Harmony in Memory: A Program to Help Harmonize a Classical Melody

Alexander E. Levinson
Bard College

Harmony in Memory:
A Program to Help Harmonize a Classical Melody

Senior Project Submitted to

The Division of Science, Math, and Computing

of Bard College

by

Alexander Levinson

Annandale-on-Hudson, New York

May 2022

Acknowledgements

I would like to thank those on my Senior Project Midway board, consisting of Kerri-Ann Norton, Robert Mcgrail, and Erica Kiesewetter, for helping me narrow my focus on what exactly I needed to write and code. I would also like to thank Bard College for helping me get access to some academic papers that I read for this project. An extended, special thank you goes to Kerri-Ann Norton who advised me throughout this past year, keeping me on track with deadlines as well as suggesting great ideas on how to improve my code.

Table of Contents

Abstract

Harmony plays a crucial role in classical music: supporting the melody. There are, however, so many different ways to harmonize one melody that one person may not even consider them all. I developed a program in Python 2.7 which would intake a melody of notes, for example in the form of "C, D, E, C", and return all possible harmonizations to that melody (excluding some more complicated harmonic cases that are not yet implemented into the program). The first draft of the program that was made was clearly optimizable, leading to two versions of the programs being made: the "unoptimized" and "optimized" programs. Comparing runtimes and memory consumption of these programs showed that while the optimized program was always predicted to be faster than the unoptimized one, for larger and larger melody inputs, the unoptimized program actually used less memory than the optimized one.

Introduction

Without harmony, a melody can seem bare and unsupported, at least in the tradition of classical music. Harmony serves the purpose to further enrich a melody, making it seem more complex and therefore more interesting to listen to. In fact, there are many different possible classical harmonizations that can fit one unique melody, so many that the average musician would not feasibly contemplate all possibilities. In order to assist with this problem, my goal was to develop an algorithm to assist the writing of a musical composition in the classical style. The idea was that a user could input a melody and my algorithm would output all possible classical style figured bass progressions that would harmonize that melody.

It is difficult to explain what "classical" styled composition really is because it is disputed what "classical" means. It can be said that music in the classical style is the process of ordering musical notes and sounds into a piece of music following rules that were written by European composers that lived around the 1800s. One book of rules is the *Gradus ad Parnassum* by Johann Joseph Fux (Fux, 2000). This work is one of the written baselines for classical composition rules, and it helped guide composers to come up with conventions for figured bass chord progressions. **Figured bass** means labeling chords with roman numerals, such as a “I” chord, or “III” chord,

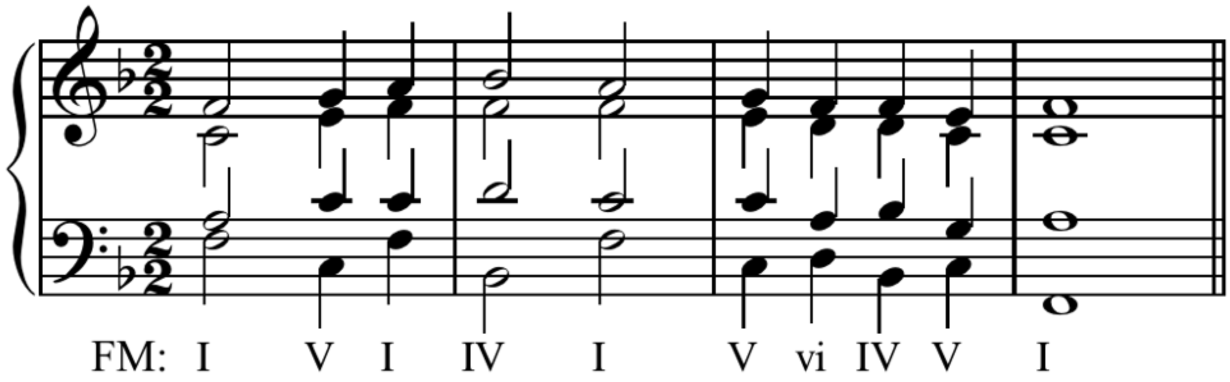


Figure 1. A 4-Part Harmony. (Image from Benward & Saker, 2003)

which represent three or, in some cases, more notes played at the same time in order to create complex music (see Figure 1).

A classical piece of music is mostly based on rising and falling waves of pitches, tension and release of that tension. This is called the **melody**. The pitches exist in different **scales**, for example, a C major **scale** contains the pitches: C, D, E, F, G, A, and B, ordered like in the alphabet. This alphabetical relationship makes "D" a higher pitch than "C", and "E" a higher pitch than "D" (if the notes stay within one octave, which in my program, they will). Going from any note to a note above it in a piece of music creates a feeling of rising tension, and going from a note to a note below it releases that tension. Also, whatever the three or so pitches that the ear hears first is regarded as the baseline for pitches coming after it. This baseline is what sets the **key** of a piece of classical music.

The **key** of C major refers to the **scale** that was aforementioned. This key holds within it relationships of tension (and release of that tension) between notes. A typical melody in a piece of classical music will use notes in a scale to express different relationships between its pitches to create a feeling of tension and release to the listener. If I were to create a simple C major melody it could be: C, D, E, G, D, E, C. There is rising tension between "C" and "D", "D" and "E", "E" and "G", and release between "G" and "D", and "E" and "C". A classical melody should typically have a release for every instance of tension, give or take, meaning that my building tension in C, D, E, G is released by G, D and my D, E is released by E, C. Knowing this tension and release is useful for understanding how chords come into the music.

Chords in classical music allow the melody to be supported; they make the melody sound more complex and "rich" to the listener because of how chords reinforce harmonic overtones in the melody. In my program there are 12 chords that are used. For example, in C major (which is not the same for another key, like A major), one can use these chords:

1. I chord = [C,E,G] consisting of the notes C, E, and G.
2. I6 chord = [E,G,C] (also) consisting of the notes C, E, and G. (I and I6 are related by their "I")
3. I64 chord = [G,C,E] ...
4. II = [D,F,A]
5. II6 = [F,A,D]
6. III = [E,G,B]
7. IV = [F,A,C]
8. V = [G,B,D]
9. V7 = [G,B,D,F]
10. VI = [A,C,E]
11. VII = [B,D,F]
12. VII7 = [B,D,F,A]

(Levinson, 2021)

(Note that I refer to all chords in one key to be in uppercase format even though Figure 1 uses the convention of using lower case for certain chords. Also, some more complex chords, such as V9s and "borrowed" chords did not make it into this program due to time constraints.)

These chords are essentially groups of notes that when played at the same time have a purpose to propel the music into tension or release. It is important to note that these chords' numeral orders are just like the order of a scale; a I chord is below a II chord, and a II chord is below a V chord, etc… The numbers that come after the roman numerals indicate the order in which the elements of the chords are played in relation with each other, changing how the chord will sound, which plays an important role in harmonic progressions. A I chord going to a V chord and then back to a I chord is one of the most basic tension and release chord progressions. This is true partly because the first element in the I chord is a C, and the first element in a V chord is G; there is tension going from C to G. Then from V to I is a release between G and C. The second and third, and sometimes fourth or fifth elements in a chord are played at the same time as the first element because they conventionally enrich the music when played that way. To tie chords with melodies, essentially any chord can be played at the same time as any melody note as long as the chord contains the melody note in its elements.

Classical chord progressions consist of conventional rules; if there is a melody to be harmonized, one cannot simply assign any chord to any melody note and have the music work. In the following rules, note that any time I list a chord without any numbers in its name, I mean that any chord that starts with the roman numeral can be used even if it is followed by numbers (but I will mark exceptions). For example, IV can mean IV, or IV6, or IV64 unless specified otherwise. The following rules are (for a major key, like C major):

1. The last chord of any piece of classical music must be a I chord (with no inversion).
2. If the piece does not start on a I chord (with no inversion, no numbers preceding it), a I chord must be used within the first third of the harmonization.
3. A V7 chord can only happen in the last 1/3 of the piece.
4. A II chord can only be used after a I and a IV has been used in the piece already.

5. Any chord can proceed itself but it should be that this proceeding chord is a different inversion than the first one. But also, if a chord can be repeated twice in a row, and the following ⅓ of the melody's length of notes can be harmonized by that same chord, then one can assign all of those notes with the same chord inversion (which is called a drone). Which inversions of chords to use are at the discretion of the user.

6. If a I chord (but not I64) has been played, then, if there is a next note, a IV, V, I6, I64, or II can be assigned to it (however II should only be used after a I and a IV have been used in the piece already).

7. If a I64 has been played then, if there is a next note, a V can be used on it.

8. If a IV has been played, then a I chord, or I64, or II, or V can be used on the next note.

9. If II was used, then a V, or VII65 can be used on the next note.

10. If a III was used, then a VI can be used on the next note.

11. If a V was used, then a I, or IV, or VI can be used on the next note.

12. If a V7 was used, then a I can be used on the next note.

13. If a VI was used, then IV, or II, or V can be used on the next note.

14. If a VII or VII65 or VII7 was used, then a I can be used on the next note.

(Fux, 2000) (Levinson, 2021).

Figure 1 is an example of a four-part harmony (that conforms to the above rules). The melody consists of the topmost notes (F, G, A, Bb, A, G, F, F, E, F) while the three notes directly below each melody note represent the chord harmonizing that melody note. The "FM" that is written in the figure corresponds to the key of F Major. The chord progression rules apply to this harmony: a I chord is used for the first melody note, F, and a I chord in F Major (= [F,A,C]) contains the first melody note, F. The second chord, V in F Major (= [C,E,G]), contains the second melody note, G. One can also see that rule #6 allows the I chord to be followed by the V. The third melody note, A, is harmonized by a I chord because A exists within a I chord (in F major) and also because rule #11 states that a I chord can be used after a V chord was used. The fourth melody note, Bb exists within a IV chord (in F major) and rule #6 allows a IV to come after a I. The fifth melody note, A exists within a I chord (in F major) and by rule #8, a I can come after a IV. The sixth melody note, G exists within a V chord (in F major) and by rule #6, a V can come after a I. The seventh melody note, F exists within a vi chord (in F major), which I notate as VI in my program, and by rule #11, a VI can come after a V. The eighth melody note, F exists within a IV chord (in F major) and by rule #13, a IV can come after a VI. The ninth melody note, E exists within a V chord (in F major), and by rule #8 a V can come after a IV. Finally, the last melody note, F exists within a I chord (in F major) and by rule #1, the I chord ends the harmonization.

The melody that I made earlier, C, D, E, G, D, E, C, can be harmonized with a few different progressions. To show a couple: I, V, I, I, V, V, I, meaning the first C is harmonized by the first 'I' chord, the first D is harmonized by the first V chord, etc…, or I, V, I, I6, IV, V, I could also work as a progression for this melody. Also note that I, V, I, I6, IV, V, I could mean I, V, I6, I, IV, V7, I, because the inversions of the chords can be chosen by the user to their discretion, but with two exceptions: a I chord cannot always be replaced by a I64 chord (as is specified in rule #6), and the last chord of the harmony can only be a I chord (with no inversion) by rule #1. These harmonies can work because each melody note exists in each chord corresponding to it, and they also follow the rules of how chords can be used, for example in the first progression, every time a

V chord is seen, it is preceded by either a 'I' chord, due to rule #6, or a V chord, due to rule #5. If the progression were to be I, II, I, I, V, V, I, while every melody note exists within its corresponding chord, this progression is invalid because the II chord appears without a IV chord anywhere behind it in the progression, violating rule #4.

It seems that most classical style melodies will have no trouble conforming to these chord progression rules, but if there somehow is a case that the input melody is not classical, like a Noise Rock melody, and possibly no harmony can be given to it, then my program will not output any harmonization. For example, I do not see how the melody: [C, B, F, F] in C major can be harmonized, mainly because it violates rule #1: that the harmony must end with a 'I' chord. "F" does not exist within a 'I' chord in C major. My program will output every harmonic chord progression (based on my set of "classical" harmony chords and rules) that could be used for a classical style melody.

Other research has focused on finding harmonizations in music, for instance in the paper "Bach in a box: The evolution of four-part baroque harmony using the genetic algorithm" (Mc Intyre, 1994). The researchers input a user-given melody and developed an algorithm to create a 4-part Baroque style harmony based on rules of counterpoint. Their genetic algorithm would generate a value that represented how fit, or good, the harmony was for the melody. The value was increased for following counterpoint rules such as contrary motion, where, say, one harmony note would lead to the next harmony note in one direction while another note in the music would lead to its next note in the opposite direction (see Figure 2, Rule 16). However some number of fitness points were taken away if the program encountered parallel motion (see Figure 2, Rule 14). Parallel motion sometimes sounds odd in Baroque style music. Their genetic algorithm worked by mutation. "Mutation is modified to randomly change a note value…" meaning that the algorithm would run over and over again, slightly changing which notes were in the harmony each time and testing whether the fitness was higher than before (Mc Intyre, 1994). Then the harmonies of highest fitness were shown to the user. Another paper, "Automatic Generation of Four-part Harmony" also assigned a fitness method while constructing harmonies but did so through the use of Markov MDPs (finite state machines) (Yi et al., 2007). The authors argue that "The choice of a chord is similar to the choice of an action in MDP planning. If we use utilities to decide the goodness of the harmony, then we will want to pick a set of chords which can maximize the utilities" (Yi et al., 2007). The Markov MDPs will act on given data about the "appropriate ranges" that, in four-part harmony, the soprano, alto, tenor, or bass notes could be. This involved work with finite state spaces. Because of the different "appropriate ranges", the authors had to deal with different octaves of notes, because the note C can exist in many numbers

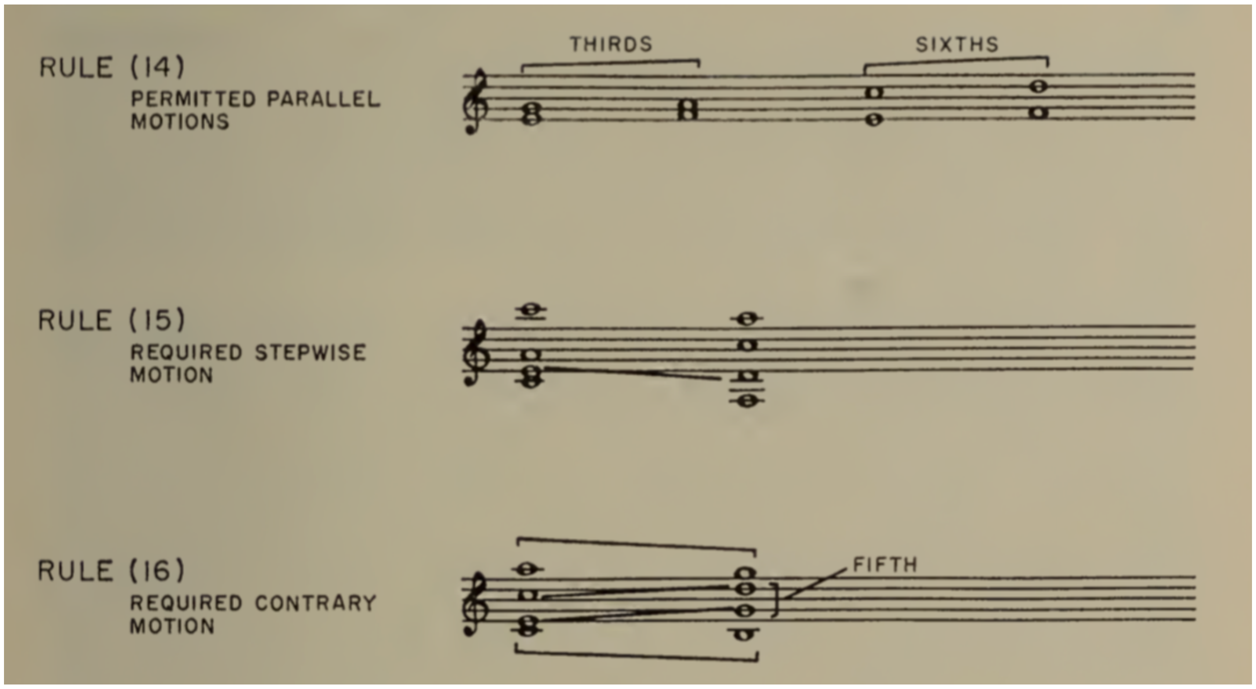


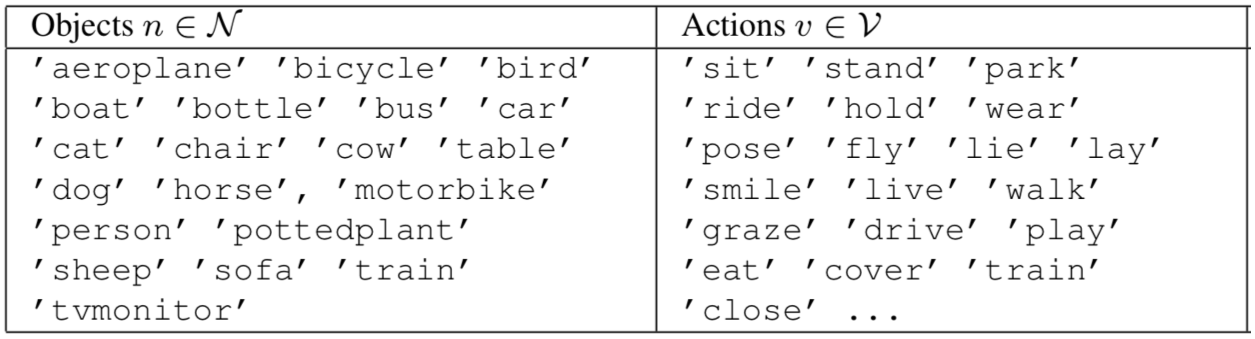
Figure 2. Examples of parallel, stepwise, and contrary motion (Hiller et al., 1959).

of audible Hz values: 16.35hz, 32.70hz, 65.41hz, 130.81hz, 261.63hz, 523.25hz, etc... The paper distinguished between different octaves by writing "C," "C" "c" "c′ " in order of lower to higher octaves. The program needed to distinguish this because it was dealing with counterpoint rules (as did the "Bach in a box" researchers) (Yi et al., 2007). They also specified that they would only use melodic notes at each beat and only consider adding chords at each beat, and keep their melody inputs as only in C major. The MDPs would then rely on dynamic Bayesian networks (DBNs) to have a probability of which notes would be restricted to be used in the harmony, as to control the output (Yi et al., 2007).

My program, like "Bach in a Box" is using a user defined melody as input except that I am using Classical style harmony based on rules of figured bass chord progressions. And while Baroque and Classical harmony are not unimaginably different from each other, a difference between their work and my project idea is that they used a genetic algorithm to show the user configurations of harmony notes (which would end up forming chords) that they could use for their melody through many iterations of trial and error; while my chords are already figured out: I, I6, II, V, etc…. meaning that while my harmonies do not specify positions of notes in the harmonies, as the "Bach in a Box" researchers did, they show what notes can be used in the harmony.

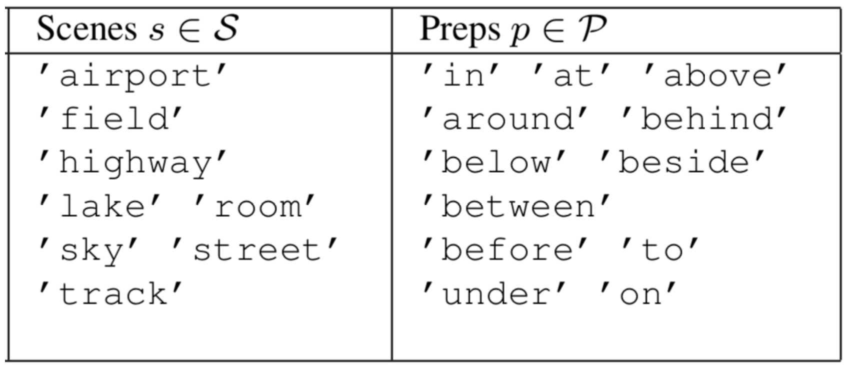
Natural language processing proved a different method of approaching computer harmonization. "Corpus-Guided Sentence Generation of Natural Images" is a paper where the writers,Yang, Yezhou, et al., discussed a program that would create specific sentences that would describe images fed to the program, (which, for my program, was just like creating harmonizations to inputted melodies) (Yang, Yezhou, et al., 2011). They theorized working with

(Objects are represented by the set *N)* (Actions are represented by the set *V*)



Objects $n \in \mathcal{N}$	Actions $v \in \mathcal{V}$
'aeroplane' 'bicycle' 'bird'	'sit' 'stand' 'park'
'boat' 'bottle' 'bus' 'car'	'ride' 'hold' 'wear'
'cat' 'chair' 'cow' 'table'	'pose' 'fly' 'lie' 'lay'
'dog' 'horse', 'motorbike'	'smile' 'live' 'walk'
'person' 'pottedplant'	'graze' 'drive' 'play'
'sheep' 'sofa' 'train'	'eat' 'cover' 'train'
'tvmonitor'	'close' ...

(Scenes are the set S) (Preps are the set P)



Scenes $s \in \mathcal{S}$	Preps $p \in \mathcal{P}$
'airport'	'in' 'at' 'above'
'field'	'around' 'behind'
'highway'	'below' 'beside'
'lake' 'room'	'between'
'sky' 'street'	'before' 'to'
'track'	'under' 'on'

Figure 3. Sets for Objects, Actions, Scenes, and Prepositions.

"trained detection algorithms" to extract sets of words based on the objects that could be identified within the image.

> "Key to our approach is the use of a large generic corpus such as the English Gigaword [Graff, 2003] as the *semantic grounding* to predict and correct the initial and often noisy visual detections of an image to produce a reasonable sentence that succinctly describes the image" (Yang, Yezhou, et al., 2011)

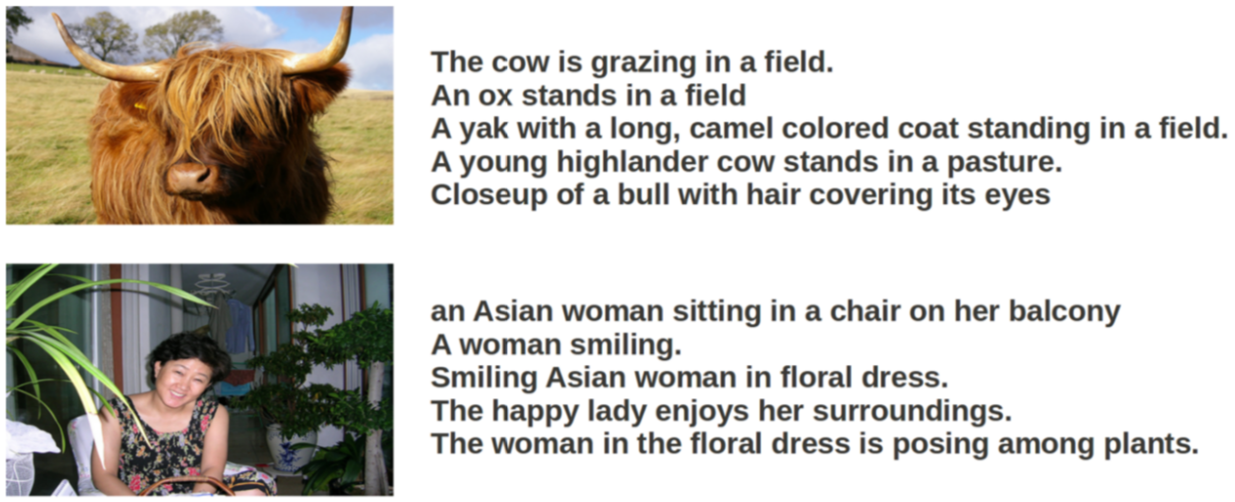
This means that they used the English Gigaword dataset (which can range to 400gb of data) to more clearly detect what kind of images they were dealing with.

Given an image of a bull in a field, an image of a man on his bike, a woman smiling among plants, etc… the program outputs this table. Their program only stores up to 20 words per set. In my program, I can say that each melody note has its own set of chords that can harmonize it. The sentence generation program has sets: Objects (nouns), Actions (verbs), Scenes (well, scenes), and Preps (prepositions). Elements from the sets in Figure 3 will then be used to construct sentences, annotations (which can be seen in Figure 4), using the rules of English grammar. "...each annotation is usually short – around 10 words long" (Yang, Yezhou, et al. 2011). Their algorithm more specifically works by creating object classes, for example a *chair* object may have synonyms such as chaise, daybed, rocker, armchair, wheelchair, etc…, and then they

> "...can now compute from the Gigaword corpus [Graff, 2003] the probability that a verb exists given the detected nouns, $P_r(v|n_1,n_2)$ [where v is a verb and n1, n2 are nouns]. We do this by computing the log-likelihood ratio [Dunning, 1993] , λ_{nvn}, of *trigrams* ($\langle n_1\rangle$, v, $\langle n_2\rangle$), computed from each sentence in the English Gigaword corpus [Graff, 2003]" (Yang, Yezhou, et al., 2011).

This means that they use the Gigaword data to create links between verbs and nouns, and eventually to objects, like the object *chair* and the subject *woman* (Yang, Yezhou, et al., 2011).

The outputs of the above "Corpus-Guided Sentence Generation…" paper reminded me of my program idea because where they gather sets for words to describe an image, I too gather



The cow is grazing in a field.
An ox stands in a field
A yak with a long, camel colored coat standing in a field.
A young highlander cow stands in a pasture.
Closeup of a bull with hair covering its eyes

an Asian woman sitting in a chair on her balcony
A woman smiling.
Smiling Asian woman in floral dress.
The happy lady enjoys her surroundings.
The woman in the floral dress is posing among plants.

Figure 4. Images side by side with their annotations.

chords to describe a melody. Yi et al.'s paper also prompted me to decide on a notational system for my notes. I decided that I will not be using just "C" "C#/Db" "D" "D#/Eb" "E" "F" "F#/Gb" "G" "G#/Ab" "A" "A#/Bb" "B", but rather, "0" "1" "2" "3" "4" "5" "6" "7" "8" "9" "10" "11" to make a uniform account for sharp and flat notes, not just the basic notes in C major. Sharp and flat notes are necessary if I am to harmonize different keys, which is something that I want to implement into this program in the future. For example, while a C major scale (as I mentioned before) contains "C" "D" "E" "F" "G" "A" "B", A major has "A" "B" "C#" "D" "E" "F#" "G#". So, C major in my notation would be "0" "2" "4" "5" "7" "9" "11", and A major would be "9" "11" "1" "2" "4" "6" "8". But like the researchers for the "Automatic Generation…", I also started my program in a simple fashion and confined myself to the key of C major (Yi et al., 2007).

Methods

There were a few program choices that could have been used to implement my algorithm, for instance the SWI Prolog language because of its strict definition-based system that could have beautifully represented chords. However, I used the Python 2.7 language because of a couple reasons: 1.) Its simplicity for implementing more complex rules of harmony, such as allowing a harmony not to start with a 'I' chord as long as a 'I' chord is within the first third of the length of the harmony (which is rule #2), and 2.) the complete control one has over each stage of their program for debugging, for example if the output of chords was somehow wrong, I knew I would be able to more precisely address the problem; in Prolog one needs to define all of the rules and facts but after that it is much more difficult to control the flow of the program once those facts are stated. Also, even though Prolog may also have executed the program faster, Python was fast enough when I started the process.

My first idea on how to find all possible harmonizations was a brute-force approach: to sift through all permutations of chords of an array the same length as the user-given melody. However, this turned out to be too complex; if the user gave, for example, my melody that I made earlier in this paper [C,D,E,G,D,E,C], then the program would test if [I, I, I, I, I, I, I] or [II, I, I, I, I, I, I] or [III, I, I, I, I, I, I] etc… would work with my chord progression rules. When in theory this could permutate into all valid classical harmonic progressions, the complexity, the amount of operations that the computer would execute, would be the number of chords, *N*, raised to the power of the length of the melody, *L*, which can be a large number, N^L, for a long melody and require the computer to store that many arrays of chords. The amount of memory that an array of chords takes can be calculated with a formula from the *Algorithms 4th ed* textbook by R. Sedgwick and Kevin Wayne (Sedgewick, R., Wayne, K., 2011). They write

> "An *array of primitive-type values* typically requires 24 bytes of header information (16 bytes of object overhead, 4 bytes for the length, and 4 bytes of padding) plus the memory needed to store the values. For example, an array of *N* int [integer] values uses 24 + 4*N* bytes (rounded up to be a multiple of 8)..."

The program stores strings, however, which are not primitive-type values. They also write "A String of length *N* typically uses 40 bytes (for the String object) plus 24 + 2*N* bytes (for the array that contains the characters) for a total of 64 + 2*N* bytes…" (rounded to be a multiple of 8). So my equation for an array of strings will be 64 + *XN* ~*N* bytes where *X* is the amount of memory that a chord (string) takes, and *N* is the number of chords in an array (Sedgewick, R., Wayne, K., 2011). Using the above melody, which has a length of 7 notes, and the amount of chords that I am using, 12, python would be managing $12^7 = 35,831,808$ different arrays of chords (each permutation). These would add up to $35,831,808 \cdot (64 + X(7))$ bytes of memory. Because X is the value of the string, it is the average number of characters used to define each chord; for example "I64" has three characters. The total number of characters used in all strings combined, 28, divided by the number of chords, 12: so 28 / 12 is roughly 2.3333 characters per chord. And since the size of a string is 64 + 2*N* bytes where *N* is the number of characters, the average amount of memory that a chord takes is $64 + 2(2.3333)\,bytes = 68.6666$ bytes. So the total amount of memory needed for all of these permutations would be $35,831,808 \cdot (64 + (68.6666)(7))\,bytes = 19.5164$ gigabytes of memory, which seems like too much for such a short melody, given that my program with its constraints would only output a few classical style harmonies for a short melody of 7 notes, and each of those harmonies take only a few hundred bytes at most, much less gigabytes.

Harmonizations can, in fact, be found more efficiently if the program applies the harmonization rules, getting rid of some options for chords to be used for certain melody notes, before finding all possible harmonic permutation, instead of first finding all permutations of the chords and applying the harmonization rules to all of them afterwards. Actually, the most chords that can apply to a melody note is 6 for the note A; the note A exists in a II chord, II6 chord, IV, VI, VII, and VII7 chord. So at this point, in a bad case (because the **worst** case is difficult to theorize due to the harmonization rules), the program would only deal with 8^6amount of harmonies, which is still a fairly large number when imagining possible harmonies, but is much less than 12^7. To do this, the program uses a loop to run through every chord progression rule for each individual element of the array, each individual melody note separated by a comma, and save all of the possible harmony progressions into other arrays which it will show to the user of the program, once it has finished. An example output would be:

```
[C,   D,   E,   G,   D,   E,   C] ← Input Melody.
[I,   V,   I,   I6,  V,   I,   I,] ← Harmony 1.
[I,   V,   I,   I64, V,   I,   I]  ← Harmony 2.
[I,   V,   I,   V,   V,   I,   I]  ← Harmony 3.
[I,   V,   I,   I,   V,   I,   I]  ← …
[I,   V,   VI,  V,   V,   I,   I]
```

Finding all possible chord progressions would have the program take a melody note, match it with all chords that it can be harmonized by, and then do the same with the next melody note. Then I will take these chords and combine them together in a way that follows the chord progression rules. Using a shorter melody, for simplicity's sake, C, E, C in this program might look like:

1. The user input melody should be in C major.

2. Take the first note of the melody: C

3. Is C an element of a 'I' chord ([C,E,G])?

    4. Yes, it is, so save the I chord to a new array.

5. Is it an element of a II chord?

    6. No.

    7. Is it an element of a III chord?

    8. …

(And go on to try for every chord)

9. Take the next note: E. Because it is the second to last note, it can only match with chords that can go to a 'I' chord, because the last note must be harmonized by a 'I' chord.

10. Is e an element of a I chord?

    11. Yes, so add I to its own new array. It so happens that E can only have a 'I' chord assigned to it if it is the penultimate note because E does not exist in any other chords (such as IV, V, V7, VII, and VII7) that lead to the 'I' chord of the final melody note.

14. Take the last note: C. By rule 1, it must be harmonized by a 'I' chord.

The above will leave us with a list per melody note which will contain all chords that the melody note it corresponds to can match with:

Chords that can harmonize C: ['I', 'I6', 'I64', 'IV', 'VI']

Chords that can harmonize E: ['I'] (Only this chord because it is the penultimate)

Chords that can harmonize C: ['I'] (Only this chord because it's the last note)

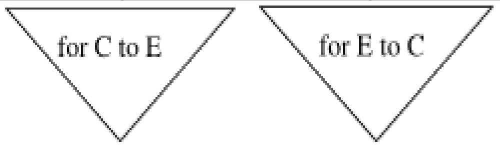
A piece of the code for this looks like:

```
for note in range(len(mel)):
        #Checks if it can be harmonized by a I.
        if (mel[note] in I): #mel is the user input melody list
                #harMemList is a listing that will hold all of the chords
                harMemList[note].append('I')
        if (melody[note] in IV):
                harMemList[note].append('IV')
        #etc… for all chords.
```

A code snippet that creates arrays of chords that contain a certain melody note.

The program then pairs up chords from these arrays together following the harmonization rules. Each pair of chords corresponds to the relationship between adjacent melody notes, because most of the simple harmonization rules deal with adjacent relationships of chords. Using the simple melody shown before:

Melody Notes:	C	E	C
Usable Chords	['I', 'I6', 'I64','IV', 'VI']	['I']	['I']



Pairs of Chords: Each pair is created by a rule.	Rule 5: ['I', 'I'], Rule 8: ['IV', 'I']	Rule 5: ['I','I']

Figure 5. The Pairing Process.

A piece of the code for this looks like:

```
#"count" and "i" iterate through harMemList revealing its chords.
if harMemList[count-1][i] == 'I': # If the chord in a melody note's list is a 'I'...

    # Searching for chords though the next melody note's list:
    for k in range(len(harMemList[count])):
        # If the chord in the next melody note's list is a 'IV'...
        if harMemList[count][k] == 'IV':
            # then put I and IV together in a pair:
            harmony[count-1].append([])
            pairs=pairs+1
            harmony[count-1][pairs].append('I')
            harmony[count-1][pairs].append('IV')

elif harMemList[count-1][i] == 'IV': # If the first chord is not a 'I', test if it's a IV…

    etc… for all the chord possibilities.
```

The above code snippet is basically saying that a IV can come after a 'I'.

Combining the pairs is the next step. The program combines them by checking if the last element of one pair matched the first element of another, if so then it creates a new list that would hold the combined pairs (omitting the first element of the second pair because it is essentially a duplicate). For example, combining [I,IV] and [IV,I] would result in a new list, [I,IV,I] (notice

Pairs of Chords: Chords that are highlighted will be combined because they share the same symbol.	['I', 'I'], ['IV', 'I']	['I','I']

Combined pairs:	['I', 'I', 'I'] ['IV', 'I', 'I']

Figure 6. Combining the Pairs.

that there is only one “IV”). If it were to append [I,V] to the previous example’s product, [I,IV,I], then it would extend the previous list to be [I,IV,I,V]. It can be visualized as such in Figure 6, continuing with the previous melody, C, E, C. These resulting combined pairs actually turn out to be the final harmonies: for melody C, E, C, the final harmonies are ['I', 'I', 'I'] and ['IV', 'I', 'I'] (with one chord ordered in a row for each note in a row). If, for example, there were four melody notes, these combined pairs, of length three, may combine once more with pairs for the fourth melody note, to get a length of four and become the final harmony(ies). Some of the code for this is below:

```
for count in range(len(harmony)): # harmony is an array that holds all of the pairs of chords.

  # We will be looking backwards into the harmony array to match previous pairs to future pairs.
  # So skip the first case for count to align with the position in the harmony array
  if count > 0:
    for i in range(len(harmony[count-1])):
        for k in range(len(harmony[count])):
            …

            if harmony[count-1][i][len(harmony[count-1][i])-1] == harmony[count][k][0]:

               # Replace the previous pair with the combination of two pairs:
               #harmony[count-1][i] and harmony[count][k][1:]. "[1:]" is used for skipping
               #over the duplicate chord.
               replace.append(harmony[count-1][i] + harmony[count][k][1:])

    harmony[count] = replace
…
```

The above code combines the pairs of chords together into one array which will become a harmony if it terminates.

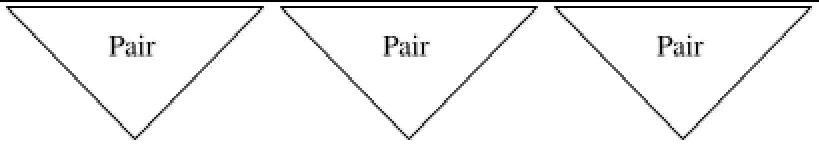
After the above method, there is still a step to further process the harmonies: to delete harmonizations that have been made by the simple pairing system that do not pass the more complex rules based on seeing melody notes in advance, like rule 2. For example, given the arbitrary harmony [II,IV,V,I], a third (floored) of the length of the melody to produce this harmony is 4//3 = 1. Because there is no 'I' at position 1, rule 2 has been violated. To get rid of these incorrect harmonies, the program marks them by appending “&” symbols (an arbitrary choice of symbol) to the end of them. The final output of the program will not pass lists that contain “&” symbols, so the user will only see valid harmonies. So this harmony would look like [II,IV,V,I,&] (and it means that it is not valid).

At this stage, a problem occurred: the program on an input of around 10 melody notes was sometimes taking ~ 10 seconds to output only 5 harmonies. Due to my approach of building the harmonies, I was dealing with pairs of chords and those pairs began to take up large amounts of memory for user inputs of around 10 notes. Half of the information per pair was not even used in the algorithm: it is checking if one pair ends on the same chord that another pair starts on and then throwing away one of those matching chords just to add the other chord in that pair to the first pair: for example [I,IV] and [IV,I] would result in the new list, [I,IV,I], and the other "IV" is never used. This could be an option for optimization.

Melodies can often be 20 notes long, and I would like to support at least that many. But when running my program on an input of 20 notes, it ended up needing to use about 20 gigabytes of memory (which was saved to storage) over one hour, which turned out to be quite large for 20 melody notes (because later I was able to harmonize over 20 melody notes using less than 20 gbs). I terminated the program due to time constraints. I reasoned that because my program required 20 gbs of space over time and my old laptop only has 8 gbs to work with, my computer was probably taking a lot of time getting rid of memory that wasn't being used so that it could have more memory to work with; if cleared many times in one program, it takes a noticeable

For melody C, D, E, C:

Notes:	C	D	E	C



Pairs:	['I', 'II'], ['I', 'V'], ['I6', 'II'], ['I6', 'V'], ['I64', 'V'], ['IV', 'II'], ['IV', 'V'], ['VI', 'II'], ['VI', 'V']	['V', 'I'], ['V7', 'I'], ['VII', 'I'], ['VII7', 'I']	['I', 'I']

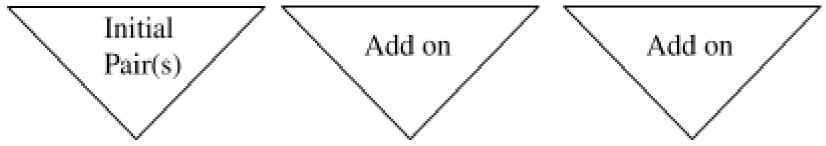
Figure 7. Failed Harmony Example.

amount of time in (about 1 second) to clear filled up memory, and it was the case that a bunch of the pairs still existed in memory even though the harmonies had already used them up and moved on to later melody notes to analyze.

Going back to the reason for the search for optimality, one way to cut the time to output 5 harmonies for a 10 note melody would be to change the method of combining all of the pairs together. What I realized was happening was that the combination algorithm was processing not only valid and working harmonies, but what I will call "failed harmonies" as well: combinations of chords that cannot be continued because they are restricted by the rules. The combination method looks at every single harmonization, even if it is a failed one, and then determines whether to show it to the user. Note that I only want to show the user valid, full harmonizations, not failed ones. A failed harmonization can be seen in Figure 7. The turquoise highlighted pairs in the first column, if combined with the highlighter pair in the second column would turn out to be failed harmonies because they violate rule 2 which states that if the harmony does not start on a I chord (with no inversion), a I chord must happen within the first third of the harmony (floored), which essentially means that there must be a I chord as the 1st chord of the harmony (because length of melody = 4 notes, and 4 // 3 = 1). So, if the turquoise pairs develop into a harmony, it will clearly not start with a 'I' chord, and they cannot change their first chords, 'I64', 'IV', and 'VI' to a 'I'; they will sit in memory and be developed throughout the whole harmonization process but will be invalid harmonies in the end. So, to free up some memory and help speed up the program, the failed harmonies should be removed from memory as soon as they are detected, so as to not waste time considering them for the combination process. This optimization could be more easily implemented using another approach to harmonizing a melody.

Although the pairing approach to my program seemed to me at first like a more understandable or "beautiful" approach to harmonizing, I decided to instead use just one array per harmonization and attach new chords onto it based on the order of melody notes. This would make it easier to clear any failed harmonies as the program ran because they would just be sitting together in one convenient place. This approach can be seen in Figure 8. Figure 8 shows the chord progression with the final harmony (to melody C, D, E, C) as ['I', 'V', 'I', 'I'], and instead of relying on pairs of chords in memory to be paired themselves (or combined) into harmonies, the harmonies build onto themselves. This could save some running time (in terms of the pairing method) because the chords are being combined together as they are being built instead of after the fact; so instead of having the method to create the pairs (Figure 5) and then another method

Melody Notes:	C	D	E	C



Straight to Harmony Building:	['I', 'V'] (Notice that all chords except for ['I', 'V'] have been preemptively removed due to rule 2!)	['I', 'V', 'I'], (Given note E, by rule 11 it can only add on a 'I' chord to the previous pair because it only exists within a 'I' chord out of rule 11s options)	['I', 'V', 'I', 'I'] (Given note C, it is the last note (and exists within a 'I' chord) so it adds on a 'I' to finish the harmony)

Figure 8. Harmonizing by using less memory.

to combine the pairs (Figure 6), there is a method that combines both ideas (Figure 8); it adds chords, based on the rules, to a growing harmony array instead of having to deal with merging chords (which takes more memory) together, if the chords match each other. The newer method also incorporates manual clearing of memory using the python garbage collector, gc() method. Using the new harmonization method, the program was able to stably harmonize from 2 up to around 30 notes (as opposed to only 2 up to 20 with the old pair method) in less than 2 hours, and without running out of memory.

Below is a table filled with runtimes of the unoptimized program along with the optimized program given an average case melody (ranging from 2 to 18 notes), based on the tune Frère Jacques, C, D, E, C, C, D, E, C, etc…, and a bad case melody, a melody filled with C's, C, C, C, C, …, because the note C has of the highest amounts of available chords it can be harmonized by as well as the highest number of rules that apply to the chords it can harmonize. (n values for 3, 5, 6, 7, and 9 were left out because they all basically had the same runtimes as for n values of 2, 4 and 8).

Number of Melody Notes	Unoptimized Program Average Input Case Time (sec)	Unoptimized, Bad Case Input Time (sec)	Optimized Program, Average Input (s)	Optimized, Bad Case (s)
n = 2	0.018	0.018	0.020	0.018
n = 4	0.017	0.018	0.021 Surprisingly, for small inputs, the unoptimized harmony seems to perform a bit quicker than the optimized!	0.022
n = 8	0.024	0.036	0.024	0.028
n = 10	0.036	0.252	0.027	0.057
n = 11	0.090	0.806	0.036	0.110
n = 12	0.209	2.748	0.052	0.278
n = 13	0.301	9.382	0.063	0.648
n = 14	0.971	31.917	0.104	1.518
n = 15	2.082	106.465	0.217	4.098

n = 16	5.764	405.241	0.506	10.093
n = 17	9.072	1403.843 But it actually took more like 4080 seconds for my computer to give an output. This might be due to clearing memory…	0.572	25.115
n = 18	27.406	Did not terminate. Killed 9 Error. Ran out of memory.	1.119	62.968 Where the unoptimized bad case failed completely, this optimized case takes only a minute!

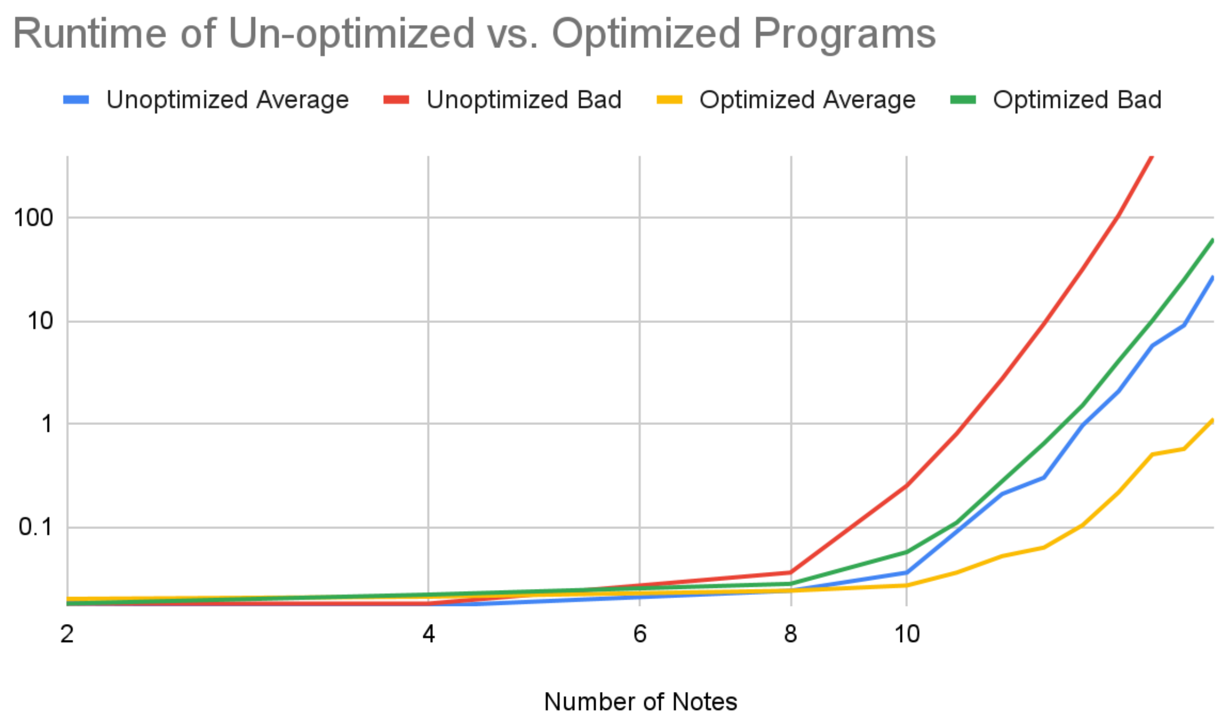


Figure 9. Log / Log Comparison of Runtimes Between the Un-optimized and Optimized Programs.

A graph for the above runtimes can be seen in Figure 9, where one can clearly see in the graph how the bad case for the unoptimized program, the red line, rises above the rest of the cases, however, for Numbers of Notes less than 6, the unoptimize program is actually faster than the optimized program. However, after Number of Notes more than 6, The unoptimized program's lines rise faster than those of the optimized program. The proximity of the average case for the unoptimized program to the bad case of the optimized program shows how much faster the optimized program is (compared to the unoptimized one). The optimized program's runtimes continue on into the 30s of melody notes, but this runtime data table shows that the optimized program (for $\sim n > 6$) is faster than the unoptimized one.

Results:

There are methods with which one can approximate the amount of memory space that a program will need to run without actually running the program. These methods come from the *Algorithms* textbook (Sedgewick, R., Wayne, K., 2011). They are useful to my program because, as can be seen in the timing results of n = 17 and n = 18 for the unoptimized program's bad case, my computer was sometimes taking extra time to clear its memory multiple times during a run of my program, which does not have so much to do with the efficiency of my program, but rather the memory constraints of my computer. Finding this memory data will help prove how much better the optimized algorithm is compared to the unoptimized one regardless of computer hardware.

The programs primarily use lots of strings and arrays, strings which represent the chords, and arrays that hold combinations and progressions of chords together. But also, the program tends to store my array of chords per note (each set) inside of another array (which just holds all of these sets together). They write that "When array entries are objects, a similar accounting leads to a total of [for an M-by-N array] $8NM + 32M + 24 \sim 8NM$ bytes for the array of arrays filled with references to objects, plus the memory for the objects themselves" (Sedgewick, R., Wayne, K., 2011).

To calculate how much memory my programs will need in order to execute. Both the unoptimized and optimized programs first keep track of all possible chords that can harmonize each note. This is done by keeping an array of chords for each note. Each chord is a string, and all of these arrays of chords are grouped using one large array. So in total we have an array of arrays filled with strings. Using what is shown in Figure 10, the memory used for an array of arrays of objects (strings), it will take 24 + 8M + M(24 + (size of strings)N) bytes where M is the length of the melody and N is the amount of chords per note (Sedgewick, R., Wayne, K., 2011). We are left with a total of 24 + 8M + M(24 + (68.6666)N) ~ 69NM bytes. But more memory is actually being used: the memory that comes from appending elements to all of the arrays, because each append to an array creates a whole new array: the contents of the old array plus the appended element. The array that holds each set of chords was also appended on to, with each chord set, so we must add on this memory to the total that the chord set process takes. This can be calculated as

$$\sum_{i=1}^{M-1} (24 + 8i + i(24 + (68.6666)N))$$

which can be simplified to

$$\sum_{i=1}^{M-1} (24) + 8\sum_{i=1}^{M-1} (i) + (24 + 68.6666N)\sum_{i=1}^{M-1} (i)$$

$$(M - 1)(24) + 8(M - 1)(M) + (24 + 68.6666N)(M - 1)(M)$$

$$24M - 24 + 8M^2 - 8M + (24 + 68.6666N)M^2 - M$$

Whis is, in final

$$(68.6666N)M^2 + 38M^2 + 15M - 24 \quad \sim 69NM^2$$

The sums go to *M*-1 (because the memory of the final array of the sets, at *M*, has already been found in the above paragraph). The sum is similar to that of the final array of sets, except that instead of having the final state, M, *i* increments for every time a new set is appended, for each new melody note.

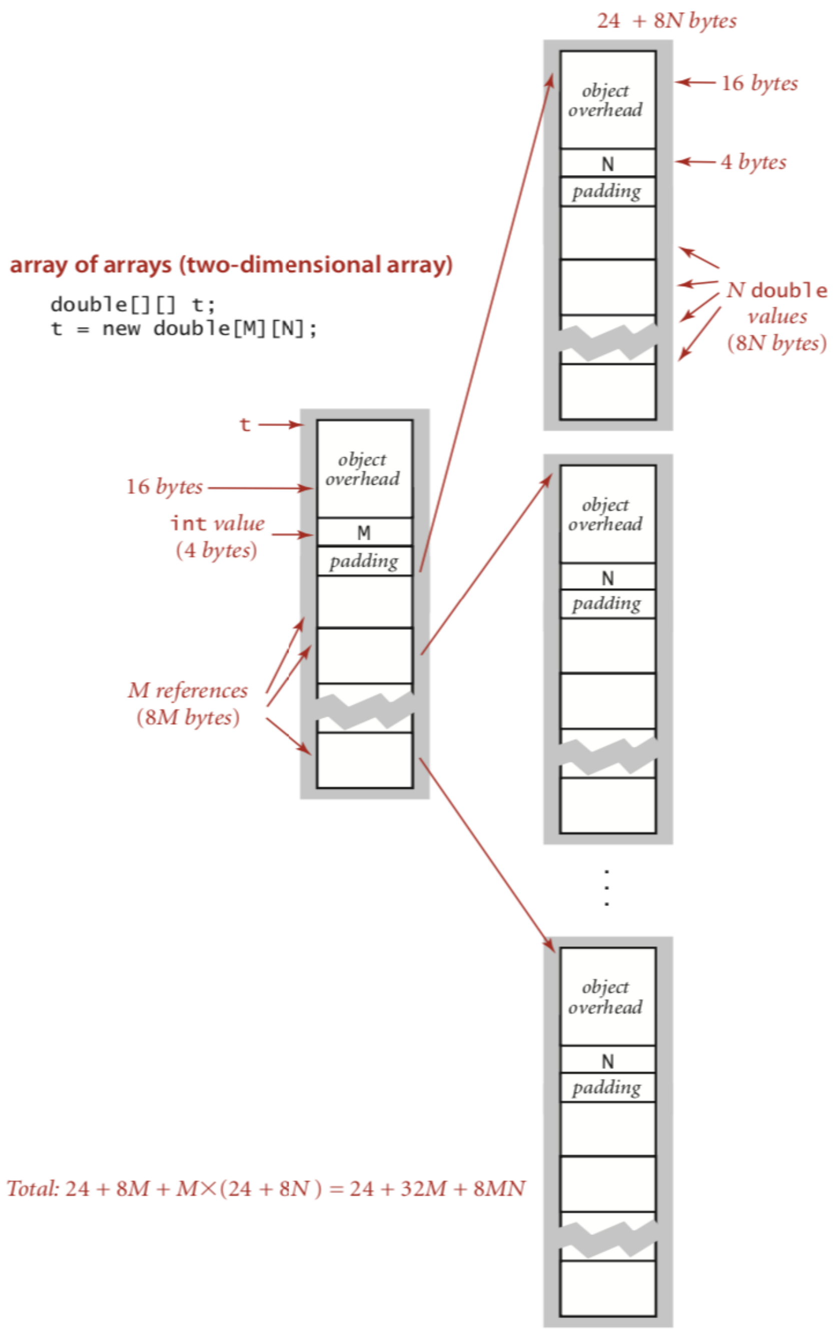


Figure 10 (above) (Sedgwick & Wayne, 203). From the *Algorithms* textbook, a figure depicting their explanation of how much memory a two dimensional M by N array filled with double values (of 8*N* bytes each) will take.

In total, the amount of memory the sets will take is the size of the final chord sets, 24 + 8M + M(24 + (68.6666)N) bytes, added to the leftover arrays in memory, for a total of

$(68.6666N)M^2 + 38M^2 + 15M - 24$ bytes =

$[8M + M(24 + (68.6666)N)] + [(68.6666N)M^2 + 38M^2 + 15M] \sim 69NM^2$ bytes

The above equation can be labeled as *Equation 1.*

Equation 1 finds the total amount of memory taken by the chord sets, where M is the size of the melody input and N is the amount of chords per note. Given the worst case for the amount of chords per note, N = 6 (with notes D or F), an example of a 14 note melody will take

$(8(14) + (14)(24 + 68.6666(6))) + ((68.6666(6))(14)^2 + 38(14)^2 + 15(14))$

bytes, which is

$6215.9944 + 88409.9216$ bytes

for a total of

94625.916 bytes = 94.6259 kilobytes used in the building of the chord sets which is unchanged in both unoptimized and optimized programs.

The Unoptimized Program:

The next part of this (unoptimized) program is the creation of the pairs. All of these pairs eventually end up in one master array that holds an array of pairs for each two melody notes: An array of arrays filled with arrays of length 2 (a pair) of chords. We know, from before, that an array of arrays of objects (pairs) takes 24 + 8M + M(24 + (Y)N) ~8NM bytes where M is the length of the melody minus 1 (due to pairs being for every two notes), N is the number of pairs per array, and X is the amount of memory a pair takes. To find Y is to see how much memory an array of two chords, an array of size 2 with two strings inside, takes. This equation was used previously in the methods section and, to reiterate, is 64 + *XN* ~*N* bytes where *X* is the amount of memory that a chord (string) takes (on average), and *N* is the number of chords in the array. Now given that pairs are arrays of size 2 with two strings inside, each will take 64 + (2.3333)(2) bytes = 64 + (4.6666) bytes = 68.6666 bytes. But every pair of chords is appended together, meaning that there is an extra array of length 1 just with the first chord of the pair, sitting in memory. This array (of length 1) will take 64 + (2.3333)(1) bytes = 66.3333 bytes of memory. So in total, a pair (plus its extra array to create it) takes A + (68.6666) bytes where A is the memory of the final pair. This means that the total for one pair, our Y, is (68.6666) + 66.3333 bytes = 134.9999 bytes = Y. Putting Y into our equation for the building of the pairs, we have 24 + 8M + M(24 + (134.9999)N) bytes.

The next step is to find N, the number of pairs for each two melody notes that are next to each other. For any melody length, if the melody is valid, which it should be, then on an average melody input, C,D,E,C,C, there are a total of 36 pairs over all 5 notes. But because there is only a pair for every two notes, we divide 36 by 4, which would mean an average of 9 pairs per two melody notes. 9 pairs = $9 \cdot 134.9999$ bytes = 1214.9991 bytes = N. Putting this as into our equation for building the pairs, 24 + 8M + M(24 + (134.9999)(1214.9991)) bytes = 24 + 8M + M(164048.757) bytes = 24 + 164056.757M bytes.

However, there still remains the extra memory used due to all of the appending to the two-dimensional array that holds all of the pairs. This can be written similarly to before:

$$\sum_{i=1}^{M-1} (24 + 164056.757i) \text{ bytes}$$

$$(\sum_{i=1}^{M-1} 24 + 164056.757 \sum_{i=1}^{M-1} i) \text{ bytes}$$

$[\,(M - 1)(24) + 164056.757(M - 1)(M)\,]$ bytes

$[\,24M - 24 + 164056.757M^2 - 164056.757M\,]$ bytes

$[\,164056.757M^2 - 164032.757M - 24\,]$ bytes

So, *Equation 2*, for the total amount of memory used for the creation of pairs is:

$[24 + 164056.757M] + [\,164056.757M^2 - 164032.757M - 24\,] \sim 164057M^2$ bytes

The final part of the unoptimized program is the combining of pairs into complete harmonies which are held by a master array (of the length of the melody) of arrays that are the same length as the melody, and filled with chords. This part of the program, however, is very difficult to compute by hand because it involves arrays that increase (and stay the same length) by factors of the harmonization rules, for example, for the input of an average melody, F, G, E, C, the combination algorithm produces something like:

```
1. [[['II', 'V'], ['II6', 'V'], ['IV', 'I'], ['IV', 'I64'], ['IV', 'V'], ['V7', 'I'], ['V7', 'V7'], ['VII', 'I'], ['VII7', 'I']],
2. [['II', 'V', 'I'], ['II6', 'V', 'I'], ['IV', 'I', 'I'], ['IV', 'V', 'I'], ['V7', 'I', 'I'], ['V7', 'V7', 'I'], ['VII', 'I', 'I'], ['VII7', 'I', 'I'],
3. ['II', 'V', 'I', 'I'], ['II6', 'V', 'I', 'I'], ['IV', 'I', 'I', 'I'], ['IV', 'V', 'I', 'I'], ['V7', 'I', 'I', 'I'], ['V7', 'V7', 'I', 'I'], ['VII', 'I', 'I', 'I'],
['VII7', 'I', 'I', 'I']],
4. [['II', 'V', 'I', 'I','&'], ['II6', 'V', 'I', 'I','&'], ['IV', 'I', 'I', 'I'], ['IV', 'V', 'I', 'I','&'], ['V7', 'I', 'I', 'I'], ['V7', 'V7', 'I', 'I'.'&'],
['VII', 'I', 'I', 'I'], ['VII7', 'I', 'I', 'I']]]
```

in four stages: 1. being just the pairs, 2. the combinations of the pairs from part 1. with other pairs, 3. is a further combination of the arrays from part 2. with other pairs, and finally 4. holds all of the final harmonizations, marking failed ones with a '&' symbol so they will not be shown to the user. Having these 33 arrays, 9 of length 2, 8 of length 3, 12 of length 4, and 4 of length 5 for this average 4 note melody looks like there is no linear pattern. Also, to know that another average melody of 4 notes: C,D,E,C, produces 9 arrays of size 2, 5 of size 3, 5 of size 4, and 4 of size 5 gives not much correlation to the previous melody, making calculating by hand the total memory produced too difficult. Therefore comparing both algorithms, unoptimized and optimized, by final output could be a way to see which is more efficient.

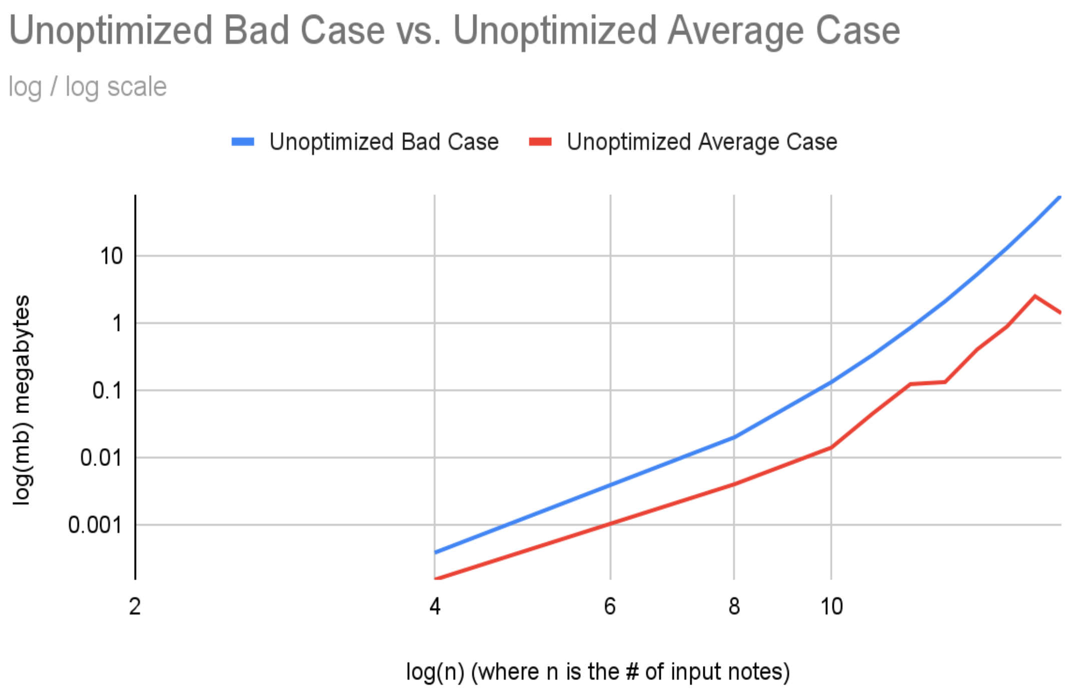


Figure 11. Log / log scale of Unoptimized Bad Case and. Unoptimized Avrg. Case. Note that values at n = 2 were too small, so are essentially equal to 0.

It can be seen in Figure 11 that of the bad case or average case functions (the "Bad Case" and "Average Case" lines), neither are linear, so regression on a semi-log graph will have to be employed in the Analysis section, in order to find an equation for the program's output. The equation found could also predict memory consumption for inputs of over 100 notes, something not able to be run on my computer as of now.

The Optimized Program:

The optimized program does share *Equation 1's*

$[8M + M(24 + (68.6666)N)] + [(68.6666N)M^2 + 38M^2 + 15M] \sim 69NM^2$ bytes

of memory needed to keep all the initial sets of chords that can be used per melody note as the unoptimized program does. So we know that one will need at least that much memory space in order to run the optimized program. But where both programs differ, optimized and unoptimized, is in the building of the harmonies. However, similarly to with the unoptimized program, calculating the amount of memory that it uses in building the harmonies will depend on nonlinear factors that are the harmonizing rules, which is not an easy task. So following the idea of comparing final output results with the unoptimized program.

Analysis

Using the data from both optimized and unoptimized programs, we will be able to approximately predict how much time and memory both programs will take for larger melody inputs, say 100 notes, which is larger than I could compute within a reasonable time with my hardware; this analysis will also show more clearly the relationship in efficiency between both programs. Starting with the runtimes of the bad case for the unoptimized program, Figure 9, one can see that the unoptimized bad case's line (the topmost, red one) has some sort of power or exponential trend. In order to create an equation for these points, as to predict the outcome of a 100 note input, we can plot its same points onto a semi-log (base 10) graph and find its trendline:

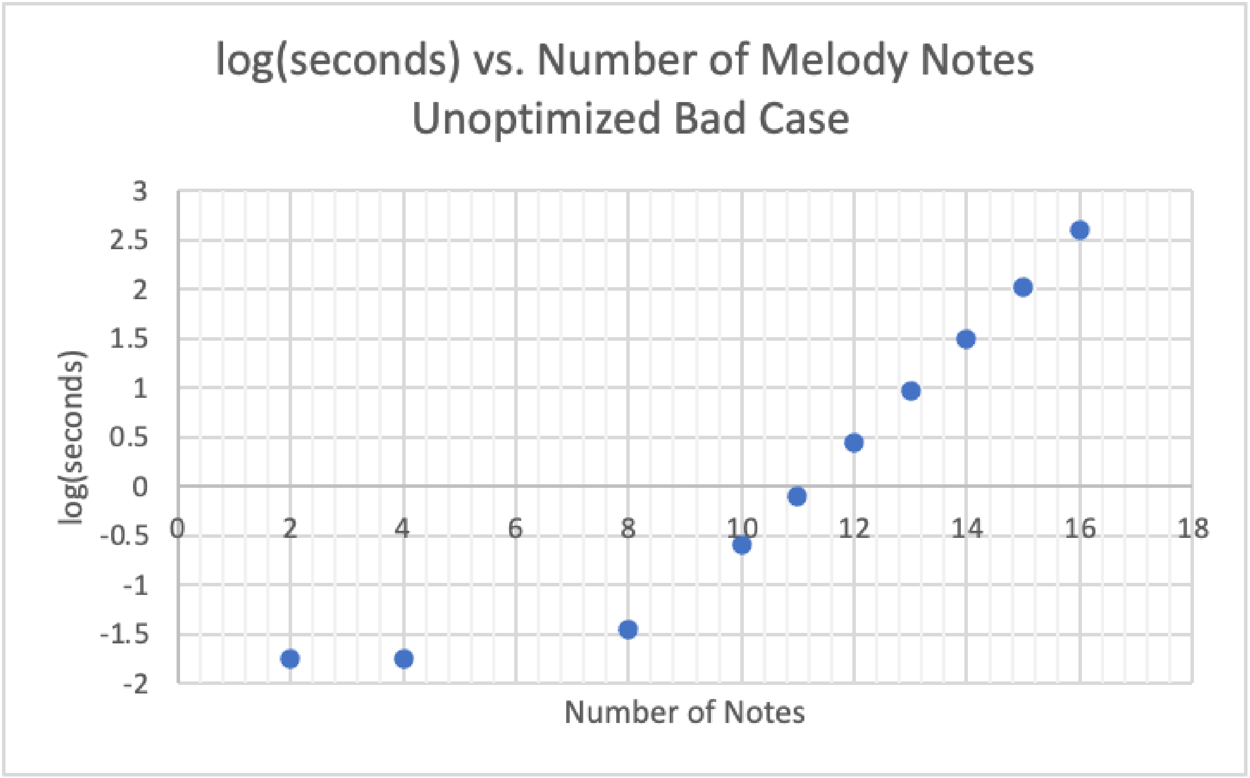


Figure 12. A semi-log graph of the seconds to the number of notes (up to 16) for the unoptimized bad case's runtime.

It can be seen that the semi-log graph in Figure 12 shows a much more linear relationship to the number of notes input, than the log-log graph in Figure 9, and so it allows for a linear trendline to fit the data more closely. However, the first two points, at 2 and 4, do not align with the rest of the points, and because we are only interested in output values for inputs of larger than 16 notes, we can remove these first two points from the graph and fit a trendline to the rest of the points, as can be seen in Figure 13.

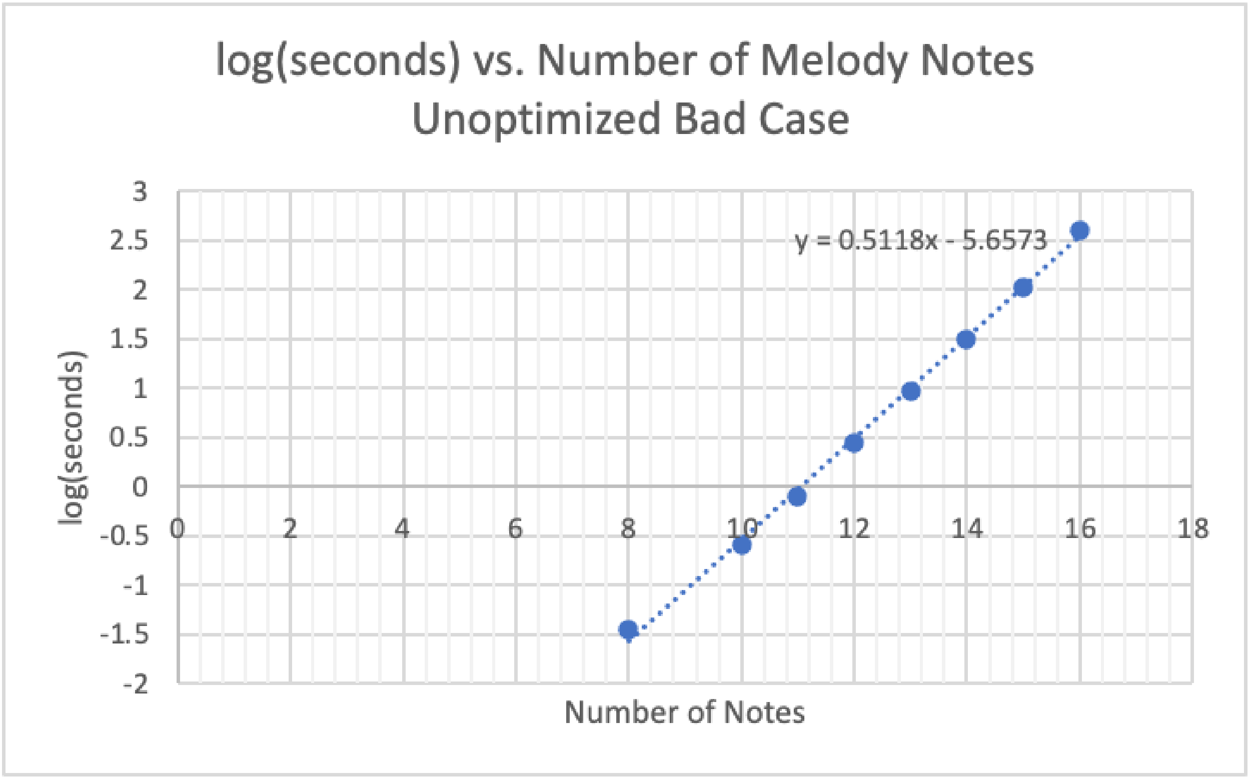


Figure 13. Trendline of y = 0.5118x - 5.6573.

The equation of the trendline is y = 0.5118x - 5.6573, however, this trendline's equation must be converted using the Exponential formula for semi-log graphs, seeing the trendline in the form y = mx + b, T(N) = ki^N where i = 10^m, k = 10^b, and N is the number of notes. So we will have a final formula of $T(N) = 10^{-5.6573}(10^{0.5118})^{N}$ seconds. For example, for an input of 17 notes, the unoptimized algorithm's bad case would take 1104.8416 seconds = 18 minutes and 24 seconds to finish. An input of 100 notes would take this program 3.332E+45 seconds to finish, which is an extremely large amount of time, centuries… so there would be not much point in waiting for a result given that input, given that the computer will have fully eroded by then.

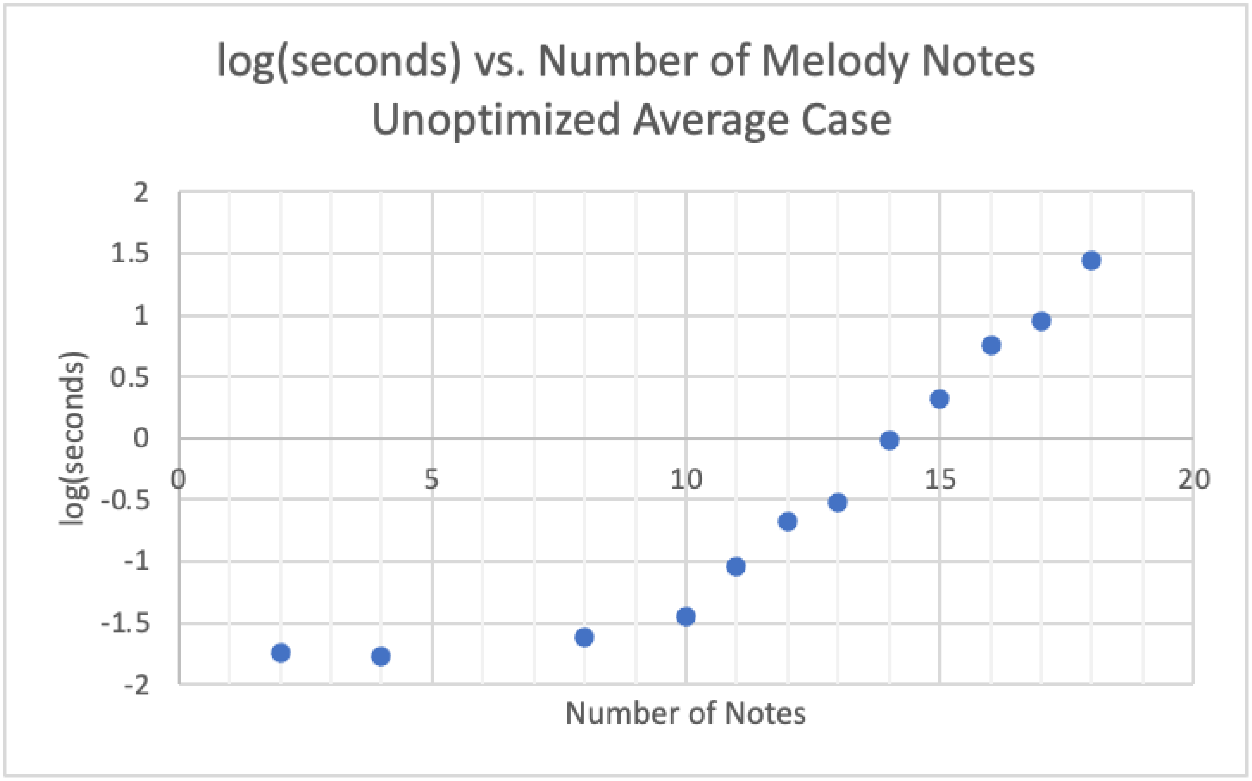


Figure 14. A semi-log graph of the log(seconds) to the number of notes (up to 18) for the unoptimized average case's runtime.

The unoptimized program's average input case can be seen in Figure 14. Although the points in Figure 14 are not as aligned as they could be, their trend was no worse than in a log-log scale. This time, the first three points at 2, 4, and 8 seem to disrupt the trend for numbers of notes past 10, so these first three points will be removed, as can be seen in Figure 15.

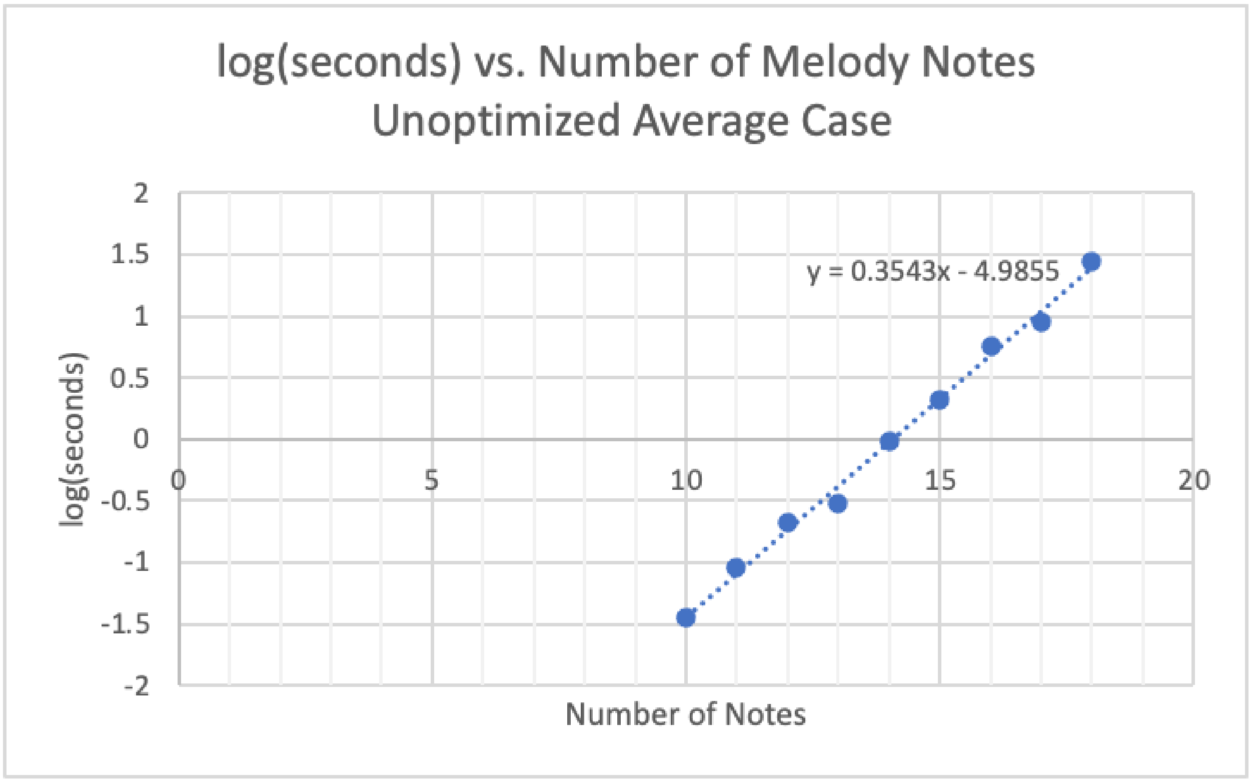


Figure 15. Trendline of y = 0.3543x - 4.9855.

Converting the trendline from Figure 15 into an Exponential form, we get T(N) = $10^{-4.9855}(10^{0.3543})^{N}$ seconds, where N is the number of notes input into the program. So given 100 notes, the unoptimized program will take 2.783E+30 seconds to finish, which is a large amount of time but is still much faster than the unoptimized program's bad case runtime of 3.332E+45 seconds.

Concerning the optimized program's runtimes, using the same method employed above for the unoptimized program, the runtime equation for the **optimized** program's bad case turned out to be T(N) = $10^{-5.1785}(10^{0.3862})^{N}$ seconds, and for an input of 100 notes would take 2.764E+33 seconds, which is a large amount of time, but is actually not too far off from the runtime of the unoptimized program's average case. But the difference between the unoptimized and optimized bad cases is that of E+12, which shows that the optimized program is faster than the unoptimized one for their bad cases. The T(N) for the optimized program's average case is $T(N) = 10^{-3.785}\left(10^{0.2098}\right)^{N}$ seconds, which for 100 notes is 1.567E+17 seconds, which is still a long time, but is less than the time for the unoptimized average case, further sealing the superiority in runtime of the optimized program.

Predicting how much memory the programs will create can be done in much the same way as calculating the runtime. The **unoptimized** program's bad case final memory output up to 17 notes can be seen in Figure 16.

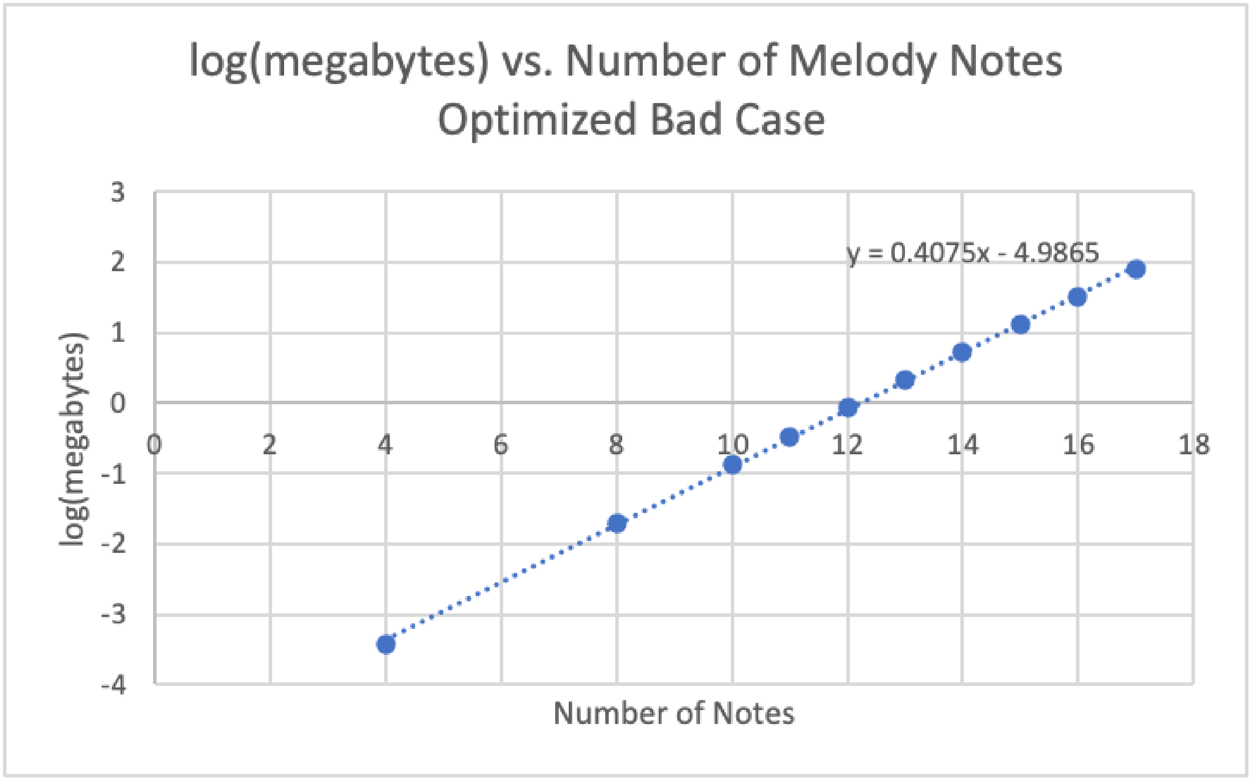


Figure 16. Final output memory of unoptimized bad case.

Given our calculations of the amount of memory used in the creation of all of the chord sets, *Equation 1,* for an 100 note melody, given the worst case for the amount of chords per note, N = 6 (with notes D or F), it can be said that it would take at least

$[8(100) \ + \ (100)(24 \ + \ (68.6666)(6))]$

$+ \ [(68.6666(6))(100)^2 \ + \ 38(100)^2 + \ 15(100) \ ]$ bytes

= [44399.96] + [4501496] bytes

= 4545895.96 bytes = 4.5459mb of memory on the computer.

The predicted memory output for 100 notes, by Figure 16, would be, for function T(N) = $10^{-4.9865}(10^{0.4075})^N$ megabytes, is 5.801E+35 mb which is 5.801E+29 Terabytes, which is a lot of space.

The amount of memory for the unoptimized average case will at least require the same 4.5459mb of memory for the chord sets, and the trendline for the expected memory output can be seen in Figure 17.

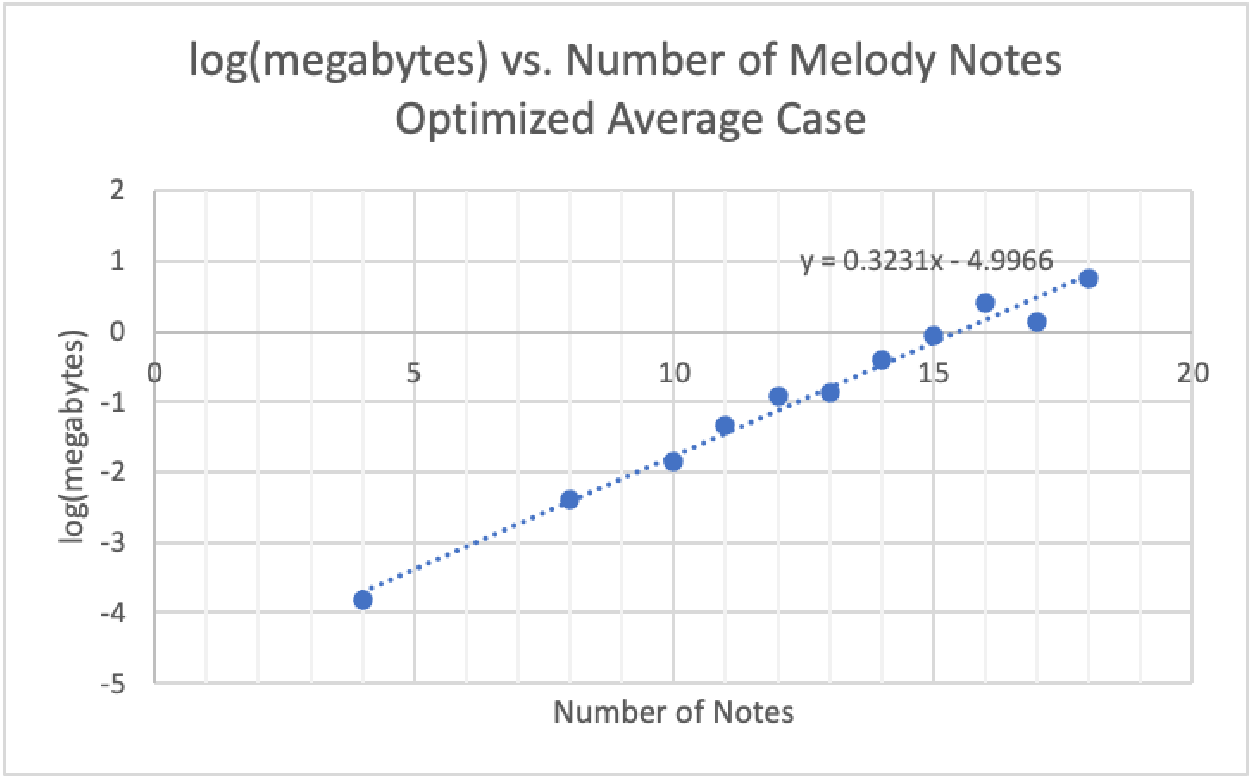


Figure 17. Unoptimized Average Case.

The equation for the amount of memory predicted to be output by the unoptimized average case is T(N) = $10^{-4.9966}(10^{0.3231})^{N}$ megabytes. For 100 notes it yields 2.058E+27 mb, which is also many Terabytes of memory. This shows just how many possibilities there are to harmonize 100 notes of melody even restricted to classical harmony in just C major.

With 100 notes, the equation for the optimized bad case memory is T(N) = $10^{-5.5853}(10^{0.4373})^{100}$ mb = 1.395E+38 mb, which is surprisingly more than the unoptimized bad case of 5.801E+35 mb. This is certainly unexpected considering that given a table of their memory outputs, all entries of the optimized program are lower than those of the unoptimized program:

# Notes	Unoptimized Bad Case (mb)	Optimized Bad Case (mb)
4	0.000381	0.0001
8	0.02	0.009
10	0.132	0.075
11	0.335	0.191
12	0.848	0.57
13	2.1	1.4
14	5.3	3.6
15	13.1	9.8
16	32.3	24.2
17	79.5	59.8
18	crashed	154.7

Figure 18.

The fact that in the long run the optimized program is predicted to use more memory may be due to a too small amount of data gathered about the programs (although the unoptimized program physically would not make it past 17 notes). There could possibly be a discrepancy in the definitions of the harmonization rules in both programs, although they seemed to produce the same outputs of final harmonies. This would require a thorough revision of both program's code to see if they indeed matched. The expected memory output for the optimized average case is $T(N) = 10^{-5.4924}(10^{0.3373})^{N}$ mb, which for 100 notes would be 1.728E+28 mb, which is also more than the unoptimized's average case output of 2.058E+27 mb. This could be true for the same reasons the unoptimized beat the optimized bad case memory. So it is that for lower inputs, the "optimized" program performs better than the "unoptimized" one, but for larger values the inverse is true.

Conclusion

The harmonizing program has the ability to intake a C-major melody defined by the user, and output all figured bass progressions–limited to C major and simple harmonization rules–in the "classical" style. The program underwent phases of optimization to lower its overall memory usage, although it turns out that for an 100 note input, even though being faster, the "optimized program" would use more memory than the "unoptimized" program. Ways to further optimize the program would be the method of proper resizing of arrays. Conventionally, arrays should not have one item (like a chord) at a time be appended to them but instead should be doubled in size whenever they run out of space for new items; every time that my program appended a new chord onto an array, or a new array of chords onto an array, the program would create entirely new arrays, one length longer than the old array, to hold this data, and the previously used arrays (which would now be one length shorter than the current array) would sit in memory and never be used. But if my program used the conventional resizing of arrays, when doubled in size, they would have empty space that could be filled by new chords or arrays. It is true that doubling the arrays would also create entirely new arrays and so the old ones would get forgotten in memory, however the rate at which the arrays would double in size would be much less than that of an array that is being appended onto one element at a time, and this would be because at a certain point, doubling a large array will create one twice as large, with half of it empty space so it would take as long to fill as the array before it did. All in all, this would lower the amount of memory used by the program.

Seeing the results of the analysis section, one can see how the optimized and unoptimized programs differed in efficiency. However, it would be interesting to know not just how much output the programs make, but the total amount of memory they create in their combination steps. This could be done if there was a method employed to measure how many pairs of chords are made in the combination algorithm of the unoptimized program. This method could be directly counting how many pairs are made by adding some code to the combination algorithm. Knowing the total amount of memory used, provided the length of the user's melody, is useful to know in regards to deciding what kind of machine the user would need to most efficiently run the code on.

Another future addition to the program, one that could make it better represent the range of classical music that exists in the world, could be adding key changes, the ability to input a melody in, say, A Major, or B Harmonic Minor instead of only C major. This would be quite a simple addition to the code: adding a variable into each chord definition (I chord = [C, E, G]). As of now, the program reads "'I' chord = [C, E, G]" as "I = [0, 5, 7]" where the numbers represent the notes: C = 0, E = 5, and G = 7, and it so happens that all of the chords hold a relationship to their key, meaning that if the key is G major, a 'I' chord is [G, B, D], which is [5, 11, 2] for the computer, where G = 5, B = 11, and D = 2. So if we say that the default key of the program is C major, then we could say 'I' chord = [(0 + X)%12, (5 + X)%12, (7 + X)%12] where X = 0 for "C" in "C major" and we perform %12 because of the one octave our notes are restricted to. Then [(0 + (0))%12, (5 + (0))%12, (7 + (0))%12] = [0, 5, 7], which is in fact a 'I' chord in C major. Now, given G major, we use the same method, but X = 5 because of the " G " in "G major". This would yield 'I' chord = [(0 + (5))%12, (5 + (5))%12, (7 + (5))%12] = [5, 11, 2], which is a 'I' chord in G major. On a different note, given, say, the key of B Harmonic Minor, some of the harmonization rules would have to change; the current rules only support major keys, like C or A major. So given the key change to B Harmonic Minor, the program would need its own new version of the harmony combination algorithm but for harmonic minor keys.

Another step further with the program would be to incorporate modulation, a technique where the key of the harmony could change in the middle of a melody. This would add even more options to harmonize a melody, which unfortunately could lower the amount of melody input supported in the program, but if added, could make the harmonies more interesting. It could be done by analyzing the outputted final harmonizations of the current program and then apply new rules concerning modulation, which determine whether the harmonizations can incorporate a key change. The harmonizations that could incorporate one would then be duplicated, and the duplicates would be passed through the appropriate combination algorithm for the key it was changed to (as explained in the above paragraph), so that the chords applied are in the new, modulated key.

The addition of some more complex harmonization rules involving V9 Major and Minor chords are also a part of classical music that was not included in this program. If these chords are used in the harmony, they have a rule which requires that a 'I' chord be used within the next two melody notes, meaning that the program would have to look ahead two melody notes in order to detect if a 'I' chord can even harmonize them. If this 'I' chord could be used, then the V9 chord would be added onto the harmony array but only be able to have chords that lead to a 'I' be added onto its array for the next note. If the next note added would be a 'I', then all is resolved, however if it would not be a 'I', then another condition would apply that the next chord must be a 'I' chord if it is to be added onto that harmony array.

I do believe that this project could also be done more beautifully in SWI Prolog given its rule-like nature. The difficult part about that, for me, would be how to very tightly constrain chord options. One would need to create many prolog rules that could finely manipulate a growing list of chords.

This python project has highlighted some ties between music and computer science. The language of music, although it may seem emotional and spontaneous, is surprisingly ordered and rule-bound. For example, by asserting when the piece is in C major, or that a V chord is [G, B, D] in C major and only in C major. The output of my program can also show some mathematical patterns regarding musical harmonies. Given the input C, C, C, C, C, C, C, C, some of the output looks like

```
I  I6  IV   I  I6  IV   I    I
I  I6  IV   I  I6  IV   IV   I
I  I6  IV   I  I6  I6   IV   I
I  I6  IV   I  IV   I   IV   I
I  I6  IV   I  IV   I   I    I
I  I6  IV   I  IV  IV   I    I
…
…
```

It is true that this output is ordered by an algorithm so it can inherently seem to have a pattern, but it is also true that any of these chord progressions can be applied to the input melody and work in the classical style.

I believed, when I started this project, that I would end up with a program that could intake a melody of any length and in any key and produce all possible classical harmonies, maybe even including modulation. But due to the deeply complex nature of music that I did not account for, as of now I only have a program that can harmonize only C major melodies, excluding some harder harmonization rules. In order to further this program efficiently, I would need to come up with a more realistic view of how much time and effort each part of the program will need in order to be built. If I want to incorporate more difficult harmony rules that require looking at melody notes far in advance of the current state of the melody, such as with the V9 chords, I may need to change my combination algorithm in such a way that it is not based on checking for adjacent relationships of chords. This change could allow for all sorts of possibilities, ones also regarding the rhythm of the melody to be a factor in how it should be harmonized. All of these advancements could more efficiently generate figured bass to melodies using a computer, which was the goal of this project.

I started this project with the idea to keep track of all possible harmonies for one user given melody. Looking into some computer science literature on the subject helped generate information to help me decide what methods I would implement into my program, such as having strict harmony rules and a clear notation style. I created methods in Python that produced valid harmonies, and two different algorithms emerged, the unoptimized and optimized. Analyzing these algorithms was an interesting experience because although the optimized program was created to use less memory than the unoptimized program, and it was always predicted to be faster than the unoptimized one, it turned out that for longer and longer melodies, the unoptimized program used less memory than the optimized program.

Works Cited

Benward & Saker (2003). Music: In Theory and Practice, Vol. I, p. 159. Seventh Edition. ISBN 978-0-07-294262-0.

Dunning, T. (1993). Accurate methods for the statistics of surprise and coincidence. *Computational Linguistics*, 19(1):61–74.

Hiller, Lejaren, and Leonard M. Isaacson. *Experimental Music*. McGraw-Hill, 1959.

Fux, Johann Joseph, and Jean-Philippe Navarre. *Gradus Ad Parnassum (1725)*. Mardaga, 2000.

Levinson, I. (2021, September 20). Personal interview [Personal interview].

Mc Intyre, R. A. (1994). Bach in a box: The evolution of four part baroque harmony using the genetic algorithm. *IEEE Conference on Evolutionary Computation - Proceedings*, *2/-*, 852–857. https://doi.org/10.1109/ICEC.1994.349943

Sedgewick, R., Wayne, K. (2011). *Algorithms, 4th Edition.*. Addison-Wesley. ISBN: 978-0-321-57351-3

Yang, Yezhou & Teo, Ching & III, Hal & Aloimonos, Yiannis. (2011). Corpus-Guided Sentence Generation of Natural Images. EMNLP 2011 - Conference on Empirical Methods in Natural Language Processing, Proceedings of the Conference. 444-454.

Yi, L., BMA, J. G.-, & 2007, undefined. (2007). Automatic Generation of Four-part Harmony. *Researchgate.Net*.

https://www.researchgate.net/profile/Judy-Goldsmith/publication/221404321_Automatic_Generation_of_Four-part_Harmony/links/0912f513825c1f284c000000/Automatic-Generation-of-Four-part-Harmony.pdf