DIGITAL ACCESS TO SCHOLARSHIP AT HARVARD
DASH.HARVARD.EDU

HARVARD LIBRARY
Office for Scholarly Communication

Order Out of Chaos: Randomized Heuristics and Their Application to the Harmonization Problem.

The Harvard community has made this article openly available. Please share how this access benefits you. Your story matters

Citable link	http://nrs.harvard.edu/urn-3:HUL.InstRepos:38811539
Terms of Use	This article was downloaded from Harvard University's DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA

Abstract

Music harmonization is a difficult problem. If you take a melody, and you assign chords to each note in that melody, how do you judge the quality of that chord progression? Furthermore, if you then assign notes within each chord to three accompanying "voices" following the rules of classical chorale harmony, how do you judge the quality of those note assignments? How can we judge two chorales against one another objectively?

I postulate that these aspects put melody harmonization in a category of problems known as NP. These problems are, simply put, very difficult for a computer to solve quickly. However, decades of research have yielded a number of interesting methods for approximating solutions to these problems quickly enough. And what is musical composition but a human approximation of the absolute? Perhaps, in some cases, a human approximation of the Divine?

I intend to study the application of these heuristics to the harmonization problem. These heuristics are approximation algorithms based on randomness. They involve generating random solutions, checking their quality, randomly changing the solution, and then checking the improvement. If the new solution is better, we keep it and repeat the process a certain number of times. We'll explore a few variations of this algorithm and their difference in harmonization quality across multiple trials.

Acknowledgements

This thesis would not have been possible without the support, assistance, and encouragement of many very special people. I am incredibly grateful for both of my advisors, Professors Michael Mitzenmacher and Suzannah Clark – this project would not have been possible without your input and support. I must thank my professor of computational music theory, Beth Chen, for inspiring me to create my own computational conceptions of music and see where they guide me, and my first professor of classical music theory, Richard Beaudoin, for consistently pulling back the curtain and showing me the wonderful ambiguities and delights of tonal music. I am thankful for many wonderful friends and classmates who supported me and humored me as I droned on and on about this paper (especially Dylan). And finally, I must mention my incredibly caring family for instilling a love for music and math within me from a very young age. I wouldn't be where I am today without my amazing parents and my irreplacable sister.

Contents

1 Introduction. **1**

2 Musical motivations. **4**
2.1 Composition vs. harmonization. 4
2.2 Hymns. 5

3 The harmonization problem. **6**
3.1 Preliminaries. 6
3.2 Formalization. 11
3.3 Complexity. 15

4 Brief survey of algorithmic music composition. **17**
4.1 Markov chains. 18
4.2 Neural networks. 19

5 Randomized algorithms and application. **21**
5.1 Approximation. 21
5.2 Local search. 22
5.3 Variants. 23

6 Experimentation and conclusions. **25**
6.1 Algorithm analysis. 26
6.2 Musical analysis. 28
6.3 Future work. 31

Appendices **33**

A Music theory fundamentals. **33**

B Complexity theory fundamentals. **37**

Bibliography **40**

Chapter 1

Introduction.

Algorithmic music composition has taken many different forms over the last century. In recent decades, the general shift in attention towards the power and flexibility of neural networks in the computer science community has also occured in the smaller field of computational musicology. Computer scientists have trained neural networks on the music of the masters – Bach, Chopin, Lennon and McCartney, Coltrane, and beyond – to study the potential of a program to imitate the work of a composer. These networks "train" by observing the trends in the bodies of work of these composers and drawing conclusions from them. Some of these networks learn in a supervised manner, trying to learn how to solve specific musical problems by watching the masters. Other networks are unsupervised, and merely make general observations and predictions about the genre or field of a piece of music after training on a large data set of many different pieces of music.[1] These types of unsupervised learning algorithms are often better suited for musical analysis rather than musical composition.[2]

However, these academic investigations into the possibilities of machines to create music that sounds "human" end up drawing a divisive line between the man and the

[1] Jan Wülfing and Martin Reidmiller. "Unsupervised learning of local features for music classification". In: *International Society for Music Information Retrieval Conference* (2012).

[2] Heinrich Taube. "Automatic Tonal Analysis: Toward the Implementation of a Music Theory Workbench". In: *Computer Music Journal* 23.4 (1999).

machine. The creators of these programs are often determined to avoid any "human" interference in the compositional process, save the inevitable role of the composer of the training data. It is possible for a human to train a neural network on their own music, but this is doomed to be either inaccurate or impractical based on the size of one's body of work.

How, then, can we bridge this gap between academic curiosity and practical application? As it turns out, one of the most important academic works that studied the intersection of mathematics and music had this as its central focus. This work is Iannis Xenakis' *Formalized Music*. In it, Xenakis applies the probabilistic concept of stochastic Markov chains to his musical compositions in various ways. However, as Xenakis does this, he is also choosing to represent music and its various elements in a way that best suits his own compositional style, in which music contains "rare sonic events" whose temporal placement and duration (among other qualities) must be determined by mathematical processes.[3] This is what Xenakis considers beautiful, while musical concepts such as functional harmony are left out. Of course, by doing this, Xenakis' algorithms produce biased music - although they are not "trained" to imitate Xenakis' compositional style in the machine learning sense, the initial conditions that define the algorithms are based heavily on Xenakis' subjective definition of what makes music beautiful.

In this way, the machine is unable to compose music of its own conception and without partiality - it is doomed to remain a tool of the composer, at least from a high-level perspective.[4] However, as Xenakis' explores so thoroughly in his work, the lower-level details are the ones where the machine really has the opportunity for something resembling inspiration.

In this paper, I intend to investigate a conception of music theory that stems from

[3]Iannis Xenakis. *Formalized Music: Thought and Mathematics in Composition.* Harmonologia Series No. 6. Pendragon Press, 1992.

[4]Dmitri Tymoczko. "Three Conceptions of Musical Distance". In: Mathematics and Computation in Music (2009).

my own musical philosophy regarding functional harmony, and eventually use it as a compositional tool. I will postulate a formal "harmonization problem" and apply various randomized heuristics to approximate solutions. In doing so, I hope to codify aspects of my own compositional process in the program I write, and observe the effect of changing the parameters of this program on the resulting "style" of the harmonization.

Outline.

In Chapter 2, we will discuss the personal and practical motivations for formulating the harmonization problem the way we do, and why to balance human and computational portions of the creative process in order to draw clear boundaries upon which the algorithm can be built. I will also briefly mention my intended use for this music generation algorithm and the implications on any future work.

In Chapter 3, we will formalize the harmonization problem using musical set theory. We will also show that the harmonization problem is in NP.

In Chapter 4, we will explore the history of algorithmic music generation to gain awareness of what has and has not been done. This survey will also show the motivations behind the use probabilistic tools in music generation.

In Chapter 5, we will discuss the specific randomized algorithms that we will be applying to solve the harmonization problem. We will discuss why approximation algorithms are an effective tool for music in particular and how they reflect the human process of composition.

In Chapter 6, we will examine the harmonizations produced by my program and draw conclusions regarding the power of our compositional tools and its real-world applications. We will also examine the effects produced by fine-tuning various weights and parameters of our cost function, and how this fine-tuning allows my program to be of general use to a large variety of tonal composers.

Chapter 2

Musical motivations.

2.1 Composition vs. harmonization.

Many existing methods for algorithmic music generation attempt to simulate the entire compositional process from start to finish. However, fragmentation of this general process into smaller subtasks allows more room for human control and thus more guiding principles for the machine. Consider the efforts of Douglas Eck to train a neural network to improvise over a blues progression.[1] Since the chord progression is fixed, the neural network can base its choice of melody and phrase structure on this foundation.

In this investigation, we will be similarly pre-processing a portion of the compositional process. However, instead of determining the harmonic structure in advance and procedurally generating the melody, we will do the opposite by determining a melody in advance and allowing the algorithm to generate the appropriate harmony. I argue that this approach will be better suited to the capabilities of the machine as well as general psychological trends of music listeners.

[1] Douglas Eck. "Finding Temporal Structure in Music: Blues Improvisation with LSTM Recurrent Networks". In: *Neural Networks for Signal Processing XII, Proceedings of the 2002 IEEE Workshop* (2002), pp. 747–756.

2.2 Hymns.

A common concern raised with exploratory papers of this type is that the rigid compositional form outlined and formalized in this paper is no longer put to practical use. Many individuals believe that four-part chorale writing is obsolete. Some arguments base this on the atonal trends of contemporary music and the slow decline in popularity (at least in the academic community) of functional harmony over the last hundred years. Others base it on the infrequency of choral music to be purely homophonic (this argument will make more sense after we formally outline the set theoretical approach to music theory in the next chapter, but essentially, “chorale” music has 4 voices, and the top voice sings the melody - each time the top voice changes to a new pitch (or possibly just repeats its current pitch), the other 3 voices change underneath at the same time. This is called a “homophonic” texture, because the voices move at the same time, although not necessarily in the same direction). The objection states that even pieces of music which contain homophonic sections are rarely entirely homophonic.

However, these protestors fail to consider the most practical modern application of this algorithmic harmonization in the chorale style: contemporary hymn harmonization. Hymns are songs of praise sung in religious (mostly Christian) services, and always in a 4-part texture. The church has codified the soprano / alto / tenor / bass paradigm, such that any constituent of the church who walks in and reads a hymnal will know how to assign themself to one of the four parts. As a result, the melodies we input to our harmonization algorithm at the experimentation phase of this project will be mostly drawn from the Harvard Memorial Church Hymnal.[2]

[2]Peter F. Gomes et al., eds. *The Harvard University Hymn Book*. Harvard University Press, 2007.

Chapter 3

The harmonization problem.

In this chapter, I will outline a mathematical representation of functional tonal harmony. I will use this representation to construct a formal definition of the musical concept of "melodic harmonization" as a computational problem. Once I've done this, I will show that this computational problem is in NP, which will prepare us for the following chapters. For a more detailed description of tonal harmony and the motivations behind our mathematical model, refer to Appendix A.

3.1 Preliminaries.

Music is inherently mathematical, insofar as "chords" can be described well using sets and basic discrete math. In fact, a set theoretical approach to music is not a very novel idea.[1] Using set theory, we can begin to lay the foundation for a theoretical reduction from melodic harmonization to more traditional computational problems, to set the stage for our algorithmic approach. However, in order to do this, we must be thorough and concise in our generalization of western music theory to mathematical primitives. First, I'll discuss the two axes of analysis for musical harmonization we discussed earlier

[1] Dmitri Tymoczko. *A Geometry of Music: Harmony and Counterpoint in the Extended Common Practice*. Oxford Studies in Music Theory. Oxford University Press, 2011.

in this thesis from a mathematical / set theoretical perspective. Next, we'll discuss the complexity class NP and its relevance to our musical problem.

Note: in this section, we will use colloquial terms to describe certain musical motions and rules. Words like "bad", "difficult", and "okay" are intentionally ambiguous – they will be formalized and made unambiguous in section 3.2.

"Vertical" analysis.

In western music theory, there are 12 pitch classes. We can represent these pitch classes as numbers in a mod 12 space.[2] For classical music theorists, we can say that 0 is the pitch class C, 1 is the pitch class C♯/D♭, and so on. We will define the set of numbers from 0 to 11 to be the set of pitch classes P. The distance between two adjacent pitch classes say, for example, G and G♯ - is called a *semitone*. Two notes of the same pitch class can have different pitches, depending on their *octave*. Formally speaking, doubling the frequency of a pitch will yield the same pitch class one octave above the starting pitch. To clearly identify an exact pitch, we write the pitch class followed by the octave – for example, C4 is one octave above C3. We will define a "chord" C to be a set of three pitch classes, $C = \{p_0, p_1, p_2\} \bmod 12$, where $p_0, p_1, p_2 \in P$ (we call p_0 the "root" of the chord C). A chord C is major iff $\exists p \in P$ s.t. $C = \{p, p+4, p+7\}$. Similarly, a chord C is minor iff $\exists p \in P$ s.t. $C = \{p, p+3, p+7\}$. Finally, a chord C is diminished iff $\exists p \in P$ s.t. $C = \{p, p+3, p+6\}$. For now, we are only concerned with these three types of chords. (And remember, these numbers are all in a mod 12 space, because there are only 12 pitch classes in P). For example, the chord $C = \{8, 0, 3\}$ is major because $p = 8$ and $0 = 8+4 \bmod 12$ and $3 = 8+7 \bmod 12$. One more important quality of chords is their inversion. Simply put, inversion indicates which note in the sounding chord has the lowest frequency. In "root position", the root p of a chord is used as the lowest note (again, in terms of frequency). In "first inversion", the next

[2] Robert Morris. *Composition With Pitch-Classes: A Theory of Compositional Design.* Yale University Press, 1987.

note of the chord is the lowest. In "second inversion", the third note of the chord is the lowest. If we call inversion some constant $v \in \{0, 1, 2\}$ where 0 is root position, 1 is first inversion, and 2 is second inversion, the bass note of a chord $C_v = \{p_0, p_1, p_2\}$ is p_v. As a simple example, take the chord of G major, {G, B, D}. If this chord were in root position, then the note with the lowest frequency would be G. If it were in first inversion, the note with the lowest frequency would be B. And if it were in second inversion, the note with the lowest frequency would be D.

Now we must consider another set of subsets of P, known as a "key". A key K is a collection of pitch classes. Once again, we call p the "root" of the key K. There are also major and minor keys - for now we will only consider major keys. A key K_p is major iff $\exists p \in P$ s.t. $K_p = \{p, p+2, p+4, p+5, p+7, p+9, p+11\}$. This is an interesting collection of pitch classes because we can see our major and minor and diminished chords all appear naturally in the major key. For example, the chord $C = \{p+2, p+5, p+9\}$ can be formed out of the notes in a major key, and it is also a minor chord, because we can represent it as $C = \{(p+2), (p+2)+3, (p+2)+7\}$.)

By the way, for those readers who are very unfamiliar with music theory, the difference between chords and keys is rather straightforward. A chord is a collection of pitch classes to be played at the same time. A key is a collection of pitch classes to be used over the course of the song. Thus, ideally, a chord used in a song will only contain pitch classes in the key of that song. It is very possible for a chord to contain pitch classes that are not in the key of the song, but we will discuss this later.

A melody M is a list of pitch classes to be played one at a time over the course of a song. As an example, imagine the first line of "Happy Birthday". It can be represented as a melody $M = [0, 0, 2, 0, 5, 4]$. The process of songwriting, for the purposes of our formalization of the harmonization problem, can be boiled down to the following: if we have a melody M in a certain key K, we can assign a chord C_i to each note of the melody M_i. We'll call that set of all chords C_i the chord progression of a song.

How do we decide whether a given chord progression is harmonically valid? According to the western musical tradition, chord progressions were considered reasonable if:

- The melody note is always within the chord assigned to it.
- Most of the chords were within the key.
- The bass notes of the chord tended to move by step (by a pitch class distance of either 1 or 2) or by perfect fourth (by a pitch class distance of 5).
- Chords resolve from first and second inversion chords to root position chords, which are considered to be the most stable chord.
- Unstable chords like chords that are not in root position or diminished chords should be followed by stable chords (major or minor chords in root position)

We can see a computational problem beginning to emerge. However, this chordal analysis is only one axis along which western classical music faces judgment.

"Horizontal" analysis.

Once we have selected a chord progression for a melody, there remains the matter of "voicing" that chord. Imagine we have four singers. One will sing the melody, while the other three will sing notes within the chord of each melody note. This means that, if we make sure to choose chords such that each chord contains the melodic note its assigned to, there will always be exactly two singers singing the same pitch class of the chord, since there are three notes to a chord (thus, by the pidgeonhole principle... I mean, common sense).

When we deal with voices, we must consider an important distinction from vertical analysis. When we were analyzing the chords of a song, each we were only concerned with pitch classes that is, we treated all Cs as the same, all C♯'s as the same, and so on.

We did not consider the relative octave of these pitches. A chord is major regardless of the relative octave of all its element pitch classes. However, with voices, we do care about the relative octave, because we are primarily concerned with the distance between two adjacent notes in a vocal part. If we did not consider the relative octave of a pitch, then a vocal part which jumps from one note to a note 15 semitones above would be considered the same as one which jumps up 3 semitones, since all distances are evaluated in a mod 12 space. We do want to consider this difference though, because a jump of 3 semitones is much easier to sing than a jump of 15 semitones. Thus, we will represent notes as integers, and only put them into mod 12 space when we are considering their membership in the underlying chords. As we just mentioned, we want to assign chordal notes to each voice so that the progression of notes contains movements that are not particularly large (lots of semitones) and thus are easy to sing. We can boil this down into some pretty straightforward rules:

- For the most part, smaller distances between notes are better.
- Leaps of 6 semitones (the western "tritone") are very bad.
- Leaps of 8 semitones or above are bad.
- In all voices but the lowest, any leap should generally be at most 4 semitones wide.
- In the lowest voice, leaps of 5 semitones or 7 semitones are perfectly okay.
- Each voice has its own unique range of valid pitches. You cannot make a singer sing notes higher or lower than their range.

For example, consider the following vocal part $V = [4, 6, 7, 6, 9, 7, 6, 4, 12]$. In this vocal part, most of the distances between notes are very small (either 1 or 2 semitones). However, the last distance from 4 to 12 is 8 semitones wide and thus "difficult to sing". We will formalize this further in the next section.

Now we arrive at some of the most important rules of western classical music theory, which all deal with relative motions between multiple parts. Let us consider two vocal parts, V_1 and V_2, where the vocal note at time step i is written as $V_1[i]$. For instance, if we consider the above vocal part $V = [4, 6, 7, 6, 9, 7, 6, 4, 12]$, $V[3] = 6$. Here are some of the strict invariants presented by western classical music theory:

- If at some moment i two voices $V_1[i]$ and $V_2[i]$ are seperated by exactly 7 or 12 semitones, then the distances $V_1[i+1]-V_1[i]$ and $V_2[i+1]-V_2[i]$ must be different. If they are the same, this is considered "parallel" motion, and it is very bad.
- The above rule also applies if $V_1[i] = V_2[i]$.

We will be considering harmonizations where V_1 represents the melody or "soprano" voice, V_2 and V_3 are the alto and tenor voices respectively, and V_4 is the bass voice. For our purposes, we will stipulate that for a given time step i, $V_4 \leq V_3 \leq V_2 \leq V_1$.

As we mentioned before, the supporting voices V_2, V_3 and V_4 are responsible for filling out the harmonic texture. This means that, for a given time step i, all three notes of the chord C_i must be representable as a subset of the four vocal notes $V_1[i], V_2[i], V_3[i]$, and $V_4[i]$ (here projected onto a mod 12 space). More formally, for a given time step i, $C_i \subset \{V_1[i] \bmod 12, V_2[i] \bmod 12, V_3[i] \bmod 12, V_4[i] \bmod 12\}$.

3.2 Formalization.

Now that we have laid the mathematical foundation for our musical set theory, it's time to formalize our problem. Basically, the harmonization problem is one that we hope a computer will understand. In order for this to be the case, we cannot express our classical rules as general guidelines - for instance, leaps of 6 semitones are "bad" - but instead must assign some error value e to each of these violations. That way, the algorithms we develop will be looking for a solution which minimizes the sum of all errors. There are a few domains in which we must establish clear error scores.

ℓ_1 vs. ℓ_2 norms.

Before we get into the specifics of the formal problem, I would like to take a moment to discuss my justification for many of the error functions which follow. In most cases, as with chordal common tones, chord tones within the key, voice-leading, and more, our error functions are quadratic – that is, the error score is calculated by squaring the distance between the given solution and some optimal rule. I chose to use quadratic functions, or the ℓ_2 norm, instead of linear functions, or the ℓ_1 norm, because the ℓ_2 norm is known to be very sensitive to outliers. Thus, our algorithm will not be quick to forgive a tritone leap in the voice-leading – that motion will create a huge error score that the algorithm will have to try hard to correct.

Vertical errors.

First, we want to set up error mechanics for chord progression. For a chord progression C where $C[i]$ is the chord at time step i, the error score will be largely determined by the bass notes and by the number of common tones between chords. The number of common tones between two chords C_i and C_{i+1} is defined as $|C_i \cap C_{i+1}|$. n is the number of chords in a progression. We will define an error

$$E_c = \sum_{i=0}^{n-2} \Big(|C_i \cap C_{i+1}| - 2 \Big)^2$$

as the error score of common tones. This error score gives the highest value to adjacent chords which share 2 notes. If we had given the highest value to adjacent chords which share 3 notes, our algorithm would likely avoid ever changing the chord (since this is a nice way to guarantee that the intersection between adjacent chords is as large as possible) which isn't very musically interesting.

Rather than assign error scores to chord depending on whether they are major or minor chords (although we may explore this in Chapter 3), for now we will just assume

that all chords in the progression have been constructed using one of the rules for major / minor / diminished chords.

Finally, we want chords to mostly be made up of notes within the key K. We'll assign an error to the size of the difference between K and a chord C_i. If C_i only contains notes that are not in the key K, then the size of $C_i - K$ will be 3. We can define a simple error score E_k:

$$E_k = \sum_{i=0}^{n-1} \Big(C_i - K\Big)^2$$

Horizontal errors.

There should be an error to account for proper bass motion between chords. Simply put, a movement from one chord to another is considered to be functionally legitimate if their bass notes are seperated by a distance of 5 or 7 semitones, or are otherwise as close as possible. What we will do is score this error based on the proximity to a distance of 6 semitones, and then add an additional error score if the distance is exactly 6 semitones. More formally, we define $V_4[i]$ to be the bass note of a chord C_i.

$$E_b = \sum_{i=0}^{n-2} \Big(|(V_4[i+1] - V_4[i])| - 6\Big)^2 + \sum_{i=0}^{n-2} \Big(V_4[i+1] - V_4[i]\Big)^2$$

We will consider in the future a clearer alternative to this error scoring mechanism - a fifth degree polynomial function with the following qualities: maximums at $(100, 6)$ and $(10, 3.5)$, minimums at $(0, 0)$, $(5, 0)$, and $(7, 0)$. If we treated this as the error function for bass note distances, we would accurately weight each potential distance the way classical music theory dictates.

The other three voices have only slightly different and simpler distance measure. For them, a larger distance merits a larger error, with no real exceptions. Thus, we

will say the error $E_d(x)$ of a voice V_x is defined as follows:

$$E_d(x) = \sum_{i=0}^{n-2} \Big(V_x[i+1] - V_x[i]\Big)^2$$

Remember, $V_x[i]$ must be an element of C_i. We won't assign an error score to this because it is a firm requirement. From an algorithmic perspective, imagine we first create our chord progression and then generate the 3 voices V_2, V_3, and V_4 out of that progression - we can guarantee that we will only choose notes for $V_x[i]$ that are elements of C_i.

$V_x[i]$ must also be within the range of acceptable notes for V_x. Since they are different for every voice, we'll call the minimum V_x.min and the maximum V_x.max. This is another firm requirement. Thus, from an algorithmic perspective, we will make sure to randomly choose notes *within this range* when we are generating random harmonizations.

Now that we've established a decent precedent for "rigid" rules (where we don't weight an error but rather deem the entire solution flawed, similar to an error score of infinity), we needn't be too stressed about our parallel voice requirements. Recall that we want to avoid parallel motion in voices that are seperated by 0, 7, or 12 semitones. If we want to remain rigid, this can be a firm requirement. Otherwise, if we want to weight this parallelism, we can, using an error E_p. First, define the set of pitches at a given timestep i as $V[i] = \{V_1[i], V_2[i], V_3[i], V_4[i]\}$. We'll define a set of all pairs of voices $P[i] = \{(a, b) | a \in V[i] \text{ and } b \in V[i] \text{ and } a \neq b\}$. Our error score is thus:

$$E_p = \sum_{i=0}^{n-2} \sum_{j=1}^{4} \sum_{k=j+1}^{4} \Bigg(V_j[i] - V_j[i+1] == V_k[i] - V_k[i+1] \wedge$$

$$\Big(|V_j[i] - V_k[i]| == 0 \vee |V_j[i] - V_k[i]| == 7 \vee |V_j[i] - V_k[i]| == 12 \Big) \Bigg) * w$$

where w is some constant weight we assign to parallelism.

Statement of problem.

Now that we have established all of the error scores, the statement of our problem is simple. For a given key K, and a given melody V_1 of length n, can we create a chord progression C and vocal parts V_2, V_3, and V_4 such that the sum of all error scores is as low as possible?

However, we will present a slight variation of this problem in order to better suit it for complexity theory and computational analysis. The variation is as follows: for a given key K, and a given melody V_1 of length n, can we create a chord progression C and vocal parts V_2, V_3, and V_4 such that the sum of all error scores is lower than some given constant k?

3.3 Complexity.

There exists a class of problems in computer science complexity theory known as NP.[3] In layman's terms, problems are in NP if they can be verified in polynomial time. For a more detailed description of terms like NP and polynomial time (and Big-O notation, which appears in our complexity analysis below), refer to Appendix B.

Easily verifiable solution.

For a solution to be easily verifiable, we must be able to check that it satisfies the requirements of the problem in polynomial time. Thus, let us consider the theoretical runtime of a harmonization-checker.

In order to check the error scores E_c and E_k of a chord progression, we only need to iterate through the list once, taking $O(n)$ steps. If we use hashed sets to lookup membership in our sets (like lookup in the key K, for instance) then lookup will be in $O(1)$ time. If we are performing $O(1)$ operations for each of our $O(n)$ chords, so our

[3] Michael Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

overall runtime is $O(n)$.

Next, we need to check the error scores E_b, $E_d(x)$ for all integers $1 \leq x \leq 4$, and E_p. To check E_b and all $E_d(x)$ we need only iterate over the list once for each voice and calculate the distance error as we go, which means that check will run in $O(4n) = O(n)$ time. For the parallelism error score E_p, for each pair of adjacent notes in each voicing we must compare it to all corresponding pairs in other voices. If we had a general number of m voices, this would mean that there would potentially be up to $m - 1$ voices to check per voice. Thus, if we need to perform that many operations per element in the list to fully check one voice, then checking a voice for parallelism errors would run in $O(n * m)$ time. Since we need to do this for every voice, our overall general runtime is $O(n * m^2)$. Luckily, we are not dealing with a general version of this problem. In our problem, specific to the classical traditions of chorale harmony, there are always only 4 voices. Thus, $m^2 = 16$ and our runtime is $O(n)$.

Since we have proven we can check the error score of a harmonization in $O(n)$ time, we have shown that the harmonization has easily verifiable certificates. If the certificate is a full harmonization and chord progression, the certificate can be checked to see if the error score is less than some integer k in polynomial time.

Now we hopefully see why we rephrased the problem - if we had left it in its original form, where we are searching for the harmonization with the lowest possible error score, there would be no way to quickly verify that a given solution was the optimal one. It would be possible that there was another solution with a lower error score.

Chapter 4

Brief survey of algorithmic music composition.

Theorists have been attempting to represent music with mathematical models for centuries, enough that the concept of an algorithm for musical composition has always dwelled within the realm of possibility. However, these models must carefully tread a line between the human practice of composition and the rules governing the style of harmony being used. For example, a naive computer program attemping to generate a chord progression might choose a random chord to start, and then greedily choose the most technically correct chord to follow it, and continue this process until they reach the end of the piece. However, this doesn't quite reflect the composition process, as a human might begin composing this way but eventually double back and change the chords they have already written once they see where the progression ends. And humans are very likely not to choose the most "obvious" chord to follow the current one every time - in fact, going against this urge seems to yield dramatic and emotional results. Thus, it is a logical conclusion that any computer program meant to generate music must have some inherent element of probability embedded within it. We will briefly discuss some of the existing techniques that have been used in the paths before

transitioning to the technique I will be experimenting with later in this paper.

4.1 Markov chains.

Markov chains are a mathematical concept formalized in the early 1900s to model probabilistic systems. They can be formally described as follows: we have a set of states $S = s_1, s_2, s_3, ..., s_n$. We start at a certain state, and at each timestep we move to another state from the one we are in. If we are in state s_i, then we move to state s_j with some probability p_{ij}. It is possible to stay at the same state s_i, which occurs with probability p_{ii}.

We can see how this model might be used to represent the act of musical composition from a harmonic perspective.[1] Each state can represent a chord, and we move from chord to chord in a harmonic progression by the probabilities given by the Markovian transition matrix. We can set this starting chord to be the tonic chord of our composition. Creating a piece of music using this stochastic process is similar to through-composing a piece from start to finish - only ever considering (in a semi-greedy / semi-probabilistic fashion) where to go to next from your current position.

This model is general enough such that the transition matrix M (where element $m_i j$ represents the probability of moving from state i to state j at any timestep) cannot take into consideration any constraints - only an initial condition (in the form of the matrix M). Thus, while it is an interesting algorithmic approach to the general compositional process, it does little to consolidate the disconnect between melody and harmony that most contemporary academic music fails to address.

Of course, there are ways to encode the harmonization of a specific melody into the transition matrix, but this means that the transition matrix will change for each timestep, as the likelihood (probability) of one chord changing to another would depend

[1] Charles Ames. “The Markov Process as a Compositional Model: A Survey and Tutorial”. In: *Leonardo* 22.2 (1989), pp. 175–187.

on the following melodic note, which is different at each timestep. Thus, the creation of the matrices necessary to solve our specific harmonization problem inevitably deviate from the standard Markov chain model with only one relevant transition matrix.

4.2 Neural networks.

The application of machine learning and neural networks to music generation has been the subject of many an academic pursuit, including many theses written by undergraduates here at Harvard.[2] The important feature of neural networks is their capacity to "train" on existing data and learn from it. Put simply, neural networks analyze the statistical trends of existing pieces of music – what chords tend to go where, the general phrase structure of the music, the melodic contour – to a degree of depth dependent on whether the learning is supervised (has some preprocessed directive of what to look for, and knows what features are present or absent in the training data) or unsupervised (is simply trying to draw general conclusions about data sets by dividing them into unique groups).

These machine learning algorithms are inherently data-driven, and require a large corpus of study to improve accuracy.[3] Interestingly enough, many academics point to machine learning algorithms as a step away from the inevitable biases of algorithmic music generation based on hand-tuned parameters, but even this data-centric approach to musical composition is based on the machine's ability to learn from and eventually predict or imitate the work of a human being. David Cope is a composer who turned to machine learning algorithms for music generation when he was faced with writer's block, but ended up training the network on the works of famous composers like Chopin

[2] Dylan Jeremy Nagler. *SCHUBOT: Machine Learning Tools for the Automated Analysis of Schubert's Lieder.* Bachelor's thesis, Harvard College, 2014.

[3] Jamshed J. Bharucha. "Modeling the Perception of Tonal Structure with Neural Nets". In: *Computer Music Journal* 13.4 (1989), pp. 44–53.

and Bach because his own music felt too nebulous.[4] This is where Cope seems to fail at his initial goal, and where Xenakis was able to succeed – Xenakis used computational tools to define and guide his work and philosophy of music, whereas Cope shied away from this possibility because he felt his music lacked definition.

If our algorithm is successful, it will codify harmonic practices indicated by the initial conditions we fine-tune. We will discuss in later chapters the effects of changing some of our generalized concepts such as chord construction and keys.

[4] David Cope. "Experiments in Music Intelligence (EMI)". in: *ICMC Proceedings* (1987), pp. 174–181.

Chapter 5

Randomized algorithms and application.

5.1 Approximation.

An approximation algorithm, simply put, sacrifices accuracy in the interest of speed. The algorithm does not guarantee that the solution generated will be the optimal one, but it does promise to find that sub-optimal solution quickly (usually this means in polynomial time) and it also allows an analysis of exactly how accurate or inaccurate the approximation may be. This is very useful in the context of our harmonization problem. Consider two possible solutions, one of which has a slightly lower error score than the other. It may take a very long time to find that better solution – the only way to find it for certain is by checking every possible harmonization. There are 36 possible chords in our formalized model (each of the twelve pitch classes times the number of chordal types – major, minor, and diminished) and each of these chords can be in three different inversions, meaning that for a melody with n notes there are $(36 * 3)^n = 114^n$ possible chord progressions (and that doesn't even consider each of the three harmonizing voices). However, if we sacrifice this need for the optimal

solution and accept that a solution we've found within a certain amount of time is "good enough" we can generate more solutions more quickly. After all, since our formalization of the harmonization problem is based on my own conception of music theory, these sub-optimal solutions may still have a lot of musical merit.

5.2 Local search.

Now we'll consider some standard approximation heuristics that are frequently used to solve NP-complete problems. They are in a category known as "local search" heuristics. In local search heuristics, there are some fundamental concepts that should be explained. The first is the idea of a solution - in the harmonization problem, this would take the form of a full chord progression C and three complete voicings V_2, V_3, and V_4. The next is the idea of a "neighbor" solution. A neighbor solution is another solution which is somehow "close" to our first solution. For instance, in the context of the harmonization problem, imagine we start with some solution, and then we alter it slightly by, say, rearranging the voices at timestep $i = 3$. The rest of the solution stays the same. This new solution is a neighbor of our first solution. We could also get a neighbor solution by changing one particular chord C_i, or even two chords, or perhaps changing one chord and rearranging the voices of two other chords, etc. The basic idea should be clear enough - rather than generating a new solution from scratch, we simply alter an existing one slightly to find a neighbor solution.

The last important concept to consider when setting up a local search algorithm is a cost function f. If x is a solution to our problem, $f(x)$ gives us the cost of that solution. In the harmonization problem, this cost function would clearly be our error score outlined in Chapter 3.

Hopefully, the local search algorithm should be apparent from our preliminaries. To begin, choose a starting solution. Then, check a neighbor of that solution. If the neighbor solution is better (i.e. has a lower error score, in the case of the harmonization

problem) then choose that as your new solution. Otherwise, keep the original solution. This process continues until the algorithm ends, when the best solution that was found is returned. The exact variant of the local search algorithm used determines how the algorithm decides when to stop – we'll go into more detail regarding some of these variants in a moment.

Local search algorithms must confront the issue of global and local maxima and minima. In the most basic local search, we keep improving our solution until we can no longer do so. Once we reach this point, we can consider it a "local maximum" since we can't move to a neighboring solution any more to find any improvement. However, it is very possible that this local maximum is not the global maximum (also known as the optimal solution), since the optimal solution doesn't necessarily have to be a neighbor of the solution we end up with. Again, connecting this to the harmonization problem, we can imagine a chord progression where changing any one chord will increase the error score, but where changing 5 or 6 chords might reduce the error score. Changing a solution this drastically probably wouldn't be considered a "neighbor" solution, so this improvement would be inaccessible to our local search algorithm. Thus, we need to make sure that our method for randomly choosing a neighbor of our current solution is such that our search will cover as large a portion of the solution space as possible.

5.3 Variants.

Different variations of the local search algorithm try to compensate for this local / global maximum issue in different ways. We will describe some of them in more detail now.

Metropolis rule.

In this variation, we still move to a neighboring solution if that solution is better. However, if the solution is worse, we still move to that solution with some probability

(which is usually based on exactly how much worse the solution is). In this case, we must also keep track of the best solution we have ever found while we progress through potentially worse solutions. The motivation behind this variant is that sometimes moving to a worse solution can prevent us from getting trapped at a local optimum.

Simulated annealing.

This variation uses the same new mechanic as the metropolis rule, pursuing bad neighbors in the event that they lead to a better local optimum. However, in the simulated annealing algorithm, the probability that the search chooses to examine a worse neighbor gets lower and lower over time, so that eventually the algorithm reaches a decision when the probability is zero and there are no neighboring solutions that are better than the current one.[1] The algorithm is named this way because it is essentially cooling down and solidifying over time.

Tabu search.

The tabu search variation is meant to constrain the local search heuristic from returning to recently visited solutions, referred to as cycling. The strategy of the approach is to maintain a short term memory of the specific changes of recent moves within the search space and preventing future moves from undoing those changes. Additional intermediate-term memory structures may be introduced to bias moves toward promising areas of the search space, as well as longer-term memory structures that promote a general diversity in the search across the search space.[2]

[1] S. Kirkpatrick; C. D. Gelatt Jr., and M. P. Vecchi. "Optimization by Simulated Annealing". In: *Science* 220.4598 (1983), pp. 671–680.

[2] Fred Glover. "Tabu Search – Part 1". In: *ORSA Journal on Computing* 1.2 (1989), pp. 190–206.

Chapter 6

Experimentation and conclusions.

Our algorithm has some specifics that should be outlined. First, there is the matter of generating the first complete solution. To embrace our randomized approach to harmonization, the solution is generated by considering each note in the melody and using it to randomly construct a major, minor, or diminished chord. To do this, we randomly choose whether the melodic note is the "root" of the chord, the third, or the fifth. Then we randomly select which kind of chord to construct, and we do so. The probabilities of selecting each kind of triad and each note within the triad the melody should be can be weighted and fine-tuned for further experimentation.

We tested three different randomized approaches to music harmonization. Each algorithm ran for 500 iterations before outputting a solution. The "random" algorithm generated a completely new random harmonization at each iteration. The "local search" algorithm generated a random starting solution, then at each iteration randomly chose a neighboring solution and compared error scores. If the new error score was lower, that solution was the new running solution and the algorithm moves on to the next operation. The "simulated annealing" algorithm also checked neighboring solutions as with local search, but chose the worse solution as its running solution with probability $p(i)$ (where p is a function on the iteration i). If $f(x)$ is the error score

of our current solution and $f(y)$ is the error score of our new solution, then $p(i)$ was defined as follows:

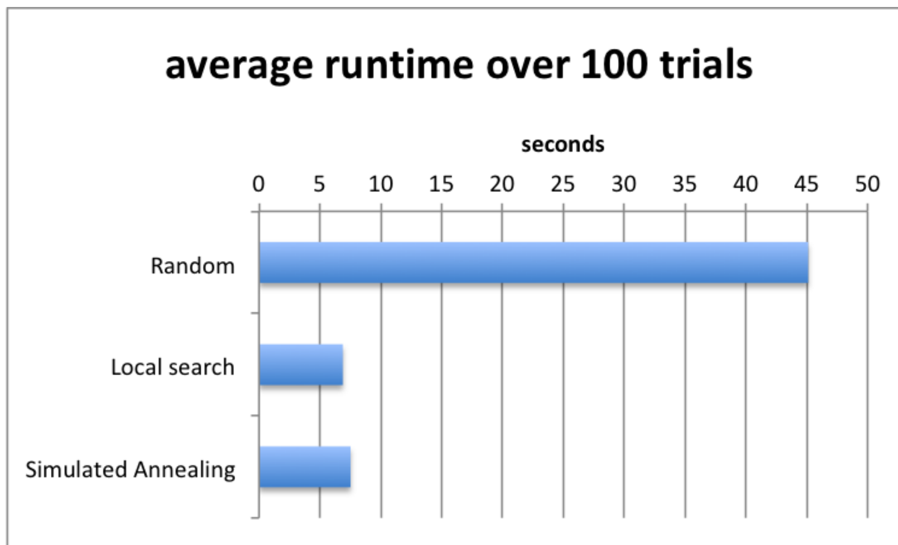
$$p(i) = \exp\Big(- \big(\frac{f(y) - f(x)}{10^{10}(0.8)^{\lfloor \frac{i}{60} \rfloor}}\big)\Big)$$

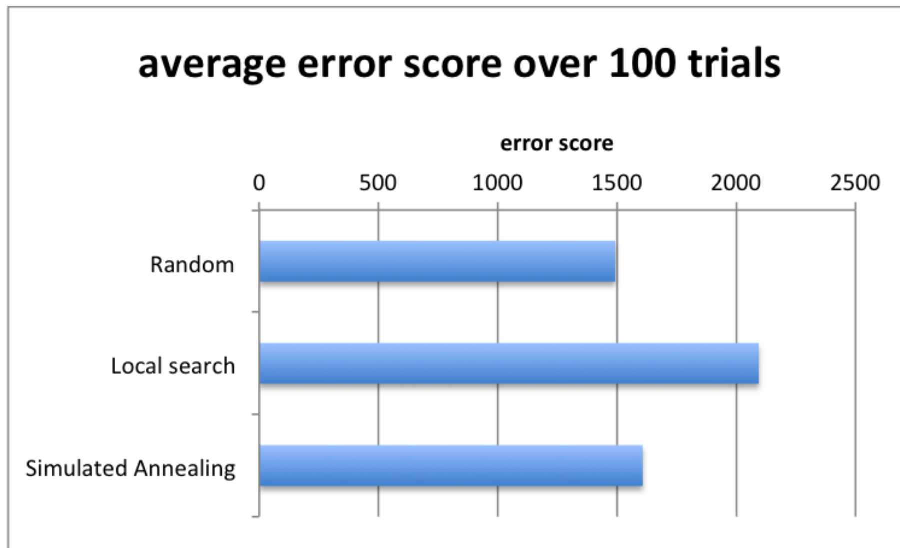
6.1 Algorithm analysis.

Data.

average running time (sec)		
random	local search	simulated annealing
45.10273	6.88101	7.5132

average error score		
random	local search	simulated annealing
1492.4301	2094.381	1607.8819





Conclusions.

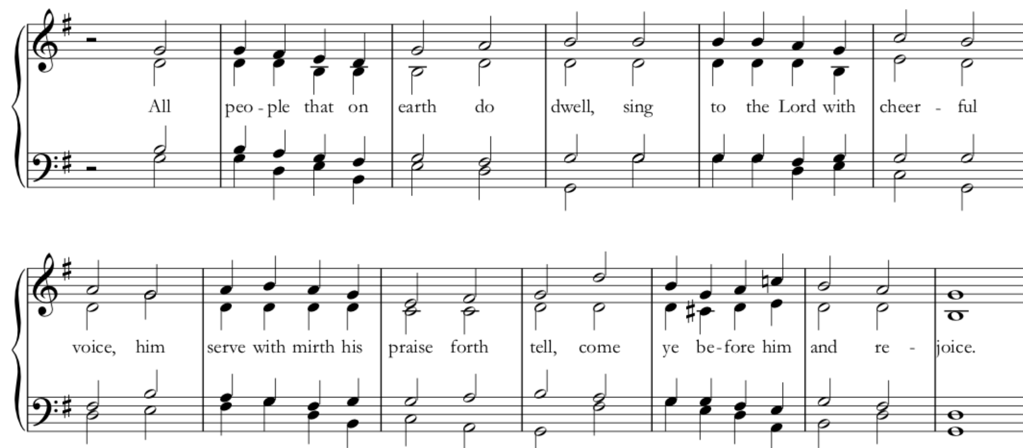
In general, the random and simulated annealing algorithms generate better solutions than the local search algorithm. This makes sense, as the local search algorithm is the only one of the three that is guaranteed to get trapped at a local optimum. The random algorithm is free to explore the entire solution space, and the simulated annealing algorithm uses probability to escape local optimum for a certain amount of time.

On average, the local search and simulated annealing algorithm performed much faster than the pure random algorithm. This makes intuitive sense as well – after all, for each iteration, the random algorithm is required to generate an entirely new solution from scratch. On the other hand, the other two algorithms merely need to adjust an existing solution to find a "neighbor", and furthermore, their error score update need only analyze the region that has been changed to update the error score. Thus, although the random algorithm seems to perform slightly better on average, the tradeoff with speed makes simulated annealing the well-rounded choice for our purposes.

6.2 Musical analysis.

Let's examine a harmonization produced by the simulated annealing algorithm against the harmonization published in the Harvard University Hymn Book. The melody is a very common sacred hymn, All People That on Earth Do Dwell.

First, let's examine the published harmonization.



This is an excellent example to study, as the harmonization only makes use of major, minor, and diminished triads. In addition, all of the formalized principles incorporated into our algorithm appear in this harmonization. For example, in the first two chords of the second line, the alto part jumps up 5 semitones. It begins on a D, and the next chord is E minor {E, G, B}. Technically, the closest note in E minor the alto could jump to from the D is the E, but the alto goes up to the G to avoid parallel octaves with the bass.

The bass part of this harmonization has a few interesting deviations from our formalization – namely, the octave leap in the fourth measure of the first line and the major 7th (11 semitone) leap in the fourth measure of the second line. We'll see how our algorithm does its best to avoid precarious leaps like this.

Here is one of the best (lowest error scores, and also subjectively pleasant to my own ears) harmonizations of this melody offered by our simulated annealing algorithm:

This is when a very special kind of music theory begins to emerge – one that seems to favor voice-leading over all other factors. See, even when voice-leading is meant to be balanced against other criteria, there are rarely any huge compromises in the resulting harmonizations, whereas other criteria are sometimes ignored or beat out by voice-leading. Of course, our formalization of the harmonization problem does not explicitly incorporate cadences from dominant chords to tonic ones (see Appendix A), but voice-leading still seems to allow these gestures to appear. Consider the final two measures of this harmonization - even though the E major and D major in the penultimate measure of the piece do not have a clear subdominant/dominant relationship, the bass motion from G♯ to A implies an applied dominant to A minor. This A minor would function as a proper subdominant to the dominant of G, D major. However, instead of A minor, the harmonization immediately skips that step and goes straight to the dominant, D

major. In this way, the voice-leading implies a kind of functional harmony without stating it explicitly.

Perhaps more exciting than this implicit functional harmony is the fleeting presence of some neo-Riemannian concepts (see Appendix A). This should not come as a huge surprise – after all, neo-Riemannian theory was born out of the musicological community's interest in creating a descriptive theory of non-functional harmony where voice-leading played the main role in chord progressions. In the above harmonization, we see an **L** shift in the fourth measure of the first line, from a G major chord {G, B, D} to a B minor chord {B, D, F♯}. This, of course, is not a particularly striking example, as these chords work well together in the context of functional harmony. However, a more exciting instance of neo-Riemannian residue appears in the second and third measures of the second line. Here, a B major chord is immediately followed by an E♭ major chord. This transformation can be represented using two neo-Riemannian shifts in succession. First, an **L** shift turns the B major chord {B, D♯, F♯} into a D♯ minor chord {A♯, D♯, F♯}, which is enharmonically equivalent to E♭ minor {B♭, E♭, G♭}. Next, a **P** shift transforms this E♭ minor chord {B♭, E♭, G♭} into an E♭ major chord {B♭, E♭, G}. There is no reasonable explanation for this chord progression in the context of functional harmony, as neither of these chords appear naturally in the melody's key of G major (which contains the notes {G, A, B, C, D, E, F♯}) and they do not share any applied dominant relationship to one another.

When we adjust the weights of each type of error, we can create dramatic differences in the resulting harmonizations. For instance, consider the following harmonization which was generated by removing the error score associated with pitches falling outside of the key.

All peo-ple that on earth do dwell, sing to the Lord with cheer - ful voice, him

serve with mirth his praise forth tell, come ye be - fore him and re - joice.

Now that voice-leading is the primary agent of musical motion, we see that neo-Riemannian concepts begin to emerge more frequently and organically from the harmonization. In the very first two chords, we see a complete lack of functional harmony and a total dominance of neo-Riemannian voice-leading. The first chord, an E♭ major triad {E♭, G, B♭}, is transformed with a **R** shift to C minor {E♭, G, C}, then with a **P** shift to C major {E, G, C}, and then finally with an **L** shift to E minor {E, G, B}. Once again, these chords can not be described by any functional harmonic analysis. The small and tightly controlled voice-leading exercises of the neo-Riemannian primitives guide the harmony through unusual and seemingly foreign chords like B major {B, D♯, F♯} and G♯ minor {G♯, B, D♯}. Best of all, by creating a computational conception of functional music theory, I have uncovered my own compositional preferences regarding the balance between harmonic function and seamless voice-leading.

6.3 Future work.

It is possible to generalize our algorithm in a few ways. First of all, we have only been applying our within-the-key error score to the major key. If we replace this key with a different scale, like the octatonic scale (represented with the set $[0, 1, 3, 4, 6, 7, 9, 10]$ or $[0, 2, 3, 5, 6, 8, 9, 11]$) we can see the dramatic effect that has on our resulting harmo-

nizations.

In addition, our method for generating random chords for a given melodic note only allows for three possibly constructions, based on the three chord qualities we defined in chapter 3 (major, minor, and diminished). If we add more of these possible constructions, our harmony will be more complex. Of course, since this loosens the rigidity of our problem formulation, it means that the program is likely to become less consistent in its harmonizations. However, if we consider this from a practical perspective where the program is meant as a compositional tool, then a greater breadth of style in the resultant harmonizations is a good thing!

Taking this one step further, it should be possible to generalize the number of voices involved in generating the harmonization. As long as it is clear that the lowest voice (in terms of frequency) operates according to slightly different principles from the other voices, our algorithm should be equipped to handle this generalization.

Most of my hopes for the future of this project involves the incorporation of more complex harmonic structures. First of all, the program should be expanded to allow four-note or even five-chords (of course, you can only have as many notes in the chord as there are voices in the harmonization, but we have already discussed the generalization of our algorithm to incorporate more voices). Second, it should consider the possibility of having more than one chord occur over the course of a single melodic note. Finally, it would be most exciting to generalize the chordal construction methods to allow for chords that do not strictly classify as a major, minor, diminished, or other kind of chord. Instead, weights can be assigned to dissonances within a chord (whether there are semitone clusters, a set of three or more notes seperated only by a semitone, such as $[0, 1, 2]$), but that's all – that way, the computer might end up producing extremely exotic harmonies that a human might not have encountered on their own.

Appendix A

Music theory fundamentals.

For the sake of thoroughness, here I present a brief crash course on tonal harmony and neo-Riemannian music theory so that the analyses presented in Chapter 6 will be better supported by music theory fundamentals. The material in Chapter 3 should not require an understanding of tonal harmony to understand the mathematical and set notation representation given, but to grasp tonal harmony might offer greater justification for the various formulas postulated.

Classical harmony.

Western music theory, there are twelve *pitch classes*: {C, C♯/D♭, D, D♯/E♭, E, F, F♯/G♭, G, G♯/A♭, A, A♯/B♭, B}. Here they are organized by ascending fundamental frequency, also known as *pitch*. If the frequency of a pitch is doubled, it will become the same pitch class one *octave* above. Scientific pitch notation is used to describe exact pitches by naming their pitch class followed by their octave. C5, for example, is the C one octave above C4, which is one octave above C3, etc. The distance between two adjacent pitch classes (e.g. E4 to F4) is called a *half-step/semitone*, and a distance of two semitones (e.g. G3 to A3) is called a *whole-step/tone*. In general, the distance between two pitches is called an *interval*.

In tonal music, one of the twelve pitch classes plays the dominant role in any composition. This central note is called the *tonic*. From the tonic, we can derive most of the structures upon which tonal harmony is based: the *scale* and the *chord*.

A scale is the collection of pitch classes that will be primarily used in the majority of a given piece. For example, the *major* scale is created by starting on the tonic pitch class and then stepping upwards by the following distances: whole step, whole step, half step, whole, whole, whole, half. Thus, the scale of G major would be comprised of the following pitch classes: {G, A, B, C, D, E, F♯(, G)}. G is the *tonic* of this scale, and is also sometimes called the *first scale degree*. A is the second scale degree, B the third, and so on.

A chord, on the other hand, is a collection of pitch classes that sound at the same time. In this paper, we are primarily concerned with chords with three notes called *triads*, where the chord is constructed from a *root* pitch and two notes a certain distance away from that root. For instance, to construct a major triad, we choose a root, and then the pitches that are 4 and 7 semitones above that root. The chord A major would be {A, C♯, E}. In this chord, A is the root, C♯ is the *third* and E is the *fifth*. Here is an important detail about chords - the general conception of a chord is that the root and quality (e.g. major or minor) of a chord does not depend on their absolute frequency. Put another way, the root, although it is the foundation of the chord, does not necessarily have to have the lowest frequency. Consider an A major chord constructed from the pitches A4, C♯4, and E2. This is still an A major chord because all of the relevant pitch classes are represented, but E2 is the lowest (by frequency) note in the chord. However, the lowest frequency note in a chord does change the chords quality - chords with the root as the lowest frequency feel much more stable and resolute than chords with the third as the lowest frequency. Thus, we use the term *inversion* to distinguish between the three possible lowest frequencies. If the root of the chord is the lowest frequency, the chord is in *root position*; if the third is the lowest,

the chord is in *first inversion*; if the fifth is the lowest, the chord is in *second inversion*.

Much of our analysis in Chapter 6 assesses music through the lens of *functional harmony*, where each chord serves a functional role in the overall progression. In functional harmony, the most important relationship is that between the tonic chord of the scale and the chord built on the fifth scale degree, also known as the *dominant*. This relationship can be generalized: the most fundamental motion in functional harmony is from one chord to another where the root of the second chord is 5 semitones above or 7 semitones below the first chord. These distances are equivalent because our pitch classes exist in a mod 12 space: going up 5 semitones from D, for instance, gives us a G – going down 7 semitones yields the same result. *Subdominant* chords are chords which bridge the gap between a tonic chord and any following dominant.

Neo-Riemannian theory.

One newer branch of music theory is called neo-Riemannian theory. This descriptive model of music theory sought to draw connections between adjacent harmonies and their voice-leading "proximity" to one another, rather than their respective relationship with the tonic.[1] In this model, there are three transformations which can connect to adjacent chords:

- The **P** shift changes a major triad to its parallel minor triad and vice versa. The parallel triad of a chord is that which shares the same root but not the same chord quality. Thus, C major and C minor are parallel triads. In a major triad, the third is moved down a half-step to form a major triad, and in a minor triad, the third is moved up a half-step to form the parallel major. As an example, the shifts from {C, E, G} to {C, E♭, G} and back are both P shifts.
- The **R** shift transforms the chord to its relative triad. For major chords, this

[1] Richard Cohn. "An Introduction to Neo-Riemannian Theory: A Survey and Historical Perspective". In: *Journal of Music Theory* 42.2 (1998).

means moving the fifth up a whole step (for C major, we'd move the G to A and create an A minor triad), and for minor triads, it means moving the root down a whole step. As an example, the shifts from {C, E, G} to {C, E, A} and back are both R shifts.

- The **L** shift shifts the *leading tone* of a major or minor chord by a semitone. For major chords, this means moving the root down by a semitone. For minor chords, the fifth is moved up a semitone. As an example, the shifts from {C, E, G} to {B, E, G} and back are both L shifts.

These shifts clearly prioritize voice-leading – the distance that each note in the chord must move to transform into the next chord – over harmonic function.

Appendix B

Complexity theory fundamentals.

Big-O notation.

Big-O notation is used to classify algorithms based on how their runtimes grow as the input size approaches infinity. The formal definition of Big-O is as follows:

$$f(n) = O(g(n)) \text{ iff } f(x) \leq c \cdot g(x) \text{ for all } x \geq x_0$$

For some constants c and x_0. As an example, if $f(n) = 2n + 1$, then $f(n) = O(n)$, because $2x + 1 \leq 3 \cdot x$ for all $x \geq 1$. However, $n^2 \neq O(n)$ because n^2 always eventually becomes greater than n as n approaches infinity. ($\lim_{n\to\infty} \frac{n}{n^2} = 0$)

How do these apply to algorithms? Well, it basically allows us to ignore any small insignificant constants when we analyze the runtimes of our algorithms. For instance, if we had an algorithm that found the double the sum of all of the numbers in a list of n elements, it might do one of two things. It might start with a running sum of 0, check each number in the list, adding it to the running sum, and then double that sum once we reach the end of the list. In this case, the algorithm does one addition for each of the n elements in the list, and one multiplication at the end, for a total of $n + 1$ operations. However, it could also start with a running sum of 0, and for

each element in the list, double that number and add it to the running sum. In this case, the algorithm does 2 operations (multiplication then division) for each of the n elements of the list, for a total of $2n$ operations. Although these runtimes are slightly different, Big-O notation allows us to pinpoint their similarity. Both $n+1 = O(n)$ and $2n = O(n)$. Saying that both of these algorithms runs in $O(n)$ time identifies the fact that they both have to look at each element of the list once (or a constant number of times).

If an algorithm is said to run in "polynomial time", that simply means that its runtime $f(n)$ satisfies $f(n) = O(g(n))$ where $g(n)$ is some polynomial (a polynomial is simply a function $h(n)$ that can be written as $h(x) = \sum_{k=0}^{n} c_k x^k$ where $c_k, \ldots, c_0$ are constants).

Complexity classes.

Another building block of complexity theory one level above asymptotic Big-O notation is the complexity class. These categories are more general, and allow us to classify problems (not algorithms) into broader groups. For example, the complexity class P is defined as the class of all problems for which there exists a polynomial time algorithm to solve them. The problem we outlined above in our description of Big-O notation would be in P, since we provided algorithms to solve the problem in $O(n)$ time.

On the other hand, NP is defined as the class of problems for which a proposed solution can be verified in polynomial time. For instance, consider the following problem: given an integer n, does n have a prime factor $< k$? None of the known solutions to this problem run in polynomial time. However, if a solution p was given, and $p < k$ and p divides n, then we know the answer to our problem is yes, since the prime factors of p are sure to be $< k$. Thus, we can easily verify this solution, and the problem of prime factorization is in NP.

There is one more subset of problems in NP, known as NP-complete problems. A

problem is NP-complete if every problem in NP can be reduced to that problem. This is mentioned in the future work section of the final chapter.

Bibliography

Ames, Charles. "The Markov Process as a Compositional Model: A Survey and Tutorial". In: *Leonardo* 22.2 (1989), pp. 175–187.

Bharucha, Jamshed J. "Modeling the Perception of Tonal Structure with Neural Nets". In: *Computer Music Journal* 13.4 (1989), pp. 44–53.

Burkholder, J. Peter; Grout, Donald Jay, and Palisca, Claude V. *A History of Western Music (Eighth Edition)*. Norton: New York, 2010.

Cohn, Richard. "An Introduction to Neo-Riemannian Theory: A Survey and Historical Perspective". In: *Journal of Music Theory* 42.2 (1998).

— "Neo-Riemannian Operations, Parsimonious Trichords, and their Tonnetz Representations". In: *Journal of Music Theory* 41.1 (1997).

Cope, David. "Experiments in Music Intelligence (EMI)". In: *ICMC Proceedings* (1987), pp. 174–181.

Eck, Douglas. "Finding Temporal Structure in Music: Blues Improvisation with LSTM Recurrent Networks". In: *Neural Networks for Signal Processing XII, Proceedings of the 2002 IEEE Workshop* (2002), pp. 747–756.

Garey, Michael and Johnson, David S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, 1979.

Glover, Fred. "Tabu Search – Part 1". In: *ORSA Journal on Computing* 1.2 (1989), pp. 190–206.

Gomes, Peter F.; Burt, Matthew F.; Cooman, Carson P.; Huff, Harry Lyn, and Jones, Edward Elwyn, eds. *The Harvard University Hymn Book.* Harvard University Press, 2007.

Kirkpatrick, S.; Jr., C. D. Gelatt, and Vecchi, M. P. “Optimization by Simulated Annealing”. In: *Science* 220.4598 (1983), pp. 671–680.

Koops, H. V. *A Model Based Approach to Automatic Harmonization of a Melody.* Utrecht University, 2012.

Lerdahl, Fred. “Tonal Pitch Space”. In: Music Perception: An Interdisciplinary Journal 9.2 (1991).

Lerdahl, Fred and Jackendoff, Ray. *A Generative Theory of Tonal Music.* The MIT Press, 1996.

Mitzenmacher, Michael and Upfal, Eli. “Some Practical Randomized Algorithms and Data Structures”. In: *Computing Handbook, 3rd ed.* Ed. by Teofilo F. Gonzalez; Jorge Diaz-Herrera, and Allen Tucker. 2014.

Morris, Robert. *Composition With Pitch-Classes: A Theory of Compositional Design.* Yale University Press, 1987.

Nagler, Dylan Jeremy. *SCHUBOT: Machine Learning Tools for the Automated Analysis of Schubert’s Lieder.* Bachelor’s thesis, Harvard College, 2014.

Taube, Heinrich. “Automatic Tonal Analysis: Toward the Implementation of a Music Theory Workbench”. In: *Computer Music Journal* 23.4 (1999).

Tymoczko, Dmitri. *A Geometry of Music: Harmony and Counterpoint in the Extended Common Practice.* Oxford Studies in Music Theory. Oxford University Press, 2011.

— “Three Conceptions of Musical Distance”. In: Mathematics and Computation in Music (2009).

Wülfing, Jan and Reidmiller, Martin. “Unsupervised learning of local features for music classification”. In: *International Society for Music Information Retrieval Conference* (2012).

Xenakis, Iannis. *Formalized Music: Thought and Mathematics in Composition.* Harmonologia Series No. 6. Pendragon Press, 1992.