3.3.7 Excerpt of String Quartet

Input: SQ-Original-fixed.xml, SQ-Original-fixed.txt
Number of chord events: 106
Chords recognized: 106 (100.0%)

Exactly correct: 58 (54.72%)
Overly recognized: 8 (7.55%)
Similar function: 2 (1.89%)
Wrong bass, same consonance: 28 (26.42%)
Wrong chords: 10 (9.43%)

Correctness analysis:
Correct chords (hard): 58
Chord correctness (hard): 54.72%
Overall correctness (hard): 54.72%
Correct chords (normal): 68
Chord correctness (normal): 64.15%
Overall correctness (normal): 64.15%
Correct chords (soft): 96
Chord correctness (soft): 90.57%
Overall correctness (soft): 90.57%

It only attains 64.15% correctness at normal level with a considerable amount of “wrong bass, same consonances” chords. The reason is the weak presence of bass notes which easily fluctuates. The bass part in the score is only notes in quarter or eighth duration, which is not long enough to support melody lines. Therefore, more chords are recognized wrong in terms of inversion.

7.1.4 Conclusion

After testing on correctness, it is concluded that the system performs best on music with slow changing of bass and homophonic melody line such as the first movement in Moonlight Sonata. It is because a slow bassline creates strong presence of harmony that bass notes do not easily fluctuate while the dissonances in melody line help chord identification. As general classical music follows this kind of composition pattern, with strong presence of bass and “free walking” melody. It is concluded that the system should yield an acceptable result of these scores.

7.2 Test on Scores with Wrong Notation

7.2.1 Test Case

The input .xml score is a wrongly notated “Moonlight Sonata III Movement excerpt” (moonlight_sonata_III_wrongly_notated.xml)

7.2.2 Test Result

The output .xml score has many notes marked in red, which suggests that they are considered as dissonances by the program. Professor Lucas Wong has suggested that these wrongly notated notes are “dissonances” because they are enharmonic equivalents. While these notes share same acoustic properties compared with rightly notated counterparts, which is heard the same, they are considered as different notes. For example, D# and Eb, the two notes are same in pitches but they are not the same notes in music theory. In the key of Eb, D# is considered as dissonance even D# has the same pitch as Eb.

8. Ideas Experimented

8.1 Measure Analyser

The Measure Analyser is the work done in the first semester. It is an idea that identify the chord using all note in the measure. It is examined because it is simple to develop and tested. It can be used to test the chord identification method on exact chord match by providing suitable sample data.

However, since it requires an assumption that each measure can only contain one chord, it cannot be used as a chord identification analyzer for classical music. It has been very common for various chord change occur within a measure.

8.2 Consecutive Modulation Chord Identification

In the ***EventAnalyzer***. Another chord identification method is introduced. In this method, after generating a list of possible chords for an event, it immediately selects the chord using two previous flows. It is very rapid method that only a single run can define all selected chord and return the analysis result. However, it is possible to select the chord wrongly due to the local maximum weight. It is possible that the modulation taking place rapidly using this method and the results are incorrect as it cannot try to minimize the key change.

9 Conclusion

Overall, in this academic year, we have achieved main objective - chord identification. As discussed with Prof. Lucas Wong, the test result is illuminating on how machines can do music analysis even though the machine does not possess the perception of human. It is because the machine does analysis one by one in events and optimizes results with different additional components.

It is similar to putting layers of analyzers with the first layer being the crudest. The layers get precise each time to converge a good result. Still, it is just a machine following rules defined by music theory. It cannot imitate how professionals analyze musically and artistically. In order to yield better results, an adaptive approach using chord progressions and modulations is implemented so as to imitate how human percept music dynamically.

For bonus objectives, we have enhanced the automatic piano reduction neural network by student NG Chiu Yuen in KY1302 with several new features, including new reduction algorithm based on chord identification results and server API. Although the new reduction algorithm does not have an obvious effect in piano reduction on our testing scores, it successfully incorporates dissonance markings into neural network for training in programmatic aspect. It can serve as a starting point for future enhancement of neural network and piano reduction.

10 Future Work

10.1 Neural Network Improvement

As there is no enough time for testing of the dissonance feature on the neural network implementation, more testing on neural network especially on dissonance feature is suggested in the future.

Besides, the setting of dissonance feature on the neural network can also be modified in the future. A middle weight of 0.5 can be given to the note that cannot be recognized to be dissonance or not. Also, the weight of seventh note in the seventh chord in the dissonance feature can also be considered.

Also, the ***EventAnalyzer*** can also provide the information of roman numeral and key of each note. In fact, this information can also be inputted as features into the neural network to improve the reduction diversity.

10.2 Chord Recognition Improvement

Although the chord recognition can provide a positive result on testing especially the soft correct chord. However, it is still not optimistic on middle and hard testing. The handle of inversion and seventh chord can be further improved in the future.

Also, it is also possible to implement the idea of resolving dissonance by step to the chord identification. It can be used to provide a better correctness checking together with the current rule, consecutive dissonance.

10.3 Modulation Improvement

The current ***HarmonicAnalyzer*** provides positive result but the time and space complexity of the algorithm can be improved in the future. A single run of harmonic analysis is possible for ***EventAnalyzer***, as the design of multiple run of harmonic analysis in the ***EventAnalyzer*** is just ease of implement.

The numbers of possible roman numeral flow can also be increased in the future. In the test case of moonlight sonata, the chord progression cannot find some specific flow in the flow weight table as some of the roman numeral of these flow are not supported in the music library, especially the rules of inversion. More roman numeral rules can be implement into the music library and flow weight table in the future.

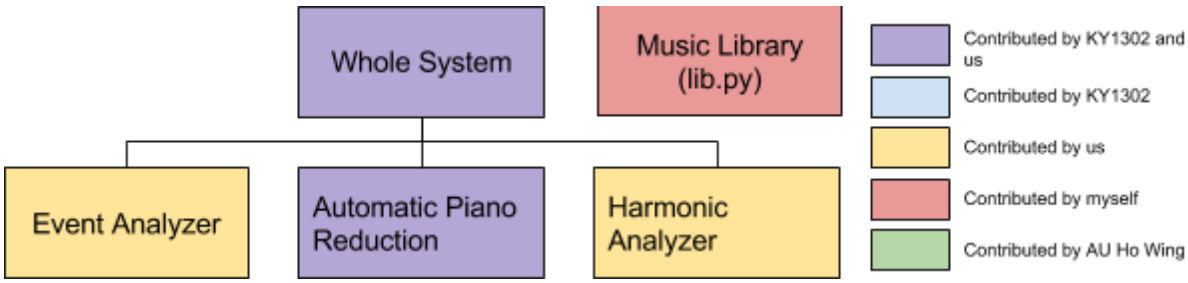
The data of flow weight table can also be improved in the future. In the current flow weight table. Since there are only among 30 score is used for data mining and generate the current flow weight table, the current algorithm has to convert the chord from inversion to its original

form to produce accurate result. It is obvious that further analysis is required with more score as testing data.

Although it is used to recognize classical music, HarmTrace, a functional harmonic analysis approach, can also be implemented into the chord progression analysis as a reference.

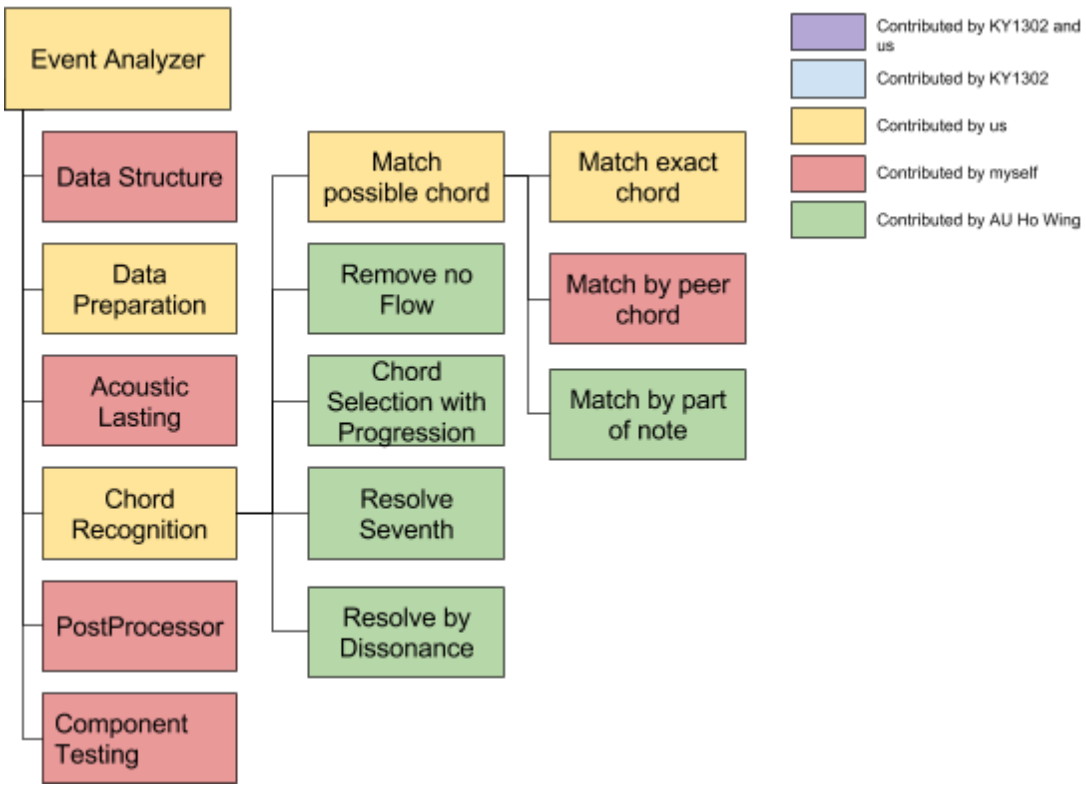
11 Responsible Work

In the whole system, I collaborate with the teammate, AU Ho Wing to all individual components including ***EventAnalyzer***, Automatic Piano Reduction and ***HarmonicAnalyzer***. Moreover, I have developed a music rules library, which is the backbone of the whole chord identification project.



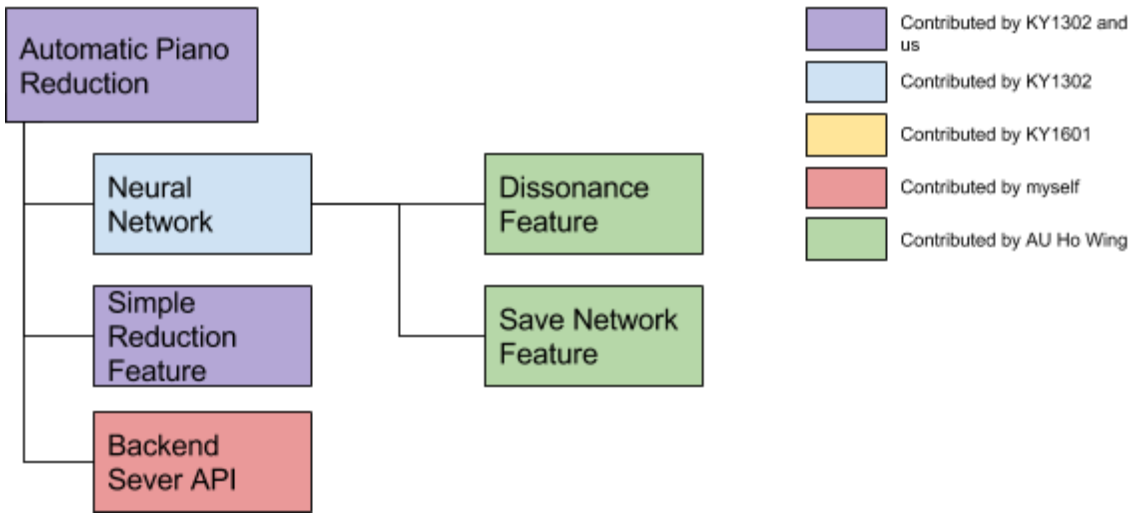
11.1 Chord Identification

In the ***EventAnalyzer*** component, my contribution goes to components Data Strucutre, **DataPreparation**, **Acoustic Lasting**, *ChordRecognition, Postprocessor, Component Testing* and sub-component Match Possible Chord with **Match_Exact_Chord()** and **Match_By_Peer_Chord()** in *ChordRecognition*.



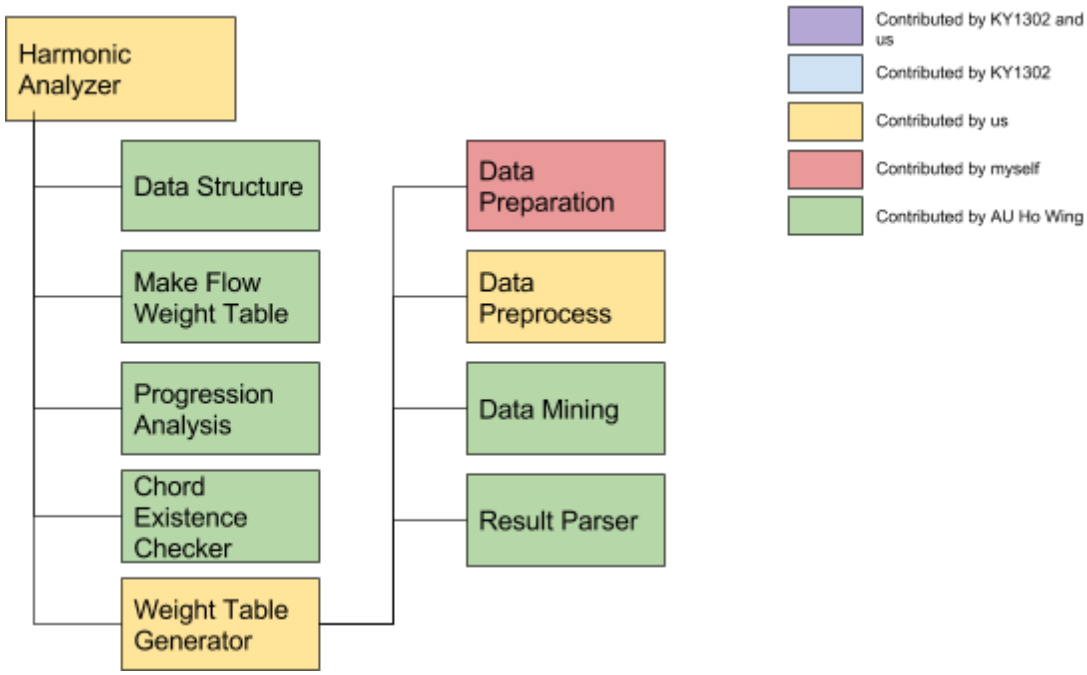
11.2 Automatic Piano Reduction

In Automatic Piano Reduction section, I have implemented backend server API and simple piano reduction by threshold.



11.3 Harmonic Analyzer

In ***HarmonicAnalyzer***, I contribute to components Weight Table Generator and sub-component Data Preparation.



12 References

Schoenberg, Arnold (1975). The Modern Piano Reduction. Style and Idea: Selected Writings of Arnold Schoenberg. pp. 348-350.

Stefan, Kostka (n.d.). 46 MIDI Excerpts. Tonal Harmony. Retrieved from http://www.cs.northwestern.edu/~pardo/kpcorpus.htm

13 Appendix

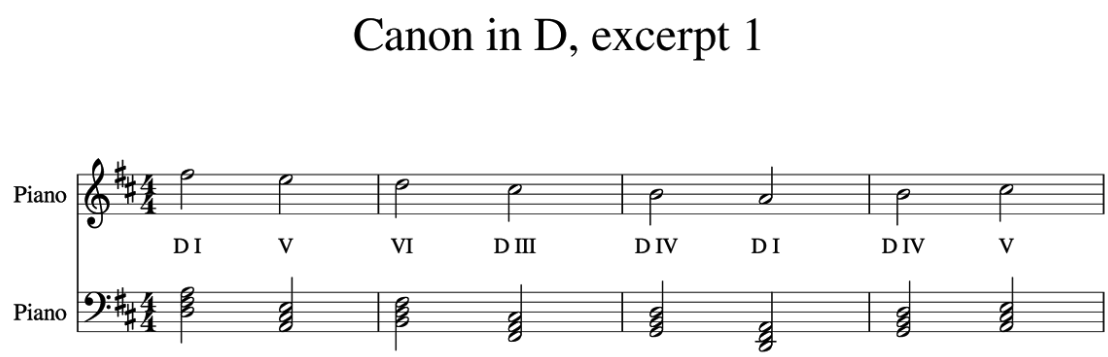
13.1 Chord Identification and Harmonic Analysis Results

The appendix shows the result of chord identification and harmonic analysis. The scores are marked with roman numerals and modulations and dissonances are marked in red colour.

For chord identification correctness statistics. Refer to section 7.1 Chord Correctness Test.

13.1.A canon_in_D_excerpt_1.xml

13.1.A.1 Analysis Result

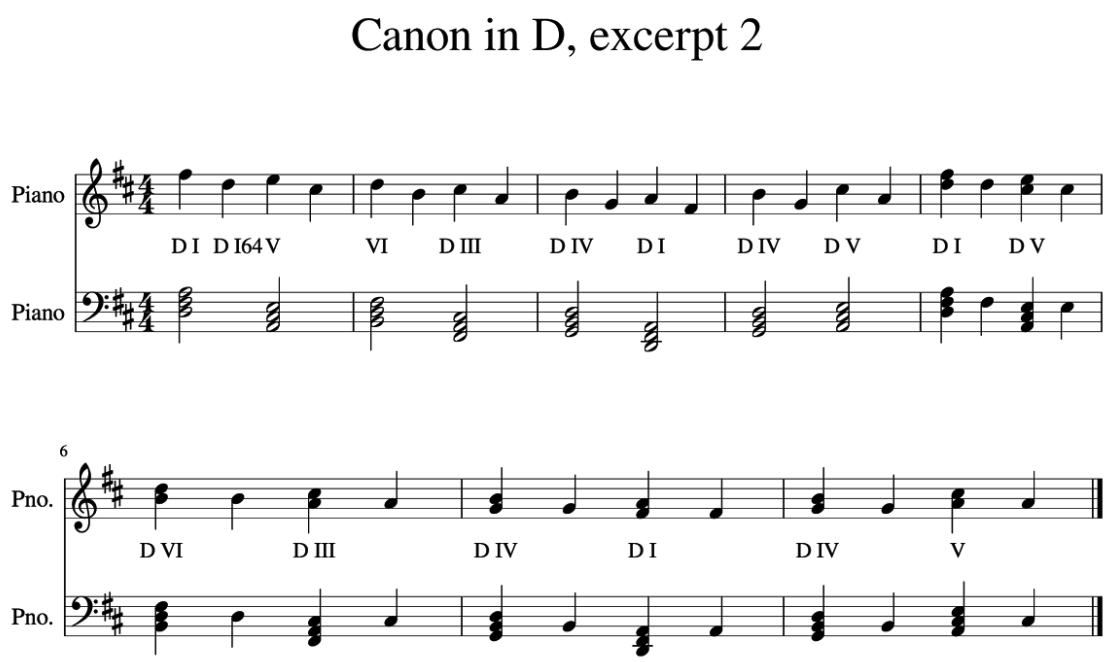


13.1.A.2 Analysis Log

Event 1 Recognized: ['A', 'D', 'F#'] Correct: ['D', 'F#', 'A'] Reason: wrong bass
Event 5 Recognized: ['F#', 'B', 'D'] Correct: ['B', 'D', 'F#'] Reason: wrong bass
Event 9 Recognized: ['D', 'G', 'B'] Correct: ['G', 'B', 'D'] Reason: wrong bass

13.1.B canon_in_D_excerpt_2.xml

13.1.B.1 Analysis Result

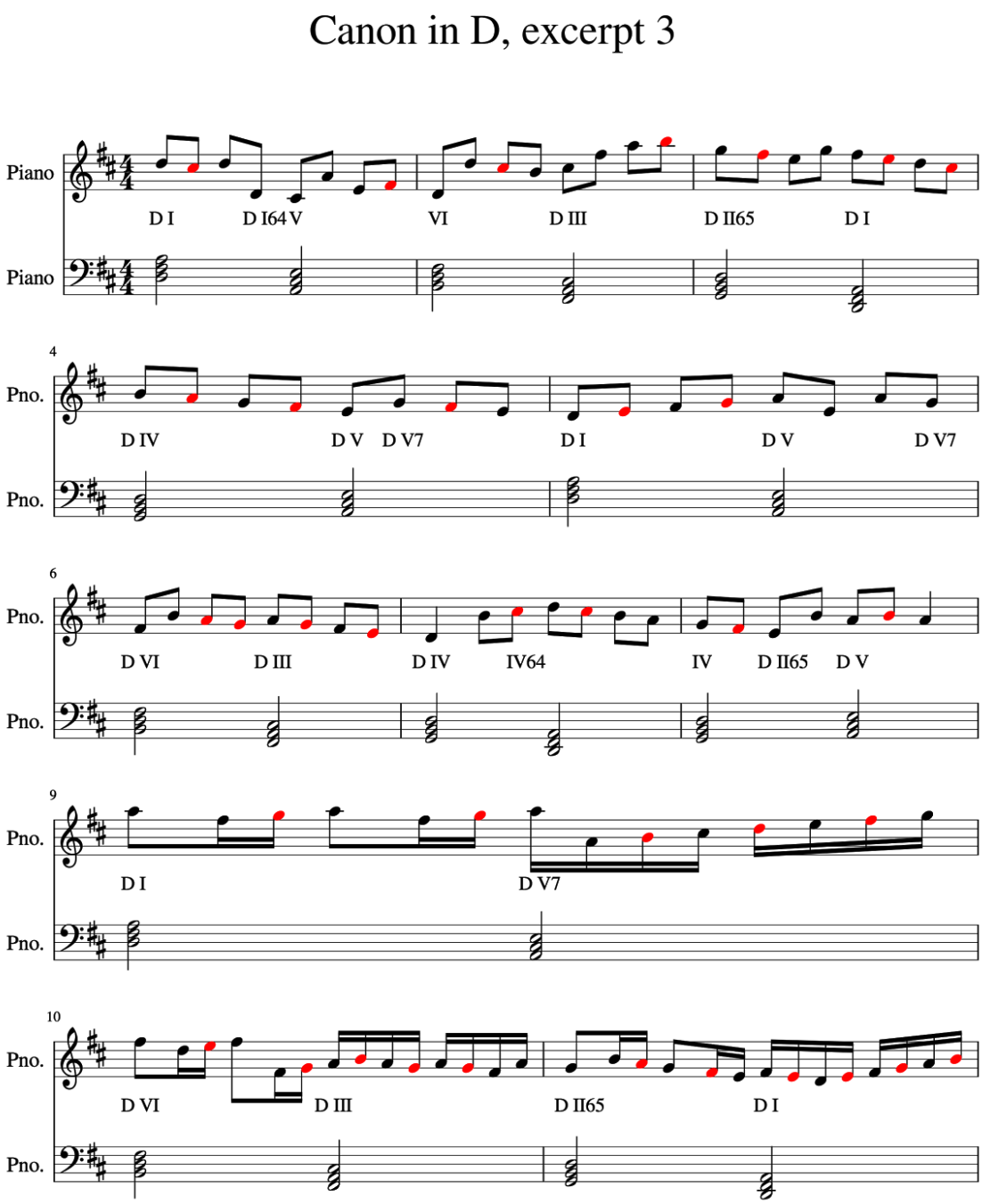


13.1.B.2 Analysis Log

Event 1 Recognized: ['A', 'D', 'F#'] Correct: ['D', 'F#', 'A'] Reason: wrong bass
Event 5 Recognized: ['F#', 'B', 'D'] Correct: ['B', 'D', 'F#'] Reason: wrong bass
Event 9 Recognized: ['D', 'G', 'B'] Correct: ['G', 'B', 'D'] Reason: wrong bass
Event 17 Recognized: ['A', 'D', 'F#'] Correct: ['D', 'F#', 'A'] Reason: wrong bass
Event 21 Recognized: ['F#', 'B', 'D'] Correct: ['B', 'D', 'F#'] Reason: wrong bass
Event 25 Recognized: ['D', 'G', 'B'] Correct: ['G', 'B', 'D'] Reason: wrong bass

13.1.C canon_in_D_excerpt_3.xml

13.1.C.1 Analysis Result



13.1.C.2 Analysis Log

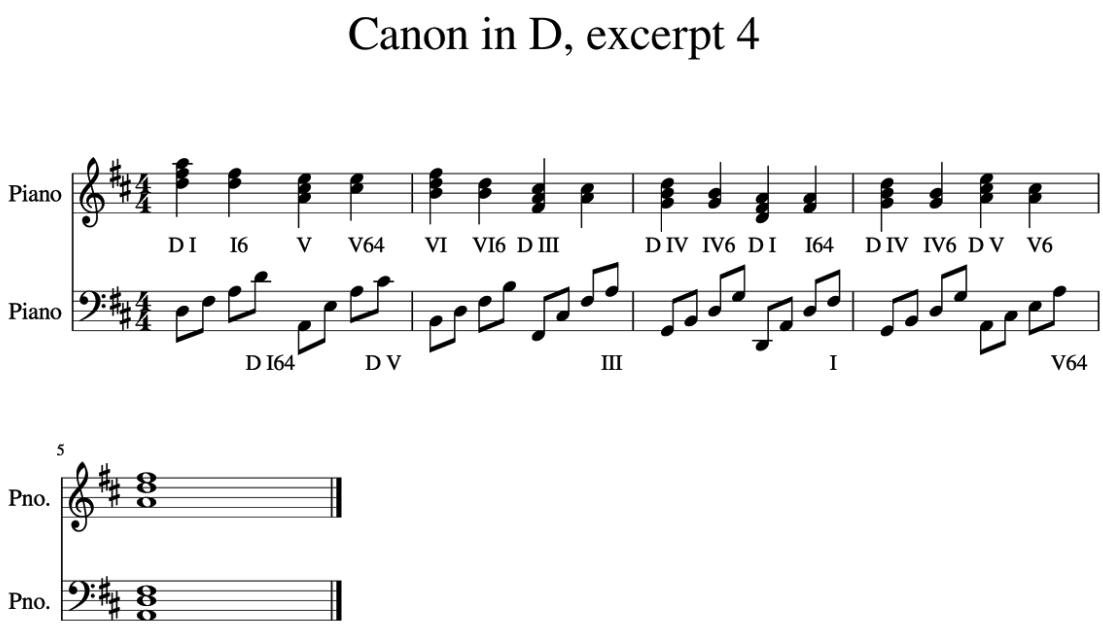
Event 3 Recognized: ['A', 'D', 'F#'] Correct: ['D', 'F#', 'A'] Reason: wrong bass
Event 11 Recognized: ['F#', 'B', 'D'] Correct: ['B', 'D', 'F#'] Reason: wrong bass
Event 16 Recognized: ['G', 'B', 'D', 'E'] Correct: ['G', 'B', 'D'] Reason: overly recognized

Event 17 Recognized: ['G', 'B', 'D', 'E'] Correct: ['G', 'B', 'D'] Reason: overly recognized
Event 18 Recognized: ['G', 'B', 'D', 'E'] Correct: ['G', 'B', 'D'] Reason: overly recognized
Event 19 Recognized: ['D', 'G', 'B', 'E'] Correct: ['G', 'B', 'D'] Reason: overly recognized
Event 29 Recognized: ['A', 'C#', 'E', 'G'] Correct: ['A', 'C#', 'E'] Reason: overly recognized
Event 30 Recognized: ['A', 'C#', 'E', 'G'] Correct: ['A', 'C#', 'E'] Reason: overly recognized
Event 31 Recognized: ['A', 'C#', 'E', 'G'] Correct: ['A', 'C#', 'E'] Reason: overly recognized
Event 35 Recognized: ['A', 'D', 'F#'] Correct: ['D', 'F#', 'A'] Reason: wrong bass
Event 39 Recognized: ['A', 'C#', 'E', 'G'] Correct: ['A', 'C#', 'E'] Reason: overly recognized
Event 43 Recognized: ['F#', 'B', 'D'] Correct: ['B', 'D', 'F#'] Reason: wrong bass
Event 50 Recognized: ['D', 'G', 'B'] Correct: ['G', 'B', 'D'] Reason: wrong bass
Event 51 Recognized: ['D', 'F#', 'A', 'B'] Correct: ['D', 'F#', 'A'] Reason: overly recognized
Event 52 Recognized: ['D', 'F#', 'A', 'B'] Correct: ['D', 'F#', 'A'] Reason: overly recognized
Event 53 Recognized: ['D', 'F#', 'A', 'B'] Correct: ['D', 'F#', 'A'] Reason: overly recognized
Event 54 Recognized: ['D', 'F#', 'A', 'B'] Correct: ['D', 'F#', 'A'] Reason: overly recognized
Event 57 Recognized: ['G', 'B', 'D', 'E'] Correct: ['G', 'B', 'D'] Reason: overly recognized
Event 58 Recognized: ['G', 'B', 'D', 'E'] Correct: ['G', 'B', 'D'] Reason: overly recognized
Event 67 Recognized: ['A', 'D', 'F#'] Correct: ['D', 'F#', 'A'] Reason: wrong bass
Event 68 Recognized: ['A', 'C#', 'E', 'G'] Correct: ['A', 'C#', 'E'] Reason: overly recognized
Event 69 Recognized: ['A', 'C#', 'E', 'G'] Correct: ['A', 'C#', 'E'] Reason: overly recognized
Event 70 Recognized: ['A', 'C#', 'E', 'G'] Correct: ['A', 'C#', 'E'] Reason: overly recognized
Event 71 Recognized: ['A', 'C#', 'E', 'G'] Correct: ['A', 'C#', 'E'] Reason: overly recognized
Event 72 Recognized: ['A', 'C#', 'E', 'G'] Correct: ['A', 'C#', 'E'] Reason: overly recognized
Event 73 Recognized: ['A', 'C#', 'E', 'G'] Correct: ['A', 'C#', 'E'] Reason: overly recognized
Event 74 Recognized: ['A', 'C#', 'E', 'G'] Correct: ['A', 'C#', 'E'] Reason: overly recognized
Event 75 Recognized: ['A', 'C#', 'E', 'G'] Correct: ['A', 'C#', 'E'] Reason: overly recognized
Event 81 Recognized: ['F#', 'B', 'D'] Correct: ['B', 'D', 'F#'] Reason: wrong bass
Event 90 Recognized: ['G', 'B', 'D', 'E'] Correct: ['G', 'B', 'D'] Reason: overly recognized
Event 91 Recognized: ['G', 'B', 'D', 'E'] Correct: ['G', 'B', 'D'] Reason: overly recognized
Event 92 Recognized: ['G', 'B', 'D', 'E'] Correct: ['G', 'B', 'D'] Reason: overly recognized
Event 93 Recognized: ['G', 'B', 'D', 'E'] Correct: ['G', 'B', 'D'] Reason: overly recognized
Event 94 Recognized: ['G', 'B', 'D', 'E'] Correct: ['G', 'B', 'D'] Reason: overly recognized
Event 95 Recognized: ['D', 'G', 'B', 'E'] Correct: ['G', 'B', 'D'] Reason: overly recognized
Event 111 Recognized: ['A', 'C#', 'E', 'G'] Correct: ['A', 'C#', 'E'] Reason: overly recognized
Event 112 Recognized: ['A', 'C#', 'E', 'G'] Correct: ['A', 'C#', 'E'] Reason: overly recognized
Event 113 Recognized: ['A', 'C#', 'E', 'G'] Correct: ['A', 'C#', 'E'] Reason: overly recognized
Event 114 Recognized: ['A', 'C#', 'E', 'G'] Correct: ['A', 'C#', 'E'] Reason: overly recognized
Event 115 Recognized: ['A', 'C#', 'E', 'G'] Correct: ['A', 'C#', 'E'] Reason: overly recognized
Event 116 Recognized: ['A', 'C#', 'E', 'G'] Correct: ['A', 'C#', 'E'] Reason: overly recognized
Event 117 Recognized: ['A', 'C#', 'E', 'G'] Correct: ['A', 'C#', 'E'] Reason: overly recognized
Event 123 Recognized: ['A', 'D', 'F#'] Correct: ['D', 'F#', 'A'] Reason: wrong bass
Event 137 Recognized: ['F#', 'B', 'D'] Correct: ['B', 'D', 'F#'] Reason: wrong bass
Event 151 Recognized: ['D', 'G', 'B'] Correct: ['G', 'B', 'D'] Reason: wrong bass
Event 152 Recognized: ['D', 'F#', 'A', 'B'] Correct: ['D', 'F#', 'A'] Reason: overly recognized
Event 153 Recognized: ['D', 'F#', 'A', 'B'] Correct: ['D', 'F#', 'A'] Reason: overly recognized
Event 154 Recognized: ['D', 'F#', 'A', 'B'] Correct: ['D', 'F#', 'A'] Reason: overly recognized
Event 155 Recognized: ['D', 'F#', 'A', 'B'] Correct: ['D', 'F#', 'A'] Reason: overly recognized

Event 156 Recognized: ['D', 'F#', 'A', 'B'] Correct: ['D', 'F#', 'A'] Reason: overly recognized
Event 157 Recognized: ['D', 'F#', 'A', 'B'] Correct: ['D', 'F#', 'A'] Reason: overly recognized
Event 158 Recognized: ['D', 'F#', 'A', 'B'] Correct: ['D', 'F#', 'A'] Reason: overly recognized
Event 159 Recognized: ['D', 'F#', 'A', 'B'] Correct: ['D', 'F#', 'A'] Reason: overly recognized
Event 172 Recognized: ['A', 'C#', 'E', 'G'] Correct: ['A', 'C#', 'E'] Reason: overly recognized
Event 173 Recognized: ['A', 'C#', 'E', 'G'] Correct: ['A', 'C#', 'E'] Reason: overly recognized
Event 110 Recognized: ['G', 'C#', 'E'] Correct: ['A', 'C#', 'E'] Reason: wrong chord

13.1.D canon_in_D_excerpt_4.xml

13.1.D.1 Analysis Result

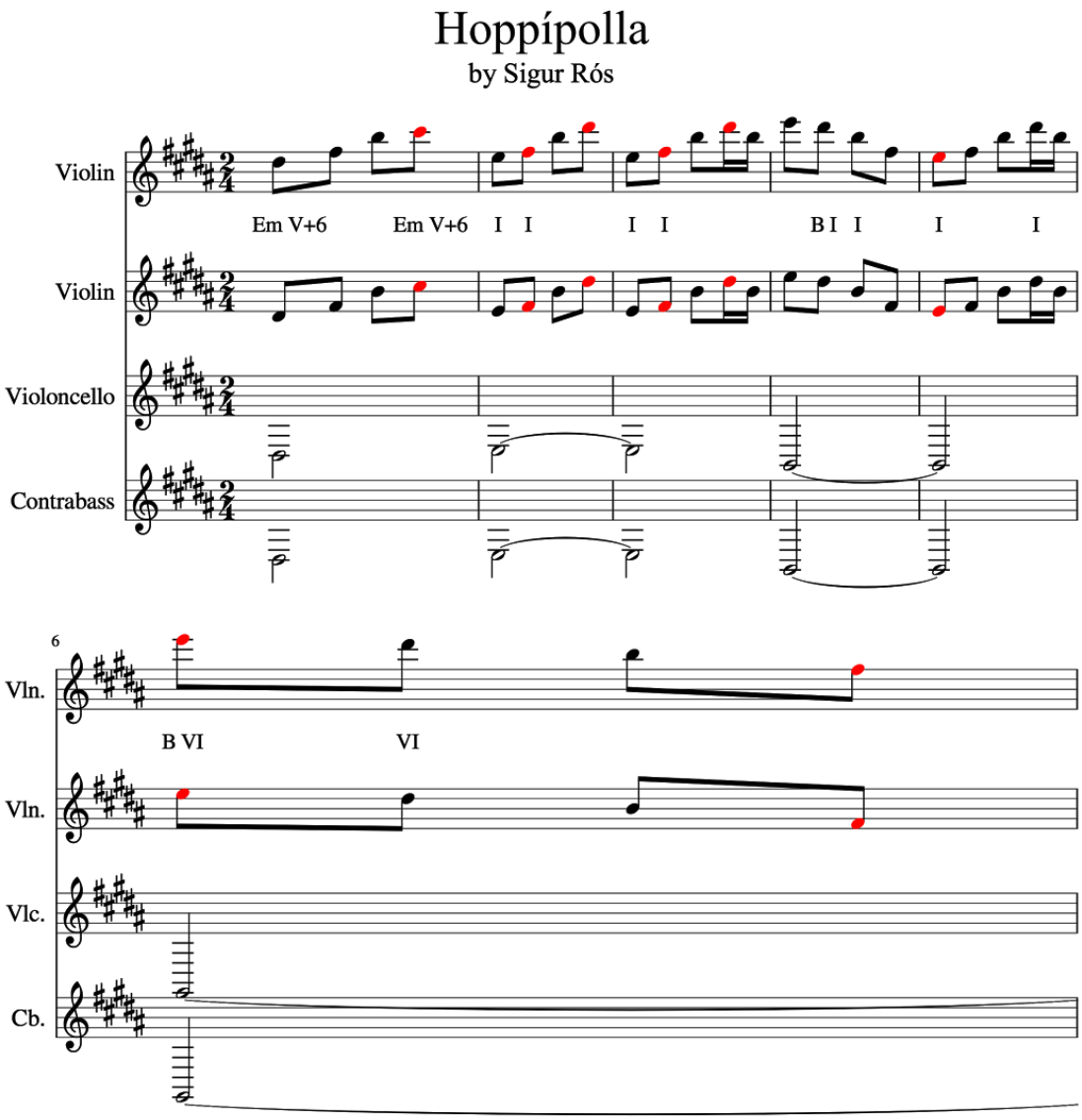


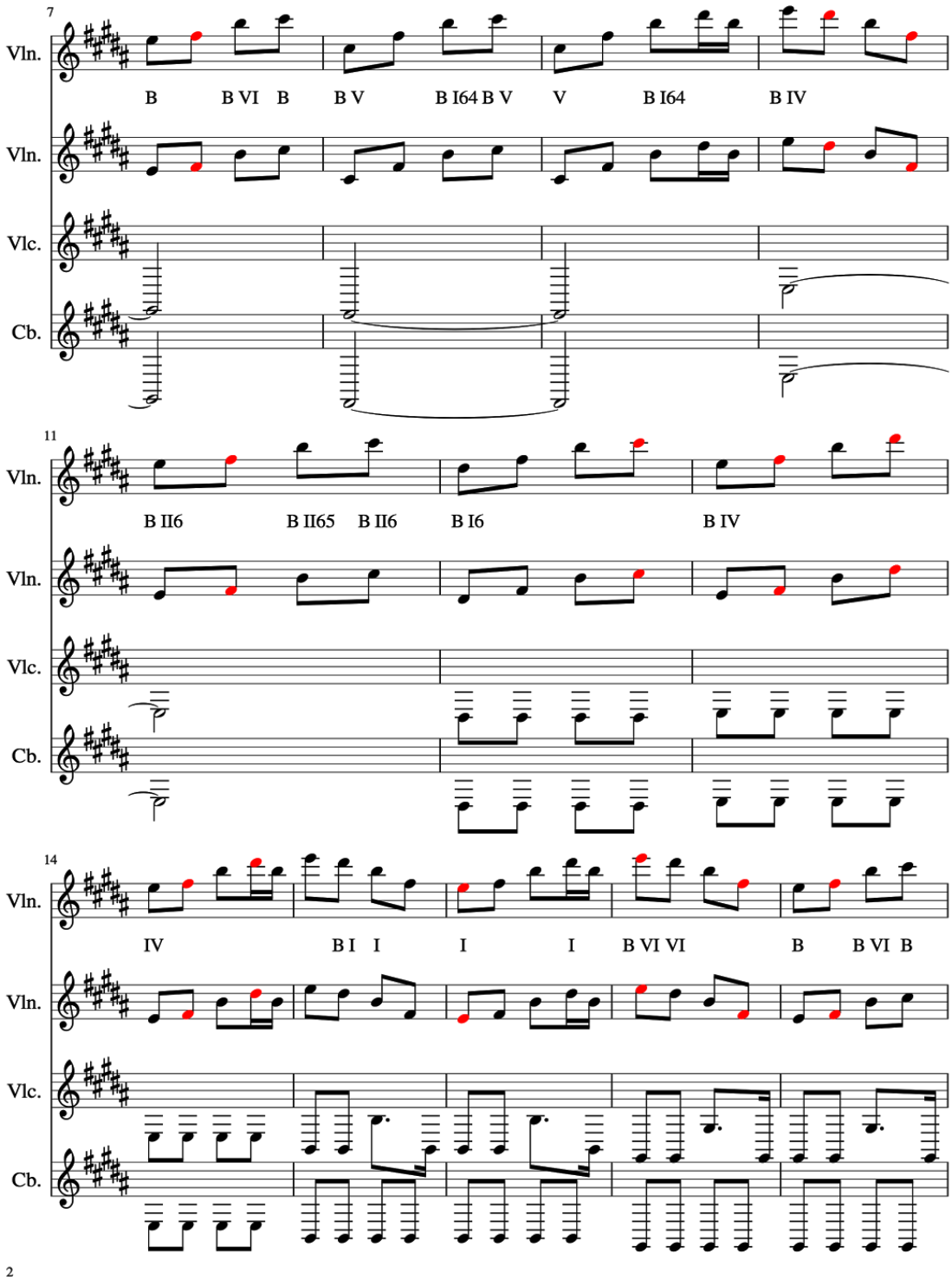
13.1.D.2 Analysis Log

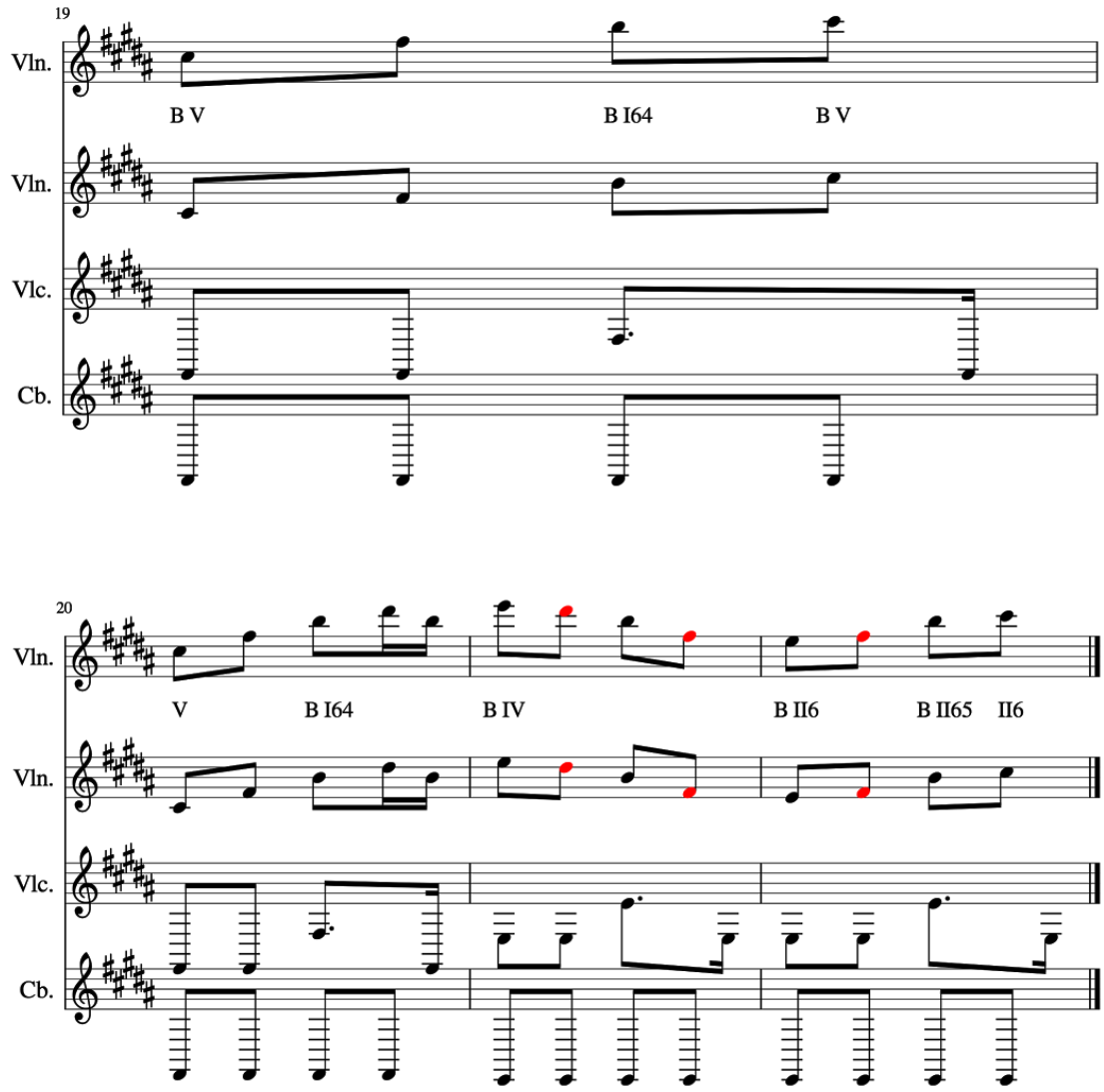
Event 2 Recognized: ['F#', 'A', 'D'] Correct: ['D', 'F#', 'A'] Reason: wrong bass
Event 3 Recognized: ['A', 'D', 'F#'] Correct: ['D', 'F#', 'A'] Reason: wrong bass
Event 6 Recognized: ['E', 'A', 'C#'] Correct: ['A', 'C#', 'E'] Reason: wrong bass
Event 10 Recognized: ['D', 'F#', 'B'] Correct: ['B', 'D', 'F#'] Reason: wrong bass
Event 11 Recognized: ['F#', 'B', 'D'] Correct: ['B', 'D', 'F#'] Reason: wrong bass
Event 14 Recognized: ['C#', 'F#', 'A'] Correct: ['F#', 'A', 'C#'] Reason: wrong bass
Event 18 Recognized: ['B', 'D', 'G'] Correct: ['G', 'B', 'D'] Reason: wrong bass
Event 19 Recognized: ['D', 'G', 'B'] Correct: ['G', 'B', 'D'] Reason: wrong bass
Event 22 Recognized: ['A', 'D', 'F#'] Correct: ['D', 'A', 'F#'] Reason: wrong bass
Event 26 Recognized: ['B', 'D', 'G'] Correct: ['G', 'B', 'D'] Reason: wrong bass
Event 27 Recognized: ['D', 'G', 'B'] Correct: ['G', 'B', 'D'] Reason: wrong bass
Event 30 Recognized: ['C#', 'E', 'A'] Correct: ['A', 'C#', 'E'] Reason: wrong bass
Event 31 Recognized: ['E', 'A', 'C#'] Correct: ['A', 'C#', 'E'] Reason: wrong bass

13.1.E hoppipolla_excerpt.xml

13.1.E.1 Analysis Result




7
Vln.
B
B VI
B
B V
B I64 B V
V
B I64
B IV
Vln.
Vlc.
Cb.
11
Vln.
B II6
B II65
B II6
B I6
B IV
Vln.
Vlc.
Cb.
14
Vln.
IV
B I
I
I
I
B VI VI
B
B VI
B
Vln.
Vlc.
Cb.
2



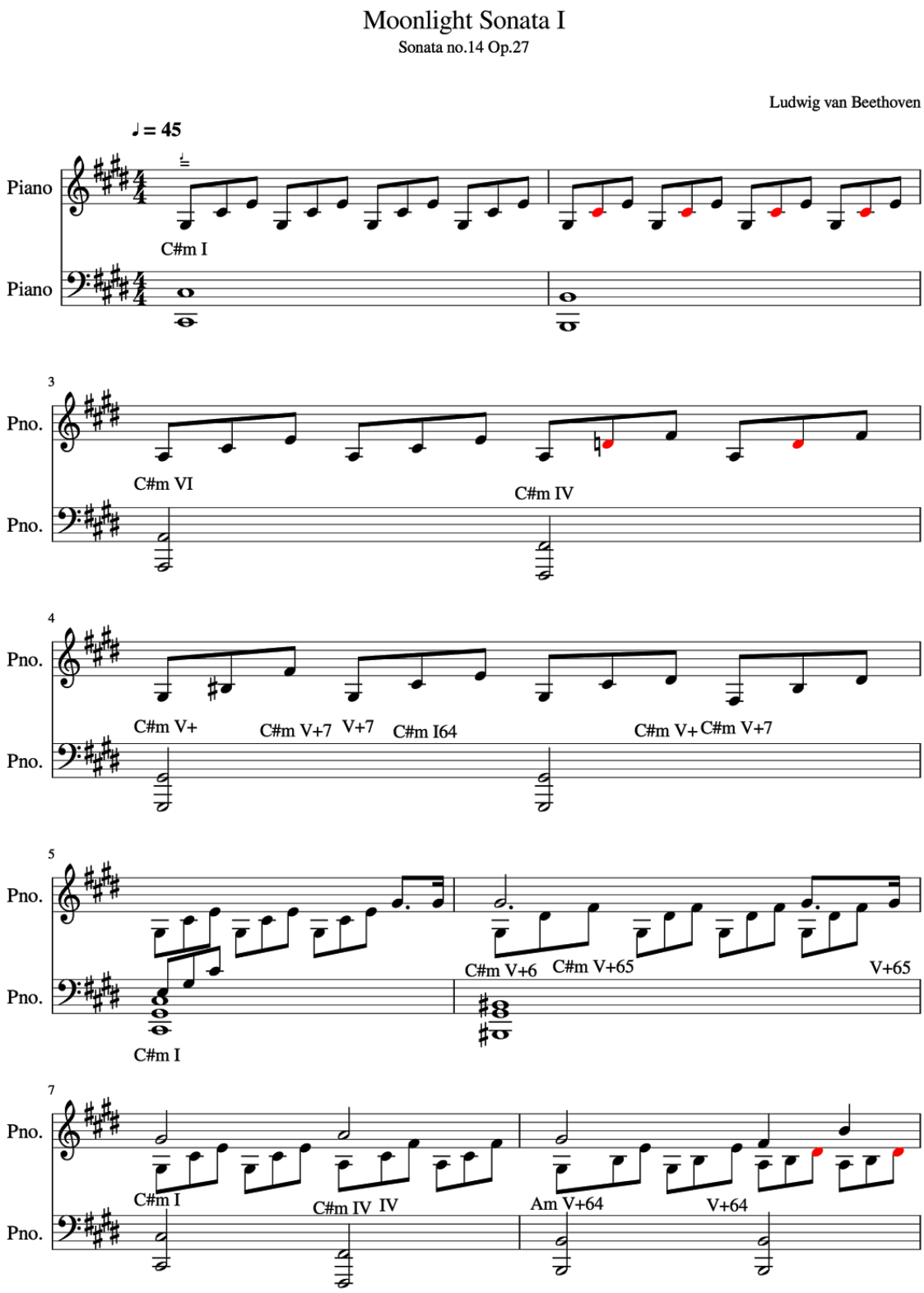
13.1.E.2 Analysis Log

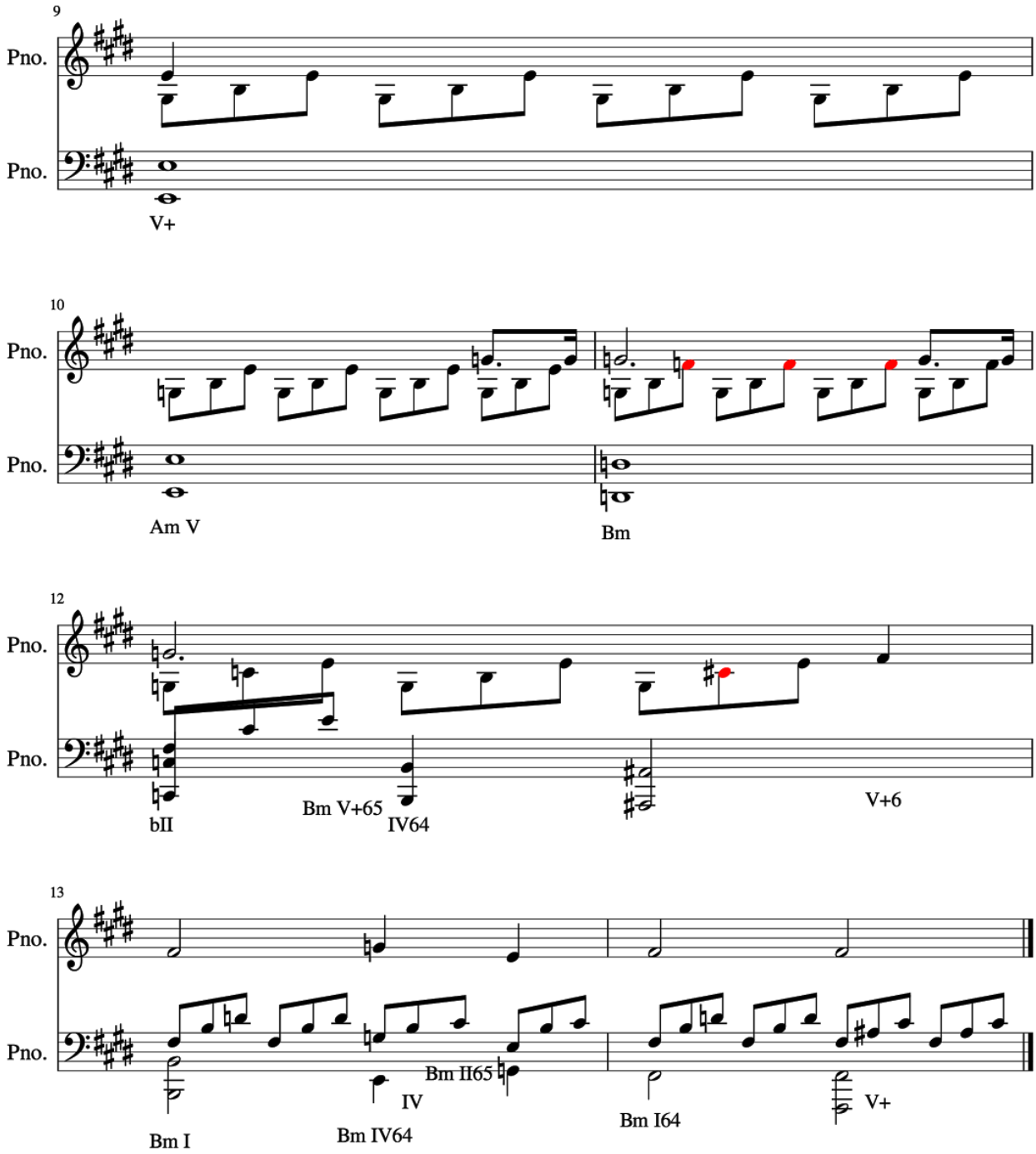
Event 4 Recognized: ['E', 'G', 'B'] Correct: ['E', 'B', 'G#'] Reason: same functionality
Event 5 Recognized: ['E', 'B', 'G'] Correct: ['E', 'B', 'G#'] Reason: same functionality
Event 6 Recognized: ['E', 'B', 'G'] Correct: ['E', 'B', 'G#'] Reason: same functionality
Event 7 Recognized: ['E', 'B', 'G'] Correct: ['E', 'B', 'G#'] Reason: same functionality
Event 8 Recognized: ['E', 'G', 'B'] Correct: ['E', 'B', 'G#'] Reason: same functionality
Event 9 Recognized: ['E', 'B', 'G'] Correct: ['E', 'B', 'G#'] Reason: same functionality
Event 10 Recognized: ['E', 'B', 'G'] Correct: ['E', 'B', 'G#'] Reason: same functionality
Event 11 Recognized: ['E', 'B', 'G'] Correct: ['E', 'B', 'G#'] Reason: same functionality
Event 12 Recognized: ['E', 'B', 'G'] Correct: ['E', 'B', 'G#'] Reason: same functionality
Event 27 Recognized: ['G#', 'E', 'C#'] Correct: ['G#', 'D#', 'B'] Reason: same functionality
Event 43 Recognized: ['E', 'C#', 'G#'] Correct: ['E', 'G#', 'B'] Reason: same functionality
Event 44 Recognized: ['E', 'C#', 'G#'] Correct: ['E', 'G#', 'B']]Reason: same functionality
Event 45 Recognized: ['E', 'B', 'C#', 'G#'] Correct: ['E', 'G#', 'B'] Reason: overly recognized
Event 76 Recognized: ['G#', 'E', 'C#'] Correct: ['G#', 'B', 'D#'] Reason: same functionality

Event 95 Recognized: ['E', 'C#', 'G#'] Correct: ['E', 'G#', 'B'] Reason: same functionality
Event 96 Recognized: ['E', 'C#', 'G#'] Correct: ['E', 'G#', 'B'] Reason: same functionality
Event 97 Recognized: ['E', 'B', 'C#', 'G#'] Correct: ['E', 'G#', 'B'] Reason: overly recognized
Event 13 Recognized: ['B', 'E', 'G'] Correct: ['B', 'D#', 'F#'] Reason:
Event 26 Recognized: ['G#', 'E', 'C#'] Correct: ['G#', 'D#', 'B'] Reason: wrong chord
Event 29 Recognized: ['G#', 'C#', 'E'] Correct: ['G#', 'D#', 'B'] Reason: wrong chord
Event 32 Recognized: ['F#', 'B', 'D#'] Correct: ['F#', 'C#', 'A#'] Reason: wrong chord
Event 36 Recognized: ['F#', 'B', 'D#'] Correct: ['F#', 'C#', 'A#'] Reason: wrong chord
Event 37 Recognized: ['F#', 'B', 'D#'] Correct: ['F#', 'C#', 'A#'] Reason: wrong chord
Event 38 Recognized: ['F#', 'B', 'D#'] Correct: ['F#', 'C#', 'A#'] Reason: wrong chord
Event 46 Recognized: ['E', 'C#', 'G#'] Correct: ['E', 'G#', 'B'] Reason: wrong chord
Event 60 Recognized: ['B', 'E', 'G#'] Correct: ['B', 'D#', 'F#'] Reason: wrong chord
Event 75 Recognized: ['G#', 'E', 'C#'] Correct: ['G#', 'B', 'D#'] Reason: wrong chord
Event 78 Recognized: ['G#', 'C#', 'E'] Correct: ['G#', 'B', 'D#'] Reason: wrong chord
Event 79 Recognized: ['G#', 'C#', 'E'] Correct: ['G#', 'B', 'D#'] Reason: wrong chord
Event 82 Recognized: ['F#', 'B', 'D#'] Correct: ['F#', 'A#', 'C#'] Reason: wrong chord
Event 87 Recognized: ['F#', 'B', 'D#'] Correct: ['F#', 'A#', 'C#'] Reason: wrong chord
Event 88 Recognized: ['F#', 'B', 'D#'] Correct: ['F#', 'A#', 'C#'] Reason: wrong chord
Event 89 Recognized: ['F#', 'B', 'D#'] Correct: ['F#', 'A#', 'C#'] Reason: wrong chord
Event 98 Recognized: ['E', 'C#', 'G#'] Correct: ['E', 'G#', 'B'] Reason: wrong chord
Event 99 Recognized: ['E', 'C#', 'G#'] Correct: ['E', 'G#', 'B'] Reason: wrong chord

13.1.F moonlight_sonata_I_excerpt.xml

13.1.F.1 Analysis Result





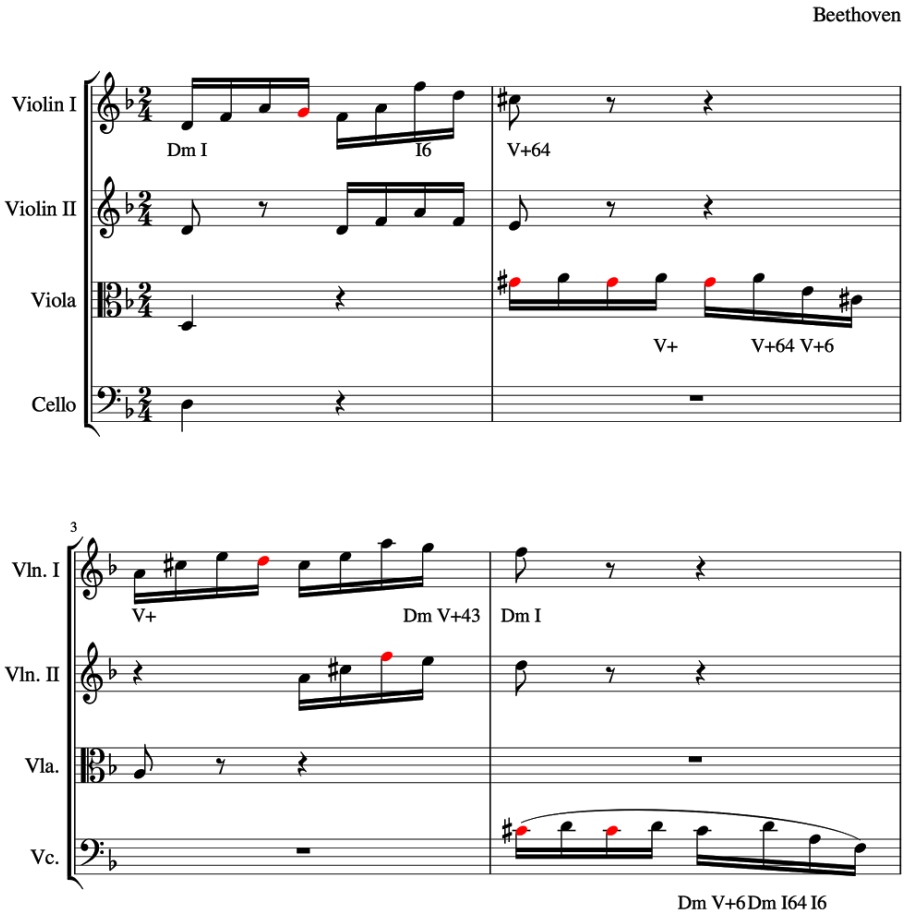
13.1.F.2 Analysis Log

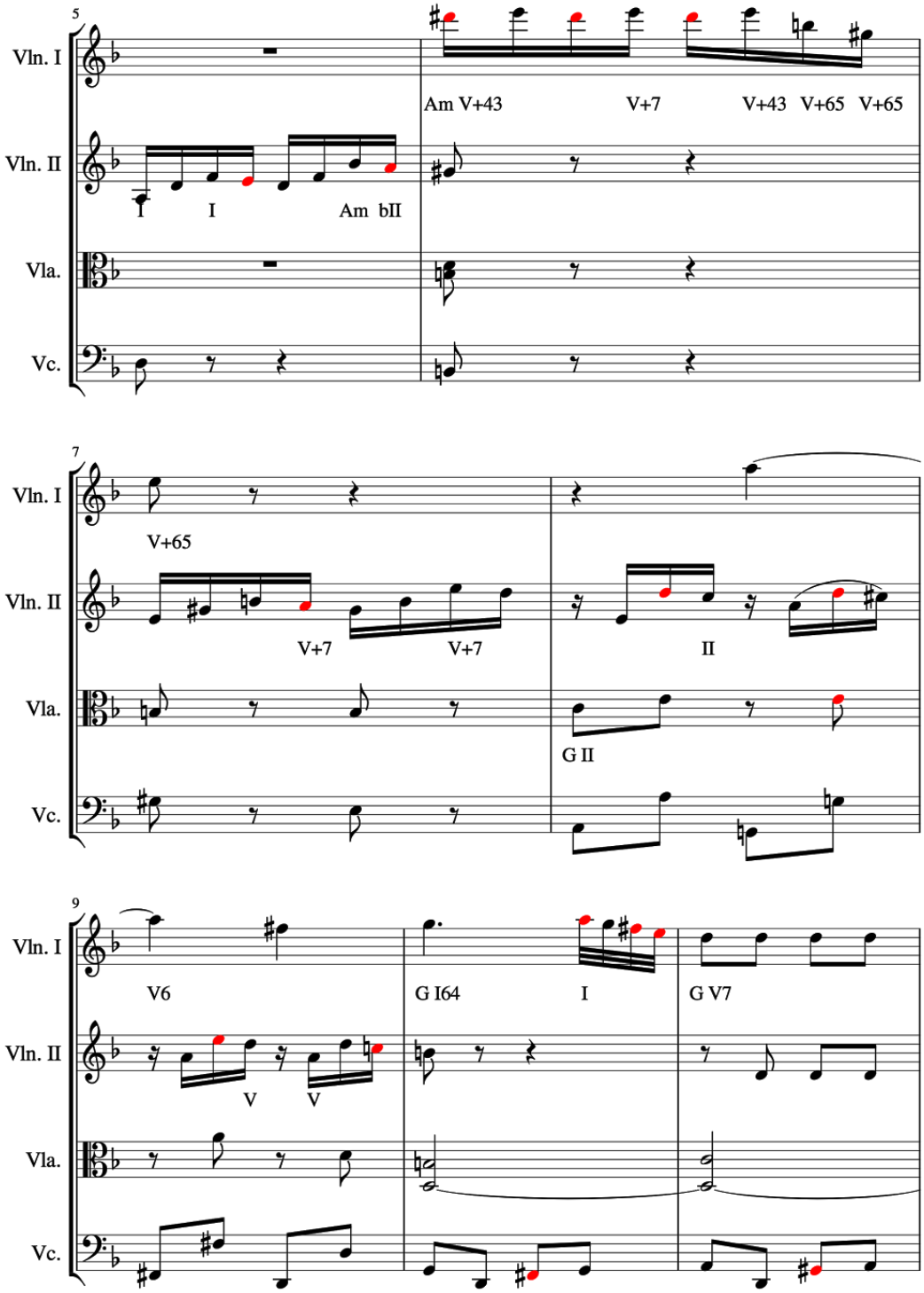
Event 30 Recognized: ['F#', 'A', 'C#'] Correct: ['F#', 'A', 'D'] Reason: same functionality
Event 32 Recognized: ['F#', 'A', 'C#'] Correct: ['F#', 'A', 'D'] Reason: same functionality
Event 33 Recognized: ['F#', 'A', 'C#'] Correct: ['F#', 'A', 'D'] Reason: same functionality
Event 35 Recognized: ['F#', 'A', 'C#'] Correct: ['F#', 'A', 'D'] Reason: same functionality

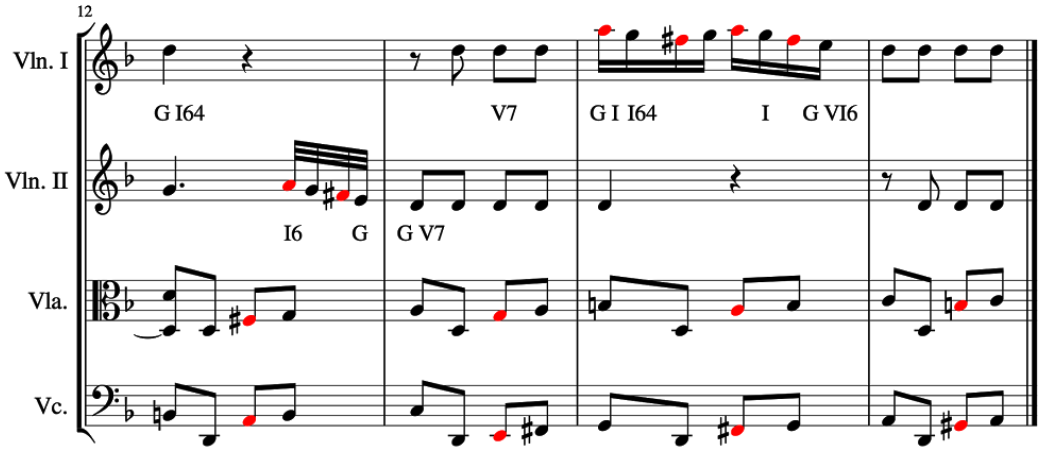
Event 39 Recognized: ['G#', 'F#', 'B#', 'D#'] Correct: ['G#', 'C#', 'E'] Reason: same functionality
Event 42 Recognized: ['G#', 'C#', 'E'] Correct: ['G#', 'B#', 'D#'] Reason: same functionality
Event 154 Recognized: ['B', 'E', 'G'] Correct: ['E', 'G', 'C#'] Reason: same functionality
Event 156 Recognized: ['E', 'G', 'B', 'C#'] Correct: ['E', 'G', 'C#'] Reason: overly recognized
Event 157 Recognized: ['E', 'G', 'B', 'C#'] Correct: ['G', 'C#', 'E'] Reason: overly recognized
Event 158 Recognized: ['G', 'E', 'B', 'C#'] Correct: ['G', 'C#', 'E'] Reason: overly recognized
Event 159 Recognized: ['G', 'E', 'B', 'C#'] Correct: ['G', 'C#', 'E'] Reason: overly recognized
Event 166 Recognized: ['F#', 'D', 'B'] Correct: ['F#', 'A#', 'C#'] Reason: same functionality
Event 31 Recognized: ['F#', 'A', 'C#'] Correct: ['F#', 'A', 'D'] Reason: wrong chord
Event 34 Recognized: ['F#', 'A', 'C#'] Correct: ['F#', 'A', 'D'] Reason: wrong chord
Event 43 Recognized: ['G#', 'C#', 'E'] Correct: ['G#', 'B#', 'D#'] Reason: wrong chord
Event 155 Recognized: ['E', 'G', 'B'] Correct: ['E', 'G', 'C#'] Reason: wrong chord

13.1.G SQ-Original.xml

13.1.G.1 Analysis Result




5
Vln. I
Vln. II
Vla.
Vc.
Am V+43
V+7
V+43 V+65 V+65
I
I
Am bII
7
V+65
V+7
V+7
II
G II
9
V6
G I64
I
G V7
V
V



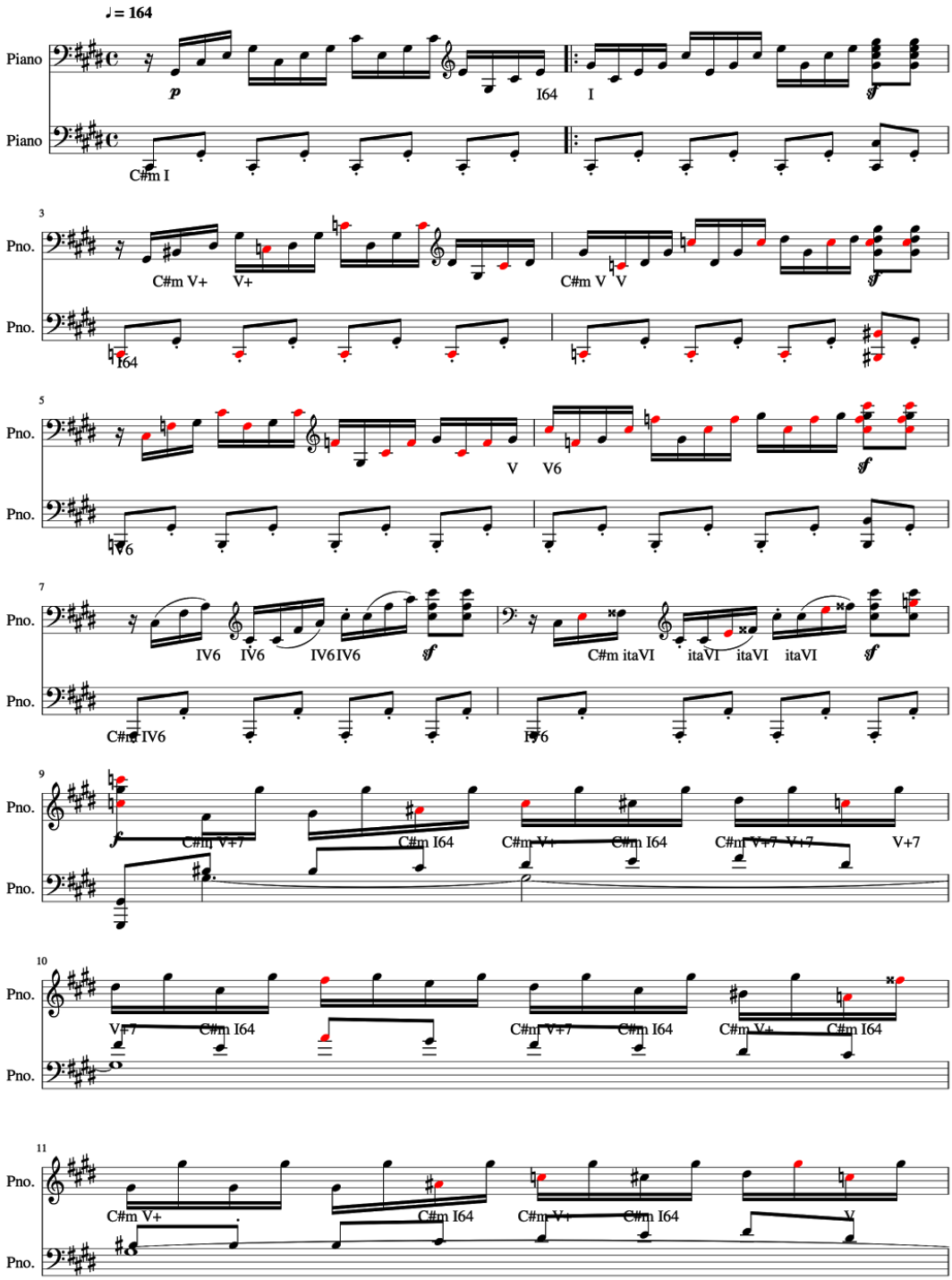
13.1.G.2 Analysis Log

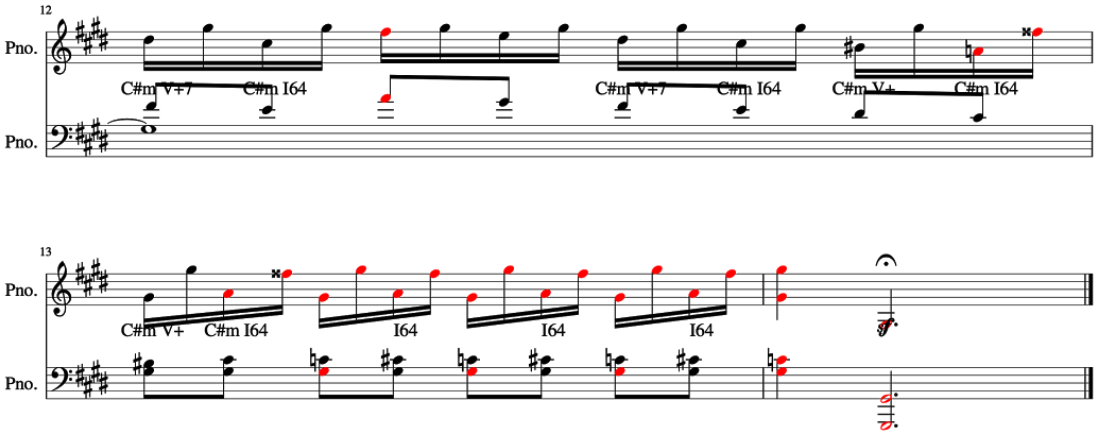
Event 6 Recognized: ['F', 'A', 'D'] Correct: ['D', 'F', 'A'] Reason: wrong bass
Event 7 Recognized: ['F', 'A', 'D'] Correct: ['D', 'F', 'A'] Reason: wrong bass
Event 8 Recognized: ['E', 'A', 'C#'] Correct: ['A', 'C#', 'E'] Reason: wrong bass
Event 9 Recognized: ['E', 'A', 'C#'] Correct: ['A', 'C#', 'E'] Reason: wrong bass
Event 10 Recognized: ['E', 'A', 'C#'] Correct: ['A', 'C#', 'E'] Reason: wrong bass
Event 13 Recognized: ['E', 'A', 'C#'] Correct: ['A', 'C#', 'E'] Reason: wrong bass
Event 14 Recognized: ['C#', 'E', 'A'] Correct: ['E', 'A', 'C#'] Reason: wrong bass
Event 22 Recognized: ['C#', 'E', 'A'] Correct: ['A', 'C#', 'E'] Reason: wrong bass
Event 23 Recognized: ['E', 'G', 'A', 'C#'] Correct: ['A', 'C#', 'E', 'G'] Reason: wrong bass
Event 29 Recognized: ['A', 'D', 'F'] Correct: ['D', 'A', 'F'] Reason: wrong bass
Event 30 Recognized: ['F', 'A', 'D'] Correct: ['A', 'F', 'D'] Reason: wrong bass
Event 43 Recognized: ['E', 'B', 'D', 'G#'] Correct: ['B', 'D', 'E', 'G#'] Reason: wrong bass
Event 44 Recognized: ['E', 'B', 'D', 'G#'] Correct: ['B', 'D', 'E', 'G#'] Reason: wrong bass
Event 46 Recognized: ['G#', 'B', 'E', 'D'] Correct: ['B', 'D', 'E', 'G#'] Reason: wrong bass
Event 47 Recognized: ['G#', 'B', 'D', 'E'] Correct: ['B', 'D', 'E', 'G#'] Reason: wrong bass
Event 48 Recognized: ['G#', 'B', 'E', 'D'] Correct: ['G#', 'B', 'E'] Reason: overly recognized
Event 49 Recognized: ['G#', 'B', 'E', 'D'] Correct: ['G#', 'B', 'E'] Reason: overly recognized
Event 50 Recognized: ['G#', 'B', 'E', 'D'] Correct: ['G#', 'B', 'E'] Reason: overly recognized
Event 51 Recognized: ['E', 'B', 'G#', 'D'] Correct: ['G#', 'B', 'E'] Reason: overly recognized
Event 52 Recognized: ['E', 'B', 'G#', 'D'] Correct: ['E', 'G#', 'B'] Reason: overly recognized
Event 53 Recognized: ['E', 'B', 'G#', 'D'] Correct: ['E', 'G#', 'B'] Reason: overly recognized
Event 54 Recognized: ['E', 'B', 'D', 'G#'] Correct: ['E', 'G#', 'B'] Reason: overly recognized
Event 67 Recognized: ['D', 'F#', 'A'] Correct: ['F#', 'A', 'D'] Reason: wrong bass
Event 72 Recognized: ['D', 'G', 'B'] Correct: ['G', 'B', 'D'] Reason: wrong bass
Event 73 Recognized: ['D', 'G', 'B'] Correct: ['G', 'B', 'D'] Reason: wrong bass
Event 75 Recognized: ['G', 'D', 'B'] Correct: ['D', 'G', 'B'] Reason: wrong bass
Event 76 Recognized: ['G', 'D', 'B'] Correct: ['D', 'G', 'B'] Reason: wrong bass
Event 79 Recognized: ['D', 'A', 'C', 'F#'] Correct: ['A', 'C', 'D', 'F#'] Reason: wrong bass

Event 83 Recognized: ['D', 'B', 'G'] Correct: ['B', 'D', 'G'] Reason: wrong bass
Event 84 Recognized: ['D', 'B', 'G'] Correct: ['B', 'D', 'G'] Reason: wrong bass
Event 86 Recognized: ['B', 'G', 'D'] Correct: ['D', 'G', 'B'] Reason: wrong bass
Event 87 Recognized: ['B', 'G', 'D'] Correct: ['D', 'G', 'B'] Reason: wrong bass
Event 90 Recognized: ['D', 'C', 'A', 'F#'] Correct: ['A', 'C', 'D', 'F#'] Reason: wrong bass
Event 91 Recognized: ['D', 'C', 'A', 'F#'] Correct: ['D', 'F', 'A', 'C#'] Reason: same functionality
Event 92 Recognized: ['D', 'F#', 'A', 'C'] Correct: ['D', 'F', 'A', 'C#'] Reason: same functionality
Event 93 Recognized: ['F#', 'A', 'D', 'C'] Correct: ['F#', 'A', 'D'] Reason: overly recognized
Event 95 Recognized: ['D', 'G', 'B'] Correct: ['G', 'B', 'D'] Reason: wrong bass
Event 102 Recognized: ['D', 'A', 'C', 'F#'] Correct: ['A', 'C', 'D', 'F#'] Reason: wrong bass
Event 28 Recognized: ['C#', 'A', 'E'] Correct: ['D', 'A', 'F'] Reason: wrong chord
Event 38 Recognized: ['F', 'B-', 'D'] Correct: ['D', 'A', 'F'] Reason: wrong chord
Event 39 Recognized: ['B-', 'F', 'D'] Correct: ['D', 'A', 'F'] Reason: wrong chord
Event 89 Recognized: ['B', 'G', 'E'] Correct: ['B', 'D', 'G'] Reason: wrong chord
Event 96 Recognized: ['D', 'G', 'B'] Correct: ['D', 'F#', 'A'] Reason: wrong chord
Event 97 Recognized: ['D', 'G', 'B'] Correct: ['D', 'F#', 'A'] Reason: wrong chord
Event 98 Recognized: ['D', 'G', 'B'] Correct: ['F#', 'A', 'D'] Reason: wrong chord
Event 99 Recognized: ['G', 'B', 'D'] Correct: ['F#', 'A', 'D'] Reason: wrong chord
Event 100 Recognized: ['G', 'B', 'D'] Correct: ['F#', 'A', 'D'] Reason: wrong chord
Event 101 Recognized: ['G', 'B', 'E'] Correct: ['F#', 'A', 'D'] Reason: wrong chord

13.2 Wrongly Notated Score

The score below is the program output from inputting a wrongly notated *Moonlight Sonata, III*




12
Pno.
Pno.
C#m V+7
C#m I64
C#m V+7
C#m I64
C#m V+
C#m I64
13
Pno.
Pno.
C#m V+
C#m I64
I64
I64
I64

13.3 Piano Reduction Score Results

13.3.A Dissonance Reduction Algorithm On

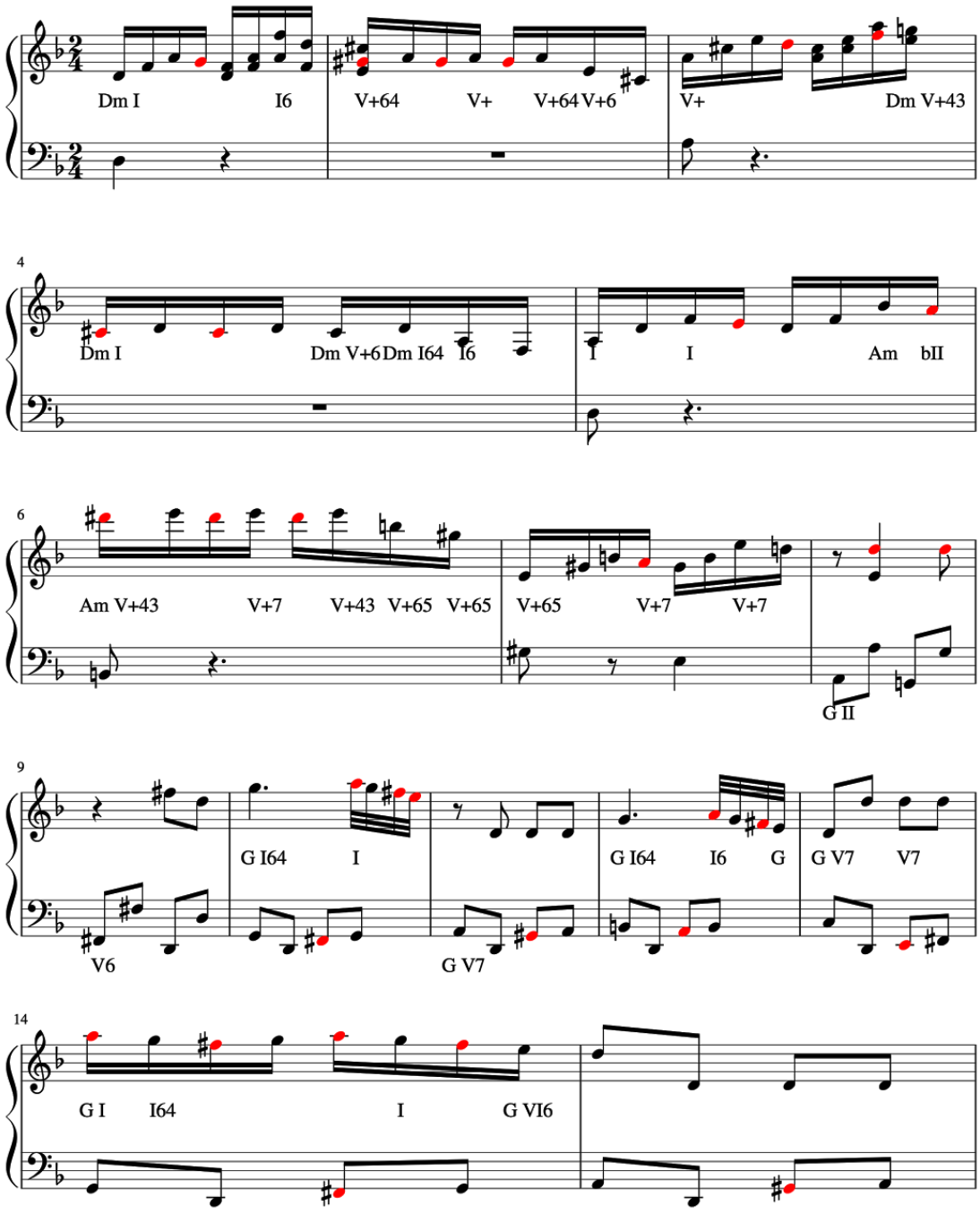
‘OnsetAfterRest’ : 1
‘StrongBeats” : 1
‘StrongBeatsDivision’ : 0.5
‘ActiveRhythm’ : 1
‘SustainedRhythm’ : 1
‘RhythmVariety’ : 1
‘VerticalDoubling’ : 1
‘Occurrence” : 1
‘PitchClassStatistics’ : 1
‘PitchClassStatisticsBefore’ : 0
‘PitchClassStatisticsAfter’ : 0
‘BassLine’ : 1
‘EntranceEffect’ : 1
‘Dissonance’ : 1

‘targetXml’ : ‘SQ-Original.xml’
‘sampleInXml’ : [‘SQ-Original.xml’]
‘sampleOutXml’ : [SQ-Important entrances plus bass line 1.xml’]

‘epoch’ : 300

Piano reduction: SQ-Original w/ Dissonance On

Arranged by KY1601



13.3.B Dissonance Reduction Algorithm Off

‘OnsetAfterRest’ : 1
‘StrongBeats” : 1
‘StrongBeatsDivision’ : 0.5
‘ActiveRhythm’ : 1
‘SustainedRhythm’ : 1
‘RhythmVariety’ : 1
‘VerticalDoubling’ : 1
‘Occurrence” : 1
‘PitchClassStatistics’ : 1
‘PitchClassStatisticsBefore’ : 0
‘PitchClassStatisticsAfter’ : 0
‘BassLine’ : 1
‘EntranceEffect’ : 1
‘Dissonance’ : 0

‘targetXml’ : ‘SQ-Original.xml’
‘sampleInXml’ : [‘SQ-Original.xml’]
‘sampleOutXml’ : [SQ-Important entrances plus bass line 1.xml’]

‘epoch’ : 300

Piano reduction: SQ-Original w/ Dissonance Off

Arranged by KY1601