Autonomous Music Synthesis

A Thesis
Presented to the

Department of Computer Science
College of Information Science & Technology

and the

Faculty of the Graduate College
University of Nebraska

In Partial Fulfillment
of the Requirements for the Degree

Master of Science

University of Nebraska at Omaha

by

Andrew J. McAuliffe

April, 2021

Supervisory Committee:
Victor Winter, Ph.D.
Mahadevan Subramaniam, Ph.D
Professor Jeremy Baguyos

Autonomous Music Synthesis

Andrew J. McAuliffe

University of Nebraska Omaha, 2021

Advisor: Victor Winter, Ph.D.

Abstract

Computers are commonly used to perform complex calculations, solve problems, store and retrieve information, and assist humans with creative tasks, such as writing, drawing, and human-directed music composition. This project will explore the automated synthesis of music. Rather than simply assisting with the mechanics of placing notes on a staff, this project will create a computer program that chooses both the notes and where they are placed. It will employ artificial intelligence techniques to model human creativity by synthesizing original works of music based upon specified constraints. The program will randomly generate song components and employ an adjudication algorithm that ranks them and selects the "best" options, most consistent with the constraints and maximizing their subjective utility.

To Amy

Acknowledgments

I wish to thank my thesis equivalent project supervisory committee. Thank you, Victor Winter, Ph.D. for your time, ideas, guidance, and motivation. Thank you very much, Mahadevan Subramaniam, Ph.D. and Professor Jeremy Baguyos for kindly investing your time and expertise toward making this project a success.

I am very grateful to the Computer Science professors of the University of Nebraska Omaha for their time, expertise, dedication, and willingness and ability to teach.

Thank you to the Peter Kiewit Foundation for underwriting the majority of my undergraduate education.

Thank you to Union Pacific Railroad, my employer, for underwriting the majority of my graduate education.

Thank you to my parents, Ray and Ruthie, for all the love, trust, emotional, and financial support!

Thank you to my daughter, Megan, for turning me on to the Kostka book!

Thank you to my son, Joseph, for all the late-night “what if?” conversations!

Most of all, thank you to my wife, Amy, for all the love, support, patience, gentle critique, and encouragement throughout this endeavor!

Table of Contents

1. INTRODUCTION 1

1. 1. PROJECT SCOPE 5

1. 2. SIZE OF THE SEARCH SPACE 5

2. PREVIOUS WORK 10

2. 1. MUSICAL DICE GAME 10

2. 2. PATTERN EXTRACTION 11

2. 3. FRACTAL MUSIC 11

2. 4. MUSICAL SEQUENCE COMPARISON 12

2. 5. EMMY 12

2. 6. ARTIFICIAL NEURAL NETWORKS 13

2. 7. DATA STRUCTURE CONSIDERATIONS 13

2. 8. CHAOTIC OPERATOR 14

2. 9. RULES + MUSICAL MEMORY DATABASE 15

2. 10. LOOSE VS EXACT CONSTRAINTS 16

2. 11. PITCH-CLASS GRAPHS 17

2. 12. THE ILLIAC SUITE 17

2. 13. TEXT TO MELODIES 17

2. 14. ARS MAGNA 18

2. 15. K-GRAMS 19

2. 16. MARKOV MODELS 20

3. IMPLEMENTATION 25

3. 1. AMS FILE FORMAT 25

3. 2. MIDI FILE FORMAT 27

3. 3. SCALES 29

3. 4. CHORD PROGRESSIONS 31

3. 5. DISCRETE LIKELIHOOD WEIGHTING 32

3. 6. GENERATING A RANDOM LINE OF MUSIC 38

3. 7. SIGNATURES 41

3. 8. ADJUDICATION 43

3. 9. HILL CLIMBING 45

3. 10. SIMULATED ANNEALING 46

3. 11. ISSUES ENCOUNTERED 47

4. RESULTS 50

4. 1. HILL CLIMBING 50

4. 2. SIMULATED ANNEALING 51

5. IMPROVEMENT / EXTENSION OPPORTUNITIES 55

6. CONCLUSION 56

7. REFERENCES 57

8. APPENDIX A 60

8. 1. ORIGINAL AMS FILE FORMAT 60

Table of Figures

Figure 1 - AMS High Level Architecture 2

Figure 2 - Gamow’s Mythical Printing Press (Gamow, p. 12) 3

Figure 3 - Mythical Printing Press Output 4

Figure 4 - Whole Note Decomposition to Sixteenth Notes 9

Figure 5 - Ars Magna Diagram A (Nierhaus, p. 18) 18

Figure 6 - One Kind of Note Duration 22

Figure 7 - Two Kinds of Note Duration 23

Figure 8 - Three Kinds of Note Duration 23

Figure 9 - A Line of Music Example 25

Figure 10 - .ams File Format Example 26

Figure 11 - .ams File Format 26

Figure 12 - Format of Keys, Chords, and Notes 27

Figure 13 - Scale Format: Key to Bit Assignments 29

Figure 14 - Scale Format Example 30

Figure 15 - Scales 30

Figure 16 - Chord Progressions 31

Figure 17 - DLW Interval Offset Structure 33

Figure 18 - Interval Offset Distribution Example 34

Figure 19 - Distribution Buckets 34

Figure 20 - Distribution Buckets Converted to Half Steps 35

Figure 21 - Downward Drift Example 36

Figure 22 - Downward Drift Solution Example 37

Figure 23 - Discrete Likelihood Weighting for Absolute Notes Example 38

Figure 24 - ams Program Command Line Options... 39

Figure 25 - Discrete Likelihood Weighting for Note Durations Example....................... 40

Figure 26 - Note Classifications 42

Figure 27 - Hill Climbing Score Progression 50

Figure 28 - A Random Line.. 51

Figure 29 - After Hill Climbing.. 51

Figure 30 - Simulated Annealing Score Progression.. 52

Figure 31 - A Random Line.. 52

Figure 32 - After Simulated Annealing 53

Figure 33 - Simulated Annealing Score Progression.. 53

Figure 34 - Simulated Annealing Score Component Progression 54

Figure 35 - Original.ams - AMS File Format Example.. 61

Figure 36 - Original 62

Table of Tables

Table 1 - Unique Melody Counts for Whole Note Decompositions 9

Table 2 - Chord Progressions.. 32

Table 3 - A Line of Music.. 48

Table 4 - Major Note Offset Classifications 49

1. Introduction

The product of the research described in this paper is a computer program named "AMS," which stands for "Autonomous Music Synthesis." The AMS program produces randomly generated lines of "monophonic" (one note played at a time) music and improves the random lines, subject to specified constraints, using two artificial intelligence algorithms: hill climbing and simulated annealing.

Regarding the word "improves" in the previous paragraph, a question asked of this researcher repeatedly throughout this process was: "What is good music to you is not necessarily good music to me, so how can your program decide what is 'good'"? A fair question. Indeed, music is art, and there are as many opinions as to what constitutes "good" music as there are opinion holders. Although the constraints chosen for this implementation are generally consistent with accepted norms for traditional Western music (a.k.a. "Tonal Harmony" (Kostka, p. ix)), it would be possible and equally valid to substitute a set of constraints that describe any other genre of music, for instance any of the various African, Asian, Latin American, etc. genres, including those which seek to break specifiable rules.

For this project, the choice of the rules to follow when "adjudicating" (assigning a numeric score to a line of music where a high number is "good", meaning sought-after, and a low number is "bad") a melodic line is arbitrary and modifiable. The primary objective is to implement algorithms that use rules effectively to transform a line of music that is inconsistent with a set of constraints to one that is consistent with them while preserving to the greatest extent possible the randomness of the line of music originally generated.

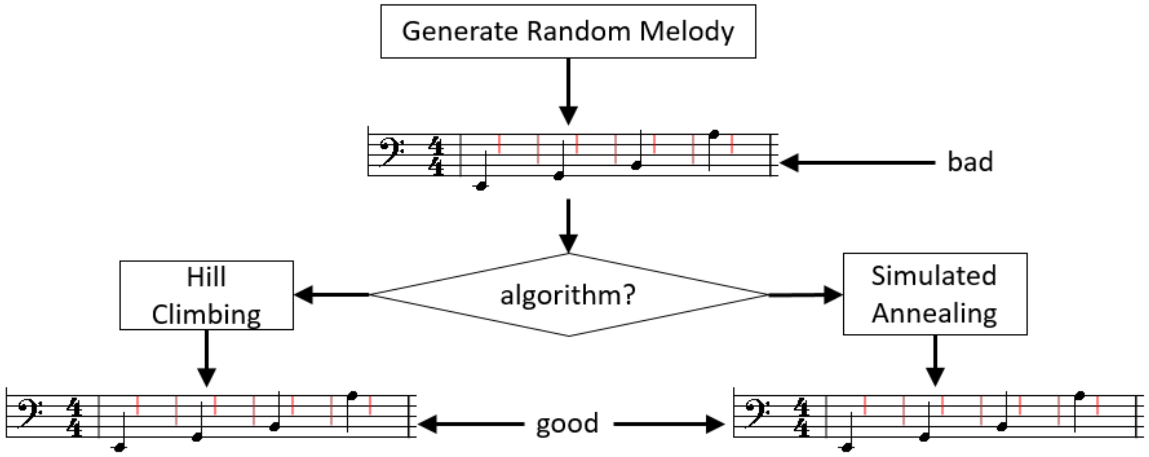


Figure 1 - AMS High Level Architecture

To that end, AMS will generate lines of music and store them in a text file, one line of music per line of text, using a proprietary format. The program also creates MIDI files corresponding to the generated lines of music. The MIDI files include chords that go along with the melody and may be loaded into MIDI editing software, such as Anvil Studio (Anvil), which was used to create the sheet music contained in this document.

In addition to musical notes, each line of music has a corresponding score -- a number that quantifies how “good” the line sounds. The program succeeds if lines of music with high scores “sound better,” than lines with low scores, according to criteria discussed earlier.

Building on the theme of subjectivity in art, the purpose of AMS is not to discover the “best melody.” Indeed, several different pieces of music may be thought of by an individual to be "best in class." The purpose of AMS is to find many instances of music that are "best in class" even if their numerical scores differ slightly.

In the book *One Two Three… Infinity*, George Gamow envisioned a printing press consisting of 65 coaxial wheels, each of which embosses at its perimeter the 50 letters, digits, and punctuation symbols of the English language (Gamow, pp. 11-14).

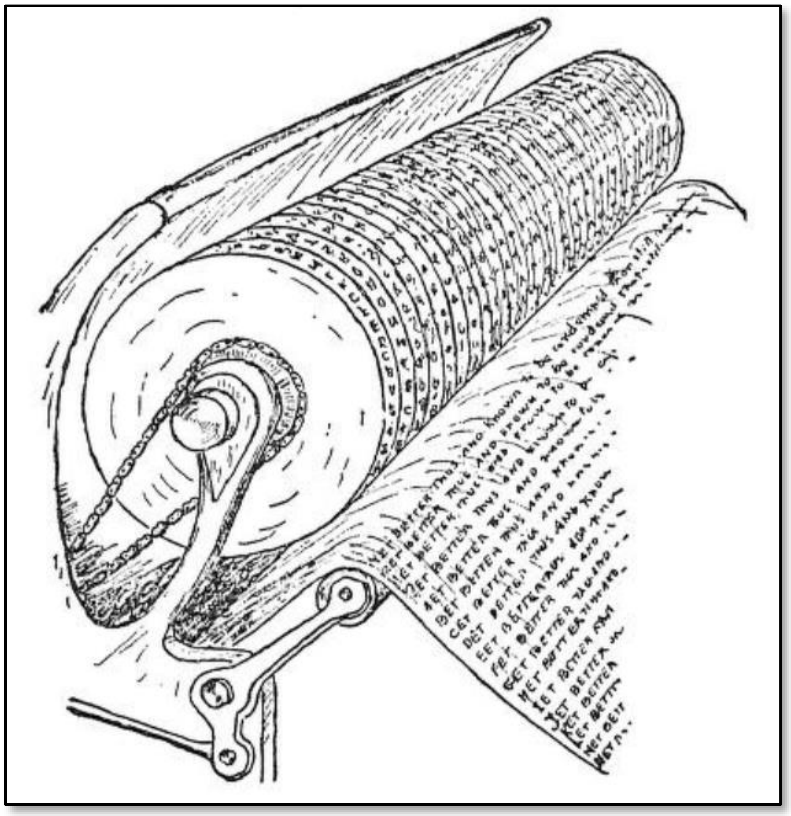


Figure 2 - Gamow's Mythical Printing Press (Gamow, p. 12)

The wheels are geared as an automobile's odometer, such that a complete revolution of a wheel results in the rotation by one position of its neighboring wheel to the left. The printing press works by printing a line of text, advancing the rightmost wheel to the next position, and printing the next line of text.

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaab
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaac
...
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa}
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaba
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaabb
...
}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}]
}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}{
}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}}
```

Figure 3 - Mythical Printing Press Output

Upon reaching the end of its printing run, this mythical printing press will have printed every possible line of text. This consists mostly of random garbage, but also lines of perfectly formed words, including every line from every book that has ever been written. Also, it will include every line from every book that will ever be written. (Gamow)

Besides the implausibility of actually generating this unreasonably large number (10^{110}) of lines of text, the idea is transformative. It changes the classification of authorship from a process of creation to one of discovery. In mathematics, the calculus is said to have been “discovered,” not “invented.” Likewise, every line of text that has yet to be written will be discovered, and the writers are less creators and more finders.

This project takes Gamow’s printing press idea and applies it to the domain of music composition. Instead of attempting to visit every possible line of music, the program generates random lines of music, subject to a few basic constraints, and discovers “good” melodies that have never before been composed.

The goal of this project is threefold. The first goal is to calculate a number that is a score for a line of music, such that the higher the score, the better the line of music sounds. Second, this score will be used as an objective function to be maximized by two artificial intelligence algorithms: hill climbing and simulated annealing. Finally, the program will produce lines of music, and melodies with high scores should be found and should sound better than those with low scores.

An alternative and complimentary presentation of this project is available in video format here: https://youtu.be/VHUL0PNXi_0.

1. 1. Project Scope

The AMS program shall produce "monophonic" (one note played at a time) melodies. The user may specify parameters that govern aspects of the music, such as the key, the number of measures and the time signature. The largest duration note to be used is the whole note, the smallest duration note to be used is the 128th note, and dotted notes are supported. To simplify algorithm experiments in this thesis, note durations are limited to eighth, quarter, and half notes, the key is restricted to C major, and all melodies generated are in $\frac{4}{4}$ time.

1. 2. Size of the Search Space

The search space consists of every combination of notes that comprise a monophonic melody for the specified number of measures. To compute the size of this space, it is necessary to define the number of possible note pitches and determine the distinct decompositions of a whole note in $\frac{4}{4}$ time (4 beats per measure, quarter note gets one beat).

This calculation shall be limited to one measure in $\frac{4}{4}$ time. Also, this calculation shall include the following note durations: whole (W), dotted half (h), half (H), dotted quarter (q), quarter (Q), dotted eighth (e), eighth (E), and sixteenth (S). The number of unique melodies per whole note decomposition = $(88^n)\left(\frac{n!}{\prod(|t_i|!)}\right)$. In this expression, 88 represents every key on a piano and *n* is the number of note durations in a decomposition. This calculation does not include rests but could do so by substituting 89 for 88, where the rest is considered the 89th key.

The first factor, (88^n), is the number of combinations of notes on the piano for a given decomposition. The second factor, $\left(\frac{n!}{\prod(|t_i|!)}\right)$, is the number of permutations of the decomposition note durations. The numerator counts all permutations. The denominator "un-counts" duplicate permutations where two or more of the same note duration switch places, but doing so does not result in multiple unique permutations. This is also known as "multiset permutation". In the denominator, t_i is a note duration, $|t_i|$ is the number of notes in the decomposition with duration t_i, and $\prod(|t_i|!)$ is the product of the factorials of the number of notes of each note duration in the decomposition. Table 1 below lists counts of the numbers of unique melodies for all 101 whole note decompositions, given the constraints indicated above.

Decomposition	**# Melodies**	**Total # Melodies**
W	88	88
HH	7744	7832
Qh	15488	23320
EEh	2.04E+06	2067740
HQQ	2.04E+06	4112160

Decomposition	# Melodies	Total # Melodies
EHq	4.09E+06	8200990
Seh	4.09E+06	12289820
QQQQ	6.00E+07	72259320
EEqq	3.60E+08	432076320
EEHQ	7.20E+08	1151710320
EQQq	7.20E+08	1871344320
ESSh	7.20E+08	2590978320
HSSq	7.20E+08	3310612320
HQSe	1.44E+09	4749882320
EEEEH	2.64E+10	31136482320
SSSSh	2.64E+10	57523082320
EEQQQ	5.28E+10	1.10296E+11
EEEQq	1.06E+11	2.15842E+11
QQQSe	1.06E+11	3.21388E+11
ESSqq	1.58E+11	4.79708E+11
HSSee	1.58E+11	6.38028E+11
QQSSq	1.58E+11	7.96348E+11
EEHSe	3.17E+11	1.11299E+12
EHQSS	3.17E+11	1.42963E+12
EQSeq	6.33E+11	2.0629E+12
EEEEEq	2.79E+12	4.84932E+12
EEEEQQ	6.97E+12	1.18154E+13
SSSSqq	6.97E+12	1.87814E+13
HQSSSS	1.39E+13	3.27135E+13
EEEHSS	2.79E+13	6.05777E+13
EQQQSS	2.79E+13	8.84419E+13
QQSSee	4.18E+13	1.30238E+14
EEESeq	5.57E+13	1.85967E+14
EHSSSe	5.57E+13	2.41695E+14
QSSSeq	5.57E+13	2.97424E+14
EEQQSe	8.36E+13	3.81017E+14
EEQSSq	8.36E+13	4.64609E+14
ESSeeq	8.36E+13	5.48202E+14
EEEEEEQ	2.86E+14	8.34275E+14
QQQSSSS	1.43E+15	2.26463E+15
HSSSSSe	1.72E+15	3.98107E+15
EEEEEEEE	3.60E+15	7.57742E+15
EEEESSq	4.29E+15	1.18685E+16
EEHSSSS	4.29E+15	1.61596E+16
SSSSeeq	4.29E+15	2.04507E+16
QSSSeee	5.72E+15	2.61722E+16
EEEEQSe	8.58E+15	3.47543E+16
EEEQQSS	8.58E+15	4.33365E+16
EQSSSSq	8.58E+15	5.19187E+16

Decomposition	# Melodies	Total # Melodies
EESSSeq	1.72E+16	6.90831E+16
EQQSSSe	1.72E+16	8.62475E+16
EEQSSee	2.57E+16	1.11994E+17
EEEEEESe	2.01E+17	3.13389E+17
EHSSSSSS	2.01E+17	5.14784E+17
QSSSSSSq	2.01E+17	7.16179E+17
SSSSeeee	2.52E+17	9.67923E+17
EEEEEQSS	6.04E+17	1.57211E+18
QQSSSSSe	6.04E+17	2.1763E+18
EEESSSSq	1.01E+18	3.18328E+18
ESSSSSeq	1.21E+18	4.39165E+18
EEEESSee	1.51E+18	5.90212E+18
EEQQSSSS	1.51E+18	7.41259E+18
EESSSeee	2.01E+18	9.42654E+18
HSSSSSSSS	2.85E+18	1.22748E+19
EQSSSSee	3.02E+18	1.52958E+19
EEEQSSSe	4.03E+18	1.93237E+19
EEEEEEESS	1.14E+19	3.07169E+19
SSSSSSSeq	2.28E+19	5.35033E+19
EESSSSSSq	7.98E+19	1.33256E+20
EQQSSSSSS	7.98E+19	2.13008E+20
QSSSSSSee	7.98E+19	2.92761E+20
EEEEESSSe	1.60E+20	4.52266E+20
ESSSSSeee	1.60E+20	6.11771E+20
EEEEQSSSS	1.99E+20	8.11152E+20
EEESSSSee	3.99E+20	1.20992E+21
EEQSSSSSe	4.79E+20	1.68843E+21
QQSSSSSSSS	1.25E+21	2.94168E+21
ESSSSSSSSq	2.51E+21	5.44819E+21
SSSSSSSeee	3.34E+21	8.7902E+21
EEEEEEESSSS	5.85E+21	1.46387E+22
EQSSSSSSSe	2.01E+22	3.46908E+22
EEEQSSSSSS	2.34E+22	5.80849E+22
SSSSSSSSSSq	2.70E+22	8.50438E+22
EEEESSSSSe	3.51E+22	1.20135E+23
EESSSSSSee	3.51E+22	1.55226E+23
QSSSSSSSSSe	2.70E+23	4.24815E+23
EEEEESSSSSS	1.13E+24	1.55709E+24
EEQSSSSSSSS	1.21E+24	2.77024E+24
ESSSSSSSSee	1.21E+24	3.98339E+24
EEESSSSSSSe	3.24E+24	7.21846E+24
SSSSSSSSSSee	1.42E+25	2.14528E+25
EQSSSSSSSSSS	2.85E+25	4.99214E+25
EEEESSSSSSSS	1.07E+26	1.56678E+26

Decomposition	# Melodies	Total # Melodies
EESSSSSSSSSe	1.42E+26	2.99021E+26
QSSSSSSSSSSSS	2.47E+26	5.45749E+26
ESSSSSSSSSSSe	2.96E+27	3.50648E+27
EEESSSSSSSSSS	5.43E+27	8.93449E+27
SSSSSSSSSSSSSe	2.34E+28	3.23167E+28
EESSSSSSSSSSSS	1.52E+29	1.84301E+29
ESSSSSSSSSSSSSS	2.20E+30	2.38891E+30
SSSSSSSSSSSSSSSS	1.29E+31	1.53226E+31

Table 1 - Unique Melody Counts for Whole Note Decompositions

Therefore the total number of unique melodies one measure long in $\frac{4}{4}$ time comprised of whole through sixteenth notes, not including rests, played on an 88-key piano is 1.53226E+31 = 15.3 nonillion.

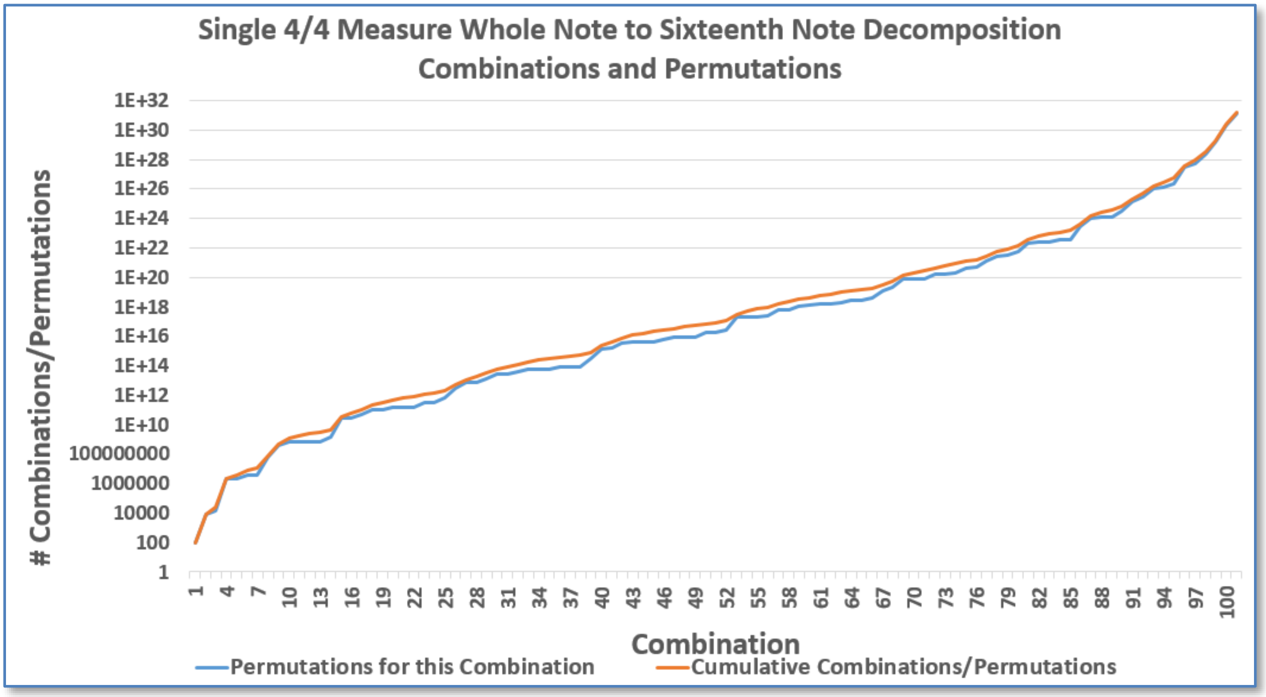


Figure 4 - Whole Note Decomposition to Sixteenth Notes

2. Previous Work

This section introduces some of the numerous efforts made throughout history to think of music composition as something other than a human being writing notes on a staff. Several of these efforts generate a line of music sequentially, one note at a time from the beginning of the line to the end. Most of these techniques preclude the opportunity to adjust earlier notes based on the pitch or duration of later notes the way AMS is able to do. For example, when two sequential notes have a pitch interval that is too large, this can be resolved by adjusting either the first or the second note of the interval. Sequential generation schemes can only adjust the pitch of the second note in the interval, because for those techniques the interval does not exist until the second note is placed. Also, not all of the techniques in this section incorporate an underlying sequence of chords to constrain the melody as AMS does.

The primary contribution of the AMS project is the use of two artificial intelligence algorithms, hill climbing and simulated annealing, to search for a better sounding line of music nearby a randomly generated line in the space of all possible lines of music.

2. 1. Musical Dice Game

The idea of randomly generating a piece of music is not new. In the 18th century, Wolfgang Amadeus Mozart produced a random musical composition by rolling a pair of dice to select notes from the scale. He used the randomly generated notes to construct a melody and its corresponding harmony (Bateman, p. 223). This style of music synthesis, Musikalishes Wurfelspiel, or musical dice game, was practiced by many composers of the time (Cope, p. 4).

2. 2. Pattern Extraction

There have been more serious efforts toward creating random works of music that focus on mimicking the style of human composers. Pierre-Yves Rolland and Jean-Gabriel Ganascia investigated "Pattern Extraction" (Miranda, Readings, p. 120) as a method of musical analysis. The goal of pattern extraction is to identify brief phrases of music that repeat during a piece of music, or even throughout the corpus of a composer's work. This repetition is thought to significantly contribute to the way humans digest music.

2. 3. Fractal Music

Music is made from a finite set of notes. Therefore, it is inevitable that a piece of music of nontrivial size will contain a measure of repetition. In *Fractals in Music*, author Charles Madden explores the use of "similarity transforms" (Madden, p. 21-26), or methods of modifying an interval, akin to ways that images can be modified. Scaling is the altering of the size of the interval. Translation is shifting the interval up or down the scale. Shear is the uneven application of scaling to subsequent intervals. Inversion is the vertical reversal of an interval. Reflection is the inversion of a series of intervals.

Given these interval transformation operations, a melody is constructed by first establishing a primary interval. This interval is the first interval in the melody and is called the "motive". Subsequent intervals are interpreted as transformations of the motive. Finally, the last note of the last interval resolves to the first note of the motive.

Describing a melody as a series of transformations of the first interval and then enumerating these transformations reveals the degree of self-similarity in the melody. Repetition is observed in both the types and values of the transformations.

The degree and nature of self-similarity in a melody can be used as factors that contribute to the evaluation function for the melody. The overall goal is to establish a tight correlation between the objective evaluation function and the subjective quality of a melody.

2. 4. Musical Sequence Comparison

“Musical Sequence Comparison” (Miranda, Readings, p. 128) is a more general approach to determining the degree of similarity between two phrases. This technique emphasizes the identification of minor changes (add, delete, or change individual notes) that can be applied to one musical phrase to transform it into another. These techniques break down individual components of a musical phrase to reveal an isolated aspect of the phrase. The end product of this breakdown is a vector of numbers that describe the aspect over the phrase. These techniques were used to study the improvisational style of Charlie Parker.

2. 5. Emmy

One of the more comprehensive attempts at randomly imitating the style of human composers was undertaken by David Cope. Cope’s program Experiments in

Musical Intelligence (or EMI, a.k.a. “Emmy”) uses signatures, earmarks, and unifications to identify and quantify the style of a composer (Cope, ch. 5) and then applies recombination, variation, and texture modification to synthesize new pieces of music based on the identified styles (Cope, ch. 4).

2. 6. Artificial Neural Networks

An interesting technique to automated synthesis is to train input and hidden layers of artificial neurons based on an input training set consisting of labeled lines of music, where the labels indicate that the line is either “good” or “bad.” The trained neural network is then applied to a labeled test set of music. If the inferences made by the network do not match the labels of the lines in the test set, the system repeats the training with an adjusted set of neuron values. This process is repeated until either a set of neuron values correctly classifies all of the samples in the test set or until improvement progress no longer occurs. (Leman pp. 285-286) The resulting neural network will be able to identify an input line of music as “good” or “bad” based on its training.

2. 7. Data Structure Considerations

The data structure choices for a computer program dealing with music are driven by the goals of the project. For example, Donncha Maidin was interested in analyzing musical scores (Maidin, pp. 67-93). The scope of analysis is the entire score – that is everything that appears on the printed page of the score. Given the high level of abstraction required, Maidin selected a linked-list data structure, where elements of the linked list are abstract objects of different types that can represent any element on the printed page. Maidin observed that the chosen data structure can represent polyphonic music by utilizing a line identification characteristic that makes unique two simultaneously played notes. However, for purposes of analysis of music and comparing lines of music, Maidin restricts consideration to monophonic melodies.

2. 8. Chaotic Operator

One approach to autonomous composition involves using a chaotic mathematical operator and mapping its output to musical notes. The use of a chaotic operator lends a degree of both randomness and structure to a stream of numbers that are then converted into musical notes. John Fitch created software to generate a series of musical note values and durations based on the Henon map (Boulanger, p. 656).

Although using a single operator produced a reasonable sequence of notes, the result was too repetitive to be interesting. To increase the variability of the note stream, Fitch used a second chaotic operator, the standard torus map, in conjunction with the Henon map.

The result of using two chaotic operators was better than that of using one. However, Fitch’s program relies on a priori knowledge regarding details of the musical piece, such as which instruments participate. This is consistent with the practice of blending rules and randomness exhibited by most practitioners of autonomous composition.

2. 9. Rules + Musical Memory Database

The essence of composition is decision making – deciding which notes get used and where. Improvisational human performers make these decisions in real-time. As a result, they cannot go back and change past notes as a consequence of the quality of the music in hindsight. They cannot un-ring the bell, as it were. Therefore, any learning resulting from past decisions can only affect future decisions (Smith, pp. 145-148).

According to Geber Ramalho and Jean-Gabriel Ganascia, musical memory comprises the total of a composer's life experiences in music. It is the foundational contributor to the intuition that guides the tasks of composition. Musical memory spans the gamut of creation and consumption, work and play, and occupational and recreational interactions with music.

One way of thinking about the composition process is in terms of two distinct musical contexts. In the first context, rules are used to select notes for the piece being created. In the second context, musicians create music in such a way that defies codification in rules. The process of synthesis is relying not on rules but upon the musical memory acquired by the musician over a lifetime of experience.

Given this framework for approaching the problem of automated synthesis, Ramalho and Ganascia propose a two-pronged approach to building a music synthesis machine. One part is comprised of a collection of rules to satisfy the rule context of composition. The other part is a database of musical memory.

To build the database, the system they propose must start, not as a highly-skilled musician, but rather as a moderately skilled musician, knowledgeable of basic rules and techniques. The system would then, through trial and error, acquire its own experience and build a corpus of musical memory from which to draw when the musical context demands.

2. 10. Loose vs Exact Constraints

Unlike mathematical problems, creative undertakings live in a separate but similar domain. Johnson-Laird identified three key characteristics that pertain to creative tasks and distinguish them from problems traditionally solvable by computers: non-determinism (i.e. there is more than one “right” answer), absence of specific goals (goals themselves are a function of the creative process), and absence of agreement in adjudicating the outcome of a creative process (subjectivity). (Johnson-Laird, pp. 255-265)

Two common high-level approaches to computer composition are making random choices 1) based on loose, hierarchical, and statistical constraints and 2) based on well-defined exact constraints. Loose constraints are applied at various levels of the composition. Random choices guide decisions made at the melody, measure, and note levels, for example. Exact constraints enumerate specific well-formed sequences of notes selected to match the musical context. The problem with loose constraints is that they are not guaranteed to produce music of acceptable quality. The problem with exact constraints is that they do not permit the freedom to break rules to create new art. (Johnson-Laird, pp. 255-265)

2. 11. Pitch-Class Graphs

Jeffrey Johnson explored the use of graph theory in the construction of harmonic melody. In his method, Johnson created graphs with vertices that correspond to pitch-classes and directed edges that represent pitch-class transitions that satisfy the harmonic characteristics corresponding to the graph. (Johnson p. 4)

2. 12. The Illiac Suite

The "Illiac Suite" was the first work of music composed using computer algorithms (Nierhaus, p. 72). It was built by Lejaren Hiller and Leonard Isaacson and made use of Markov models to guide selection of components of the music. A performance of the four "experiments" of the "Illiac Suite" is available for listening online here: https://www.youtube.com/watch?v=n0njBFLQSk8 (DiNunzio).

2. 13. Text to Melodies

In around 1000 AD, Guido of Arezzo developed a system for converting text into melodies (Nierhaus, p. 21). In his system, letters, syllables, and sequences of words were mapped to musical note pitches and phrases of music. He even incorporated natural pauses within poetic verse as pauses in the resulting melody.

2. 14. Ars Magna

In 1305, Raimundus Lullus published the “Ars Magna,” a machine designed to form logical statements by systematically combining words (Nierhaus, pp. 17-21). The machine took the form of concentric rings, each containing words. The circle in the center contained lines that form associations among the words.

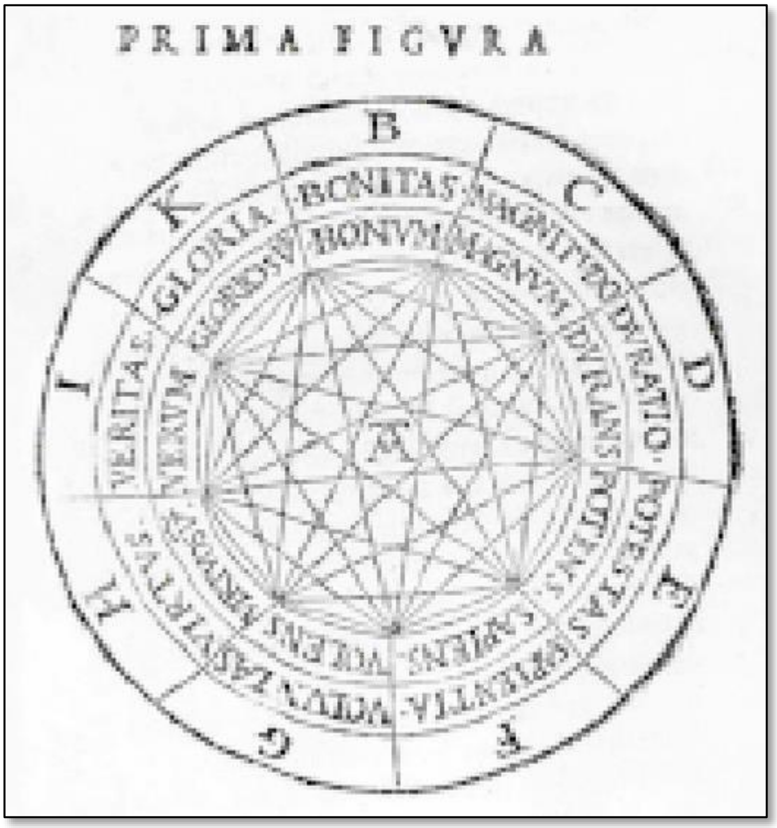


Figure 5 - Ars Magna Diagram A (Nierhaus, p. 18)

The fundamental idea behind the Ars Magna is that there is a place for both structure and randomness. If one were to produce a machine that produces random works of English literature, a reasonable expectation would be that the works produced would follow the accepted rules of grammar, at a minimum, and possibly paragraph structure, genre structure, etc. Therefore, creating a machine that simply generates random words would not be likely to yield a high number of works that follow these expected structures.

The level of constraint to place upon a system that randomly generates a piece of creative work must be considered carefully. Placing too few constraints on the randomly generated works will result in a large number of works that would not be considered “good” and that must be rejected. This will minimize the yield of the synthesis engine. However, if too many constraints are employed when generating random works, the yield, or percentage of “good” works will be higher, but the system runs the risk of failing to generate otherwise “good” works that fail to satisfy all of the imposed constraints. This shall be known as the “constraint problem.”

The solution to the constraint problem is to use as few non-quantifiable constraints as possible and to make quantifiable constraints dynamically configurable. Permitting artists the freedom to dynamically configure constraints will enable them to “dial in” the right level of constraints that produce sufficiently interesting works in a reasonable amount of time.

2. 15. k-grams

To quantify the goodness of a melody or monophonic line of music, the melody must be considered as a collection of fundamental units. The fundamental unit of a melody is arguably the note. However, considered individually, one note is as another. It is when multiple notes are strung together to form a series of intervals and melodies that two sequences of notes can be compared, and one deemed “better” sounding than the other. Therefore, the fundamental unit of a melody, for quantitative adjudication (assigning a numeric score to the melody), is the k-gram.

In the discipline of information storage and retrieval, a k-gram is an ordered set, of length k, of characters that represent a sub-sequence of a word (Manning, p. 50). For example, the word “music” includes the 3-grams “^mu”, “mus”, “usi”, “sic”, and “ic$”, where the symbol “^” represents the start of the word, and “$” represents the end of the word. Information storage and retrieval uses k-grams to facilitate wildcard searches in text-based search engines. For purposes of analyzing a melody, each substring of k notes shall be given a score indicating their sound quality. Although this will be subjective, as is the majority of this project, there will be sequences (harmonic) that are more pleasing to the ear than others (dissonant) as per the rules of tonal harmony.

One way of using k-grams to determine a numeric “goodness” score for a line of music is to take the average of all of the k-gram scores and use that as one numeric measure of the quality of a melody. Another way is to weight the value of each k-gram according to the number of times the sequence of k notes appears in the melody (i.e. repetition). A second-degree method of using k-grams is to assign scores for the presence or absence of combinations of two distinct k-grams. Also, k-grams can be comprised of raw notes, note families (e.g. “C”, “C#”, “D”, etc.), direction changes, intervals, interval plus direction changes, scale degrees, etc.

2. 16.Markov Models

One paradigm for structuring constraints for random music synthesis is to determine a set of optional values from which to choose. The next step is to assign to each value a probability with which the value will be selected. In this paradigm, the complete set of value options is the non-quantifiable constraint. That is, it is not possible to choose a value outside the set. The probabilities associated with each value in the set represent the quantifiable constraints. That is, the probabilities can be changed, and each unique set of probabilities will result in a machine that is likely to produce pieces of random music that are similar to each other.

non-quantifiable constraint a constraint that is either satisfied or not

quantifiable constraint a constraint that may be satisfied by one of a set of values

For example, if a model only allows notes corresponding to keys on a piano, that is a non-quantifiable constraint. If notes within two octaves of middle-C are made to be twice as likely to be selected as other notes, that is a quantifiable constraint (because the frequency of note preferences can be adjusted to a variety of values). In addition to randomly selecting components of music, such as note families (e.g. “C”), durations (e.g. quarter note), and intervals (e.g. three half steps), the probability distribution values themselves can also be randomly generated.

This paradigm for random selection is known as a Markov model, because values are not chosen purely at random, but are chosen at random per the probability that they will be chosen (Nierhaus, p. 71). The Markov model can be applied to multiple facets of a musical composition. For example, one aspect of a measure of music is the set of note durations that comprise the measure (e.g. eighth notes, quarter notes, half notes, etc.) and the order in which the note durations appear in the measure. Every combination of note durations that fill a single measure would then be the set of values to which probabilities would be assigned.

The obvious downside to this approach is the large number of combinations that arise from relatively few note-duration options. For example, if the time signature is $\frac{4}{4}$, and the set of possible note durations consists of only the whole note (W), then the size of the set of note duration options is 1, and that note duration (whole note) would be chosen 100% of the time. If both whole notes and half notes (H) can be used, then the set of options for a measure would have size 2: either one whole note or two half notes. Each of these options would then have assigned to them probabilities that sum to 1.

If whole, half, and quarter notes (Q) can be chosen, then the set of note duration combination options for a measure grows to 1 (whole note) plus 1 (two half notes) plus 1 (one half note and two quarter notes) plus 1 (four quarter notes) = 4. In addition to the note duration combination options, it is necessary to consider the possible permutations, or sequences, of note durations. For each of the combinations of a whole note, two half notes, and four quarter notes, there is only one permutation (switching the two half notes does not create a new permutation). For the combination of one half note and two quarter notes, more than one permutation is possible. Specifically, that combination could be represented as HQQ, QHQ, or QQH. Therefore, when dealing with whole, half, and quarter notes in $\frac{4}{4}$ time, there are 6 possible configurations of note durations.

When eight notes (E) are introduced into the mix, there are 39 possible configurations:

4 permutations of one kind of note duration:

```
W                    HH                   QQQQ                 EEEEEEEE
```

Figure 6 - One Kind of Note Duration

23 permutations of two kinds of note durations:

HQQ
QHQ
QQH
HEEEE
EHEEE
EEHEE
EEEHE
EEEEH
QQEEEE
QEQEEE
QEEQEE
QEEEQE
QEEEEQ
EQQEEE
EQEQEE
EQEEQE
EQEEEQ
EEQQEE
EEQEQE
EEQEEQ
EEEQQE
EEEQEQ
EEEEQQ

Figure 7 - Two Kinds of Note Duration

12 permutations of three kinds of note durations:

HQEE
HEQE
HEEQ
QHEE
EHQE
EHEQ
QEHE
EQHE
EEHQ
QEEH
EQEH
EEQH

Figure 8 - Three Kinds of Note Duration

The number of possible configurations of note durations increases rapidly with the addition of smaller note-durations. It is left as an exercise for the reader to determine the number of configurations possible with the introduction of sixteenth notes. Whether the size of the possible value set is in the hundreds, thousands, or above, the set size is impractical for the direct Markov model paradigm.

A better solution would be to use an indirect Markov model, where probabilities are assigned to each note-duration. Then the construction of a measure would consist of randomly selecting note-durations, per their associated probabilities, until the sum of the durations of all notes selected equals that of one measure.

There are two methods for using the Markov model to select note-pitches. In the 1st order Markov model, the set of possible values consists of each note-pitch. Each note-pitch has a probability associated with it, and the sum of these probabilities equals 1. In this method, the probability of each note-pitch is independent of every other note-pitch in the composition.

The 2nd order Markov model for note-pitches considers not the raw note-pitches themselves, but rather considers the sequential transition from one note-pitch to another. Every combination of two note-pitches (current and next) is a member of the set of values from which to choose called a transition table. Probabilities are assigned to each ordered pair of note-pitches, such that the probabilities of all note-pitch ordered pairs that share a common first note-pitch sum to 1. Then, these note-pitch transitions are selected randomly according to the current note pitch and the probabilities of selecting the next note pitch (Nierhaus, p. 72).

The transition table of a Markov model for music synthesis can be constructed by counting transitions in an actual song. The probabilities in the transition table then reflect the frequency of actual transitions in an existing song. Using this transition table to synthesize a new piece of music will result in the new piece of music "sounding like" the original piece of music, in the sense of transition selection, while still being distinct from it. Building a transition table from a set of songs by a single artist will enable the transition table to synthesize songs that mimic the style of that artist, again in terms of transitions only.

If the desire is to synthesize original songs that are in the style of a specific artist, the above technique can be applied to aspects of the song in addition to note-pitch transitions. The Markov model would consist of a set of probability tables for note duration transitions, chord progressions, etc.

3. Implementation

The AMS program was developed using the C++ programming language on the Linux operating system. AMS outputs a text file such that one line of text in the text file corresponds to one line of music. Also, AMS creates MIDI files that can be read by music editing software such as Anvil Studio (Anvil) and displayed as sheet music.

3. 1. AMS File Format

The AMS file format places an entire line of music on one line of text in a text file using delimiters to separate sections and individual items. Figure 9 below shows an example of a single line of music generated by AMS. Figure 10 below shows the same line of music as stored in text form in the .ams file.

NOTE: There are two ways to play the music examples in this document.

1. Click on the sheet music.
or
2. Scan the QR code.

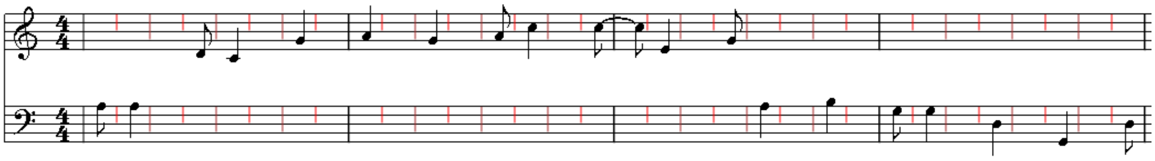


Figure 9 - A Line of Music Example

```
54;53;1;A;4500;0,500,1000,1000,1000,0;4,4;>1536,0
,0;>384,1>384,2>384,5>384,1;>48,57,>96,57,>48,62,
>96,60,>96,67,>96,69,>96,67,>48,69,>96,72,>96,72,
>96,64,>48,67,>96,57,>96,59,>48,55,>96,55,>96,50,
>96,43,>48,50,
```

Figure 10 - .ams File Format Example

.ams File Format

Each line in the file is a semicolon-delimited list of the following fields:

1.	sequence number	sequential in .ams file, corresponds to text line number
2.	parent sequence number	for lines derived by hill climbing or simulated annealing
3.	run number	sequence of hill climbing or simulated annealing run, shared by initial random line and all of its children
4.	method	'R' = random, 'C' = hill climbing, 'A' = simulated annealing
5.	score	composite score for the line used by algorithms
6.	syncopation score	currently unused syncopation score component
7.	measure beats w/ notes score	
8.	interval absolute score	
9.	reasonable octave score	
10.	avoid notes score	
11.	last note chord note score	
12.	time signature	beats per measure,1/gets one beat e.g. "4,4" is $\frac{4}{4}$ time
13.	keys	list of keys (see below)
14.	chords	list of chords (see below)
15.	notes	list of notes (see below)

Figure 11 - .ams File Format

Lists of keys/chords/notes

The keys, chords, and notes components are lists of values, where each list starts with a '>' (greater than) character and ends with either the '>' character of the next item in the list or a ';' semicolon denoting the end of the list. Each list is a comma-separated list of two or three fields, as follows:

Keys		
	duration	in MIDI ticks (96 MIDI ticks per quarter note)
	note family	0 = "C", 1 = "C#", etc.
	scale	0 = major, 1 = minor, etc.
Chords		
	duration	in MIDI ticks (96 MIDI ticks per quarter note)
	chord value	base scale degree (1-7, e.g. 1 = I (for major) or i (for minor), etc.)
Notes		
	duration	in MIDI ticks (96 MIDI ticks per quarter note)
	MIDI pitch	0 = "C", 1 = "C#", etc.
	lyric	string, currently unused

Figure 12 - Format of Keys, Chords, and Notes

3. 2. MIDI File Format

MIDI or Musical Instrument Digital Interface is a standard way of digitally encoding and communicating sound produced by a musical instrument. Note that MIDI does not encode music, per se, as there are no MIDI codes for "quarter note" or "measure" or "rest". These abstractions can be extrapolated from MIDI data, however.

An initial attempt was made to create a program that parses MIDI files, based upon the MIDI specification. The idea was that the program could build a database of music based on the millions of existing MIDI files on the Internet. This attempt was only partially successful, however, because many MIDI files have been created using message compression techniques and modes supported by the MIDI specification. These modes complicate the development of a universal MIDI parser sufficiently to push the development of a universal MIDI parser outside of the scope of this project.

There are two high-level types of MIDI messages: channel and system. Channel messages are directed to one of 16 separate channels or endpoints of MIDI message streams. System messages are sent to all endpoints in a MIDI system. System messages are further broken down into three message subtypes: Real-Time, Common, and System Exclusive. Channel messages are comprised of Mode and Voice message subtypes (Rothstein, pp. 53-55).

For purposes of implementing the AMS program, the primary message type of interest is the channel voice message. Notes are created by AMS using only two such messages: Note On and Note Off. These messages specify the channel, the note (in this context, “note” corresponds to the key on the piano keyboard), and the time since the last MIDI messages for the specified channel for the note to be started or stopped.

3. 3. Scales

A note family, or “pitch class” (Kostka, p. 51), is the set of notes that all share the same name. For example, middle C and the C one octave above middle C are two members of the note family of “C”.

Scale “denotes the tonal material of music arranged according to rising pitches (Apel, p. 662).” For example, the chromatic scale consists of all 12 note families that are found on a piano keyboard. Other scales consist of a subset of these note families.

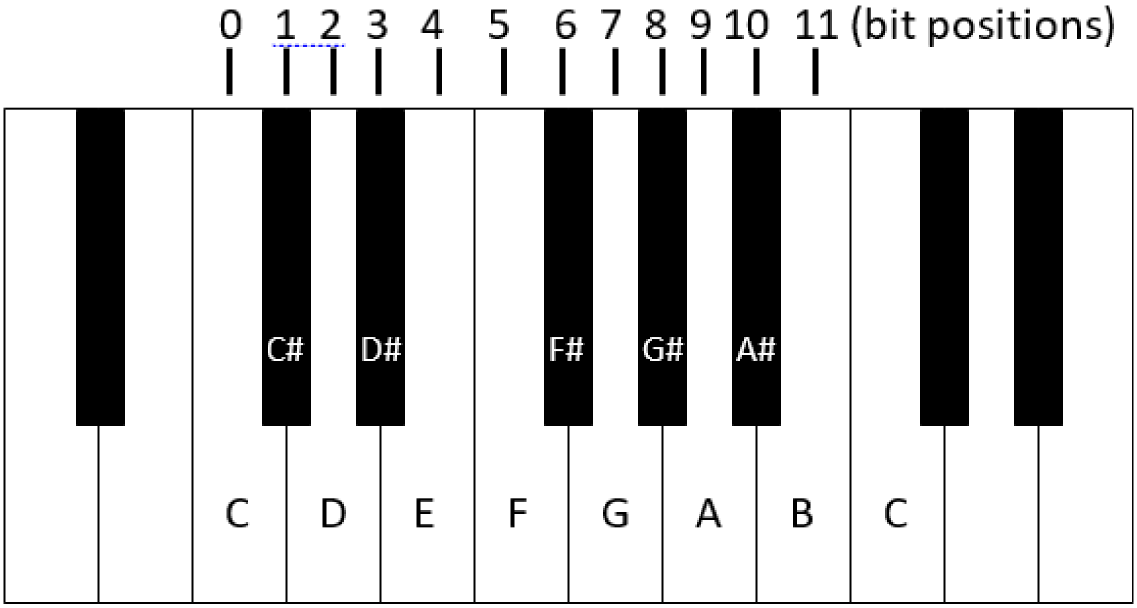


Figure 13 - Scale Format: Key to Bit Assignments

The recognition that each of the 12 note families either participates or does not participate in a given scale naturally suggests a data structure for encoding a scale. AMS uses the low-order 12 bits of an unsigned integer to encode a scale. If a note family participates in a scale, its corresponding bit is set to one. Otherwise, it will be zero. The lowest order bit corresponds to the key of the scale being represented, so the low-order bit will always be one. The AmsScales class stores the names of scales, associates each scale name with a scale number, and contains a vector of the scales of which AMS is aware.

For example, the major scale is encoded as 0xAB5. This will hold, regardless of the key, or tonal center (C Major, F Sharp Major, etc.). The following illustrates how the bits correspond to note families for the key of C Major.

```
               A       B       5
               |       |       |
            ------- ------- -------
    0xAB5 = 1 0 1 0 1 0 1 1 0 1 0 1
            | | | | | | | | | | | |
      B  <--/ | | | | | | | | | | |
(no)  A# <----/ | | | | | | | | | |
      A  <------/ | | | | | | | | |
(no)  G# <--------/ | | | | | | | |
      G  <----------/ | | | | | | |
(no)  F# <------------/ | | | | | |
      F  <--------------/ | | | | |
      E  <----------------/ | | | |
(no)  D# <------------------/ | | |
      D  <--------------------/ | |
(no)  C# <----------------------/ |
      C  <------------------------/
```

Figure 14 - Scale Format Example

AMS defines the following scales. Additional scales are straightforward to define.

```
// Heptatonic (7-note) scales:
BuildScaleVector("Ionian",          0xAB5); // Ionian (major)
BuildScaleVector("Dorian",          0x6AD); // Dorian (minor w/ raised 6th)
BuildScaleVector("Phrygian",        0x5AB); // Phrygian (minor w/ lowered 2nd)
BuildScaleVector("Lydian",          0xAD5); // Lydian (major w/ raised 4th)
BuildScaleVector("Mixolydian",      0x6B5); // Mixolydian (major w/ lowered 7th)
BuildScaleVector("Aeolian",         0x5AD); // Aeolian (minor)
BuildScaleVector("Locrian",         0x56B); // Locrian (minor w/ lowered 2nd & 5th)

// Diatonic (2-note) scales:
BuildScaleVector("MajorDiatonic",   0x005); // 2-note scale

// Pentatonic (5-note) scales:
BuildScaleVector("MajorPentatonic", 0x295);  // C-D-E-G-A
BuildScaleVector("MinorPentatonic", 0x4A9);  // A-C-D-E-G
BuildScaleVector("Egyptian",        0x4A5);  // D-E-G-A-C
BuildScaleVector("BluesMinor",      0x529);  // E-G-A-C-D
BuildScaleVector("BluesMajor",      0x2A5);  // G-A-C-D-E

// Dodecatonic (12-note) scale:
BuildScaleVector("Chromatic",       0xFFF); // a.k.a. Grand (all notes)
```

Figure 15 - Scales

3. 4. Chord Progressions

Whereas scale relates to the key, or tonal center, of a piece of music, “mode” refers to a starting point within the key. Mode “denotes the selection of tones, arranged in a scale, which form the basic tonal substance of a composition (Apel, p. 452).” For purposes of AMS, the concept of mode is used to construct the chords which underlie the line of music to be generated.

“Tonal harmony” is the style of music that was developed by classical composers, most notably Bach, between the years 1650 and 1900 (Kostka, p. x). Music based on tonal harmony features a tonal center, a pitch class to which a melody gravitates, and from which other pitch classes can be assigned a kind of quantitative potential energy.

In addition to a tonal center, tonal harmony music utilizes major or minor scales. Finally, tonal harmony utilizes chords built on thirds (intervals) known as tertian chords. These chords are related to the tonal center and each other in complex ways. (Kostka, p. xi)

From Major Chord	I	ii	iii	IV	V	vi	vii°	<--To Major Chord
I	1	1	1	1	1	1	1	
ii	0	1	0	0	1	0	1	
iii	0	0	1	1	0	1	0	
IV	1	1	0	1	1	0	1	
V	1	0	0	0	1	1	0	
vi	0	1	0	1	0	1	0	
vii°	1	0	0	0	0	0	1	

From Minor Chord	i	ii°	III	iv	V	VI	vii°	VII<--To Minor Chord
i	1	1	1	1	1	1	1	1
ii°	0	1	0	0	1	0	1	0
III	0	1	1	1	0	1	0	0
iv	0	1	0	1	1	0	1	0
V	1	0	0	0	1	1	0	0
VI	0	1	0	1	0	1	0	0
vii°	1	0	0	0	0	0	1	0
VII	0	0	1	0	0	0	0	1

Figure 16 - Chord Progressions

In the chord transition tables in Figure 16 above, based on (Kostka, pp. 107-109),

- uppercase Roman numerals represent major chords
- lowercase Roman numerals represent minor chords
- lowercase Roman numerals with “°” represent diminished chords
- “1” means the transition is allowed
- “0” means the transition is not allowed

As an alternative, the following table represents tonal major chord progressions (Chew, p. 131).

Chord	is followed by	sometimes by	less often by
I	IV, V	VI	II, III
II	V	IV, VI	I, III
III	VI	IV	I, II, V
IV	V	I, II	III, VI
V	I	VI, IV	III, II
VI	II, V	III, IV	I
VII	III	I	

Table 2 - Chord Progressions

This table does not quantify “sometimes” and “less often”. However, these frequencies can be assigned probabilities that are either specified or selected at random before generating a chord sequence for a song segment.

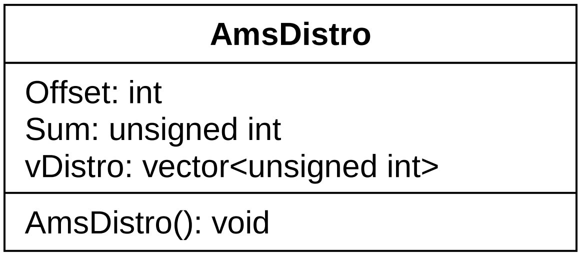
3. 5. Discrete Likelihood Weighting

In *Artificial Intelligence: A Modern Approach*, Russell and Norvig describe Monte Carlo algorithms, which are used to approximate solutions to inference problems that are too complex to solve reasonably using mathematical means. One such algorithm called “likelihood weighting” (Russell, p. 532) uses randomly generated, continuous values to select one of a finite set of events. The events are not equally likely to occur, however, so each event is weighted according to its likelihood, and generated random numbers select an event per its likelihood relative to the likelihoods of all events.

AMS taps the likelihood weighting concept for randomly generating both notes and the duration of those notes. Instead of a continuous distribution of random values, AMS randomly generates notes and durations from a discrete set of possibilities. For example, if a vocalist is capable of singing high notes occasionally but not for long periods of time, the program can be made to randomly select high notes 5% of the time and notes that are more reasonable to sing 95% of the time.

Interval Offset Model -- A First Attempt at Note Selection

The AmsDistro class stores a single discrete likelihood weighting distribution. For notes, the random number generated represents the next note as an interval offset from the current note. The offset possibilities are integer numbers of scale steps above or below the current note. The likelihoods of selecting each number of scale steps are stored in the vDistro vector. The Offset variable is used to determine which member of the vDistro vector represents zero scale steps. The Sum variable stores the sum of the members of the vDistro vector, for improved performance.



AmsDistro

Offset: int
Sum: unsigned int
vDistro: vector<unsigned int>

AmsDistro(): void

Figure 17 - DLW Interval Offset Structure

For example, the following likelihood distribution makes a random selection of one half step up or down (e.g. where a half step on a piano is the interval from C to C#) three times as likely as a random selection of two half steps up or down. This distribution example, for purposes of illustration only, does not support the selection of the unison interval (note pitch does not change) steps or ±3 or more half steps.

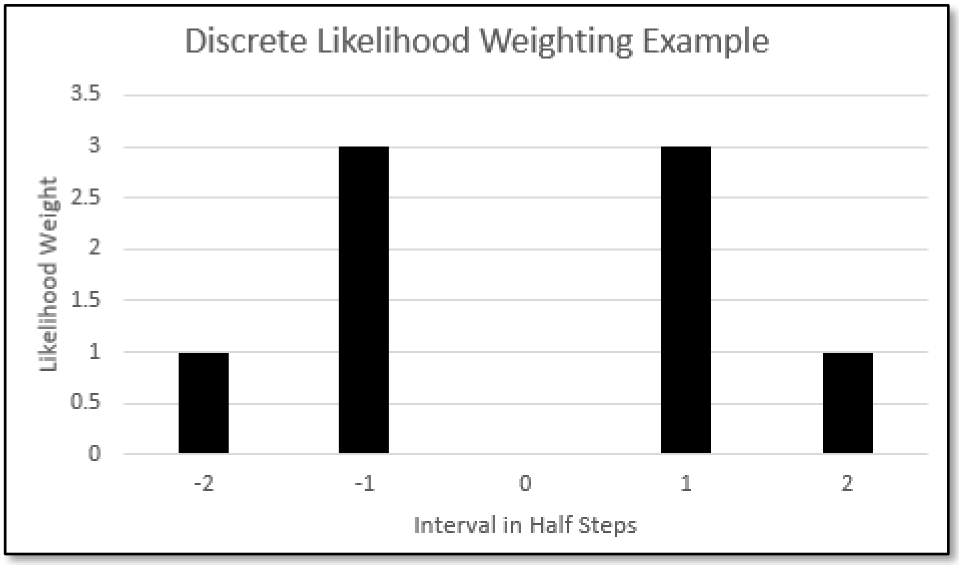
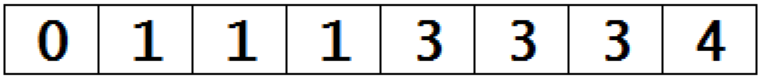


Figure 18 - Interval Offset Distribution Example

To make a random selection using this likelihood weighting, the first step is to construct a vector of unsigned integers vBuckets. For each value in the distribution vector (as illustrated in Figure 18), insert integers into the vBuckets vector, such that value of an integer corresponds to the interval in half steps and the number of times that integer is inserted into the vBuckets vector corresponds to the likelihood weight of that interval. So for the likelihood weighting in Figure 18, the vBuckets vector will look like this:

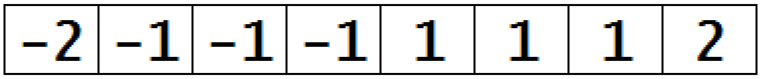


0	1	1	1	3	3	3	4

Figure 19 - Distribution Buckets

Imagine this is a game at the county fair where one tosses a ball into one of the buckets in Figure 19. One would be three times as likely to toss a ball into a “3” bucket as into a “4” bucket. Also note that one may not toss a ball into a “2” bucket, because no such bucket exists.

Next, select a random number (Rand) between 0 and “number of buckets” – 1 (8 - 1 = 7 in this example). This will yield an index into the vBbuckes vector (i.e. the “ball toss”). Finally, acquire the number of scale steps by subtracting Offset (2 in this example) from vBuckets[Rand]. This calculation converts the bucket values into the desired scale step values, as follows:



-2	-1	-1	-1	1	1	1	2

Figure 20 - Distribution Buckets Converted to Half Steps

In this conceptual vector, a purely random choice of the eight values is three times as likely to select -1 or 1 as it is to select -2 or 2. Note that in this illustration zero is not an option. Thus the likelihood of each choice is weighted according to the desired distribution. This solution is similar to the one described by Bateman for specifying a random distribution of notes (Bateman, p. 228).

It differs in two significant ways. First, Bateman’s probability distribution uses a random number to select absolute notes. The distribution originally implemented for AMS selects a note that is a relative distance above or below the previous note. Second, in Bateman’s probability distribution, the random number specifies half steps. The random number in the AMS distribution selects a number of half steps for the next interval. This aspect of AMS could be similar to Bateman by using the chromatic scale (all 12 note families "A" through “G#”) for the distribution.

Originally the discrete likelihood weighting for note selection was implemented using an integer variable that accumulated vDistro likelihood values and stopped when the accumulated likelihood value equaled or exceeded the random number generated. This technique consistently yielded note selections that did not match the intended likelihood weighting but instead "drifted down" showing favor to random notes lower than the previous note. See Figure 21 for an example of the downward drift.

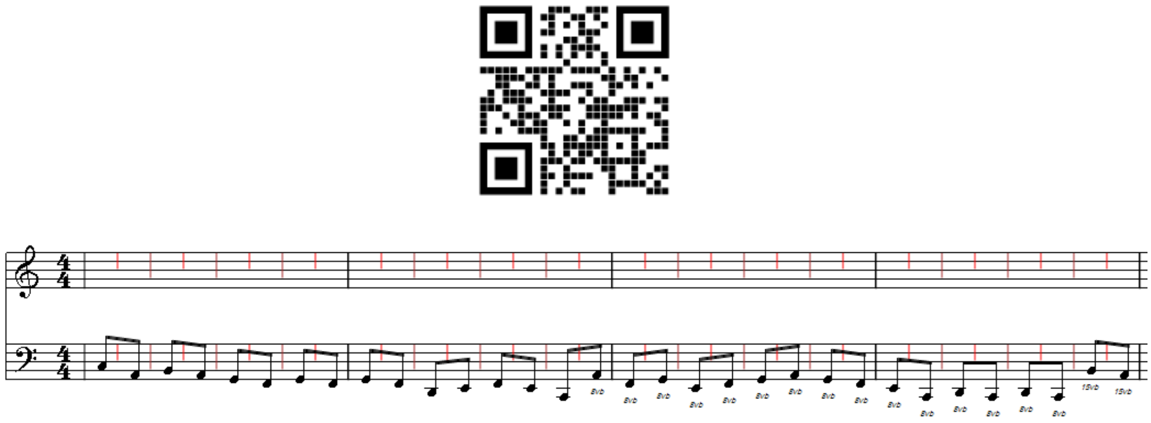


Figure 21 - Downward Drift Example

When the note selection algorithm was changed to utilize the vBuckets vector, as described above, the downward drift was eliminated, and notes were equally likely to go up or down. An example of the improvement is shown in Figure 22.

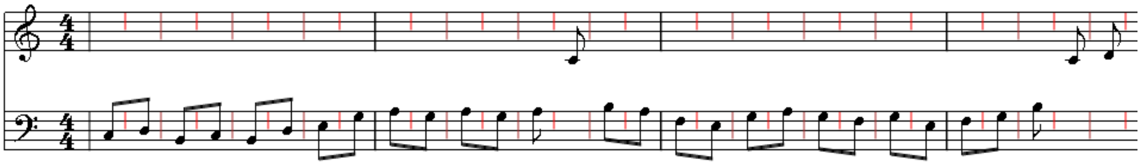


Figure 22 - Downward Drift Solution Example

Random Absolute Model -- The Note Selection Technique Ultimately Used

The problem with the offset model is that it requires the user to specify likelihoods of intervals before generating notes, which can lead to unintended consequences, such as the downward drift illustrated above. Also, by choosing an interval it becomes necessary to account for boundary conditions. For example, when the current note is the highest acceptable pitch, an interval that goes up would need to be detected and rejected. The method of rejection, e.g. turning “up” into “down” can result in some absolute notes being selected more frequently than intended, which violates the specified distribution of the interval offset method.

A better technique, and the one that is currently used by AMS, is to assign weights to every note in an acceptable range (e.g. every key on the piano keyboard) and use discrete likelihood weighting to select an absolute note. By selecting absolute notes randomly, there are no boundary conditions to be dealt with, and there are no side-effects, such as downward drift.

In the following discrete likelihood weighting example, notes in the two octaves surrounding middle C are five times as likely to be randomly chosen as notes of the two octaves flanking them, and notes higher or lower than these four octaves are omitted from random selection.

```
# Absolute_Notes_Middle_Favored: notes toward the
center of the keyboard are more likely to be
chosen
# C,C#,D,D#,E,F,F#,G,G#,A,A#,B
Absolute_Notes_Middle_Favored=\
0,\
0,0,0,0,0,0,0,0,0,0,0,0,\
0,0,0,0,0,0,0,0,0,0,0,0,\
0,0,0,0,0,0,0,0,0,0,0,0,\
1,1,1,1,1,1,1,1,1,1,1,1,\
5,5,5,5,5,5,5,5,5,5,5,5,\
5,5,5,5,5,5,5,5,5,5,5,5,\ # middle C octave
1,1,1,1,1,1,1,1,1,1,1,1,\
1,0,0,0,0,0,0,0,0,0,0,0,\
0,0,0,0,0,0,0,0,0,0,0,0,\
0,0,0,0,0,0,0,0,0,0,0,0,\
0,0,0,0,0,0,0,0
```

Figure 23 - Discrete Likelihood Weighting for Absolute Notes Example

3. 6. Generating a Random Line of Music

Specifying Options

Options such as the number of measures to generate, the algorithm to use, and discrete likelihood weight tables are specified on the command line and in configuration files.

```
usage: ams -c <config.file>

   or: ams -f <file.mid>           -l
           -n <note family>        -q <quantity of notes>
           -s <scale>              -m <num measures>
           -b <beats per measure>  -t <note distribution>
           -g <gets one beat>      -d <duration distribution>

       (must specify either -q # or -m #)

   <note family> is one of: A A# B C C# D D# E F F# G G#

   <scale> is one of: Heptatonic (7-note) Scales (or Modes):
      (7, Heptatonic) Major Minor Ionian Dorian Phrygian Lydian Mixolydian
Aeolian Locrian
      (5,  Pentatonic)  MajorPentatonic  MinorPentatonic  Egyptian  BluesMinok
BluesMajor
      (2, Diatonic)   MajorDiatonic
      (12, Chromatic) MajorDiatonic

   -l means generate a random line of music
   -s means generate the specified scale

   <note distribution> is one of (* marks unison interval):
      ZeroOrOneStep (1, 1*, 1)
      TwoOrOneStep  (2, 1, 0*, 1, 2)
      OneOrTwoStep  (1, 2, 0*, 2, 1)
      TwoUpOneDown  (1, 0*, 2)
      OneUpTwoDown  (2, 0*, 1)
      PlusOrMinusOneOctave (1, 1, 2, 5, 3, 4, 3, 1*, 3, 4, 3, 5, 2, 1, 1)

   <duration distribution> is one of:
      Durations_1     Durations_2

   eg: ams -f major.mid -n C -s Major -b 4 -g 4
       ams -f major.mid -n C -s Major -b 4 -g 4 -l -q 100
```

Figure 24 - ams Program Command Line Options

Choosing the Key

The AMS program supports multiple keys (a.k.a. “key changes”) within a line of music. For purposes of this research, one key is used for the duration of the line: C Major. Musical lines generated can always be translated externally to any key.

Choosing the Chords

For purposes of this research, the duration of chords is simplified to one chord per measure (e.g. chords are whole notes). The first and last cords are always 1 (the tonic). Intermediate chords are chosen at random from the available options, 1-7, based on the value of the previous chord and the likelihood weighting of possible next chords, which for this project are either 0 or 1.

Choosing the Notes

The duration of a note is chosen at random based on the following discrete likelihood weighting table:

```
Durations_5=\
0,\#  offset
0,\#  128th
0,\#  64th
0,\#  dotted 64th
0,\#  32nd
0,\#  dotted 32nd
0,\#  16th
0,\#  dotted 16th
100,\# 8th
0,\#  dotted 8th
100,\# quarter
0,\#  dotted quarter
20,\# half
0,\#  dotted half
0#    whole
```

Figure 25 - Discrete Likelihood Weighting for Note Durations Example

The pitch of a note is chosen at random based on the discrete likelihood weighting table in Figure 23.

The note is constructed using the randomly generated duration and pitch and is added to the line of notes vector. This process is repeated until the sum of the note durations equals the duration of the specified number of measures for the line. The duration of the last note of the line of music is truncated if its random duration would make it last beyond the last measure of the line.

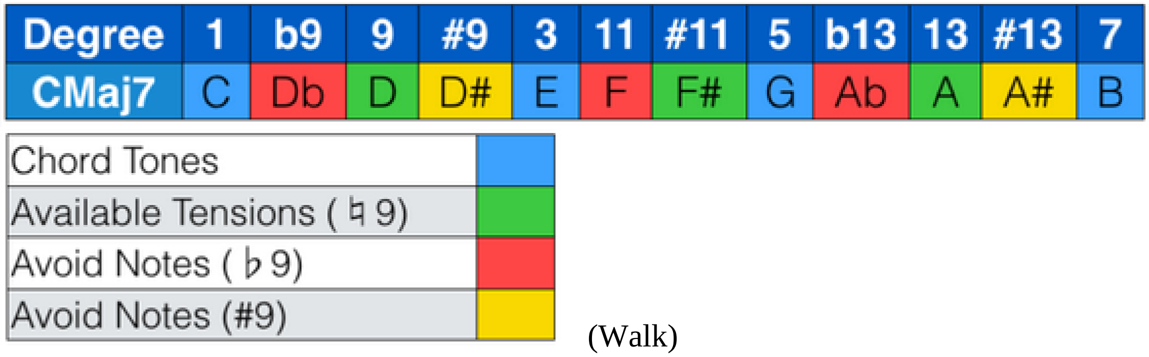
3. 7. Signatures

After a line of music is randomly generated, AMS constructs signatures for the line. A signature is an array of numbers that describe one aspect of the line of music. It is like a one dimensional slice of the data and is useful when implementing adjudication rules. The following signatures are constructed.

DurationsWithRests	one array element per note where each element is the duration of each note in the line in midi ticks including rests
NotesWithRests	one array element per note where each element is the MIDI note pitch of each note including rests
ChromaticDegreesWithRests	one array element per note where each element is the number of half steps above the tonic of the key of the line of music
IntervalsRelative	one array element per adjacent note pair where each element is the adjacent note interval in half steps
IntervalsAbsolute	one array element per adjacent note pair where each element is the absolute value of adjacent note interval in half steps
Directions	one array element per adjacent note pair where each element is -1 = down, 0 = unison, or 1 = up interval
NoteOnBeat	one array element per beat, where each element is either 0 = no note on beat or 1 = note on beat
NotesOnBeats	one array element per measure beat, where each element is the count of notes on each measure beat

AvoidNotes	one array element per note where each element is a character representing an avoid note classification: 'C' = chord tone 'T' = available tension 'a' = less dissonant avoid note 'A' = more dissonant avoid note

The “Avoid Notes” score is based on a system developed to support improvisation by jazz musicians and is graphically depicted in Figure 26 below. (Walk)



Degree	1	b9	9	#9	3	11	#11	5	b13	13	#13	7
CMaj7	🟦 C	🟥 Db	🟩 D	🟨 D#	🟦 E	🟥 F	🟩 F#	🟦 G	🟥 Ab	🟩 A	🟨 A#	🟦 B

Chord Tones	🟦
Available Tensions (♮9)	🟩
Avoid Notes (♭9)	🟥
Avoid Notes (#9)	🟨

(Walk)

Figure 26 - Note Classifications

Here are some examples of signatures, each an array of numbers or characters corresponding to one aspect of a line of music:

```
SigDurationsWithRests (19):
48,96,48,96,96,96,96,48,96,96,96,48,96,96,48,96,96,96,48

SigNotesWithRests (19):
57,57,62,60,67,69,67,69,72,72,64,67,57,59,55,55,50,43,50

SigChromaticDegreesWithRests (19):
9,9,2,0,7,9,7,9,0,0,4,7,9,11,7,7,2,7,2

SigIntervalsRelative (18):
0,5,-2,7,2,-2,2,3,0,-8,3,-10,2,-4,0,-5,-7,7

SigIntervalsAbsolute (18):
0,5,2,7,2,2,2,3,0,8,3,10,2,4,0,5,7,7

SigDirections (18):
0,1,-1,1,1,-1,1,1,0,-1,1,-1,1,-1,0,-1,-1,1

SigNoteOnBeat (16):
1,0,1,1,1,1,1,0,0,0,1,1,1,0,0,0
```

```
SigNotesOnBeats (4):
6,2,6,4

SigAvoidNotes (19):
T,T,T,C,C,C,T,C,C,C,T,C,T,C,C,C,T,C,T
```

3. 8. Adjudication

Adjudication is the assigning of a number or “score” to a line of music based on how good it sounds. The intent is that lines of music with high scores will sound better than those with low scores. The score is the sum of the following component scores.

Ratio of “Beats with Notes” to “Total Beats”

If a note lands on a beat, that beat counts toward the numerator. The denominator is the total number of beats, which equals 4 times the number of measures in $\frac{4}{4}$ time. If this ratio is between 0.75 and 0.95, this component is worth 1000. If the ratio is less than 0.5, the score is zero for this component. Otherwise, this component contributes 500 to the total score.

Intervals Absolute Score

For each interval in the line of music made by two adjacent notes, if the interval is greater than an octave, sum up the number of whole octaves. Multiply this sum by 200 and subtract the product from 1000. The greater of this number or zero is the component score.

Reasonable Octave Score

Sum up the number of notes in the line whose MIDI pitch value is not between 37 (C# on piano octave 2) and 95 (B on piano octave 6), inclusive. Multiply this sum by 200 and subtract the product from 1000. The greater of this number or zero is the component score.

Avoid Notes Score

As per section 3. 7 above, notes classified as ‘a’ (notes that are 3 or 10 half steps above the base of the current chord) are “less dissonant avoid notes,” and notes classified as ‘A’ (notes that are 1, 5, or 8 half steps above the base of the current chord) are “more dissonant avoid notes.” The numbers of ‘A’s and ‘a’s influence the avoid notes component score as follows:

$$1000 * \left(\frac{1}{2}\right)^{\#\ 'A's} * \left(\frac{2}{3}\right)^{\#\ 'a's}$$

Melodic Intervals Score

Compute the interval average as the sum of absolute intervals between adjacent notes divided by the number of intervals (which is the number of notes in the line - 1). If the interval average is less than 4.1, the melodic intervals score is 1000. Otherwise, subtract 4 from the interval average, multiply by 100, and subtract the product from 1000. If this difference is greater than zero, then it is added to the total score.

Last Note Chord Note Score

As per section 3. 7 above, notes classified as ‘C’ (notes that are 0, 4, 7 or 11 half steps above the base of the current chord) are “chord notes.” This component adds to the total score 500 if the first note of the line is a chord note and 500 if the last note of the line is a chord note. Originally both the first note and the last note needed to be chord notes for this component to contribute 1000, otherwise it was worth zero. See section 3. 11 below for an explanation of why this did not work.

3. 9. Hill Climbing

One way AMS searches for quality melodies among the space of 12.9 nonillion per measure lines of music is to employ the artificial intelligence technique of hill climbing. Each line of music is a point in a space that may be searched. Points in this space have a high score number if they sound good and a low number if they do not. To perform hill climbing, AMS searches by repeatedly finding and moving to neighboring lines of music with the highest score until a plateau is reached where there are no higher scoring lines (Russell, pp. 122-125).

Hill Climbing Algorithm

1. Start with a randomly generated line of music, known as "current line".

2. Score[current line] = the score of the line generated.

3. Consider every line of music that is the same as the current line with a change to a single note (apply these to each note of the current line, one at a time):
    a. Nudge the note up one octave.
    b. Nudge the note up one half step.
    c. Nudge the note down one half step.
    d. Nudge the note down one octave.
    e. If the note is a half note, decompose the note into two quarter notes of the same note family.
    f. If the note is an eighth note and the next note is also an eighth note, make the first eighth note a quarter note and remove the next eighth note.
    g. If the note is an eighth note and the next note is also an eighth note, make the next eighth note a quarter note and remove the first eighth note.

4. For each "new line" constructed in step 3, score the new line.

5. If Score[new line] > Score[current line], remember the new line.

6. Repeat steps 3 through 5 for every note in the current line.

7. Make the new line with the highest score the current line.

8. Go to step 3.

9. Continue until Score[new line] <= Score[current line].

10. Return current line.

3. 10. Simulated Annealing

Another way AMS searches for good sounding melodies is a technique called “simulated annealing.” Simulated annealing is a gradient search algorithm similar to hill climbing. Two configuration parameters are passed to the simulated annealing algorithm: temperature and schedule. Temperature governs the likelihood that a worse scoring neighboring line will be selected. Schedule defines the number of loop iterations spent at a specific temperature, after which the temperature is decreased by one, making the algorithm less likely to choose a worse scoring neighbor. (Russell, p. 125-126)

The idea is that whereas hill climbing can get stuck on a local maximum score that is less than higher scores in the region, simulated annealing breaks free from the local maximum by deliberately selecting worse scoring neighbors more frequently earlier in the search process and less frequently later. The search traverses downhill from the current hill to hopefully find the base of a taller hill to ascend. (Russell, p. 125-126)

Whereas hill climbing considers the score of every line that is an immediate neighbor to the current line, simulated annealing is a stochastic technique that only considers the score of a single, randomly selected neighbor. If the random neighbor has a higher score than the current line, that neighbor becomes the current line, and the process repeats. If the random neighbor has a lower score than the current line, then a decision is made to navigate to that neighbor or not. (Russell, p. 125-126)

DeltaScore is the score of the random neighbor minus the score of the current line, which is negative if the random neighbor is worse. The probability that the algorithm will navigate to the worse scoring neighbor is given by the formula:

```
Probability = 1000 * $e^{(DeltaScore / Temperature)}$
            (= 0..999)
Rand = [a random number between 0 and 999]
if (Rand < Probability) navigate to the worse neighbor
```

As Temperature goes to zero, Probability also goes to zero, and worse scoring neighbors are chosen less often. This process is akin to molecules in blown glass being allowed to cool slowly through the process of annealing. (Russell, p. 125-126)

3. 11. Issues Encountered

<u>Last Note Chord Note Scoring</u>

The "last note chord note" score component was an all or nothing score: 1000 points if both the first and last notes of the line are chord notes and 0 points if either note is not a chord note. Regularly this score component was registering as 0 for maximum scoring lines following a hill climbing or simulated annealing (which is finished off by a hill climb) run. Here is an example where line 172 of a simulated annealing run registered a score of 0 for the LastNoteChordNote component.

```
$ score.sh 172
MeasureBeatsWithNotes: 1000
IntervalAbsolute:      1000
```

```
ReasonableOctive:      1000
AvoidNotes:            1000
MelodicIntervals:      1000
LastNoteChordNote:     0
```

An analysis of line 172 shows that both the first note and the last note of the line are available tensions. The “ChordNote” in Table 3 is the offset in half steps from the base note of the underlying chord, or “ChordOfNote” (where 1 = ‘C’).

```
 0: Note=50 NoteFamily =  2 = D  ChordOfNote = 1 ChordNote= 2 AvoidNoteSig = T
 1: Note=50 NoteFamily =  2 = D  ChordOfNote = 1 ChordNote= 2 AvoidNoteSig = T
 2: Note=48 NoteFamily =  0 = C  ChordOfNote = 1 ChordNote= 0 AvoidNoteSig = C
 3: Note=57 NoteFamily =  9 = A  ChordOfNote = 1 ChordNote= 9 AvoidNoteSig = T
 4: Note=50 NoteFamily =  2 = D  ChordOfNote = 4 ChordNote= 9 AvoidNoteSig = T
 5: Note=53 NoteFamily =  5 = F  ChordOfNote = 4 ChordNote= 0 AvoidNoteSig = C
 6: Note=57 NoteFamily =  9 = A  ChordOfNote = 4 ChordNote= 4 AvoidNoteSig = C
 7: Note=52 NoteFamily =  4 = E  ChordOfNote = 4 ChordNote=11 AvoidNoteSig = C
 8: Note=47 NoteFamily = 11 = B  ChordOfNote = 7 ChordNote= 0 AvoidNoteSig = C
 9: Note=50 NoteFamily =  2 = D  ChordOfNote = 7 ChordNote= 3 AvoidNoteSig = C
10: Note=56 NoteFamily =  8 = G# ChordOfNote = 7 ChordNote= 9 AvoidNoteSig = C
11: Note=56 NoteFamily =  8 = G# ChordOfNote = 7 ChordNote= 9 AvoidNoteSig = C
12: Note=53 NoteFamily =  5 = F  ChordOfNote = 7 ChordNote= 6 AvoidNoteSig = C
13: Note=52 NoteFamily =  4 = E  ChordOfNote = 1 ChordNote= 4 AvoidNoteSig = C
14: Note=50 NoteFamily =  2 = D  ChordOfNote = 1 ChordNote= 2 AvoidNoteSig = T
15: Note=47 NoteFamily = 11 = B  ChordOfNote = 1 ChordNote=11 AvoidNoteSig = C
16: Note=42 noteFamily =  6 = F# ChordOfNote = 1 ChordNote= 6 AvoidNoteSig = T
```

Table 3 - A Line of Music

Notes are classified as one of four flavors:

'C' = chord tone
'T' = available tension
'a' = less dissonant avoid note
'A' = more dissonant avoid note

The major chord note classifications correspond to how “good” each note in the chromatic scale sounds above the underlying chord, where offset 0 is the note family of the underlying chord. The classifications are assigned to half step offsets for major chords according to the following table.

Offset	Classification
0	C
1	A
2	T
3	a
4	C

5	A
6	T
7	C
8	A
9	T
10	a
11	C

Table 4 - Major Note Offset Classifications

The question is: “Why did line 172 have a maximum score when the first and last notes are not chord notes?” The answer turned out to be a combination of the way the LastNoteChordNote score component was computed and the way neighboring lines were determined by the algorithms.

First, the first note of line 172 shown in Table 3 is offset (or ChordNote) 2. Table 4 shows that offset 2 is classified as ‘T’ (available tension). More importantly, Table 4 show that offset 2’s neighbors are classified as ‘A’ (= more dissonant avoid note = offset 1) and ‘a’ (= less dissonant avoid note = offset 3). Therefore a nudge of the first note in any direction as described in step 3 of the hill climbing algorithm (and also used in simulated annealing) in section 3. 9 did not result in the note landing on a chord note. Therefore, the LastNoteChordNote score full-score criteria would never be satisfied. This problem was corrected by adding two offsets to the nudge vector used to determine line neighbors: -2 and 2:

```
vNudge = { -12, -2, -1, 1, 2, 12 }
```

Second, the LastNoteChordNote score component was poorly written, as its value only increased as the result of two notes changing. When the algorithms are looping through the neighboring lines of a line, they only adjust one note at a time. Therefore, if both the first note and last note of a line were available tensions, the algorithms as written would never discover a neighbor where both the first and last notes are chord notes. This bug presented, because the last note of line 172 is offset 6, whose neighbor offset 7 is classified as a chord note, which would have represented a score improvement if the first and last notes of the line were being rewarded separately for being chord notes. This problem was corrected by changing the LastNoteChordNote score component to award 500 points each for the first and last notes of the line independently of each other.

4. Results

4. 1. Hill Climbing

Here are the results of a hill climbing run. This chart shows the scores of all neighbors for each iteration of the loop. Interestingly at every step there appear to be one or two neighbors whose score is considerably worse than the current line.

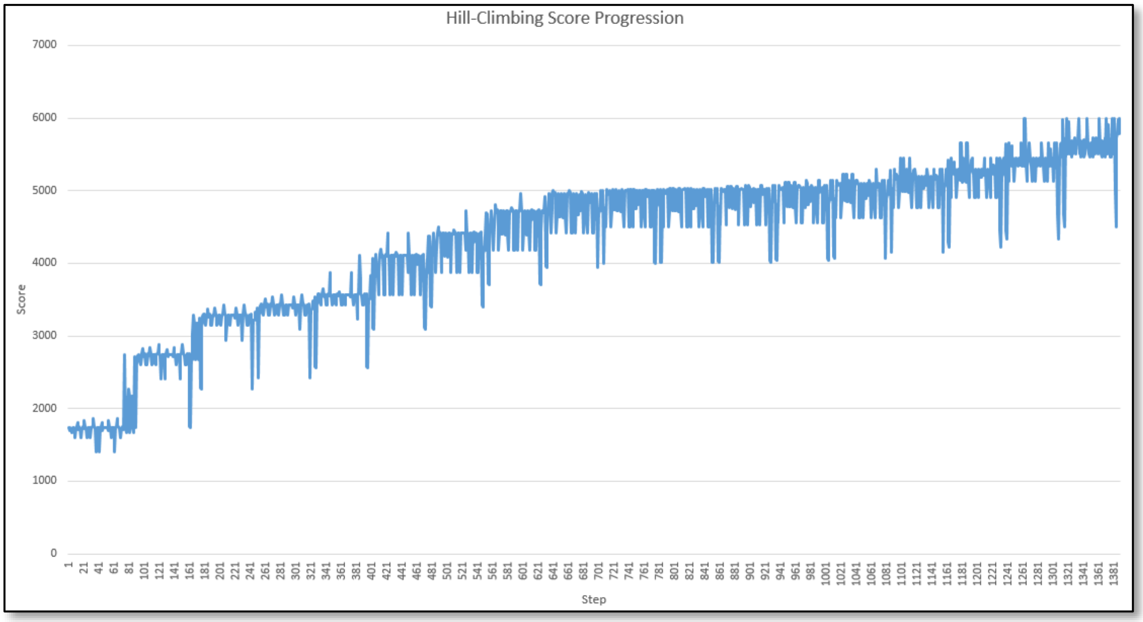


Figure 27 - Hill Climbing Score Progression

Random Line (score = 1179):

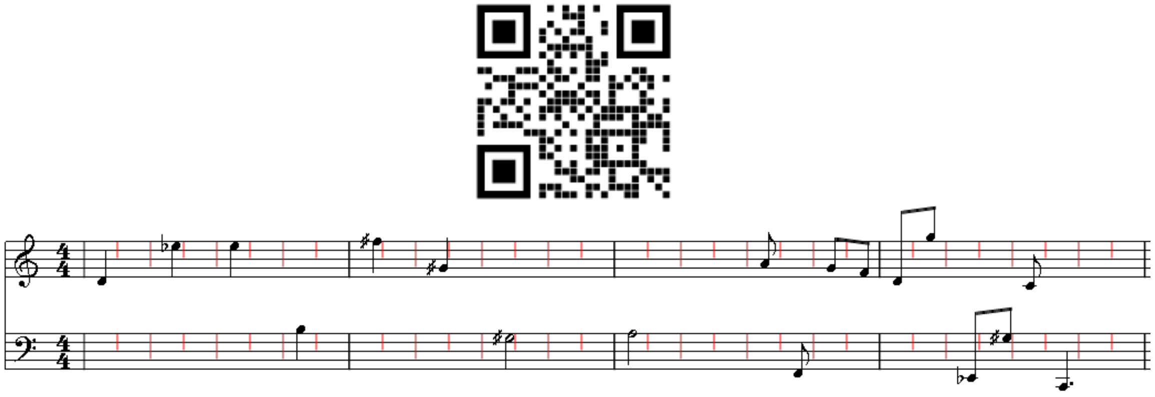


Figure 28 - A Random Line

After Hill Climbing (score = 6000):

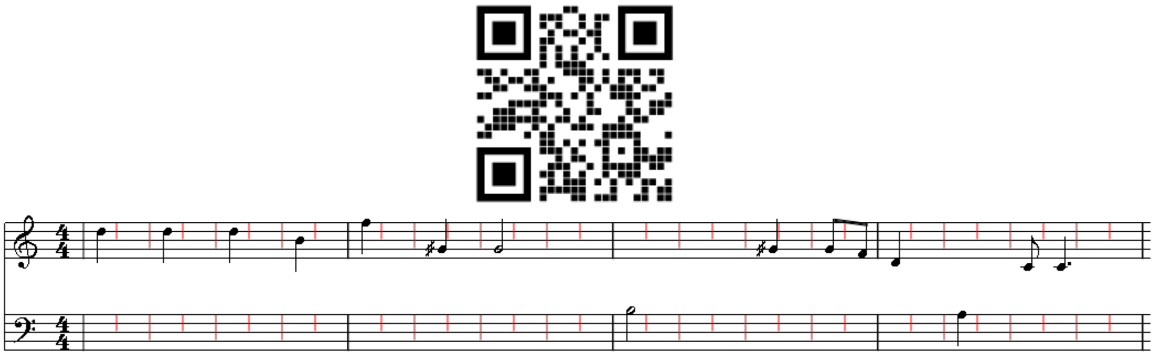


Figure 29 - After Hill Climbing

4. 2. Simulated Annealing

Here are the results of a simulated annealing run with initial temperature = 16 and schedule = 300. Note that randomly selected neighbors that are worse than the current line and are not selected are not represented in this chart.

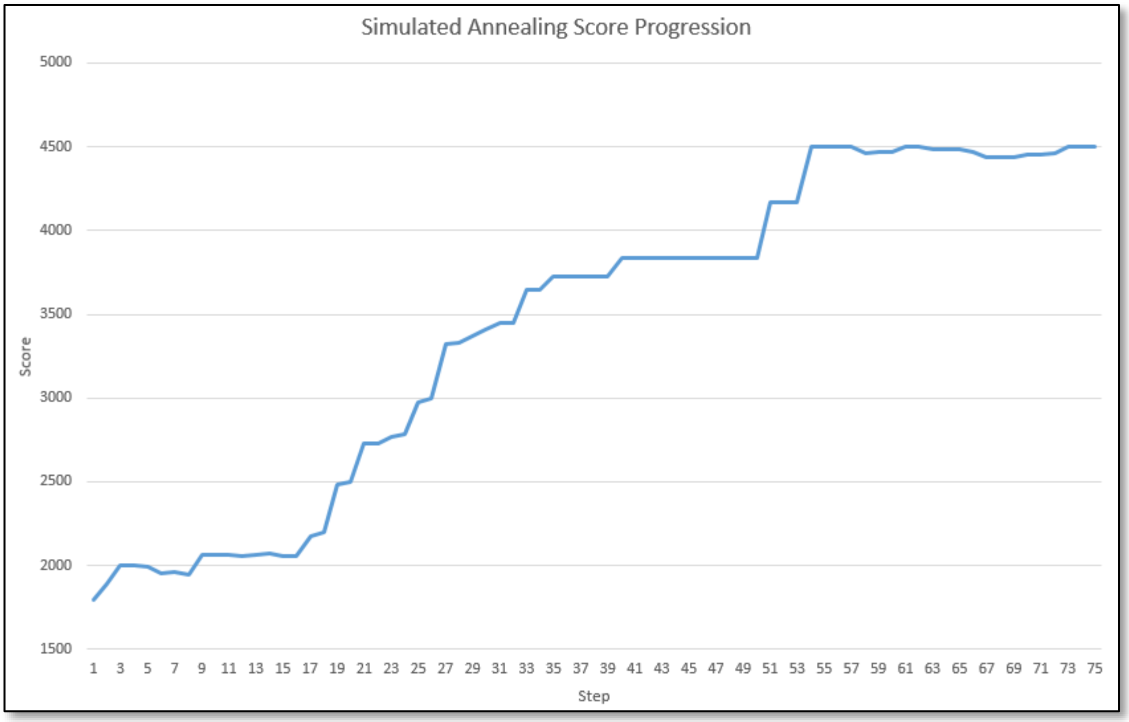


Figure 30 - Simulated Annealing Score Progression

Random Line (score = 1794):

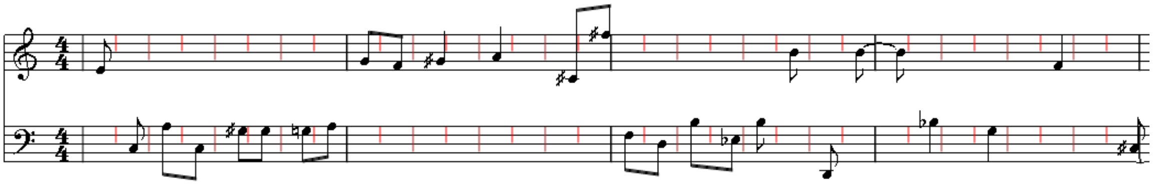


Figure 31 - A Random Line

After Simulated Annealing (score = 4500):

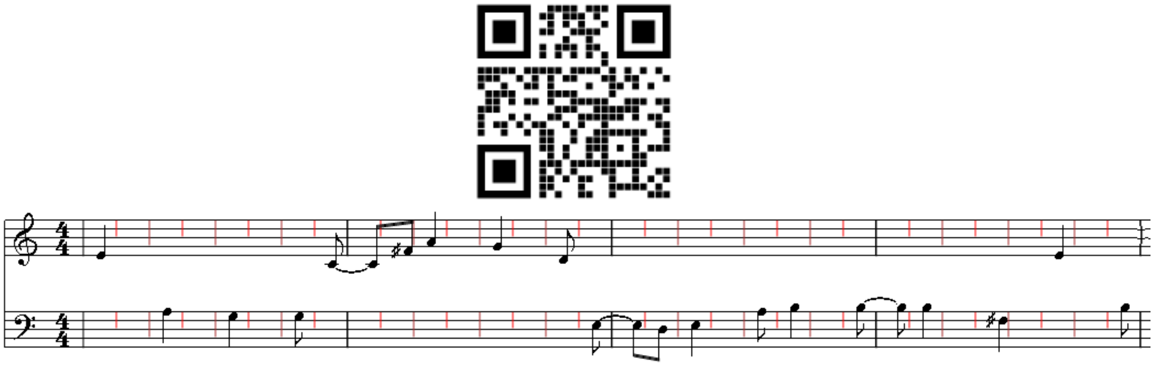


Figure 32 - After Simulated Annealing

Another simulated annealing run with initial temperature = 16 and schedule = 1000 illustrates the essence of the algorithm in Figure 31. The volatility of scores early in the process, on the left, is high. Moving from left to right, as the temperature "cools" (gets lower) the line becomes less jaggy. Overall, improvement is made, although there are local spots where the score temporarily worsens.

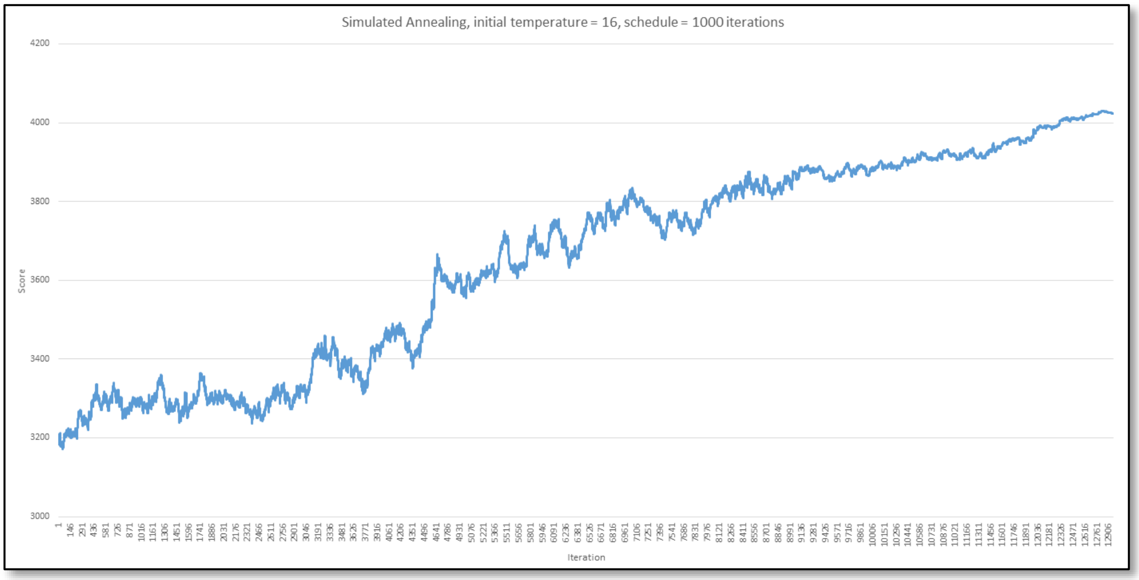


Figure 33 - Simulated Annealing Score Progression

Figure 34 below shows how the components of the score increase as the simulated annealing algorithm proceeds.

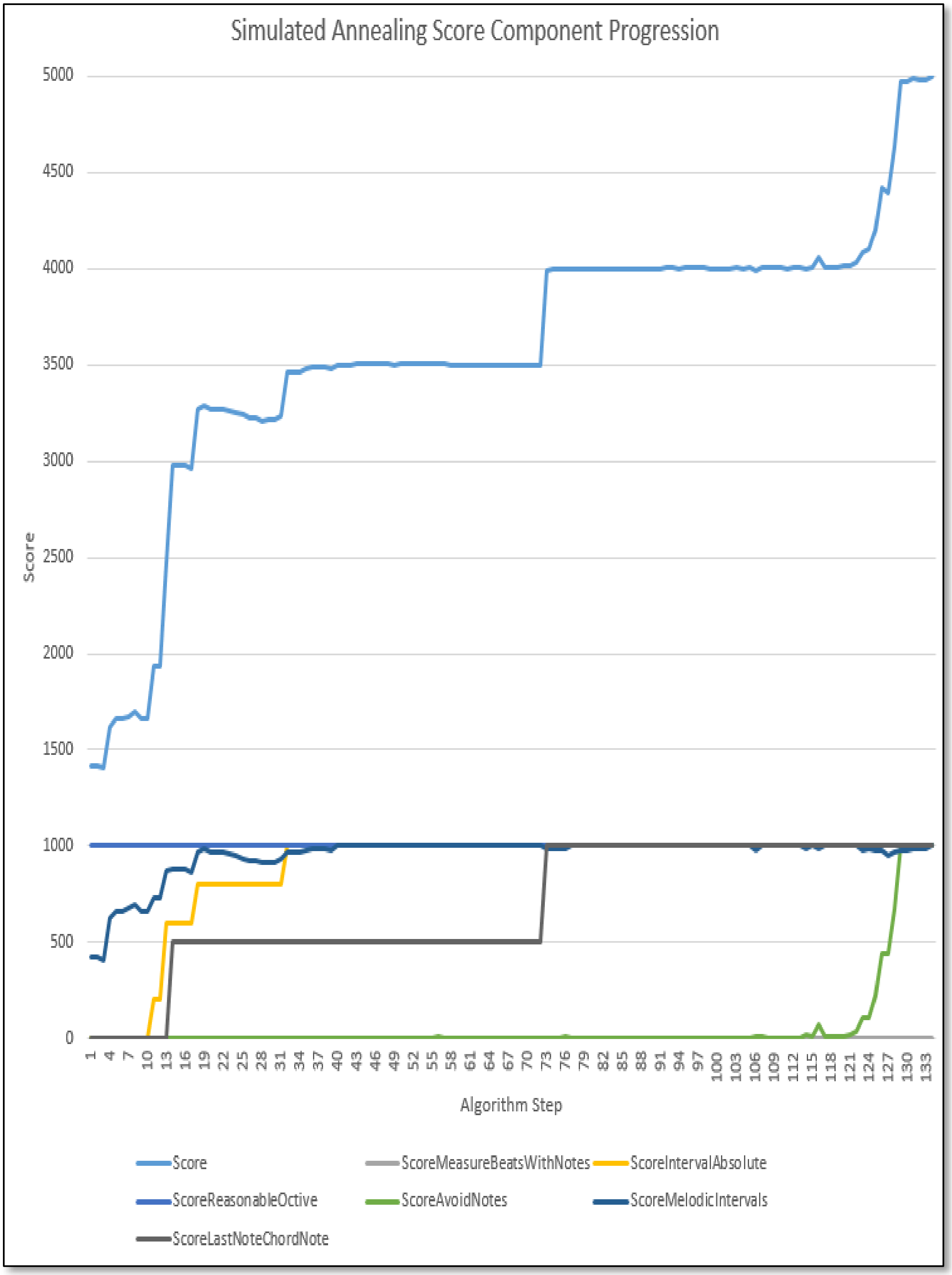


Figure 34 - Simulated Annealing Score Component Progression

5. Improvement / Extension Opportunities

Given the limited scope of the AMS project, there are straightforward opportunities to expand the adjudication algorithm to incorporate higher levels of musical abstraction, such as polyphonic harmonic melodies, verses, refrains, entire songs, and larger bodies of work.

Although AMS was written as a research program without too much care given to usability, it would not be difficult to polish the user interface and make a TUI (textual user interface) and/or GUI (graphical user interface)/Web shell wrapper.

The AMS program has no a priori knowledge about music that other composers have created in the past. One improvement would be to hook into an external database to check if a line of music discovered by AMS is substantially similar to an existing song. That way one could avoid declaring that a song randomly generated by AMS is an original piece of art when in fact it is not, but rather a coincidental discovery of a melody that already exists.

Also, AMS does not track its own history. Doing so would help to reduce duplicated effort by not examining the same line of music multiple times inside the hill climbing and simulated annealing algorithms.

AMS could also be improved by using k-grams as described in section 2. 15 above to enrich the objective function, making it more difficult to find a melody with a maximum score, resulting in high scoring melodies sounding even better and more complex.

6. Conclusion

AMS promises to be a powerful arrow in the quiver of a composer searching for a new sound. With AMS, a composer can evaluate a practically endless sampling of good sounding melodies from which to harvest and build upon.

7. References

[Anvil] “Anvil Studio: Free Music Composition, Notation & MIDI-Creation Software.” *Anvil Studio | Free Music Composition, Notation &amp; MIDI-Creation Software*, Willow Software, 10 Oct. 1998, anvilstudio.com.

[Apel] Apel, Willi. *Harvard Dictionary of Music*. Cambridge, MA: Belknap of Harvard UP, 1964. Print.

[Boulanger] Boulanger, Richard Charles, and Victor Lazzarini. *The Audio Programming Book*. Cambridge, MA: MIT, 2011. Print.

[Bateman] Bateman, Wayne. *Introduction to Computer Music*. New York: John Wiley & Sons, 1980. Print.

[Cahill] Cahill, Margaret, and Donncha O. Maidin. “Melodic Similarity Algorithms – Using Similarity Ratings for Development and Early Evaluation.” 6th International Conference on Music Information Retrieval. London, UK, 2005. Web.

[Chew] Chew, Elaine; Childs, Adrian; Chuan, Ching-Hua. Mathematics and Computation in Music. Dordrecht: Springer, 2009. Ebook Library.

[Cope] Cope, David. *Virtual Music: Computer Synthesis of Musical Style*. Cambridge, MA: MIT, 2004. Print.

[DiNunzio] DiNunzio, Alex. “Lejaren Hiller - Illiac Suite for String Quartet [1/4].” YouTube, YouTube, 4 Dec. 2011, www.youtube.com/watch?v=n0njBFLQSk8.

[Gamow] Gamow, George. *One, Two, Three-- Infinity: Facts and Speculations of Science*. New York: Dover Publications, 1988. Print.

[Johnson] Johnson, Jeffrey. *Graph Theoretical Models of Abstract Musical Transformation: An Introduction and Compendium for Composers and Theorists*. Westport, CT: Greenwood, 1997. Print.

[Johnson-Laird] Johnson-Laird, P N. *The Computer and the Mind: An Introduction to Cognitive Science.* Cambridge, Mass: Harvard University Press, 1988. Print.

[Kostka] Kostka, Stefan M., and Dorothy Payne. *Tonal Harmony, with an Introduction to Twentieth-century Music*. 5th ed., New York, NY:

McGraw-Hill, 2004, Print.

[Leman] Leman, Marc. “Artificial Neural Networks in Music Research.” *Computer Representations and Models in Music*. Academic Press, 1992. Print.

[Madden] Madden, Charles. *Fractals in Music: Introductory Mathematics for Musical Analysis*. Salt Lake City: High Art, 1999. Print.

[Maidin] Maidin, Donncha O. “Representation of Music Scores for Analysis.” *Computer Representations and Models in Music*. Academic Press, 1992. Print.

[Manning] Manning, Christopher D., Prabhakar Raghavan, and Hinrich Schütze. "K-Gram Indexes for Wildcard Queries." *Introduction to Information Retrieval*. New York: Cambridge UP, 2008. 50-51. Print.

[Miranda, Evo] Miranda, Eduardo Reck, and Al Biles. *Evolutionary Computer Music*. London: Springer, 2007. Print.

[Miranda, Readings] Miranda, Eduardo Reck. *Readings in Music and Artificial Intelligence*. New York: Routledge, 2010. Print.

[Nierhaus] Nierhaus, Gerhard. Algorithmic Composition. Dordrecht: Springer, 2009. Ebook Library.

[Owen] Owen, Harold. *Music Theory Resource Book*. New York: Oxford UP, 2000. Print.

[Rothstein] Rothstein, Joseph. *MIDI: A Comprehensive Introduction*. Madison, WI: A-R Editions, 1995. Print.

[Rumsey] Rumsey, Francis, and Tim McCormick. *Sound and Recording*. Oxford, UK: Focal, 2009. Print.

[Russell] Russell, Stuart J. and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Third ed. Upper Saddle River: Pearson, 2010. 532-35. Print.

[Smith] Smith, Matt, Alan Smaill, and Geraint A. Wiggins. *Music Education: An Artificial Intelligence Approach: Proceedings of a Workshop Held as Part of AI-ED 93, World Conference on Artificial Intelligence in Education, Edinburgh, Scotland, 25 August 1993*. London: Springer-Verlag, 1994. Print.

[Walk]	“Avoid Notes.” The Jazz Piano Site, Walk That Bass, 1 Feb. 2019, www.thejazzpianosite.com/jazz-piano-lessons/jazz-improvisation/avoid-notes.

8. Appendix A

8. 1. Original AMS File Format

Early in the development process, a file format was required to compose the notes in a line of music in a human readable fashion. This would allow manual construction of a line of music for purposes of testing various aspects of the program. As the software development proceeded this original file format was insufficient for processing the thousands of lines of music generated by the artificial intelligence algorithms. Therefore, this file format was abandoned in favor of the one-text-file-line per line of music format described in section 3. 1 above.

The AMS program was written to read music from this text file format, which has the following characteristics.

- All characters including and after a ‘;’ are considered a comment and are disregarded.
- The key is indicated by the keyword “KEY” followed by the note family and scale.
- A time signature is indicated by “TS” followed by the beats per measure and gets one beat.
- Notes are specified by a duration, a space, a note name, and an optional space and octave number.
- Durations are W (whole note), H (half note), Q (quarter note), E (eighth note), S (sixteenth note), or a number representing MIDI ticks.
- If the duration is dotted (e.g. “Q.”), the specified duration is computed to be 1.5 times the duration without the dot. For example, a dotted quarter note, “Q.” equates to a duration of 1.5 * 96 = 144 MIDI ticks.
- Note names are A, A#, Ab, B, Bb, C, C#, D, D#, Db, E, Eb, F, F#, G, Gb, G#.
- If the octave is omitted, the octave used is the one for which the pitch class named is nearest to the previous note.

The following is the content of a file named Original.ams that illustrates the features of the AMS format. It is a simple 2-measure, monophonic melody. Note that although the measures are separate in the file, the AMS file format does not explicitly demarcate measures, and notes may span measures. Such notes will be shown in Anvil Studio as two tied notes: the last note of one measure tied to the first note of the subsequent measure.

```
; Original.ams

KEY C Major

TS 4 4 ; Time signature = 4/4

; Measure 1

E C 5  ; eighth note, note C, octave 5, beat 1
E D
E D 6  ; beat 2
S A    ; sixteenth note, note C
E F
S E
E D#
E D#
E D

; Measure 2

E E 5
E G
E F
S F
E G
S F
E E
S D
S B
E C
```

Figure 35 - Original.ams - AMS File Format Example

The following program execution reads Original.ams and writes Original.mid, a corresponding MIDI file. When Original.mid is read into the Anvil Studio program, the sounds are translated into the sheet music depicted in Figure 36.

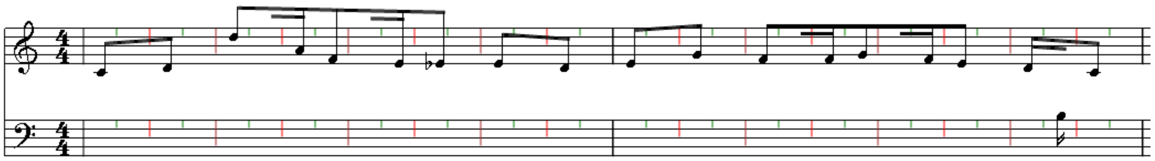


Figure 36 - Original