YaleNUSCollege

Synthesizing Musical Harmony using Equality Graphs

Lee Juwon

Capstone Final Report for BSc (Honours) in Mathematical, Computational and Statistical Sciences

Supervised by: Prof. Ilya Sergey

AY 2021/2022

Yale-NUS College Capstone Project

DECLARATION & CONSENT

1. I declare that the product of this Project, the Thesis, is the end result of my own work and that due acknowledgement has been given in the bibliography and references to ALL sources be they printed, electronic, or personal, in accordance with the academic regulations of Yale-NUS College.

2. I acknowledge that the Thesis is subject to the policies relating to Yale-NUS College Intellectual Property (Yale-NUS HR 039).

ACCESS LEVEL

3. I agree, in consultation with my supervisor(s), that the Thesis be given the access level specified below: [check one only]

✓ Unrestricted access
Make the Thesis immediately available for worldwide access.

o Access restricted to Yale-NUS College for a limited period
Make the Thesis immediately available for Yale-NUS College access only from _____________ (mm/yyyy) to _______________ (mm/yyyy), up to a maximum of 2 years for the following reason(s): (please specify; attach a separate sheet if necessary):
____.

After this period, the Thesis will be made available for worldwide access.

o Other restrictions: (please specify if any part of your thesis should be restricted)
____
____

Juwon Lee, Cendana College
____
Name & Residential College of Student

___________________________________ April 4, 2022
Signature of Student Date

Ilya Sergey ___________________________________ April 4, 2022
Name & Signature of Supervisor Date

Acknowledgements

I am indebted to the people who have shaped me at Yale-NUS College.

First, Professor Ilya Sergey for seeing me through introductory computer science courses to capstone. I could not have finished college without his patience and guidance in the past three years.

Second, friends at Verse Lab: Kiran for helping me with troubleshooting with his intelligence and openness. Yunjeong for teaching me to be diligent and infusing courage into me since Intro to Data Structures to now.

Raphael, for imparting his musical wisdom for this project in the short time that he came back to Singapore.

Lastly to my dear friends and family, thank you for fueling me with love every step of the way. The unexpected friendships in my final year that made YNC home–Sewen, Liam, Hun, Yashmit, Minu, Dodi–thank you. My graduated friends Yejin, Tiffany, Josh, Nathasha and more at CF, Fellowship, RHC, and New City Church, I'm grateful for your impact on my faith. *Soli Deo Gloria!*

YALE-NUS COLLEGE

Abstract

B.Sc (Hons)

Automating Musical Harmony Using Equality Graphs

by Juwon LEE

Harmonization is an important building block of musical composition and performance. Over the past few decades, the process of harmonization has been automated through various programming language techniques based on formalized music theory. We present a harmony synthesis strategy that builds on the previous works, especially Harmtrace [5] and FComp [6], to enhance the performance of selecting optimized output chords given an input sequence of chords. We utilize equality graphs, a data structure that allows for efficient storing and extraction of nodes in an abstract syntax tree. This report will survey basic music theory and the definition of equality graphs and explain the pipeline of our model by presenting how a sequence of chords is parsed, saturated, and extracted.

Contents

Acknowledgements **2**

Abstract **3**

1 Introduction **6**

1.1 Thesis 6

1.2 Motivation 6

1.3 Goal of this project 7

1.4 Our Approach 7

1.5 Challenges and Contributions 8

1.6 Outline 9

2 Background **10**

2.1 Basic Music Theory 10

2.2 Automated Music 11

2.2.1 HarmTrace 12

2.2.2 FComp 14

2.3 Equality Graph (E-Graph) 14

2.3.1 Introduction to e-graphs 15

2.3.2 Ego: E-graphs OCaml Library 15

2.3.3 Initialization 16

2.3.4 Rewriting and Saturation 17

2.3.5 Extraction 18
2.3.6 Using e-graphs for harmonization 20

3 Encoding **21**
3.1 Parsing 21
3.2 E-graph Rewriting 23
3.3 E-graph Extraction 25

4 Optimizations **27**
4.1 Rewrite rules 27
4.1.1 Functional Harmony Rules 27
4.1.2 Jazz Harmony Rules 29
4.2 Cost function 30

5 Evaluation **31**
5.1 Method 31
5.2 Process 32
5.2.1 Optimizing for major shorthand 32
5.2.2 Optimizing for minor shorthand 32
5.3 Results 33

6 Discussion **35**
6.1 Conclusion 35
6.2 Future Work 35

Bibliography **37**

Chapter 1

Introduction

> Music creates order out of chaos: for rhythm imposes unanimity upon the divergent, melody imposes continuity upon the disjointed, and harmony imposes compatibility upon the incongruous.
>
> *Yehudi Menuhin*

1.1 Thesis

Harmonization is an optimization problem.

1.2 Motivation

Harmony is the combination of musical notes that sound simultaneously. It enriches the melody line by imbuing emotions and modulating the tension of the musical piece at the appropriate phrase. Think orchestra or choir as opposed to a single violin playing or a single singer singing. Regardless

of whether the listener is musically trained or not, good harmony can be easily distinguished from incongruous harmony. Therefore, creating good harmony is pivotal for musical composition.

What was once thought to be in the subjective realm of musically-trained composers, the creation of music has been subject to formalization and automation in the past few decades. This is because music, more than other forms of creative art, has regularity and order, allowing the process of harmonization to be automated [4, 5, 6].

This is where music and computer science intersect. Just as compilers optimize code to enhance the program performance, for example by eliminating redundant code, harmonization can be optimized to enhance the musical piece.

1.3 Goal of this project

Musical pieces are programs that can be optimized to sound better by adding, deleting, and expanding individual chords. The goal of our project is to find enriched harmony using program optimization.

1.4 Our Approach

This project provides a proof of concept for utilizing **equality graphs (e-graphs)** as a method of optimizing musical harmony. An e-graph is a non-destructive graph data structure that stores equivalent programs. The process of storing the equivalent programs is called *rewriting*, and e-graphs are *saturated* once the rewriting is complete. Given a cost function, the saturated e-graph is traversed to output the program with the lowest cost. This

process is called *extraction* [11]. This is an alternative to traditional and sequential methods of optimization as it mitigates the problems encountered with traditional optimization techniques such as the phase-ordering problem [11]. It is a technique widely used in fields such as manufacturing [7, 12], machine learning and scientific computing [9]. With the proven efficiency of e-graphs in multiple domains, our model applies the optimization technique to musical harmony.

1.5 Challenges and Contributions

Rewriting the e-graph is a crucial part of the optimization strategy as the quality of the rewriting defines the quality of the optimization result. Since there is no prior work done that accumulates conventional harmony composition rules, the challenge of the project was gathering these theories.

The project is significant in the following aspects:

1. It provides a novel formulation of e-graphs and equivalence classes on trees of musical harmony.
2. It provides a novel set of rewrite rules guided by jazz-style harmony.
3. It allows for a user-guided cost function for music rewriting.
4. It provides an open-source implementation available for experiments and extensions, which can be found on GitHub.[1]

[1] `https://github.com/juwonzylee/emu`

1.6 Outline

The report structure is as follows. In chapter 2, we survey existing literature and toolchains on automated music and equality graphs. In chapter 3, we explain how equality graphs were used to optimize musical harmony. In chapter 4, we explain the rewrite rules and the cost function in depth. In chapter 5, we provide a specific example of a sequence of musical chords being optimized through the project's pipeline. Finally, chapter 6 concludes with a discussion on the project and possible future work.

Chapter 2

Background

In this chapter, we first provide a minimalistic explanation of functional harmony theory in section 2.1. Then we highlight the cornerstones of automated music and dive in-depth about Harmtrace and FComp in section 2.2. Finally, we provide the motivation, structure, and applications of equality graphs in section 2.3.

2.1 Basic Music Theory

A *tone* is a sound with a fixed frequency. The tones are denoted by *notes* based on their pitches. When two or more tones sound simultaneously, *harmony* is made. Harmony is denoted by *chords*, which is a group of tones. A chord is identified by its *chord root*, which is the note that the chord is built on, and the *intervals* between the notes.

Functional Harmony Theory The functional harmony theory states that each chord within a key can be reduced to one of three functions within its tonal context.

1. **Tonic:** affirms the key, releases tension

2. **Subdominant:** builds tension and prepares dominant chord

3. **Dominant:** builds maximum tension

Chords have different functions in different tonal contexts. For example, in a C major key, the C major chord is a tonic chord, whereas it would be a dominant chord in an F major key. The relationships between the keys and the functionality of the chords are shown in Figure 2.1. This functionality will be the basis of the rewrite rules in our model.

Key	*I (T)*	*ii (SD)*	*iii (T)*	*IV (SD)*	*V (D)*	*vi (SD)*	*viio (D)*
A	A	Bm	C#m	D	E	F#m	G#o
B	B	C#m	D#m	E	F#	G#m	A#o
C	C	Dm	Em	F	G	Am	Bo
D	D	Em	F#m	G	A	Bm	Bo
E	E	F#m	G#m	A	B	C#m	D#o
F	F	Gm	Am	Bb	C	Dm	Eo
G	G	Am	Bm	C	D	Em	F#o

FIGURE 2.1: Function of chords in different tonal contexts. T stands for tonic, SD for subdominant, and D for dominant. The roman numerals denote the chord root, uppercase being a major chord and lower case being a minor chord.

Formalized rules of functional harmony theory can serve as a guide to harmony composition and optimization.

2.2 Automated Music

Now that we have seen the importance of formalized music grammar, we briefly survey the history of automated music.

Based on the formalized functional harmony theory, Steedman was among the first to propose a generative grammar for blues chord progressions [10].

The first musical piece to be generated by a computer is the *Illiac Suite* by the Illiac computer [3]. The computer generated certain musical materials according to their tonal functions and selected the best option from the candidates according to various rules [1]. From then on, modern composers have integrated algorithmic generated music into their compositions. Iannis Xenakis's Atrées [14] and Morsima-Amorsima [15] are example compositions that deduced a score from a "list of note densities and probabilistic weights supplied by the programmer, leaving specific decisions to a random number generator" [1].

In the past decade, tonal harmony has been further integrated into algorithms. Tonal harmony was modeled as formal grammar by Rohrmeier [8] and the formal grammar was implemented by De Hass et al [2]. Magalhães [5] expanded on the work of Rohrmeier [8] and proposed HarmTrace, a parser in Haskell that automatically derives the input chord's harmonic function. This allowed for a better estimation of harmonic similarity between different sequences of chords. HarmTrace has been used in generating a sequence of chords to accompany an input melody line [4].

Building on the work of Koop [4], Magalhães [6] proposed FComp, a system that generates both the harmony and the melody that accompanies the harmony. FComp generates candidate chords based on the basic rules of functional harmony, which will be explained in depth below.

2.2.1 HarmTrace

HarmTrace takes in an input sequence of chords denoted with its root, accidental, and extension and parses them with chord labels and scale degrees along with the functional analysis of each of the chords and phrases.

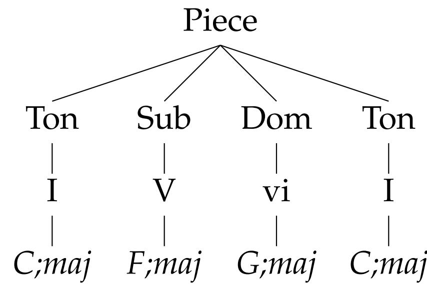

FIGURE 2.2: Pretty printed tree of the original chord sequence.

The syntax written in Haskell is shown below.

```
data Piece = Piece [Phrase]
data Phrase = PT Ton | PD Dom
data Ton = T(IMaj) Degree
data Dom = D(VMaj) Degree | D(SD,D) SDom Dom
data SDom = S(IVMaj) Degree
```

We see a musical piece as a list of phrases. A phrase is either a tonic or a dominant, which refers to the theory of functional harmony. A tonic is simply the first scale degree, while a dominant might branch into a subdominant and a dominant, or simply be the fifth degree.

Here is an example input sequence consisting of C major, F major, D7, G7, and C major.

```
C:maj F:maj G:maj C:maj
```

The example sequence is parsed into a tree, where this sequence of chords, or type piece, is divided into the tonic phrase, dominant phrase, and the tonic phrase. The parsed tree is shown in Figure 2.2.

The parsed sequence is then used for music recognition based on the harmonic structure. Research done afterward utilizes the Harmtrace parser for other applications such as melody and harmony generation [4, 6].

2.2.2 FComp

Building on HarmTrace, FComp generates foundational harmony based on random values of a datatype that encodes the rules of tonal harmony [6]. FComp encodes basic harmony rules without secondary dominants, tritone substitutions, and more.

The general harmony rules used by FComp are as follows:

1. The tonic phrase consists of the *I* chord.

2. A dominant can expand to the following: a dominant or major chord built on the *V* chord, or a diminished *VII* chord. It can also be preceded by a subdominant or a secondary dominant on the *II* chord.

3. The subdominant, in major mode, can either be *II*:Min chord or *IV*:Maj, preceded by a *III*:Min. In minor mode, a subdominant is realized with *IV*:Min chord.

We will be using the given rules and expanding on them to include more complex functional harmony rules and jazz-style harmony rules in our model in chapter 4.

2.3 Equality Graph (E-Graph)

In order to enhance the quality of the chords, we introduce equality graphs, or e-graphs for short, a data structure to store candidate chords in a non-destructive manner, resolving a phase-ordering problem of traditional optimization strategies.

2.3.1 Introduction to e-graphs

Movivation In traditional optimization, the program is made incrementally better with an applied sequence of optimization rules. Because the optimization rules are applied sequentially, if the program is optimized in a different order than the previous optimization, there would be a different optimized result. If there exists an exponential number of orderings, the traditional optimization would randomly output the optimized version, instead of traversing through the whole set of possible candidate outcomes. This is called the *phase ordering problem.* In light of this problem, equality graphs ([11]) have been proposed as an alternative optimization method to traditional optimization.

2.3.2 Ego: E-graphs OCaml Library

Ego (EGraphs OCaml) is an OCaml library that provides generic equality saturation using e-graphs.[1] This has been translated from the original egg library in Rust into OCaml by the Verse lab at NUS. The design of Ego provides two flexible interfaces, one to run basic equality saturation and one extended with custom user-defined analyses.

Among the two interfaces to the library, we use *Ego.Basic* for syntactic rewrites and visualizations of e-graphs. In order to understand how the library works, we work with a simple example.

[1] `https://verse-lab.github.io/ego/ego/index.html`

2.3.3 Initialization

Consider the mathematical program $(x \ll 1)/2$. In order to initialize the e-graph for our example program, we define the program as a sub-expression.

```
let expr =
    List [Atom "/"; List[Atom "<<"; Atom "x"; Atom "1"]; Atom "2"]
```

We then initialize an e-graph and add the sub-expression using the method *Ego.Basic.EGraph.add_sexp*. Adding the sub-expression yields an expression id, which will be used for extraction.

```
let graph = EGraph.init()
let expr_id = EGraph.add_sexp graph expr
```

Now our example program $(x \ll 1)/2$ can be visualized using the helper function *Ego.Basic.EGraph.to_dot*. The GraphViz representation of the initial e-graph is shown in Figure 2.3. Each of the equivalence classes is denoted by their expression id's, such as $e0$, $e1$, and so on.

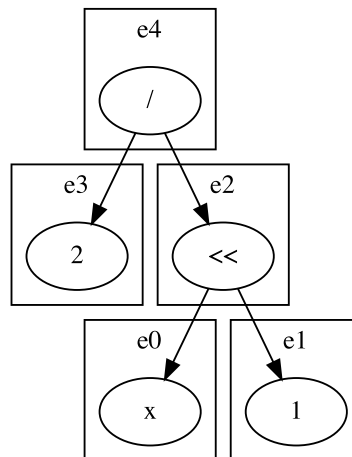

FIGURE 2.3: The e-graph representation of $(x \ll 1)/2$.

2.3.4 Rewriting and Saturation

Now here are the two rewrite rules we wish to use to saturate the e-graph:

1. $x \ll 1$ is equivalent to $x * 2$.

2. $x * 2/2$ is equivalent to x.

Using the helper function *Ego.Basic.EGraph.run_until_saturation*, we apply the two rules. Here is a simplified version of the code snippet for readability.

```
let expr1 = (a<<1) in
let expr2 = (a*2) in
let rule1 = (Rule.make ~from ~into) in

let expr3 = ((a*2)/2) in
let expr4 = (a) in
let rule2 = (Rule.make ~from:from1 ~into:into1) in

let _ = EGraph.run_until_saturation graph [rule1;rule2] in
```

Now the e-graph stores the following equivalent programs in a non-destructive form:

1. $(x \ll 1)/2$

2. $(x * 2)/2$

3. x

The saturated e-graph is visualized in Figure 2.4.

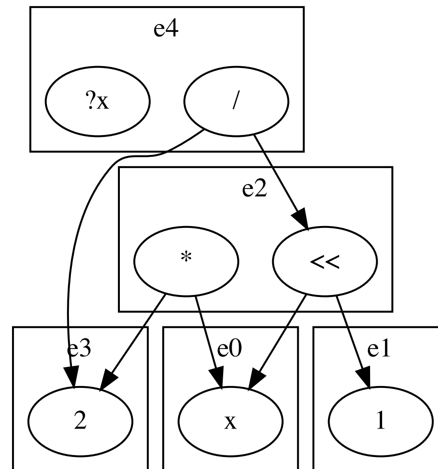

FIGURE 2.4: The saturated e-graph visualized.

2.3.5 Extraction

Next, we search the saturated e-graph to extract the optimized program. We define a cost function to do this.

```
let cost_function score (sym, children) =
 let node_score =
    match Symbol.to_string sym with
    | "*" -> 1.
    | "/" -> 1.
    | "<<" -> 2.
    | _ -> 0. in
 node_score +. List.fold_left (fun acc vl ->
                              acc +. score vl) 0. children
```

Here, we see that the left shift operator has been given the highest cost of 2. The multiplication and division operators are given costs of 1 each.

Based on this cost function, the costs of the three equivalent programs in the saturated graph are as follows:

1. Cost of $(x \ll 1)/2$ is 3.

2. Cost of $(x * 2)/2$ is 2.

3. Cost of x is 0.

Therefore, the program x has the lowest cost, meaning it is the optimized program. We can use the helper function *Ego.Basic.EGraph.extract* to extract and return the sub expression of the optimized output, as shown in Figure 2.5.

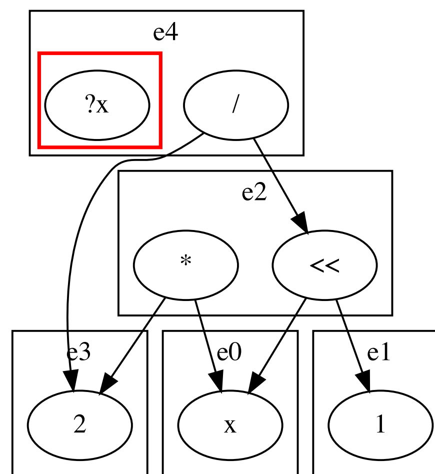

FIGURE 2.5: The red box shows the optimized program within the saturated e-graph.

Applications E-graph is a powerful and versatile optimization technique that allows new functionality and speedups in program synthesis and optimization [13]. Since the invention of the technique, there have been applications in a wide range of topics, such as 3D printing [7] where they allowed for efficient search of smaller programs, shrinking the size of the model in seconds. Diospyros [12] optimized small linear algebra kernels

for digital signal processors using e-graphs, outperforming existing DSP libraries by 3.1 on average. Pherbie [9] integrated e-graphs in precision tuning to improve the accuracy for floating point expressions. Our model expands the field of applications to music.

2.3.6 Using e-graphs for harmonization

Using the versatility of the data structure, we adapt and define the domain to apply the optimization technique to harmonization. The current generation of harmony in FComp is done in the following steps [6]:

1. An element of randomness is introduced to select candidate chords out of the valid pool of chords.
2. Each constructor has a set weight to control which rules of harmony appear more frequently over others.

Since it is impossible to consider all possible candidate chords, FComp chooses the sequence that appears more statistically. However, by introducing e-graphs to compactly store the possible candidate chord sequences, we are able to search through the space and extract the optimized chords attuned to the user's desires without introducing a random component in the optimization process.

Chapter 3

Encoding

Now that we have explored and understood the toolchain for e-graph, we can apply e-graphs to harmony automation. The pipeline of our project is shown in Figure 3.1. We first take in an input sequence of chords, which is passed to the HarmTrace parser [5], which is then parsed into OCaml sub-expressions. Using the parsed output, we initialize, saturate, and extract the e-graph to output the optimized harmony.

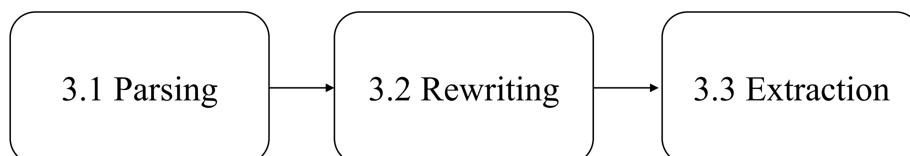

FIGURE 3.1: Implementation pipeline

3.1 Parsing

Before we can initialize an e-graph with a chord sequence, we utilize the HarmTrace parser ([5]) to parse the input sequence of chords.

The HarmTrace parser parses each chord within the input sequence into a root, extension, accidental, shorthand, and function. The function refers

to the subdivision in functional harmony theory: tonic, subdominant, and dominant.

Using this information, we encode our own datatype in OCaml. First, the input sequence is represented as type piece, which is a list of phrases. We include a constructor $DomPhraseof(phrase * phrase)$ because a dominant phrase can consist of a tuple of phrases, for example, a subdominant and a dominant.

```
type phrase =
| Ton of chord list
| Dom of chord list
| DomPhrase of (phrase * phrase)
| Sub of chord list

type piece = phrase list
```

Each constructor in the type phrase is comprised of a list of chords, whereby the type chord has four constructors: the root, accidental, shorthand, and extension. The constructors for type chord is shown below:

```
type chord = {root : root
              ; accidental : accidental
              ; shorthand : shorthand
              ; extension : extension}
type root = C | D | E | F | G | A | B
type accidental = Sharp | Natural | Flat
type shorthand = Maj | Min
type extension = Fifth | Seventh | Ninth
```

Let's take an example of a simple chord sequence in the C Major key consisting of one tonic, one subdominant, and one dominant chord. The sequence is shown in Figure 3.2.

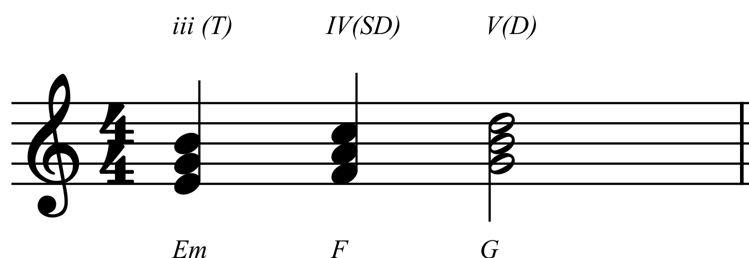

FIGURE 3.2: Chord sequence of E minor, F major, and G major in the key of C Major. The root of the chord is written in Roman numerals on top of the chords, along with the functional analysis of the chord.

3.2 E-graph Rewriting

With the input sequence parsed, we can now initialize the e-graph. We add the sub-expression of the chord sequence and obtain the e-graph visualized in Figure 3.3.

The order of the chord sequence is preserved using the cons placeholder. Reading in ascending order of the e-class expression id's, the first chord has function of Tonic ($e7$), root note of E ($e3$), and shorthand of Minor ($e1$), accidental of Natural ($e0$), and extension of Fifth ($e2$). The same applies for the next two chords.

Now let's rewrite the graph according to two the following rules:

1. **Functional Harmony Rule**: A dominant chord can be substituted by other dominant chords, and tonic chords by other tonic chords. In the tonal context of C major, the G major chord can be changed to a diminished seventh chord, and the E minor chord can be substituted by the C major chord.

2. **Extension Chord**: The fifth extension chords can be substituted by the seventh extension chords.

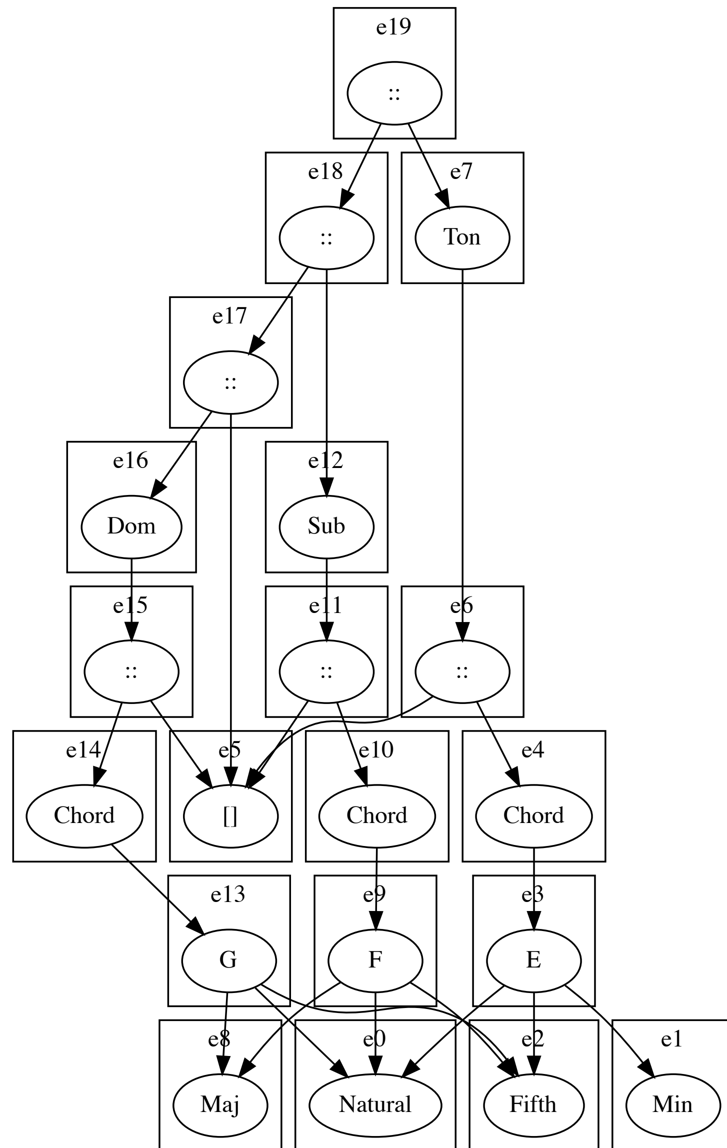

FIGURE 3.3: E-graph visualized for the original chord sequence in Figure 3.2.

Applying this rule, the saturated e-graph is shown in Figure 3.4.

We see in Figure 3.4 that the e-class with id *e*14 has been expanded to include the diminished seventh chord of G major seventh. E-class *e*4 has expanded to include the C major seventh chord.

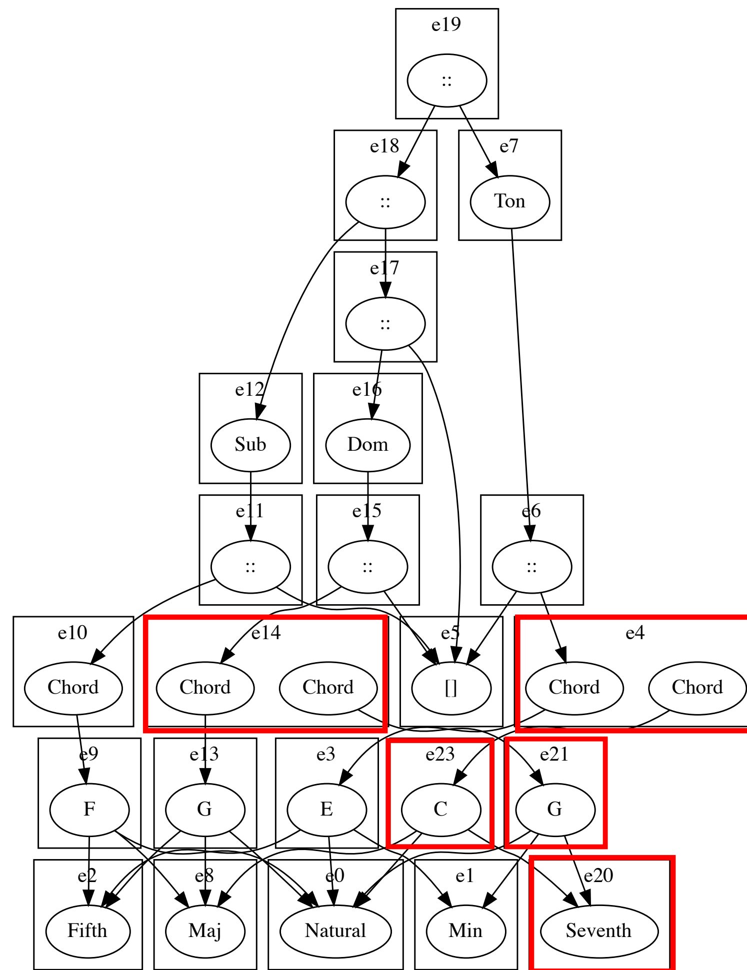

FIGURE 3.4: Visualized for the saturated e-graph. The red boxes indicate the expanded e-classes and added e-nodes.

3.3 E-graph Extraction

With the saturated e-graph example, we now wish to extract an optimized program based on a cost function. Let's assume that the cost function is defined as shown in Figure 3.5.

The minor chords are placed with a cost of 2, the major chords 1. The fifth extension has a higher cost of 2 whereas the seventh extension has cost 1. This means the major chord and the seventh extension will be prioritized for extraction.

```
let cost_function score (sym, children) =
  let node_score =
    match Symbol.to_string sym with
    | "Min" -> 2.
    | "Maj" -> 1.
    | "Fifth" -> 2.
    | "Seventh" -> 1.
    | _ -> 0. in
  node_score +. List.fold_left (fun acc vl ->
                                 acc +. score vl) 0. children
```

FIGURE 3.5: Example cost function that penalizes minor chords and fifth extensions.

According to this cost function, we obtain the following sub-expression:

```
(:: (Ton (:: (Chord (C Natural Maj Seventh)) []))
 (:: (Sub (:: (Chord (F Natural Maj Fifth)) []))
  (:: (Dom (:: (Chord (G Natural Maj Seventh)) [])) [])))
```

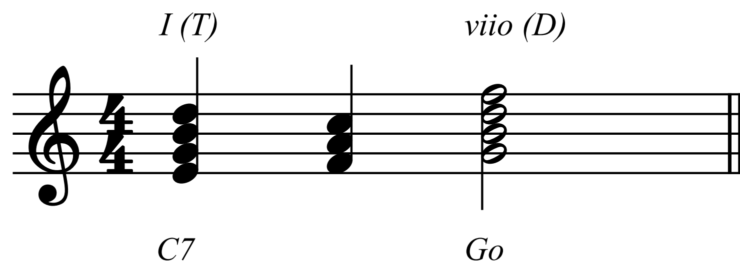

FIGURE 3.6: Output chord sequence of C major seventh, F major, and G diminished seventh major.

Based on the cost function, the seventh extensions and major shorthand have been extracted over the minor shorthand and fifth extension as visualized in Figure 3.6.

Chapter 4

Optimizations

This chapter explains the rewrite rules that are used to saturate the e-graph (section 4.1) along with the details of the cost function used for extraction (section 4.2).

4.1 Rewrite rules

Our goal is to make the input sequence of chords sound better. The more rewrite rules we have, the bigger the pool of candidate chords to choose from. We have referenced existing literature and musicians to gather widely used harmony rules.

4.1.1 Functional Harmony Rules

We begin by encoding the basic harmony rules used by Rohrmeier [6]. Refer to section 2.2 and Figure 2.1 for more details.

1. Dominant chords of a tonal context can be substituted by any other dominant chord.

2. Dominant chords can be subdivided into a dominant chord preceded by a subdominant chord.

3. Subdominant chords can be substituted by any other subdominant chord.

4. Tonic chords remain within the tonic functional chords.

We expand on the basic functional harmony rules used by Rohrmeier to include more complex rules.

5. Dominant chords can be substituted by **secondary dominant chords** as illustrated in Figure 4.1. The audio file is available here.[1]

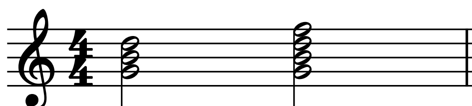

FIGURE 4.1: The first chord is the G major dominant chord. The second chord is the secondary G major seventh dominant chord.

6. **Tritone substition:** A dominant seventh chord can be substituted by a tritone (chord with three whole steps) from the original chord. This is illustrated in Figure 4.2 and the audio file is available here.[2]

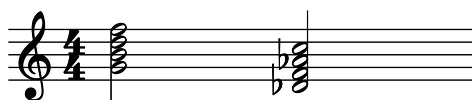

FIGURE 4.2: The first chord is the G major seventh dominant chord. The second chord is the tritone substitution of the first, the D flat major seventh chord.

[1] https://drive.google.com/file/d/1bKFG4Gl_vrIVhz4OZRvtB4LwAZ8IgPpw/view?usp=sharing

[2] https://drive.google.com/file/d/1AfJZ1nGYbJy1z2br5MfeZ_gFTGCmQs6H/view?usp=sharing

4.1.2 Jazz Harmony Rules

Next, we add the jazz harmony rules on top of the functional harmony rules.

7. **Passing Chords (2-5-1)** allow the subdominant and dominant chords to be built up with more tension. Figure 4.3 is an example of a passing chord in the C major scale. The sound is available in Google Drive.[3]

FIGURE 4.3: Example of a 2-5-1 passing chord in C Major key. The root chords are denoted in the bass clef; and the harmony is built on the seventh extension in the treble clef.

8. **Extension chords** allow the tonic, subdominant, and dominant chords to sound richer in harmony, meaning there are more emotions imbued in the chords. This is because on top of the perfect triad, the seventh or ninth note is added to the chord.

9. **Chord replacement** is another method to modulate the tension in the dominant phrase.

Based on these rewrite rules, the e-graph is saturated with all possible candidate chord sequences. Next, we explain how our program selects a chord sequence out of the equivalent programs.

[3] https://drive.google.com/file/d/1ja2NONm6yHbD25v7QtsiQUJ6CARU6EgV/view?usp=sharing

4.2 Cost function

E-graphs search through the program and all the candidate phrases and choose the one based on the given cost function. We allow the user to guide the cost function based on the emotions of the output they wish.

Emotions, as subjective as they seem, are the definition of how chords are defined. The major shorthand is known to give off happier feelings than chords with a minor shorthand. For example, the diminished seventh chords are more sad-sounding than a minor fifth chord.

With this convention, we can give different chords different costs, which is shown in Figure 4.4.

```
let cost_function score (sym, children) =
  let node_score =
    match Symbol.to_string sym with
    | "Min" -> 0.
    | "Maj" -> 1.
    | "Fifth" -> 1.
    | _ -> 0. in
  node_score +. List.fold_left (fun acc vl ->
                                  acc +. score vl) 0. children
```

FIGURE 4.4: Cost function that optimizes for minor shorthand, penalizing the major shorthand.

The cost function places a higher cost on the major chord lower cost on minor chords. In other words, the one with the major chords is being penalized. This will be the case when the user wants to obtain an output chord sequence that sounds sad. In addition, it is possible to experiment with the cost function with other aspects of the chord sequence such as the extension, accidental, and length of the sequence. For example, Figure 3.5 is a cost function that penalizes not only the shorthand of the chords but also the extension.

Chapter 5

Evaluation

5.1 Method

In this section, we take the melody of the song *Moon River* from *Tiffany's at Breakfast* as our testing example.[1] We generate two optimized outputs based on two different cost functions, one that outputs happy-sounding sequence of chords, and one that outputs sad-sounding one. The three sound files are available on google drive.[2]

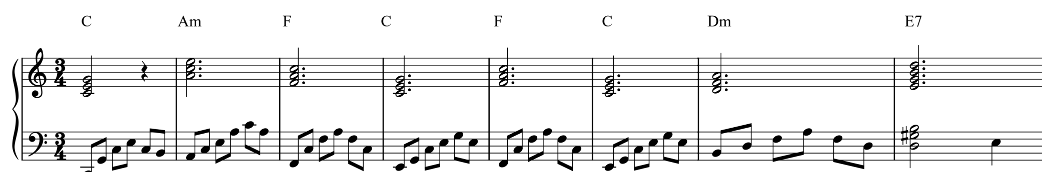

FIGURE 5.1: The original chord sequence of *Moon River*

[1] https://tabs.ultimate-guitar.com/tab/audrey-hepburn/moon-river-chords-1480951

[2] https://drive.google.com/drive/folders/1-uNNqIBGugrqaJ8wwidi8_p-Q3MPZwaX?usp=sharing

5.2 Process

5.2.1 Optimizing for major shorthand

First, we extract chord sequences that sound happy. In other words, we use a cost function that penalizes the minor shorthand. The music sheet is shown in Figure 5.2.

FIGURE 5.2: The optimized sequence that penalizes minor chords.

The red boxes in Figure 5.2 show the differences from the original chord sequence in Figure 5.1. The A minor subdominant chord has been substituted by the two-five-one passing chord sequence of C major, E minor, and G major sequence. The D minor chord has been substituted for the B flat major chord. As opposed to having two minor subdominant chords, the optimized version has substituted them for major subdominant chords.

5.2.2 Optimizing for minor shorthand

Next, we extract chord sequences that sound sad. This time we penalize the major shorthand. The music sheet is shown in Figure 5.3

This time, we see that the subdominant chord has been substituted by an extension chord of A minor seventh preceded by an E minor chord. The dominant F chord has been substituted with the passing chord sequence of

FIGURE 5.3: The optimized sequence that penalizes major chords.

G, C, and F, and the last subdominant D minor chord has been substituted for an A minor seventh.

5.3 Results

To evaluate whether the optimization heuristics correlated with the user preferences, we conducted a pilot survey, gathering ten participants of various musical backgrounds. Participants listened to the two different outputs and describe the emotions they felt.

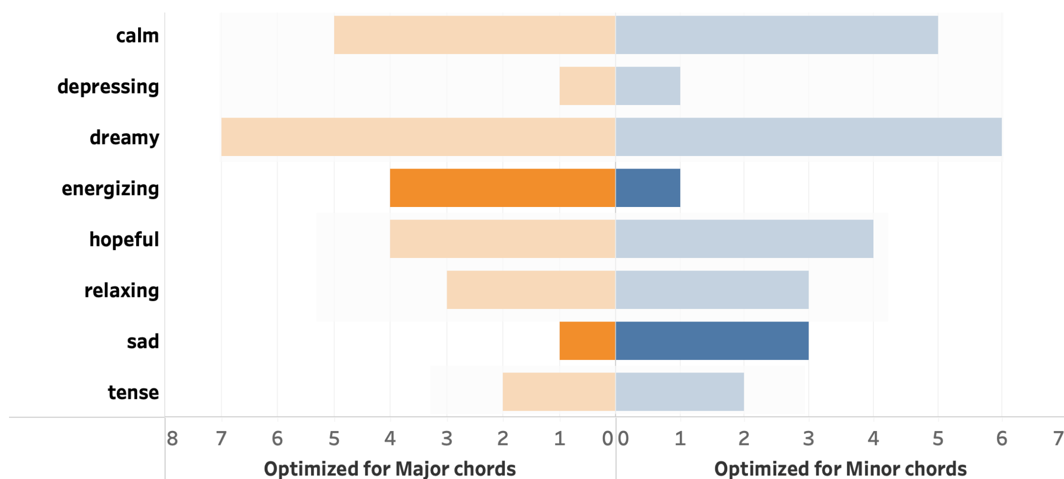

FIGURE 5.4: Survey results comparing the participant responses to the two different optimized outputs. The orange bar plot shows the response to the output optimized for the major chord, and vice versa.

Emotions As seen in Figure 5.4, four out of ten participants described the output optimized for happier sounding sequence (Figure 5.2) as energizing, whereas only one out of ten described the output optimized for sad sounding sequence (Figure 5.3) as energizing. One out of ten participants described the happier sounding sequence as sad, whereas three out of ten participants described the sad sounding sequence as sad. This is the predicted output as the major and minor shorthand are designed to convey different emotions, where the major shorthand sounds happy, and the minor sad.

Preference We asked the participants which chord sequence they preferred, and there was an even split between the two versions. Interestingly, both sides conveyed that their preferred sequence had more emotions imbued. This is where the subjectivity of music plays in. An objective standard for a better sounding harmony does not exist. Individual experiences and cultural conceptions of emotions form biases that affect one's judgment of music. Our model takes this into account by allowing the users to define the cost function to output the sound that they prefer.

Chapter 6

Discussion

6.1 Conclusion

We started with a thesis statement that harmonization is an optimization problem. The goal was to find better and richer harmony. Through the utilization of equality graphs, where there is flexibility in both the rewriting phase and the extraction phase, we gathered a pool of candidate chord sequences from which the user could extract according to their preferences of emotions.

6.2 Future Work

Because this project is built on previous works of HarmTrace [5] which by default assumes that the input sequence of chords is in the C Major context, this project does not produce complete musical pieces invariant to the tonal context but provides a proof of concept for the C major key. However, this puts the burden on the user to transpose their input chord sequence into the C major key. This could be resolved by allowing the user to designate the

tonal context along with their input sequence, and transposing the chords into the C major key.

The cost function is limited to the shorthand of the chords. As the code is open-sourced, users may experiment with the cost function to include other metrics such as the extension or the functionality of the chord sequences to output their preferred sound.

Lastly, music is constantly evolving and so is harmony composition. There may be genres that are not captured by the rewrite rules accumulated in our paper, which may be easily added later on to enhance the quality of the program.

Bibliography

[1] Adam Alpern. “Techniques for algorithmic composition of music”. In: (1995).

[2] W Bas De Haas et al. “Modeling harmonic similarity using a generative grammar of tonal harmony”. In: *ISMIR*. 2009.

[3] Lejaren A Hiller Jr and Leonard M Isaacson. “Musical composition with a high speed digital computer”. In: *Audio Engineering Society Convention 9*. Audio Engineering Society. 1957.

[4] Hendrik Vincent Koops, José Pedro Magalhaes, and W Bas De Haas. “A functional approach to automatic melody harmonisation”. In: *Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design*. 2013, pp. 47–58.

[5] José Pedro Magalhaes and W Bas de Haas. “Functional modelling of musical harmony: an experience report”. In: *ACM SIGPLAN Notices* 46.9 (2011), pp. 156–162.

[6] José Pedro Magalhães and Hendrik Vincent Koops. “Functional generation of harmony and melody”. In: *Proceedings of the 2nd ACM SIGPLAN international workshop on Functional art, music, modeling & design*. 2014, pp. 11–21.

[7] Chandrakana Nandi et al. "Synthesizing structured CAD models with equality saturation and inverse transformations". In: *POPL*. 2020, pp. 31–44.

[8] Martin Rohrmeier. "A generative grammar approach to diatonic harmonic structure". In: *Proceedings of the 4th sound and music computing conference*. 2007, pp. 97–100.

[9] Brett Saiki et al. "Combining precision tuning and rewriting". In: *IEEE Symposium on Computer Arithmetic (ARITH)*. 2021.

[10] Mark J Steedman. "A generative grammar for jazz chord sequences". In: *Music Perception* 2.1 (1984), pp. 52–77.

[11] Ross Tate et al. "Equality saturation: a new approach to optimization". In: *POPL*. 2009.

[12] Alexa VanHattum et al. "Vectorization for digital signal processors via equality saturation". In: *POPL*. 2021, pp. 874–886.

[13] Max Willsey et al. "Egg: Fast and extensible equality saturation". In: *POPL* (2021).

[14] Iannis Xenakis. *Atrées*. 1962.

[15] Iannis Xenakis. *Morsima-Amorsima*. 1962.