SPEKULATOR: GENETIC ALGORITHM CHORD PROGRESSION GENERATOR

A Project

Presented to the faculty of the Department of Computer Science

California State University, Sacramento

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

by

John Glenn Keeling

FALL
2016
SPEKULATOR: GENETIC ALGORITHM CHORD PROGRESSION GENERATOR

A Project

by

John Glenn Keeling

Approved by:

__________________________________, Committee Chair
Dr. Scott Gordon

__________________________________, Date

__________________________________, Second Reader
Dr. Isaac Ghansah

__________________________________, Date
Student: John Glenn Keeling

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the project.

____________________________, Graduate Coordinator
Dr. Ying Jin

Department of Computer Science
Abstract

of

SPEKULATOR: GENETIC ALGORITHM CHORD PROGRESSION GENERATOR

by

John Glenn Keeling

Creative blocks in music, or perhaps any artful endeavor, are often due to imaginary limitations unknowingly imposed by the artist. Education or training in an art is comprised, in part, of rules, but ultimately, these rules are intended to be broken. “Learn the rules like a pro so that you can break them as an artist” (Picasso). This master’s project presents an application named SpeKculator. SpeKculator is a tool for helping the musician think outside the box, to help break creative blocks and facilitate the creation of music from sources influenced by, but sourced from outside, the artist. SpeKculator accomplishes this by accepting a musical objective and applying a Genetic Algorithm to a population of randomized chord progressions, the goal of which is to evolve the population to produce an individual chord progression that most closely meets the user-provided objective. The musical objective is comprised of two inputs: Color, which describes the musical spectrum from Darkness to Brightness, and Quality, which describes the musical spectrum from Dissonance to Consonance.

SpeKculator produces surprisingly good results. The generated chord progressions trend toward the provided objective and most certainly do not adhere to any rules of musical composition. An unintended benefit of SpeKculator is the challenge of performing the generated chord progressions, as the chord progressions include chords that may not be familiar to the user.
SpeKculator provides audio and visual representations of chord progressions. Audio playback is easy to digest and assess. Visual assessment is intended to be assessed with instrument in hand.

_______________________, Committee Chair
Dr. Scott Gordon

_______________________
Date
ACKNOWLEDGEMENTS

I would like to thank my family. My Grandfather Sergio Ricci and Grandmother Ida Ricci impressed upon me the importance of education. My Grandma Ida valued education and asked me about my coursework and projects every time we spoke. We discussed the SpeKculator project often. My Mother Eileen Landi for being unceasingly supportive, paying for my college when we were both without resources. My daughter’s Amanda and Elizabeth, their presence in the world has motivated me to strive to be better, part of which has been my educational pursuits.

I would like to thank my nutty musical friends: Ken Verhoevon, Michael Abinante, Adam Christenson, and Lob Instagon. Our musical endeavors have brought great joy to my life. You guys have taught me the importance of discarding musical rules. Special thanks to John “Horjo” Halicki for the moniker “speke”, a term which influenced the name of this project.

I would like to thank my Project Advisor, Dr. Scott Gordon, for his enthusiasm with the project and his ever-helpful suggestions, and Dr. Cui Zhang for her continuous focus on student success.
TABLE OF CONTENTS

Acknowledgements ........................................................................................................ vii
List of Tables .................................................................................................................. xi
List of Figures ................................................................................................................ xii

Chapter
1. INTRODUCTION ........................................................................................................... 1
   1.1 Motivation ........................................................................................................... 1
   1.2 Project Overview ................................................................................................. 2

2. BACKGROUND AND RELATED WORK ...................................................................... 3
   2.1 Background ......................................................................................................... 3
   2.2 Genetic Algorithms ............................................................................................ 3
   2.3 Music .................................................................................................................. 14
   2.4 MIDI ................................................................................................................... 18
   2.5 Related Work ..................................................................................................... 19

3. EVOLVING CHORD PROGRESSIONS WITH A GENETIC ALGORITHM ......................... 22
   3.1 Encoding and Decoding ...................................................................................... 22
   3.2 Selection and Crossover Operators .................................................................... 24
   3.3 Mutation Operator ............................................................................................. 26
   3.4 Elitism ............................................................................................................... 29
   3.5 Fitness Function .................................................................................................. 29

4. SPEKULATOR ............................................................................................................... 34
   4.1 Subsystems ......................................................................................................... 34
APPENDIX L: Test Results .................................................................................................................93
APPENDIX M: Open Issues.................................................................................................................97
Works Cited ........................................................................................................................................ 99
# LIST OF TABLES

<table>
<thead>
<tr>
<th>Tables</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Triad Chords</td>
<td>16</td>
</tr>
<tr>
<td>2. Sample of Standard Chord Progressions</td>
<td>17</td>
</tr>
<tr>
<td>3. Data Points Organized by Individual</td>
<td>31</td>
</tr>
<tr>
<td>4. Data Points Organized by Objective</td>
<td>32</td>
</tr>
<tr>
<td>5. Normalized Objective Values</td>
<td>32</td>
</tr>
<tr>
<td>6. MIDI Values</td>
<td>48</td>
</tr>
<tr>
<td>7. Association of Degree Name and Const Value</td>
<td>56</td>
</tr>
<tr>
<td>8. Test Results for Elitism 0, Crossover Rate 0.95, Mutation Rate 0.1</td>
<td>93</td>
</tr>
<tr>
<td>9. Test Results for Elitism 1, Crossover Rate 0.95, Mutation Rate 0.1</td>
<td>94</td>
</tr>
<tr>
<td>10. Test Results for Elitism 0, Crossover Rate 0.7, Mutation Rate 0.3</td>
<td>95</td>
</tr>
<tr>
<td>11. Test Results for Elitism 1, Crossover Rate 0.7, Mutation Rate 0.3</td>
<td>96</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>1</td>
<td>Genetic Algorithm Process Flow</td>
</tr>
<tr>
<td>2</td>
<td>Uniform Crossover</td>
</tr>
<tr>
<td>3</td>
<td>Single Point Crossover</td>
</tr>
<tr>
<td>4</td>
<td>Cut and Splice Crossover</td>
</tr>
<tr>
<td>5</td>
<td>Bit String Mutation</td>
</tr>
<tr>
<td>6</td>
<td>Local Minimums in a Fitness Landscape</td>
</tr>
<tr>
<td>7</td>
<td>Encoding and Decoding</td>
</tr>
<tr>
<td>8</td>
<td>Piano Keyboard</td>
</tr>
<tr>
<td>9</td>
<td>Intervals</td>
</tr>
<tr>
<td>10</td>
<td>iii-IV-V Chord Progression</td>
</tr>
<tr>
<td>11</td>
<td>Un-coded and Encoded Chord Progression</td>
</tr>
<tr>
<td>12</td>
<td>Cut and Splice Crossover of Chord Progressions</td>
</tr>
<tr>
<td>13</td>
<td>NoteType Mutation</td>
</tr>
<tr>
<td>14</td>
<td>ChordType Mutation</td>
</tr>
<tr>
<td>15</td>
<td>Assessing Individual Fitness Ranges</td>
</tr>
<tr>
<td>16</td>
<td>Fitness Landscape</td>
</tr>
<tr>
<td>17</td>
<td>SpeKculator GUI</td>
</tr>
<tr>
<td>18</td>
<td>GUI Tool Tip</td>
</tr>
<tr>
<td>19</td>
<td>Use of GUI Stop Button</td>
</tr>
<tr>
<td>20</td>
<td>Class Diagram of MIDI Playback Subsystem</td>
</tr>
<tr>
<td>21</td>
<td>Direct Execution of Classes SequenceCreator and MidiPlayer</td>
</tr>
</tbody>
</table>

xii
22. Class Diagram of Genetic Algorithm Subsystem .................................................. 43
23. Population Dump ................................................................................................... 44
24. Class Diagram of Musical Objects Subsystem ..................................................... 47
25. Circle of Fifths ..................................................................................................... 49
26. ChordDB Dump Command ................................................................................. 52
27. Executing Note Class Unit Tests ..................................................................... 53
28. Executing All Unit Tests .................................................................................... 54
29. SpeKculator.log showing unidentified chords ............................................. 55
Chapter 1

INTRODUCTION

1.1 Motivation

I have played guitar for many years. My understanding of music theory is decent. I have largely taught myself elements of music theory that concern Intervals, Scales, Modes, Chords, and their relationship with one another. The process of musical composition has always fascinated me. I have written songs born out of studying and applying theory; I have also written songs that appeared out of nowhere, where I’m mindlessly strumming my guitar, hear something that catches my ear, then build upon the idea. These haphazard songs are universally my favorite, and I have pursued unorthodox ways to attempt to coax them into existence. Prior to my computer science education and an understanding of modeling and simulation, I undertook rudimentary approaches such as scribbling chord references on scraps of paper, shuffling them in a hat, and then picking out and playing the selections.

I have been mindful of the possibly leveraging a computer program to generate random musical ideas, and it was in my graduate coursework that a project idea began to germinate. I had completed, in an Artificial Intelligence class, a project that modeled musical chords and scales and sought out associations between the two; the project was useful as reference tool, and demonstrated the ability to successful model musical objects and their relationship to one another. With the application of randomness, I realized that it would possible to replicate my prior rudimentary approach to chord progression generation with a computer program, and hence the SpeKculator project was conceived.
1.2 Project Overview

SpeKculator is a tool to assist in the effort of musical composition. SpeKculator provides a GUI (Graphical User Interface) that accepts a two-part musical objective, applies a Genetic Algorithm to evolve a population of randomized chord progressions, and presents visual and audio representation of the chord progression that most closely meets the user’s objective. The GUI is comprised of four panels, two that concern the Genetic Algorithm and two that concern musical input and output. The Genetic Algorithm panels allow a user to alter parameters that influence execution and to observe details of execution. The musical-related panels support the input of a two-part objective and output of the arrived at chord progression.

SpeKculator implementation is comprised of four main subsections: GUI, MIDI playback, Genetic Algorithm, and Musical Object Models. SpeKculator is written in Java. The GUI is written in Java Swing. The MIDI Playback leverages the javax.sound.midi package.

SpeKculator creates, at runtime, log file SpeKculator.log in the current-working directory of the application. SpeKculator.log contains log entries pertaining to execution. Level of log message is controllable by editing file SpeKculator.java, changing the value passed into setLevel () method of the logger_instance (see below), and recompiling.

```java
//logger_.setLevel ( Level.CONFIG );
logger_.setLevel( Level.SEVERE );
```
Chapter 2

BACKGROUND AND RELATED WORK

2.1 Background

SpeKculator applies a Genetic Algorithm to a population of chord progressions in an effort to evolve the population towards a user-supplied musical objective. Some understanding of the Genetic Algorithms and musical terminology will be helpful in understanding the application and this paper. SpeKculator Project Report is written in font Times New Roman 11pt. Literal terms and source code excerpts, including data types, variables, methods, pseudo-code, is written in font Courier New 9pt. A variable is sometimes presented with its datatype to provide further clarification, for example, \textit{NoteType} \texttt{root}. Book titles and points of emphasis are shown in \textit{italics}.

2.2 Genetic Algorithms

Genetic Algorithms belong to a larger set of algorithms known as evolutionary algorithms. Evolutionary algorithms are inspired by processes seen in nature; these processes include mate selection and biological reproduction, mutation, and adaptation. Some evolutionary algorithms exclusively use mutation. Evolutionary algorithms were developed during the inception of AI (Artificial Intelligence) in the 1950s when early computer scientists, in an attempt to program intelligence, analyzed and attempted to model processes seen in nature. These scientists were developing algorithms for problem domains for which existing computational techniques were not well suited. These domains included complex, vast, and incomplete data, and presented problems for which optimal solution algorithms were not yet known. “Fundamentally,
all evolutionary algorithms can be viewed as search algorithms which search through a set of possible solutions looking [for] the best --or ‘fittest’-- solution” [1].

A Genetic Algorithm (GA) is an algorithmic approach that models processes of evolution, including mate selection, reproduction, mutation, and survival of the fittest. Through cycles of reproduction and mutation, GA evolves a population of candidate solutions towards an objective. Wikipedia defines genetic algorithms as “a metaheuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA)” [2]. Genetic Algorithms are well suited for problem domains in which a sufficiently good, non-optimal, solution will suffice, or problems for which the optimal solution is computationally intractable. Travelling Salesman Problem [3] comes immediately to mind. The current general-purpose algorithm for obtaining an optimal solution for the Travelling Salesman Problem is to enumerate all possible solutions and select the entry whose path is the shortest. Performance of this brute force approach is $O(n!)$, where $n$ is the city count; factorial performance makes brute force an impractical approach. Genetic Algorithms in Java Basics [1] provides a GA implementation for solving the Traveling Salesman Problem.

Genetic Algorithms are inspired by processes seen in nature and thus leverage terms analogous to those of evolutionary science; the most commonly used terms are as follows:

- **Objective** is a goal towards which a population evolves. Objective may be regarded as the optimal solution to a problem.

- **Fitness** is a Genetic Algorithm Function that measures how well an Individual meets an objective.
• **Chromosome** is an individual’s genetic information. A chromosome is comprised of individual genes. Chromosome exists in two representations. Decoded chromosome is a representation of a solution to an objective. Encoded chromosome is a representation suitable for operation against by the Genetic Algorithm operators. Mapping of chromosomes between decoded and encoded representations is part of a Genetic Algorithm.

• **Individual** is a potential solution to the Genetic Algorithm objective. An Individual maintains genetic information, referred to as Chromosome, which is assessed for fitness against an objective. The most fit Individual at time of algorithm termination, is the Genetic Algorithm solution.

• **Population** is a collection of Individuals. A Population serves as initial input to the Genetic Algorithm, as well as the data set to which Genetic Algorithm Operators are iteratively applied.

• **Evaluation** is fitness assessment of each Individual in the Population. Evaluation post-condition is a fitness value associated with each individual. Evaluation may also include population fitness, which is the collective fitness of all individuals.

• **Crossover** is a Genetic Algorithm Operator that models biological reproduction. Crossover recombines existing genetic information from two Individuals into a new Individual. The new Individual is then added to the next Population. There are numerous techniques from Crossover.
• **Selection** is a Genetic Algorithm Operator that dictates the selection of Individuals for the Crossover operation.

• **Mutation** is a Genetic Algorithm Operator. Mutation models biological mutation. Mutation operates on a single Individual and introduces new genetic information into a Population.

• **Generation** is an occurrence of an iteration of a Population through Fitness assessment and application of the Genetic Algorithm operators. Generations are counted. A common terminate condition is a maximum number of generations.

A Population of individuals is generated prior to entry into the main GA iteration. This iteration is called an evolutionary cycle and it is in the evolutionary cycle, shown in Figure 1, that the population is operated on by the GA operators. The first Population, which serves as initial input to the Genetic Algorithm, is comprised of Individuals with randomized genetic information. The nature of the randomization is highly dependent upon problem domain. The Population must at all times be comprised of Individuals whose genetic information provides a valid solution for the problem. For example, the initial Population supplied to the Travelling Salesmen Problem is comprised of Individuals whose genetic information is a randomized sequence of visited cities; the sequence, through random, must not contain a duplicate city. The initial Population supplied to SpeKculator is comprised of chord progressions with a randomized number and sequence of chords. For the SpeKculator problem domain, a duplicated chord in a chord progression is permissible. The constraint that Individuals must always provide a valid solution to the problem is also required during execution of Crossover and Mutation operators.
The Crossover Operator blends genetic information of two individuals, creating a new Individual; this operation is analogous to biological reproduction. Different Crossover techniques exist, and the one chosen can be dependent upon the domain of the problem. For example, the Crossover technique chosen for The Traveling Salesman Problem must ensure that the resulting Individual, or offspring, does not include a duplicate city in its genetic information; Ordered Crossover [4] satisfies this requirement.

Uniform Crossover blends genetic information of two parents at a per gene level. A mixing ratio dictates the probability of an offspring’s gene coming from Parent 1 or Parent 2.
Figure 2 shows an example of Uniform Crossover with a uniformly distributed gene set. Such a gene set is possible outcome with a mixing ratio of 50%.

![Diagram of Uniform Crossover](image)

Figure 2: Uniform Crossover

Single Point Crossover includes a crossover point that demarcates genetic information of Parent 1 and Parent 2 at precisely the same location. The crossover point is random chosen. The offspring is comprised of genes from Parent 1 on one side of the demarcation point and genes from Parent 2 on the other side of the demarcation point. Unlike Uniform Crossover, Single Point Crossover operates on gene segments rather than individual genes. Figure 3 shows an example of Single Point Crossover.
Cut and Splice Crossover is similar to Single Point Crossover in that a crossover point is random chosen and applied. Cut and Splice is different in that each Parent has its own crossover point. Cut and Splice Crossover results in offspring whose genetic information may be of a different length than the Parents. SpeKculator uses Cut and Splice Crossover as it was desirable to have offspring whose genetic information, a chord progression, is potentially different in length from that of either parent. Figure 4 shows an example of Cut and Splice Crossover.
Selection Operator selects, from the Population, individual candidates for Crossover.

There are numerous different selection techniques, but the general outcome of selection is to favor individuals of higher fitness but still allow for the possibility of selecting less fit individuals. This outcome serves a Genetic Algorithm assumption that the combination genetic information from two highly fit individual will lead to an even fitter offspring [1]; the inclusion of less fit individuals helps to maintain diverse genetic information in next generation’s population.

Selection is dependent upon individual fitness, and therefore a population must be evaluated prior to Selection. Different selection techniques include Fitness Proportionate, Tournament, Truncation, Stochastic Universal Sampling, and Reward-Based. SpeKculator uses Tournament Selection.

Mutation Operator modifies the genetic information of a single Individual. It is through Mutation that new genetic information is introduced into a population. Crossover does not introduce new genetic material, and is instead a recombination of existing genetic information. Mutation operates on genes. A Chromosome is a sequence of genes. Mutation alters the value of a
single gene. Different Mutation techniques exist, and the one chosen is dependent upon how genetic information is encoded. For Chromosomes encoded into a Bit String (i.e. “1 0 0 1 0 0 1”), Mutation will flip the value of a single bit. Figure 5 shows bit string mutation. For Chromosomes encoded into an Integer String (i.e. “7 9 61 2 8 81 11”), Mutation may alter a specific value to its lower or upper range. Because Mutation introduces new genetic information into a population, it is necessary to ensure the result of the mutation is still a valid solution to the problem. For example, in The Traveling Salesman Problem, Mutation must not result in the occurrence of a duplicate city in an Individual’s genetic information. In SpeKculator, Mutation must not result in the occurrence of a non-musical object in an Individual’s genetic Information.

![Figure 5: Bit String Mutation](image)

Crossover and Mutation Operators are invoked at a given probability. These probabilities are configurable settings of the Genetic Algorithm and affect the probability of the operator’s occurrence during the evolutionary cycle. Probabilities given to Crossover and Mutation are called, respectively, Crossover Rate and Mutation Rate. Biological reproduction is responsible for passing favorable traits onto the future generation, and so Crossover Rates tend to be rather high; a common range for Crossover Rate is 80% to 95%. Biological mutation is a source of genetic
diversity and introduces new gene strains into a population. The probability of a mutation resulting in a favorable trait is remote, and so Mutation Rates tend to be low; a common range for Mutation Rate is 1% to 10%. Optimal Crossover and Mutation rates are dependent upon numerous factors and are thus difficult to assign apriori. It is necessary, in practice, to assess Genetic Algorithm performance while applying different rates.

Mutation is an important operation for addressing a condition known as Local Minimums. Crossover results in the population evolving towards an objective. This evolution, through moving towards the objective, can be moving along a path that is in the direction of the objective but ultimately blocked from ever attaining the objective. Mutation helps mitigate Local Minimums as the Mutation occurs without any disposition towards the objective; this can have the effect of backing out of a path that is blocked from ever attaining the objective. Figure 6 shows the occurrence of local minimums in a 2-d fitness landscape.

![Figure 6: Local Minimums in a Fitness Landscape](image_url)
Genetic Algorithm crossover works from the assumption that the combination of genetic information from two highly fit individuals will lead to an even fitter offspring [1]; mutation occurs without any such assumption. It is possible through crossover and mutation that the most fit individual in the population is altered to be less fit. Elitism is an optional GA optimization that addresses this issue. Elitism is a count that affects Crossover and Mutation operators by permitting a specific number of the most fit individuals, the Elitism Count, to bypass the possibility of modification by the GA Operators, and pass directly, without alteration, from the current generation’s population to the next generation population. Elitism count has the effect of maintaining the genetic information of most fit individuals across generations.

The Genetic Algorithm continues until a termination condition has been met. The criteria for termination can include numerous conditions. For problems in which the optimal solution is known, the termination condition can be the identification of an individual whose fitness meets the objective. For problems in which the optimal solution is not known, the termination condition can be an individual that meets some fitness threshold [1]. A common termination criterion is a limit on the number of permitted generations. Limiting the number of generations is important for problems in which the optimum solution is not known.

An Individual is a candidate solution to an objective. The solution is coded into an individual’s chromosome. A chromosome may not be in a representation suitable for operation by the GA operators. Encoding is the process of mapping a chromosome into a representation suitable for GA operations. Decoding is the process of mapping a chromosome into representation suitable for presentation as a solution. A common encoding is to represent a chromosome as a string of ones and zeros, i.e. “1001001”. Using chromosome representative of the Traveling Salesman Problem, Figure 7 shows the encoding of a sequence of visited cites into a bit string, and the decoding of a bit string into a sequence of visited cities.
2.3 Music

This section contains a brief description of music theory elements that pertain to SpeKculator. The fundamental building block in music is a note. A note is a musical object with a name and pitch; examples of notes are A, A#, B, C, Db. Some notes, though different in name, are of the same pitch. For example, C# (C Sharp) is the same pitch as Db (or D Flat). The difference in name is due to the musical key to which the notes belong. Some keys require notes be flattened, whereas some keys require notes to sharpened. There are twelve unique notes in western music. Twelve consecutive notes form one octave. An octave is the distance between a note and another note of half or double the same frequency. For example, note A above the middle C has a frequency of 440Hz. Octaves to this note would be A with frequency of 220Hz and 880Hz. Notes are best visualized on piano keys. Figure 8 shows a piano keyboard [7] with an identified octave.
A note has a singular pitch or frequency. An interval is two notes played simultaneously. Intervals are named according to the distance between the two notes. The unit of distance, in western music, is a semitone. Adjacent keys on a piano are one semitone apart. Interval whose notes are one semitone apart is a Minor Second, or m2; an interval whose notes are two semitones apart is Major Second, or M2. Figure 9 shows complete set of intervals with root note C.

Including in the set is unison interval, which is two identical notes, and the octave intervals, which is the same note at double the frequency.

Intervals have a musical color and quality. Color and quality are somewhat subjective terms, but there are formal ways to assess their value. Musical color refers to mood or emotive value. The color spectrum is viewed from dark and somber to bright and uplifting. Quality is a
measure of harmoniousness and is viewed on a spectrum of dissonance to consonance [5]. A dissonant quality is harsh and jarring, whereas a consonant quality is harmonious and pleasing.

A chord is a collection of three or more notes played simultaneously. Chords are classified by note count and harmonic quality and can be assessed by viewing the chord as a set of intervals. Triads for example are chords comprised of three specific notes. The three notes include a root, third, and fifth. Triads are identified by the distance between the root and third note, and root and fifth note; these two distances are intervals. Table 1 shows four triads along with the composite intervals.

<table>
<thead>
<tr>
<th>Triad</th>
<th>3rd</th>
<th>5th</th>
</tr>
</thead>
<tbody>
<tr>
<td>Major</td>
<td>major</td>
<td>major</td>
</tr>
<tr>
<td>Minor</td>
<td>minor</td>
<td>major</td>
</tr>
<tr>
<td>Diminished</td>
<td>minor</td>
<td>minor</td>
</tr>
<tr>
<td>Augmented</td>
<td>major</td>
<td>major</td>
</tr>
</tbody>
</table>

Table 1: Triad Chords

Because an Interval is defined by the relationship between any two notes, this assessment can be made without regard to the root note of the chord. For example, a CMaj7 chord has the same set of intervals as a DMaj7; these intervals are major 3rd (or M3), major 5th (or M5), and major 7th (or M7). Because a chord can be viewed as a composite set of Intervals, a chord’s color and quality can be derived by assessing collective color and quality of the chord’s composite intervals.

A chord progression is a sequence of chords. Like chords, chord progressions are classified by a count and harmonic content. Chord progression are notated by the included chord
names or with Roman numerals, where the Roman numeral indicates the relationship of the chord’s root to the chord progression’s key; lowercase Roman numeral indicates the chord is a minor; a chord is minor if it includes the minor 3rd interval. Figure 10 shows an iii-IV-V chord progression; this progression is comprised of triads.

A song that sounds familiar is likely due its use of a standard chord progression. There exist numerous standard chord progressions [10]. Table 2 shows a small sampling of the available standard chord progressions.

<table>
<thead>
<tr>
<th>Name</th>
<th>Chord Progression</th>
</tr>
</thead>
<tbody>
<tr>
<td>50’s progression</td>
<td>I-vi-VI-V</td>
</tr>
<tr>
<td>Andalusian Cadence</td>
<td>iv-III-bII-I</td>
</tr>
<tr>
<td>Backdoor progression</td>
<td>ii-bVII7-I</td>
</tr>
<tr>
<td>Eight-bar-blues</td>
<td>I-V-IV-IV-I-V-I-V</td>
</tr>
</tbody>
</table>

Table 2: Sample of Standard Chord Progressions
Color and quality assessment of a chord progression is derived by assessing collective color and quality of the composite chords. Intervals and chords are set of notes played simultaneously, whereas a chord progression is a set of chord played in sequence. Some chords flow very naturally from one to another, whereas others do not; this is due to the presence and absence of melodic lines that bridge chords. Chord progression quality is affected by the nature of chord changes within a chord progression.

2.4 MIDI

MIDI (Musical Instrument Digital Interface) is a set of standards in support of the creation and exchange of digital music information between devices. The MIDI standard defines a protocol for message exchange, message format, device types, and connection types. MIDI is a very broad and deep subject, which this section on touches upon. An understanding of MIDI terminology will be helpful in understanding the SpeKculator MIDI playback source code but is not necessary for use of SpeKculator.

A MIDI system is comprised of MIDI devices exchanging MIDI messages. MIDI devices are classified by whether they generate or receive MIDI messages. Some devices are able to both generate and receive MIDI messages. There are numerous MIDI message, organized by classification [11]. MIDI message types include Channel Voice, Channel Mode, System Common, and System Real-Time. MIDI message contain both device and musical information. Device information includes the MIDI channel on which the music information is directed; music information values for pitch, velocity (volume), effect such as vibrato or tremolo, and timing related information. MIDI standard allows 16 MIDI Channels to be simultaneously transmitted over a MIDI cable. This capacity permits a single MIDI device to generate MIDI messages for 16 other MIDI devices. The followings list identifies frequently used MIDI terms:
• **MIDI Event** is a MIDI message that includes timing information.

• **Track** is a collection of MIDI Events.

• **Sequence** is a collection of MIDI Tracks. Sequence is also referred to as a MIDI Schedule.

• **Controller** is a MIDI device that generates MIDI messages. A MIDI Controller is also referred to as a Transmitter.

• **Synthesizer** is a MIDI device that receives MIDI messages. A MIDI Synthesizers is also referred to as Receiver.

• **Sequencer** is a MIDI device that both generates and receives MIDI messages.

2.5 Related Work

I have researched the availability of existing applications that apply some notion of randomization to the generation of musical passages. I did not find an application that supports user-directed generation of musical passages. The focus of related applications is on learning chords and ear training. Reviewed applications include AutoChord [12], DrumBot [13], Interactive Songwriting Chord Progression Generator [14], and Chord Progression Generator [15]. A brief description of these applications is provided below.
AutoChords is an online application that presents and plays chord progressions. The user selects song qualities under the heading of Feel and Key. Feel and Key parameters include Key, Instrument, and Name. Name is an intriguing feature that included a pull down menu with choices ranging from Cliché to Rebellious. I played with this application for some time, and it seems geared toward a beginner who is attempting to learn chord changes and standard chord progressions. Generated chord progressions were limited to two chord types: Major and Minor. There was a Randomize button that presented randomized all possible settings, including Key, Instrument and Name.

Drumbot is an online application catered to guitarists. DrumBot permits the user to drag and drop chords into a sequence. The duration and strum pattern of each chord is configurable. Chord progression BPM (Beats Per Minute) field adjusts the playback tempo. DrumBot includes a GUI slider named “Dice Craziness”. It was not clear, from documentation or audio playback, how movement of Dice Craziness slider affects chord selection. Drumbot would be useful for developing song ideas but does not itself auto generate chord progressions; generated chord progressions are completely defined by the user.

Interactive Chord Progression Generator is an online tool that assists in organizing and documenting chord progressions. Chord progressions are represented as chord transitions in a state transition diagram. The user enters in all chords and transitions; I did not find a feature that generates chord progressions. The charted chords help to visualize the relationship between the chords, for which I see some value.

Chord Progression Generator is an online application catered to guitarists. Similar to AutoChords, Chord Progression Generator application presents canned chord progressions, the purpose of which is to teach beginners how to play chords. My review of this application is
strictly based upon the webpage layout; I was unable to play any of the presented chord progressions.

Chord Progression Generator is an online application catered to guitarists. Similar to AutoChords, Chord Progression Generator application presents canned chord progressions, the purpose of which is to allow users to create a song against which to play along. The application allows the user to select a musical genre, a Key, a disposition (happy, sad), and a speed. The parameters then translate to an entire song that can allegedly be performed by selecting the Play button; the Play button did not work for the presented chord progressions.
Chapter 3

E VOLVING CHORD PROGRESSIONS WITH A GENETIC ALGORITHM

3.1 Encoding and Decoding

An individual is a candidate solution to the objective presented to a genetic algorithm. In SpeKculator, an individual is a chord progression. The chord progression is the individual’s chromosome, or genetic information. Chord progression is an instance of class ChordProgression. ChordProgression defines a chord progression as an array of Chord instances. In the course of GA execution, a ChordProgression instance is mapped between two representations: un-coded and encoded. An un-coded representation is the array of chord instances. The un-coded representation is offered, in visual and audio, as a solution the objective and is comprised as set of chord names.

SpeKculator Crossover and Mutation operators operate on an encoded chord progression. An encoded chord progression is represented as a 2-dimensional integer array. The 2-dimensional array is an array of encoded chord instances. Figure 11 shows an un-coded and encoded representation of a chord progression. The un-coded chord progression is shown vertically, with an arrow per chord, to illustrate that each row of the encoded chord progression is an encoded chord.
A chord progression is encoded by iterating over its composite chords, encoding each chord, and added the encoded chord as a row in the 2-diminsional array. The encoding source code is provided in Appendix A.

Chord instance is encoded by parent method `MusicalObject.Encode()` into an integer array. A Chord instance has two data fields that are encoded: `NoteType root_` and `long degrees_`; both fields are inherited from parent class `MusicalObject`. `NoteType root_` defines the chord’s root note; `long degrees_` is a bit field that identifies each of the chords composite intervals. `NoteType` is an enumeration of all possible notes. The ordinal value of `NoteType root_` is placed in array index[0]. `degrees_` bit field is mapped into a bit array by static method `Utils.DegreesToDegreeArray()`. The resulting bit array is copied into array index[1-N]. The entire chord is at this point encoded and is added at the appropriate row index of the 2-dimensional chord progression encoding.

An encoded chord progression is decoded by the inverse of actions taken during encoding. The encoded 2-dimensionsal integer array is iterated over the row index. Each entry, an integer array, represents an encoded chord. Array index[0] is valued back into a `NoteType`
instance; array index[1-N] is mapped back into a long bit field by static method

`Utils.DegreeArrayToDegrees()`. The degrees value is then mapped into a `ChordType` instance
by static method `ChordType.DegreesToType()`. With the chord’s NoteType and ChordType
known, the decoded representation of the chord is created by passing these values into constructor

`Chord(NoteType, ChordType)`. The entire chord is at this point decoded and is added at the
appropriate entry in the chord progression Chord array. The decoding source code is provided in
Appendix B.

### 3.2 Selection and Crossover Operators

SpeKculator crossover uses the Cut and Splice technique. Cut and Splice was chosen as a
crossover technique for its property of creating offspring whose genetic information may be of
different length than that of either parent. Length of genetic information corresponds to a chord
count in a chord progression. Furthermore, the initial population of randomized chord
progressions includes progressions of different length. The randomization creates chord
progressions whose count ranges from three to eight chords. Cut and Splice Crossover is suitable
for parents whose genetic information is of different length.

Parent candidates for crossover are selected through two different techniques. `parent1`
selection technique is adapted from book *Genetic Algorithm in Java Basics* [1]. The book does
not identify the name of this technique. The technique, described in further detail below, results in
a maximized opportunity for genetic diversity present in the current population, to be passed onto
the next generation. This selection approach will be referred to as Diversity Selection. `parent2`
selection uses Tournament Selection.

Selection occurs at the Population level in method `GeneticAlgorithm.Crossover()`. This
method iterates over the entire population, tracking the iteration count in `populationIndex`. 
The Population is sorted on Fitness, with the most fit Individual at population index[0] and least fit at population index[population size - 1]. Diversity Selection assigns into parent1 the individual at population index[populationIndex]. Because the entire population is iterated over, Diversity selection helps to ensure that a broad collection of the current population’s genetic information has an opportunity to be passed onto the next generation. parent2 is selected by Tournament Selection. Tournament selection randomly selecting a fixed number of individuals from a population, sorts the selected individuals on fitness, and returns the most fit Individual.

SpeKculator had first applied Fitness Proportionate Selection. Fitness Proportionate Selection, also known as Roulette Wheel Selection, applies a higher probability to selecting individuals with larger fitness and lower probability to a lower fitness. This selection approach was determined, in testing, to be inappropriate as Fitness Proportionate favors intrinsically larger fitness values; SpeKculator regards a smaller fitness value as more fit than a larger fitness value; Fitness Proportionate resulted in favoring the least fit for selection. Fitness Proportionate was eventually replaced with Tournament Selection, which is dependent upon a Population sorted on Fitness, in whatever sorting order is applied.

Actual application of crossover occurs at an Individual level once parent selection is complete and crossover conditions have been satisfied. Individual crossover is implemented in static method Individual.Crossover(parent1, parent2). This method applies Cut and Splice crossover technique and returns an instance of Individual. Figure 12 shows Cut and Splice technique as applied to chord progressions. Cut and Splice technique sets a randomly chosen demarcation point within each parent1 and parent2 chord progression. The left side of parent1 demarcation is added to the offspring, while the right side parent2 demarcation is added to the offspring. Implementation of Selection and Crossover operators is provided in Appendix C.
The occurrence of crossover is dependent upon two conditions: the crossover rate is met, and, the crossover candidates are not subject to Elitism. Crossover post-condition is a new population of the same size as population passed into the crossover operator. In the event crossover does not occur, parent1, the parent chosen by Diversity Selection, is passed, unaltered, from the current to the next population.

3.3 Mutation Operator

An individual’s genetic information, or chromosome, is a chord progression. A chord progression is comprised of chords, and so a chord in a chord progression is analogous to a gene in a chromosome. Chord progression crossover recombines chords from two chord progressions into a new chord progression. Individual chords are not altered, and so crossover operates at a chromosome-level. Chord progression mutation alters a single chord within a chord progression. Mutation operates on a gene-level. Mutation alters composite material of a chord, making it necessary to ensure the mutated gene is still a valid.

Figure 12: Cut and Splice Crossover of Chord Progression
Mutate operates on a chord progression that has been encoded into a 2-dimensional integer array. Each row of the 2-dimensional array holds an encoded chord. A chord is comprised of two distinct fields that are encoded differently. The chord key is a NoteType enumeration that can hold 22 distinct values; chord key is encoded into index[0] of the chord encoding. The chord type is mapped from a bit field into a bit array and occupies index[1-N] of the chord encoding. Mutation selects a random row and column of the encoded chord progression. The row/column values are passed into Chromosome instance method `ToggleDNA()`. `ToggleDNA()` discerns whether the mutation affects NoteType or ChordType. If NoteType, then a new random entry in the NoteType enumeration is entered into the row/column of the encoding. Figure 13 shows a NoteType mutation. If ChordType, then the bit at row/column is flipped. Figure 14 shows a ChordType mutation.

![Figure 13: NoteType Mutation](image)
ChordType Mutation can result in an encoding that does not decode into a valid gene. After a gene has been mutated with ToggleDNA(), a call is made to Chromosome method ValidGene(). ValidGene() attempts to decode the mutated gene into a chord instance. If this fails, then the entire mutation is a backed out and reattempted with a new set of randomly chosen row and column values. The maximum allowable attempts at mutation is capped at 10,000. Failure to mutate after 10,000 attempts results in an error message written to SpeKculator log.

The occurrence of mutation is dependent upon two conditions: the mutation rate is met, and, the mutation candidate is not subject to Elitism. Mutation post-condition is a new population of the same size as population passed into the mutation operator. In the event mutation does not occur, the individual candidate for mutation is passed, unaltered, from the current to the next population. Implementation of Mutation operator is provided in Appendix D.
3.4 Elitism

Crossover and Mutation Operator implementations include a provision for Elitism. Elitism allows a number of individuals, configurable through the SpeKculator GUI, to pass from the current generation to the next generation, without modification by Crossover and Mutation operators. Candidates for Crossover and Mutation and chosen, iteratively, from a population that has been shorted on fitness. Implementation of elitism is a comparison check between the iteration and elitism count; crossover and mutation is permitted to occur for iteration counts that exceed the elitism count. This has the effect of forwarding, without alteration, the fittest individuals from the current to next generation. Elitism optimization occurs in methods GeneticAlgorithm.Crossover() and GeneticAlgorithm.Mutate(). Crossover implementation is provided in Appendix C. Mutate implementation is provided in Appendix D.

3.5 Fitness Function

SpeKculator Fitness Function includes complexities due to the subjective nature of the objective, and that the objective is multi-valued. SpeKculator Fitness function is best described in terms of fitness evaluation, fitness normalization and fitness assessment. Fitness evaluation assigns Color and Quality values to a given chord progression. Chord progression Color is the accumulated color of each chord in the progression. Chord progression quality is a measure of the harmony between each chord in the chord progression.

Fitness evaluation of chord color assessment is somewhat heuristic. Zero is considered neutral. Positive values are in the direction of brightness, and negative values are in the direction of darkness sound. The assignment of darkness and brightness encapsulates the notion of color as described in a Coursera class on Jazz Improvisation [16]. In the class, scales were ordered from
brightest to darkest based upon the number of augmented/major notes to the number of flattened/minor notes.

Fitness evaluation of chord progression quality approximates the harmonic relationship between each adjacent chord in the progression. Zero is considered neutral. Positive values are in the direction of consonance, and negative values are in the direction of dissonance. Quality assessment entails iterating over every possible musical degree. If the degree exists between the two chords, quality is incremented; if the degree exists between one chord and not the other, quality decremented; quality is not affected if the degree does not exist in either chord. Early testing demonstrated a tendency for objectives with dissonant quality to result in numerous occurrences of the same dissonant chord but with different root note. This is a highly undesirable outcome; to mitigate the occurrence of duplicate chord types, the fitness function includes a small reward for a highly diverse chord mix. A highly diverse chord mix is (unique chords/chord count) == 1.0.

Fitness normalization assigns a normalization value to each of the objective values such that the individual fitness more closely matches the fitness space of the objective. This was accomplished by running numerous iterations of Population class test in which 130,000 random individuals were created, evaluated, and measured. Test identifies and reports the individual with lowest combined color and quality value and highest combined color and quality value. Reported information includes four data points: color & quality of individual with lowest combined color + quality, and color and quality of individual with highest combined color and quality. Figure 15 shows execution of the test.
Figure 15: Assessing Individual Fitness Ranges

The four data points are organized by individual is shown in table 3. The same data, organized by objective shown in table 4. When in objective value form, the range of values is established. This range was used to establish the objective range presented to the user in the GUI.

<table>
<thead>
<tr>
<th></th>
<th>Color+Quality</th>
<th>Color</th>
<th>Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Lowest Individual</strong></td>
<td>-65</td>
<td>-16</td>
<td>-49</td>
</tr>
<tr>
<td><strong>Highest Individual</strong></td>
<td>56</td>
<td>41</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 3: Data Points Organized by Individual
The SpeKculator GUI presents an equivalent objective range of -30 to 30 for both color and quality. To more closely map objective fitness ranges to individual fitness ranges, normalization values are added to both color and quality objective. Table 5 shows the presented color and quality ranges, the normalized color and quality ranges, and the values gathered through testing the evaluated population. With normalization, objective fitness landscape is kept very close to the individual fitness landscape.

<table>
<thead>
<tr>
<th>Color Normalization</th>
<th>Low</th>
<th>High</th>
<th>Quality Normalization</th>
<th>Low</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Input</td>
<td>-30</td>
<td>30</td>
<td>Objective Input</td>
<td>-30</td>
<td>30</td>
</tr>
<tr>
<td>Objective Normalized (+13)</td>
<td>-17</td>
<td>43</td>
<td>Objective Normalized (-17)</td>
<td>-47</td>
<td>13</td>
</tr>
<tr>
<td>Individual Fitness Range</td>
<td>-16</td>
<td>41</td>
<td>Individual Fitness Range</td>
<td>-49</td>
<td>15</td>
</tr>
</tbody>
</table>

Table 5: Normalized Objective Values

Fitness assessment is the ranking of an individual's assigned fitness against an objective. SpeKculator's fitness landscape is best viewed as a 2-d coordinate system where one axis representing color and the other axis quality. Fitness assessment is made by measuring the
distance between the objective point in the landscape and the individual fitness point in the landscape. Pythagorean Theorem is used to assess the distance between objective and individual assessment points. The most fit individual has the shortest distance between its location and the objective's location in the fitness landscape. Figure 16 shows placement of objective and individual points on a fitness landscape and individual fitness values. Fitness Function implementation is provided in Appendix E.

Figure 16: Fitness Landscape
4.1 Subsystems

SpeKculator design is comprised of four main subsections: GUI, MIDI Playback, Genetic Algorithm, and Musical Objects. SpeKculator includes, in addition to feature supporting subsections, a class that assists in unit test creation and execution, and a class that unifies command-line argument processing. Numerous classes implement a local main() method; local main allows for class specific unit and system test execution.

4.2 GUI

SpeKculator is a tool for songwriters. As such, SpeKculator GUI (Graphical User Interface) is designed to be intuitive and not require knowledge or configuration of the underlying genetic algorithm. SpeKculator GUI is written in native Java using Java Swing. Java Swing is part of Java’s Foundation Classes (JFC) and is regarded as component-based GUI framework. The fundamental component in Java Swing is class Component (java.awt.Component). Component is an abstract super class which under are defined classes for collecting user input and classes for containing other components. GUI items that serve as sources of user input are called Widgets; widgets include buttons, text fields, sliders and menus. Swing JFrame and JPanel are Components into which widgets and other components are contained. JFrames and JPanels have Layout Managers that control the location and size of contained components. Layout Managers also control the movement and resize of components when the GUI is resized.

SpeKculator GUI is implemented in class SpeKculatorGUI. SpeKculatorGUI inherits from class javax.swing.JFrame. SpeKculatorGUI Layout Manager is GridBagLayout. GridBagLayout is among the more flexible, thus complex, of the available Layout Managers. SpeKculatorGUI
contains four JPanels. These JPanels are Configuration, Objective, Status, and Output. Each of these JPanels also use GridBagLayout. Figure 17 shows the SpeKculator GUI.

SpeKculator GUI and application logic is kept separate by use of Java SwingWorker (javax.swing.SwingWorker) [17]. SwingWorker provides support for launching, from the GUI Event Dispatch Thread, long running application logic in a background thread. SpeKculator class GeneticAlgorithmWorker extends SwingWorker and maintains all Genetic Algorithm logic.

Figure 17: SpeKculator GUI

Configuration Panel presents and allows user-modification of Generic Algorithm configuration settings; this panel contains five JTextFields and a JButton. The JTextFields present default values and accept user-input. Inputted values are range checked, as not all values valid. Each JTextField has a FocusListener and tool tip. The FocusListener implements the FocusLost() method, which when executed will validate input and present an error message if necessary. Implementation of Population Size JTextField is provided in Appendix F. The tool tip presents
range of acceptable values for the selected JTextField. Tool tip for the Mutation JTextField is shown in Figure 18. Configuration panel contains a single JButton named Reset. The Reset Button is activated when a default value has been altered. Clicking the Reset Button will reset all values to their default setting.

![Configuration Panel](image)

**Figure 18: Tool Tip**

Objective Panel presents Genetic Algorithm Objective settings and controls for starting, stopping, and resuming execution of the Genetic Algorithm; this panel contains two JTextFields, two JSliders and three JButtons. SpeKculator Objective is two-part: Color and Quality. Color JSlider assigns a color objective value while the Quality JSlider assigns a quality objective value. Both JSliders assign color and quality values within range \([-30, 30]\). Each JSlider has a registeredChangeListener that invokes when the user selects and slides the JSlider. JSlider value is shown in a JTextField associated with each JSlider. Three JButtons control the state of Genetic
Algorithm execution. Start button initiates, Stop button terminates, and Resume button resumes execution. Start Button source code is provided in Appendix G. Clicking the Start Button instantiates class GeneticAlgorithmWorker, enables the Stop Button, and invokes method GeneticAlgorithmWorker.execute(). Execute() method invokes method SwingWorker doInBackground(), which launches a background thread then immediately returns control to the GUI Event Dispatch Thread; at this point, there are two threads in execution: the GUI Event Thread Dispatch and SwingWorker. The GeneticAlgorithm executes in the SwingWorker method doInBackground(). Implementation of SwingWorker.doInBackground() is provided in Appendix H. Stop Button terminates execution of GeneticAlgorithm and writes to Output Panel JTextField ChordProgression, the current Fittest Individual. Screen shot showing use of the Stop Button is provided in Figure 19. The screen shows Chord Progression Field populated with the genetic information of the current most fit Individual, the Status Panel Generation is equal to 4, indicating the generation in which execution was halted, and Start and Resume Button have been activated. Clicking the Start Button will start the Genetic Algorithm from Generation 1; clicking the Resume Button will resume Genetic Algorithm starting from generation 4.
Status Panel presents real-time status of the executing Genetic Algorithm. Status Panel includes three read-only JTextFields. Generation Field displays the current generation. Ave.Individual Fitness displays the average Individual Fitness of the entire Population. Fittest Individual displays the current fittest individual. Real time update of these is facilitated through a call to SwingWorker.publish() method in the SwingWorker.doInBackground(). Average individual fitness is an indication of general Population evolution and is thus an extremely valuable indicator of the Genetic Algorithm’s progress. It through monitoring of the average individual fitness that Local Minimums are identified.

Output Panel presents the Genetic Algorithm result. Output panel includes JTextField Chord Progression for presenting a visual representation of a chord progression, and JButton Play for presenting an audio representation of a chord progression. Clicking the Play Button will pass the genetic chord progression to the MIDI subsystem for playback.

Figure 19: Use of GUI Stop Button
4.3 MIDI Playback

SpeKculator MIDI subsystem supports creation and playback of MIDI Sequences. Figure 20 shows a UML diagram of the MIDI subsystem. MIDI subsystem is comprised of three classes: SpeKculatorPlayer, SequenceCreator, and MidiPlayer. The MIDI subsystem leverages Java classes from Package javax.sound.midi.*. SpeKculatorPlayer is a high-level class that contains instances of SequenceCreator and MidiPlayer. SpeKculatorPlayer provides the interface used by SpeKculatorGUI to play chord progressions. SpeKculatorPlayer includes overloaded method `Play()` that accepts musical objects of type Scale, Chord, and ChordProgression. Musical object argument passed into Play() is transitioned into a set of MIDI Events, collected into a MIDI Sequence, then submitted for playback to a MIDI Receiver. The interface between SpeKculatorGUI Play Button MIDI Subsystem is provided in Appendix I.

![Figure 20: Class Diagram of MIDI Playback Subsystem](image-url)
Class SequenceCreator creates MIDI Sequences. Specifically, SequenceCreator creates an object of type javax.sound.midi.Sequence. SequenceCreator constructor accepts MIDI related arguments for divisor, velocity, and resolution and instantiates a MIDI Sequence of type javax.sound.midi.Sequence and MIDI Track of type javax.sound.midi.Track. Method AddEvent() creates and populates a MIDI Event of type javax.sound.midi.ShortMessage. The ShortMessage instance added to the MIDI Track. Method GetSequence() returns a reference to the created MIDI Sequence.

Class MidiPlayer provides a rudimentary MIDI Player. MidiPlayer implementation was developed by studying Java sound package tutorial [18] and a repository of Java MIDI utilities [19]. MidiPlayer is general purpose in that it can play MIDI Sequences that contain a single channel. SpeKculator created MIDI Sequences are all single channel (channel == 0). MidiPlayer contains numerous references to classes defined in Java Package javax.sound.midi. These classes include Sequence, Sequencer, Synthesizer, Transmitter, and Receiver. Sequence data field is a handle to a passed in Sequence object. All other objects concern the acquisition and configuration of computer’s audio interface. The main access point for acquiring audio resources is static class MidiSystem (javax.sound.midi.MidiSystem). Java SE Documentation defines a MidiSystem class as follows: “The MidiSystem class provides access to the installed MIDI system resources, including devices such as synthesizers, sequencers, and MIDI input and output ports. A typical simple MIDI application might begin by invoking one or more MidiSystem methods to learn what devices are installed and to obtain the ones needed in that application” [20]. Synthesizer object is initialized with call to MidiSystem.getSynthesizer(). Sequencer object initialized with call to MidiSystem.getSequencer(). MidiPlayer source code is provided in Appendix J; command-line option and volume-setting routines are omitted for space considerations. The general use and processing of MidiPlayer is as follows:
1. MidiPlayer is instantiated with no-arg constructor.
   a. Synthesizer and Sequences are initialized with calls to static class
   b. MidiSystem.Transmitter data member is initialized with call to
      sequencer.getTransmitter().
   c. A MetaEventListener is registered with the sequencer instance. The listener
      listens for an end-of-track marker. Listener action is to close instances of
      sequencer and synthesizer instances.

2. MidiPlayer.Initialize( Sequence sequence )
   a. Sequence data member is initialized with sequence argument
   b. Sequencer instance is opened and is given reference to sequence object

3. MidiPlayer.Play()
   a. SynthReceiver object is initialized with call to synthesizer.getReceiver()
   b. Transmitter object is given reference to SynthReceiver instance
   c. Sequencer is started.

Classes SequenceCreator and MidiPlayer implement a main() method in support of
SpeKculator development and testing. Class SequenceCreator accepts command-line arguments
testType=<scale||chord> midiFile=<filename.mid>, and velocity=<integer>. Value passed to
testType= results in a random musical object of that type which is rendered into MIDI format
and both played through MidiPlayer, and written, in MIDI file format, to the filename given to
option midiFile=. Class MidiPlayer accepts command-line argument midiFile=<filename.mid>. MidiPlayer will provide audio feedback of the provided MIDI file. Figure 5.1 shows a screenshot
of SequenceCreator creating a testType=scale MIDI file, which is then played by MidiPlayer.
4.4 Genetic Algorithm

SpeKculator GA (Genetic Algorithm) subsystem supports evolving a population of randomly generated chord progressions towards a user provided, multi-part, objective. The GA subsystem is comprised of seven classes: GeneticAlgorithmWorker, GeneticAlgorithm, Population, Individual, Chromosome, Objective, and Chord. Figure 22 shows a class diagram of the GA subsystem.

GeneticAlgorithmWorker facilitates the separation of GUI and applications, and is an extension of javax.swing.SwingWorker. “SwingWorker is designed for situations where you need to have a long running task run in a background thread and provide updates to the UI either when done, or while processing. Subclasses of SwingWorker must implement the doInBackground() method to perform the background computation” [17]. GeneticAlgorithmWorker maintains, in a containment relationship, instances of class GeneticAlgorithm, Population, and Objective. The overall GA flow and iteration, shown in Figure 21: Direct Execution of Classes SequenceCreator and MidiPlayer
Figure 1, is implemented in method `doInBackground()`. Implementation of `doInBackground()` is provided in Appendix H. The GA iteration continues until termination condition is met or user halts execution by clicking GUI Stop Button.

Figure 22: Class Diagram of Genetic Algorithm Subsystem

Class GeneticAlgorithm contains methods that apply Selection, Crossover and Mutation Operators, and a Fitness Function, to a Population. Class Individual contains methods that apply Crossover and Mutation to an Individual. Both GeneticAlgorithmWorker and GeneticAlgorithm contain an Objective instance. The GeneticAlgorithmWorker Objective instance is instantiated with color and quality values collected from the GUI; GeneticAlgorithmWorker maintains an
Objective copy in support of GA Stop and Resume functionality. GeneticAlgorithm Objective instance is the instance referenced by the Fitness Function.

Population Class abstracts a collection of Individuals. GeneticAlgorithm Crossover, Mutation, and Selection Operators operator on a Population instance. Population has private data fields for maintaining an array of Individual instances, population size, and a cumulative fitness value. A Population instance can be modified by public methods AddIndividual(Individual) and SetIndividual(int offset, Individual). Method Shuffle() randomizes the population order; population is shuffled is by the Selection Operator as part of Tournament Selection.

Population can be queried with methods GetIndividual(int offset), GetFittest(int offset), and Dump(). GetFittest() sorts Population on fitness prior to returning Individual at offset. Dump() method dumps the entire Population contents to the console. Figure 23 shows use of and output from the Population dump feature.

Figure 23: Population Dump
Population dump is a valuable debugging and testing feature as it includes detailed information on each Individual in the population. This information includes the Individual’s genetic information (chord progression) and the values color and quality values which are assessed in the Fitness Function.

Individual Class abstracts a single candidate solution for the GA objective. Individual has private data fields maintaining instances of ChordProgression, Chromosome and double fitness. The majority of public methods are wrappers that provide access to internal chromosome instance methods; these methods include GetGene(int index), SetGene(int index, int[] gene), and AddGene(int[] gene). ChordProgression maintains un-coded representation of a solution whereas Chromosome is an encoded representation of a solution. Individual maintains data fields for each, and so code exists to ensure both instances are, at all times, representing an identical solution. Source code for individual.SetGene(), provided below, demonstrations how chord progression and chromosome fields are kept in sync by updating field progression_ with decoded chromosome_ when chromosome has been altered.

```java
public boolean SetGene( int chromosome_index, int[] gene ) {
    boolean setGene = chromosome_.SetGene( chromosome_index, gene );
    if( setGene == true )  // chromosome_ modified, reset chordProgression_
    {
        chordProgression_ = ChordProgression.Decode( chromosome_ );
    }
    return setGene;
}
```

GeneticAlgorithm class implements Crossover and Mutation Operators that operate on a Population. These operators work by calling Individual `Crossover()` and `Mutate()` methods on each individual instance in the population. Sections 3.2 Crossover and 3.3 Mutation cover details of Individual class Crossover and Mutate.

Chromosome class abstracts the genetic information of an Individual instance. Chromosome in representation of a candidate solution encoded into a 2-dimensional int array.

4.5 Musical Objects

Musical Object subsystem is a general-purpose framework that contains numerous classes in support of modeling musical objects and their relationships to one another. Musical Objects subsystem is comprised of the following classes: NoteType, Note, Key, Interval, MusicalObject, Chord, ChordDB, ChordProgression, Scale, ScaleDB, and Mode. Scale, ScaleDB, and Mode classes are functionally operational but do not currently contribute to evolving chord progressions through a GA. Figure 24 shows a class diagram of the Musical Object Subsystem.

Musical Object classes include numerous methods that, while important, are not strictly pertinent to main SpeKculator objective of evolving chord progressions; such methods include `toString()`, `hashCode()`, `get()/set()` pairs for private data fields, and many others; these methods are not mentioned nor is their source code included in any Appendix. Source code for the Musical Objects Subsystem is provided in Appendix K.
NoteType is an enumeration that includes entries for all possible notes. Java enumerations are classes and can have associated constructors and methods. NoteType has a single data field, MIDI_value_, for maintaining the note’s MIDI value [21]. Assignment of MIDI value occurs in NoteType declaration. MIDI values chosen for NoteType enumeration where those within the range of middle C. Middle C MIDI value is 60. Table 6 shows the MIDI values associated with each note. Notes with a different name but of the same pitch have the same MIDI number. The MIDI number identifies the pitch, or frequency, of a note when played on MIDI device. Table 6 also shows the representation of octaves. Octave down subtracts 12 from the
Note and Key classes abstract a musical note. Note instances represent all possible notes, whereas the Key instances only represent notes that are keys. Keys are notes that may serve as key signatures; these notes are included in the Circle of Fifths. The Circle of Fifths [22], shown in Figure 25, demonstrates the relationship between the 12 notes that serve as key signatures.

Note class includes an octave field indicating whether note is to be played at a different octave. Note objects are not contained in other classes but are the object type into which musical objects such as chords, chord progressions, and scales are rendered prior to MIDI Playback.

Table 6: MIDI Values

<table>
<thead>
<tr>
<th>Note</th>
<th>A</th>
<th>A#/Bb</th>
<th>B</th>
<th>C</th>
<th>C#/Db</th>
<th>D</th>
<th>D#/Eb</th>
<th>E</th>
<th>F</th>
<th>F#/Gb</th>
<th>G</th>
<th>G#/Ab</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIDI value</td>
<td>57</td>
<td>58</td>
<td>59</td>
<td>60</td>
<td>61</td>
<td>62</td>
<td>63</td>
<td>64</td>
<td>65</td>
<td>66</td>
<td>67</td>
<td>68</td>
</tr>
<tr>
<td>Octave-</td>
<td>45</td>
<td>46</td>
<td>47</td>
<td>48</td>
<td>49</td>
<td>50</td>
<td>51</td>
<td>52</td>
<td>53</td>
<td>54</td>
<td>55</td>
<td>56</td>
</tr>
<tr>
<td>Octave+</td>
<td>69</td>
<td>70</td>
<td>71</td>
<td>72</td>
<td>73</td>
<td>74</td>
<td>75</td>
<td>76</td>
<td>77</td>
<td>78</td>
<td>79</td>
<td>80</td>
</tr>
</tbody>
</table>

value; octave up adds 12. NoteType enumeration is contained in classes Key, Note, and MusicalObject.
MusicalObject is an abstract super class that serves as parent to Chord and Scale classes. MusicalObject has two private data fields. root_ is an instance of NoteType and represents the root note of a musical object. degrees_ is a long value and serves as a bit field. degrees_ is valued with power-of-2 constants defined in the Const class. Appendix K shows the definition of the power-of-2 constants. Use of the degrees_ bit field and constants allows for the representation of every possible musical chord and scale. Chord and Scale objects can have the same raw representation of their composite notes. The Chord and Scale difference is realized at MIDI playback. Note set for a chord are played simultaneously whereas the note set of a scale is played per note, with an intervening period of silence. The below code excerpt shows creation of a 7b5 chord and a Kumoi (degree 1) scale. The scale definition does not include Const.ROOT entry. All scales must have a root note, and so the Const.ROOT value is automatically included. Chords may not have a root note, and so the Const.ROOT value must be included.
\begin{verbatim}
7b5 chord: degrees += (Const.ROOT|Const.MAJ3|Const.MIN5|Const.MIN7);
Kumoi scale: degrees += (Const.MAJ2|Const.MIN3|Const.MAJ5|Const.MAJ6);
\end{verbatim}

MusicalObject includes methods that map degrees_ bit-field into other representations. Method GetNoteList() returns an array of Note objects; this method is applied during MIDI playback of chord progressions. Method Encode() returns an array of int; this method is applied during Genetic Algorithm encoding of a chord progression. Method GetIntervals() returns an ArrayList of Interval objects. Intervals are used to assess a musical object’s quality. Method GetQuality() calls GetIntervals() and build a quality values based upon the composite set of Interval; GetQuality() is used in the assessment of a chord progression quality value.

Chord class models a musical chord. Chord includes private data field chordType_ of type ChordType. ChordType is an enumeration of chord entries gathered from Hal Leonard Picture Chord Encyclopedia [23] and [24]. ChordType includes methods that map ChordType to and from degrees_ bit-field. Much of Chord classes’ applied functionality is inherited from parent class MusicalObject. Methods added in Chord class include instance method Decode(), and class method GetQuality(Chord, Chord).

ChordProgression implements an abstract representation of a musical chord progression. ChordProgression models a chord progression with an array of Chord instances. The ChordProgression class contains methods for creating random chord progression, modal chord progress, and standard chord progression. SpeKculator exclusively uses randomized chord progressions. The majority of methods are in support of Genetic Algorithm Subsystem. Methods GetChord( index ) and SetChord( index, Chord) are in support of Crossover operator. Methods ChordDiversity() and UniqueChordTypes() are used as part of the Fitness Function; these methods determine the ratio of unique chords per chord count. A more diverse chord set is
rewarded in the Fitness Function. Methods `GetColor()` and `GetQuality()` support Fitness evaluation. Methods `Encode()` and `Decode()` support encoding of a musical chord progression to and from a Chromosome instance; Chromosome is encoded representation of a chord progression.

ChordDB maintains a repository of chord instances. ChordDB and ScaleDB classes contribute the general musical framework but are not currently in frequent use in SpeKculator. ChordDB is queried when assessing whether a mutated gene (encoded chord) is still a valid chord; ChordDB can be invoked directly. Dump command-line option will dump the entire ChordDB contents; this use of dump is not particularly useful. Dump command accepts a scale name and will restrict output to chords that match the provided scale. Figure 26 shows use of the ChordDB `dump=<scale>` command to show which chords match with a Dorian scale. ScaleDB is not currently in use by SpeKculator.
4.6 Testing

SpeKculator includes a large set of unit tests and system-level tests. Unit tests are executable per class, or in total by invoking the UnitTestDriver. An individual class’ unit tests are executed by invoking the class with no arguments. Figure 27 shows execution of the Note unit tests. Command-line option testRuns=<number> will issue the unit tests <number> times. Issuing the same unit tests numerous times is valuable, as many of the tested features include randomization, and reissue of tests provides greater test coverage. Unit tests are executed in full by invoking class UnitTestDriver with no arguments. Figure 28 shows execution of the UnitTestDriver. UnitTestDriver implements overloaded class (static) methods named testCase(). Below code excerpt shows exhaustive testing of Chord class Encode() and Decode methods.
Exhaustive test of encode/decode of each ChordType.
Root note of each chord is randomly chosen.

for( ChordType chordTypeA : ChordType.values() )
{
    chord = new Chord( NoteType.GetRandom(), chordTypeA );
    encoding = chord.Encode();
    UnitTestDriver.TestCase( null, "Chord.Encode/Decode("+chord.GetType()+")", Chord.Decode(encoding), chord );
}
System level tests are comprised of tests that provide insight on internal processing. System level test are invoked by executing the class with command-line argument classTest. Figure 15 shows execution of the Population class system test. Population system test creates a population of 130,000 of Individuals. Each Individual has its genetic information (chord progression) assessed for color and quality. The test identities the Individual that has the collective lowest color + quality and highest color + quality. This test proved very useful in determining what should be the range of objective values presented in the GUI, and how the user submitted objective values should be normalized to more closely align with the values generated in the randomized population.

4.7 Extending the Implementation

New chords can be added to the ChordType enumeration. Increased number of chords will increase chord progression diversity. Increasing chords is analogous to increasing the number
of genes possible in chord progression. Increased chord types will aid the ability to search out chord progression that meet an objective. Increase chord count will also help performance by reducing number of mutated genes that are identified as invalid. The mutation operator currently has a check to ensure that the mutated gene will decode into a valid chord. An invalid gene is restored to its original state, and mutation is reattempted with a different set of random values.

Chromosome:ValidGene() checks whether a encoded gene decodes into a valid chord. This method has been instrumented as follows to log detailed information of unknown chords:

```java
else if( ChordType.DegreesToType(degrees) == null )
{
    validGene = false;
    logger_.log( Level.INFO, "Unidentified chord, degreeArray: "
+Utils.PrettyPrintDegreeArray(degreeArray) );
}
```

Logged information includes degree identifiers. Figure 29 shows a SpeKculator.log excerpt showing the occurrences unidentified chords resulting from mutation.

![Figure 29: SpeKculator log showing unidentified chords](image)

A new ChordType will be creating the first log entry in Figure 29 as an example. This log entry indicated that degree notes root, maj3, aug3, aug5, maj7 do not map to a known ChordType.
Map the degree information to associated Const entries as shown in Table 7. Appendix K shows Const entries.

<table>
<thead>
<tr>
<th>Degree Name</th>
<th>Const.&lt;value&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>Root</td>
<td>Const.ROOT</td>
</tr>
<tr>
<td>Maj3</td>
<td>Const.MAJ3</td>
</tr>
<tr>
<td>Aug3</td>
<td>Const.AUG3</td>
</tr>
<tr>
<td>Aug7</td>
<td>Const.AUG7</td>
</tr>
<tr>
<td>Maj7</td>
<td>Const.MAJ7</td>
</tr>
</tbody>
</table>

Table 7: Association of Degree Name and Const Value

Chords are named by the set of contained degrees per a set of guidelines. Wikibooks Music Theory/Complete List of Chord Patterns [24] details how chords are named. It should be possible to appropriately name any collection of two or more degree values. Using the established set of Const values, open file Chord.java and make the following edits:

1. Update ChordType enumeration with new name
2. Update ChordType.toString() to return textual representation of new chord
3. Update ChordType.TypeToDegrees() with an entry for the new chord. This method values the degrees_bit field with Const degree values. The above chord would value the bit field as follows:

```
degrees += (Const.ROOT|Const.MAJ3|Const.AUG3|Const.AUG5|Const.MAJ7)
```
The new chord will automatically be picked up by the method that creates random chord progressions. This process should be repeated for as many "Unidentified chord" log entries as possible.
Chapter 5

OBSERVATIONS AND ASSESSMENT

5.1 Test Runs

One hundred test runs were issued in which Genetic Algorithm configuration settings for elitism, crossover rate, and mutation rate were altered and issued against the following color and quality objectives: [0,0], [-30,-30], [30,30], [-30, 30], and [30 -30]. Population size and max generations were not altered and remained at their default value of 1000. Each unique Genetic Algorithm and objective combination was tested five times. Test results are provided, in table form, in Appendix L. Table data is as follows:

- **Table 8**: Elitism 0, Crossover Rate 0.95, Mutation Rate 0.1
- **Table 9**: Elitism 1, Crossover Rate 0.95, Mutation Rate 0.1
- **Table 10**: Elitism 0, Crossover Rate 0.7, Mutation Rate 0.3
- **Table 11**: Elitism 1, Crossover Rate 0.7, Mutation Rate 0.3

5.2 Assessment

Assessment is comprised of audio playback of evolved chords, data in table form shown in Appendix L., responsiveness of the Genetic Algorithm to changes in configuration, ease of repeated use of the GUI, overall quality of the application.

Audio assessment of evolved chord progressions is subjective. It is however possible to discern darkness against brightness and dissonance against consonance. Evolved chord progressions, to this listener’s ear, definitely trend towards the submitted objective.

Genetic Algorithm responds, as expected, to configuration changes. Setting elitism to 0 results in regression of the fittest individual's fitness; elitism is set to 1 or above prevents regression of fittest individual's fitness. The evolution of chord progressions is assessed by
viewing Ave. Individual Fitness field of the GUI status panel during test execution. Tests issued with lessened crossover rate of 0.7 and heightened mutation rate of 0.3, exhibited large swings in the average individual fitness. Crossover rate of 0.95 and mutation rate of 0.1 resulted in a more smoothed out evolution. This is expected behavior, as crossover is more likely to produce more fit individuals than mutation. Chord diversity is lacking. Certain chord types gravitate towards specific objectives. Test data shows that objectives with a dissonance quality attract the Petruska chord; objectives with bright color quality attract Maj13 and Tristan chords. More troubling is the repeating of the same chord type within a chord progression. The Fitness Function includes a reward for chord diversity, but data shows that this is not adequately effective. An extreme example of chord type repetition, seen in Table 10 at second entry of result for color[-30] and quality[30] is as follows:

C#add9, B5, B5, B5, Fb(petrushka), Fb(petrushka), E(petrushka)

Test data shows a possible issue with randomness of chord key randomness. Distribution of chord keys does not appear uniform. Key of C, including the application of an accidental (i.e. Cb, C#), appear in higher frequency than other keys.

One hundred test runs provided the most rigorous test to date of the SpeKculator implementation. Numerous tests were issued in single session, meaning SpeKculator was left running during different test runs. No run-time crashed occurred, and, the results from one test to another produced clearly independent results. Genetic Algorithm performance is a concern. SpeKculator is exceedingly slow for populations that approach 3000. The following two defects where identified during testing:
1. Objective of color=0 and quality=0 exposes an issue in which fittest individual's fitness equals -0.1. Range of acceptable fitness values is [0-N]. Negative value should not happen. This is likely due to the Fitness Function provision in which chord progressions with the highest possible chord diversity are rewarded by subtracted -0.1 from the evaluated fitness.

2. ChordType of Frankenstein showed up in three evolved chord progression. Frankenstein chords are not musical and thus should not show up either random chord progressions or evolved chord progressions. Furthermore, the Frankenstein chord was leveraged in a version of the Fitness Function that is no longer in use. Frankenstein chords are built, note by note, by collecting notes from each chord in the chord progression. A quality measure of the Frankenstein chord was used as a quality measure of the chord progression.
CONCLUSIONS AND FUTURE WORK

6.1 Conclusions

SpeKculator demonstrates reasonably good results at evolving a random set of chord progressions towards are a user-supplied color and quality objective. SpeKculator GUI is intuitive in use, and testing has demonstrated the implementation is robust. The Musical Objects Subsystem is a general framework that can support searches of objects other than chord progressions; I would like to extend SpeKculator to include the generation of tone rows [25]. I look forward to applying SpeKculator to its intended use as a tool to assist in creating music, and to help break creative blocks. I am eager to share SpeKculator with other musicians.

Figure 30: Manual Version of SpeKculator [26]
6.2 Future Work

There are numerous areas for improvement in the SpeKculator implementation. These areas include the additions of features, improvement of existing features, reviewing and possibly improving performance, and bug fixes.

SpeKculator GUI File Menu is not complete. Under the File menu, there are menu options for Loading and Saving a Chord Progression; selecting either option results in a pop-up window with text “Not yet implemented”. The intention was for users to be able to save and load evolved chord progressions. Audio of evolved chord progressions would be greatly enhanced by allowing the chord progression playback to be repeated. Rather than playing a single occurrence of the chord progression, it would be more useful, particularly when attempting to perform along with the chord progression, to allow the chord to repeatedly played until directed to stop. Support for adjusting the tempo of chord progression playback, particularly if paired with the ability to loop audio playback, would improve audio assessment of chord progressions.

The Musical Objective Subsystem was implemented to model chord progressions as individuals in a Genetic Algorithm. The implementation evolved into a general-purpose framework for modeling and programmatically assessing relationships between many different types of musical object. The framework supports searches beyond chord progressions. A Genetic Algorithm search for tone rows [25] would be an interesting feature. Chord Progression class can generate different classes of chord progression; these classes include random, modal, random-modal, and standard chord progressions; Table 2 shows a few of the standard chord progressions that are possible. The initial population passed into the Genetic Algorithm is currently comprised random chord progressions. The method that create populations could be updated to include standard chord progressions.
Genetic Algorithm Fitness Function is lacking. The evaluation of chord progressions is too heavily weighted upon individual chords; a more holistic view of the chord progression could address the problem chord diversity, and the tendency of certain chord gravitating towards certain objectives. One way to facilitate a holistic view of a chord progression is to leverage modes. Musical modes are model in a Mode class. Modes could be leveraged by the fitness function to identify what if any melodic lines exist across the chords in a progression. The melodic lines could searched for full or partial matches with modes. The evaluation of a mode’s color and quality are quite easy; identifying modes within chord progression could provide a more accurate evaluation of a chord progression.

A list of identified issues and bugs was maintained throughout implementation. The list entries are organized, top to bottom, by priority and by whether the item is a bug or feature. This list is provided in Appendix M.
public static long DegreePositionToConst( int degreePosition )
{
  long constValue = 0;
  switch( degreePosition )
  {
    case  0 : constValue = Const.ROOT;   break;
    case  1 : constValue = Const.MIN2;   break;
    case  2 : constValue = Const.MAJ2;   break;
    case  3 : constValue = Const.AUG2;   break;
    case  4 : constValue = Const.MXM3;   break;
    case  5 : constValue = Const.MIN3;   break;
    case  6 : constValue = Const.MAJ3;   break;
    case  7 : constValue = Const.AUG3;   break;
    case  8 : constValue = Const.MXM4;   break;
    case  9 : constValue = Const.MIN4;   break;
    case 10 : constValue = Const.MAJ4;   break;
    case 11 : constValue = Const.AUG4;   break;
    case 12 : constValue = Const.AXA4;   break;
    case 13 : constValue = Const.MXM5;   break;
    case 14 : constValue = Const.MIN5;   break;
    case 15 : constValue = Const.MAJ5;   break;
    case 16 : constValue = Const.AUG5;   break;
    case 17 : constValue = Const.AXA5;   break;
    case 18 : constValue = Const.MXM6;   break;
    case 19 : constValue = Const.MIN6;   break;
    case 20 : constValue = Const.MAJ6;   break;
    case 21 : constValue = Const.AUG6;   break;
    case 22 : constValue = Const.MXM7;   break;
    case 23 : constValue = Const.MIN7;   break;
    case 24 : constValue = Const.MAJ7;   break;
    case 25 : constValue = Const.OCTAVE; break;
    case 26 : constValue = Const.MIN9;   break;
    case 27 : constValue = Const.MAJ9;   break;
    case 28 : constValue = Const.AUG9;   break;
    case 29 : constValue = Const.MIN10;  break;
    case 30 : constValue = Const.MAJ10;  break;
    case 31 : constValue = Const.AUG10;  break;
    case 32 : constValue = Const.MIN11;  break;
  }
}
case 33 : constValue = Const.MAJ11; break;
case 34 : constValue = Const.AUG11; break;

case 35 : constValue = Const.MIN13; break;
case 36 : constValue = Const.MAJ13; break;
case 37 : constValue = Const.AUG13; break;
}

return constValue;

public static int[] DegreesToDegreeArray( long degrees )
{
    int[] degreeArray = null;

    if( degrees <= 0 )
    {
        throw new IllegalArgumentException( "Invalid degrees[" +degrees+ "] argument" );
    }

degreeArray = new int[Const.DEGREE_COUNT];

    for( int i = 0; i < Const.DEGREE_COUNT; i++ )
    {
        degreeArray[i] =((degrees & Const.DegreePositionToConst(i)) != 0) ? 1 : 0;
    }

    return degreeArray;

public int[] GetDegreeArray()
{
    return Utils.DegreesToDegreeArray( degrees_ );
}

public int[] Encode()
{
    int[] encoding = null;

    return encoding;
}
int[] degreeArray = null;

if( root_ == null )
{
    throw new RuntimeException( "root_ == null: Cannot encode" );
}

if( degrees_ == 0 )
{
    throw new RuntimeException( "degrees_ == 0: Cannot encode" );
}

encoding = new int[Chromosome.GENE_LENGTH];
encoding[0] = root_.ordinal(); // encoding[0] reserved for key
degreeArray = GetDegreeArray();

for( int i = 0; i < Const.DEGREE_COUNT; i++ )
{
    encoding[i+1] = degreeArray[i];
}

return encoding;

//========================================================================
//= ChordProgression::Encode()                                          
//= Encode the passed in chordProgression into a Chromosome instance      
//========================================================================

public static Chromosome Encode( ChordProgression chordProgression )
{
    Chromosome chromosome = null;
    Chord chord = null;
    int chordCount;
    int[][] encoding;

    if( chordProgression == null )
    {
        throw new IllegalArgumentException( "Invalid chordProgression argument: chordProgression == null" );
    }

    encoding = new int[Chromosome.CHROMOSOME_LENGTH+1][Chromosome.GENE_LENGTH];
    chordCount = chordProgression.GetCount();

    for( int i = 0; i < chordCount; i++ )
    {
        chord = chordProgression.GetChord( i );
        encoding[i] = chord.Encode();
    }

    encoding[chordCount][0] = Chromosome.ENCODING_END;
    chromosome = new Chromosome(encoding);
    return chromosome;
}
public static long DegreeArrayToDegrees( int[] degreeArray )
{
    long degrees = 0;
    if( degreeArray == null )
    {
        throw new IllegalArgumentException( "degreeArray == null" );
    }
    if( degreeArray.length > Const.DEGREE_COUNT )
    {
        System.out.println( "degreeArray: " +
                Utils.PrettyPrintDegreeArray(degreeArray) );
        throw new RuntimeException( "degreeArray length[" +degreeArray.length+
                "] > Const.DEGREE_COUNT[" +Const.DEGREE_COUNT+ "]" );
    }
    for( int i = 0; i < Const.DEGREE_COUNT; i++ )
    {
        degrees += degreeArray[i] == 1 ? Const.DegreePositionToConst(i) : 0;
    }
    if( degrees == 0 )
    {
        logger_.log( Level.WARNING, "DegreeArrayToDegrees(): degrees == 0" );
    }
    return degrees;
}

public static Chord Decode( int[] encoding )
{
    NoteType  noteType  = null;
    Note      note      = null;
    ChordType chordType = null;
    Chord     chord     = null;
    long      degrees   = 0;
    int[]     degreeArray;
    if( encoding != null )
    {
        noteType    = NoteType.values()[encoding[0]]; // key value
    }
note = new Note(noteType); // create Key object
degreeArray = Arrays.copyOfRange(encoding, 1, Chromosome.GENE_LENGTH);
degrees = Utils.DegreeArrayToDegrees(degreeArray);
chordType = ChordType.DegreesToType(degrees);
chord = new Chord(noteType, chordType);
chord.SetRoot(note);
}

return chord;
}

//========================================================================
//= ChordProgression::Decode( Chromosome chromosome )
//= ------------------------------------------------
//= Decode passed in Chromosome instance into a chord progression
//===============================================

class ChordProgression
{
public static ChordProgression Decode(Chromosome chromosome)
{
    ChordProgression chordProgression = null;
    Chord chord = null;
    int chromosomeLength = 0;
    boolean lengthFound = false;
    int[][] encoding;
    int i;

    if(chromosome == null)
    {
        throw new IllegalArgumentException("chromosome == null");
    }

    encoding = chromosome.GetEncoding();
    chordProgression = new ChordProgression(); // Creates chordProgression
    chordProgression.Clear(); // Clear chordprogression

    for(i = 0; i < Chromosome.CHROMOSOME_LENGTH+1; i++)
    {
        if(encoding[i][0] == Chromosome.ENCODING_END)
        {
            chromosomeLength = i;
            break;
        }
    }

    if(chromosomeLength == 0)
    {
        throw new RuntimeException("Cannot identify chromosome length");
    }

    for(i = 0; i < chromosomeLength; i++)
    {
        chord = Chord.Decode(encoding[i]);
        chordProgression.SetChord(i, chord);
    }

    return chordProgression;
}

APPENDIX C: Selection and Crossover Operator Implementation

//=======================================================
//= Individual::Crossover_CutAndSplice()
//= Implementation of Cut And Splice Crossover operation
//=======================================================

public static Individual Crossover_CutAndSplice( Individual parent1, Individual parent2 )
{
  Individual offspring = new Individual();
  int parent1_gene_count = parent1.GetGeneCount();
  int parent2_gene_count = parent2.GetGeneCount();
  int parent1_crossover_point = parent1.GetCrossoverPoint();
  int parent2_crossover_point = parent2.GetCrossoverPoint();
  int[] parent1_gene = new int[parent1_gene_count];
  int[] parent2_gene = new int[parent2_gene_count];
  int parent_index = 0;
  int offspring_index = 0;
  for( parent_index = 0; parent_index < parent1_crossover_point; parent_index++ )
  {
    parent1_gene[ parent_index ] = parent1.GetGene( parent_index );
    if( offspring.SetGene(offspring_index, parent1_gene) == false )
    {
      logger_.log( Level.INFO, "offSpring.SetGene() failure. chromosome_index[" +parent_index+ "]" );
    }
    offspring_index = parent_index;
  }
  for( parent_index = parent2_crossover_point; parent_index < parent2_gene_count; parent_index++ )
  {
    parent2_gene[ parent_index ] = parent2.GetGene( parent_index );
    if( offspring.SetGene(offspring_index, parent2_gene) == false ) // will fail if max gene count reached
    {
      break;
    }
    offspring_index++;
  }
  offspring.SetEnd( offspring_index );
  return offspring;
}

//=======================================================
//= Individual::Crossover()
 //= ------------------------------------
//=======================================================

//======================================
//= Individual::Crossover()
// Apply Crossover on Individual. This method invokes the current
// crossover implementation
public static Individual Crossover( Individual parent1, Individual parent2 )
{
    return Crossover_CutAndSplice( parent1, parent2 );
}

private Individual Selection_Tournament( Population population )
{
    Individual individual = null;
    Population tournament = new Population( tournamentSize_ );

    population.Shuffle();
    for( int i = 0; i < tournamentSize_; i++ )
    {
        individual = population.GetIndividual( i );
        tournament.SetIndividual( i, individual );
    }

    return tournament.GetFittest( 0 );
}

private Individual Selection( Population population )
{
    return Selection_Tournament( population );
}

public Population Crossover( Population population_in )
{
    Population population_out = new Population( population_in.Size(), false );
    Individual parent1        = null;
    Individual parent2        = null;
    Individual offspring      = null;
    Random     random         = new Random();
generations_++;  

for( int populationIndex = 0; populationIndex < population_in.Size();  
    populationIndex++ )  
{  
    parent1 = population_in.GetFittest( populationIndex );  

    if( crossoverRate_ > Math.random() && populationIndex >= elitismCount_ )  
    {  
        parent2 = Selection( population_in );  
        offspring = Individual.Crossover( parent1, parent2 );  
        CalculateFitness( offspring );  
        population_out.AddIndividual( offspring ); // crossover applied  
    }  

    else // parent1 transitions, without modification, from current to next  
        generation  
    {  
        population_out.AddIndividual( parent1 ); // crossover not applied  
    }  

return population_out;  
}
APPENDIX D: Mutation Operator Implementation

```java
public boolean ValidGene( int chromosome_index )
{
    boolean validGene = false;
    int[] gene;
    int keyOrdinal;
    int[] degreeArray;
    long degrees;

    if( chromosome_index > geneCount_ )
    {
        throw new IllegalArgumentException( "chromosome_index[" +chromosome_index+ "] > geneCount_[" +geneCount_+ "]" );
    } else if( ChordType.DegreesToType(degrees) == null )
    {
        validGene = false;
        logger_.log( Level.INFO, "Unidentified chord, degreeArray: " +Utils.PrettyPrintDegreeArray(degreeArray) );
    } else if( NoteType.IsValid(keyOrdinal) == false )
    {
        validGene = false;
        logger_.log( Level.SEVERE, "ValidGene(): invalid keyOrdinal[" +keyOrdinal+ "]" );
    } else
    {
        validGene = true;
    }

    return validGene;
}
```

public void ToggleDNA( int chromosome_index, int gene_index )
{
    Random random = new Random();
    int DNA;

    if( chromosome_index >= geneCount_ )
    {
        throw new IllegalArgumentException( "chromosome_index[" +chromosome_index+ "] >= geneCount_" +geneCount_+ "]" );
    }

    DNA = GetDNA( chromosome_index, gene_index );

    if( gene_index == 0 ) // if modifying the key
        DNA = random.nextInt( NoteType.GetCount() ); // was hard-coded to 21
    else
        DNA = (DNA == 0) ? 1 : 0;

    SetDNA( chromosome_index, gene_index, DNA );
}

public void Mutate()
{
    ChordProgression chordProgressionPreMutate = chordProgression_;
    Random random = new Random();
    boolean valid_mutation = false;
    int geneCount = GetGeneCount();
    int chromosome_index = 0;
    int gene_index = 0;
    int mutate_attempts = 0;

    while( valid_mutation == false && mutate_attempts < Const.MAX_ATTEMPTS )
    {
        chromosome_index = random.nextInt( geneCount ); // random
        chord (was GetGeneCount() -1)
gene_index = random.nextInt(Const.DEGREE_COUNT); // random degree within chord
chromosome_.ToggleDNA(chromosome_index, gene_index); // flip the bit

if( chromosome_.ValidGene(chromosome_index) == false ) // Is gene (chord) invalid?
    chromosome_.ToggleDNA(chromosome_index, gene_index); // undo failed mutation
else
    valid_mutation = true;
mutate_attempts++; // constrain mutate attempts
}

if( valid_mutation == true ) // chromosome modified, set chord progression
{
    chordProgression_ = ChordProgression.Decode(chromosome_);

    if( chordProgression_.equals(chordProgressionPreMutate) == true )
    {
        logger_.log(Level.WARNING, "");
        logger_.log( Level.WARNING, "mutate chord progression did not change chord progression" );
        logger_.log( Level.WARNING, " pre[" +chordProgressionPreMutate+ "]" );
        logger_.log( Level.WARNING, " post[" +chordProgression_+ "]" );
        logger_.log( Level.WARNING, " chromosome_index[" +chromosome_index+ "], gene_index[" +gene_index+ "]" );
    }
}
else
{
    logger_.log( Level.SEVERE, "failure to mutate: " + chordProgressionPreMutate );
}

//========================================================================
//= GeneticAlgorithm::Mutate() //=
//= --------------------------- //=
//= Apply Mutation Operator to entire population. Mutation implementation //=
//= exists in the Individual class. //=
//========================================================================

public Population Mutate(Population population_in )
{
    Population population_out = new Population(population_in.Size(), false );
    Individual individual = null;
    Random random = new Random();

    for( int populationIndex = 0; populationIndex < population_in.Size(); populationIndex++)
    {
        individual = population_in.GetFittest(populationIndex );
if( mutationRate_ > Math.random() && populationIndex >= elitismCount_ )
{
    individual.Mutate();
}

population_out.AddIndividual( individual );

return population_out;

APPENDIX E: Fitness Function Implementation

//========================================================================
//= Utils::PythagoreanDistance()
//= ------------------------------
//= Return the distance between two points
//========================================================================

public static double PythagoreanDistance( int X1, int Y1, int X2, int Y2 )
{
    double distance = Math.sqrt( Math.pow((X2 - X1), 2) + Math.pow((Y2 - Y1), 2));

    return distance;
}

//========================================================================
//= ChordProgression::ChordDiversity()
//= ----------------------------------
//= Return a chord diversity measure: (unique chord count / chord count)
//========================================================================

public double ChordDiversity()
{
    double chordDiversity = (UniqueChordTypes() * 1.0)/GetCount();

    return chordDiversity;
}

//========================================================================
//= GeneticAlgorithm::CalculateFitness()
//= ------------------------------------
//= Calculate and assign fitness of an individual.
//========================================================================

private double CalculateFitness( Individual individual )
{
    double fitness;
    double geneDiversity;
    double reward;

    if( individual == null )
    {
        throw new IllegalArgumentException( "individual == null" );
    }

    fitness = Utils.PythagoreanDistance( objective_.GetColor(),
                                        objective_.GetQuality(),
                                        individual.GetColor(),
                                        individual.GetQuality() );

    // do not allow fitness < 0.0
    reward = (individual.GeneDiversity() == 1.0) ? (fitness > 0.2 ? 0.0 : -0.1) :
    0.0;
    fitness += reward; // Add reward if chord set is diverse
//fitness = (fitness - geneDiversity <= 0.0) ? 0.0 : (fitness -
geneDiversity);  // old approach
   individual.SetFitness( fitness );

   return fitness;
}

//=================================================================================
//= GeneticAlgorithm::Evaluate()
//= -----------------------------------------
//= Evaluate the fitness of each individual in the population.
//= Iterate through entire population, assessing and assigning a fitness
//= to each individual. The population fitness is the cumulative fitness
//= of all individuals. Includes some accounting related code.
//=================================================================================

public void Evaluate( Population population )
{
   double fitness = 0.0;

   for( Individual individual : population.GetIndividuals() )
   {
      fitness += CalculateFitness( individual );
   }

   population.SetFitness( fitness );

   //----------------------------------------------------------------------------
   //----------------------------------------------------------------------------

double fittestIndividual    = population.GetFittest(0).GetFitness();
double avePopulationFitness = population.GetFitness()/population.GetSize();

   if( generations_ == 0 )
   {
      indFitStart_ = fittestIndividual;
      popFitStart_ = avePopulationFitness;
      popFitLow_   = avePopulationFitness;
   }
   else
   {
      if( fittestIndividual < indFitCurr_ )
      {
         indFitIncs_++;
      }

      if( avePopulationFitness < popFitLow_ )
      {
         popFitIncs_++;
         popFitLow_ = avePopulationFitness;
      }

      indFitCurr_ = fittestIndividual;
      popFitCurr_ = avePopulationFitness;
   }
}
MusicalObject::GetColor()
---
= Provide a somewhat heuristic assessment of a musical object's color.
= Zero is considered neutral. Higher positive Color value indicates a
= brighter sound, whereas a higher negative value indicates a darker sound.
= Method attempts to encapsulate the notion of color as described in a
= Coursera class on Jazz Improvisation. In the class, scales were ordered
= from brightest to darkest (i.e. Color) based upon the number of
= augmented/major notes to the number of flattened/minor notes.
---

```java
public int GetColor()
{
    int color = 0;

    if( HasDegree(Const.MIN2) ) color -= 1;
    if( HasDegree(Const.NAJ2) ) color += 1;
    if( HasDegree(Const.AUG2) ) color += 2;

    if( HasDegree(Const.NXM3) ) color -= 2;
    if( HasDegree(Const.MIN3) ) color -= 1;
    if( HasDegree(Const.NAJ3) ) color += 1;
    if( HasDegree(Const.AUG3) ) color += 2;

    if( HasDegree(Const.NXM4) ) color -= 2;
    if( HasDegree(Const.MIN4) ) color -= 1;
    if( HasDegree(Const.NAJ4) ) color += 1;
    if( HasDegree(Const.AUG4) ) color += 2;
    if( HasDegree(Const.AXA4) ) color += 3;

    if( HasDegree(Const.NXM5) ) color -= 2;
    if( HasDegree(Const.MIN5) ) color -= 1;
    if( HasDegree(Const.NAJ5) ) color += 1;
    if( HasDegree(Const.AUG5) ) color += 2;
    if( HasDegree(Const.AXA5) ) color += 3;

    if( HasDegree(Const.NXM6) ) color -= 2;
    if( HasDegree(Const.MIN6) ) color -= 1;
    if( HasDegree(Const.NAJ6) ) color += 1;
    if( HasDegree(Const.AUG6) ) color += 2;

    if( HasDegree(Const.NXM7) ) color -= 2;
    if( HasDegree(Const.MIN7) ) color -= 1;
    if( HasDegree(Const.NAJ7) ) color += 1;

    // HasDegree(Const.OCTAVE) intentially omitted.

    if( HasDegree(Const.MIN9) ) color -= 1;
    if( HasDegree(Const.NAJ9) ) color += 1;
    if( HasDegree(Const.AUG9) ) color += 2;

    if( HasDegree(Const.MIN10) ) color -= 1;
    if( HasDegree(Const.NAJ10) ) color += 1;
```
if( HasDegree(Const.AUG10) ) color += 2;
if( HasDegree(Const.MIN11) ) color -= 1;
if( HasDegree(Const.MAJ11) ) color += 1;
if( HasDegree(Const.AUG11) ) color += 2;
if( HasDegree(Const.MIN13) ) color -= 1;
if( HasDegree(Const.MAJ13) ) color += 1;
if( HasDegree(Const.AUG13) ) color += 2;

return color;
}

//===================================================================
//= ChordProgression::GetColor()
//= -----------------------------
//= Establish and return a chord progression Color. Chord progression
//= color is the accumulated color of the chord progression chords
//===================================================================
public int GetColor()
{
    Chord chord = null;
    int color = 0;

    for( int i = 0; i < progression_.size(); i++ )
    {
        chord = progression_.get( i );
        color += chord.GetColor();
    }

    return color;
}

//================================================================================================
//= Chord::GetQuality()
//= -------------------
//= Return the Quality (Dissonance/Consonance) between the two passed in
//= chords. The assessment is to check every degree, if shared between then
//= quality is incremented (a movement towards consonance), if the not is
//= not shared, then the quality is decremented (a move towards dissonance).
//================================================================================================
public static int GetQuality( Chord chordA, Chord chordB )
{
    Chord chord = new Chord( ChordType.FRANKENSTEIN );
    long chordA_degrees = chordA.GetDegrees();
    long chordB_degrees = chordB.GetDegrees();
    int quality = 0;

    for( long i = Const.ROOT; i >= 1; i /= 2 )
    {
        if( chordA.HasDegree(i) == true && chordB.HasDegree(i) == true )
        {
            quality++;
        }
        else if((chordA.HasDegree(i) == true && chordB.HasDegree(i) == false) ||
                  (chordA.HasDegree(i) == false && chordB.HasDegree(i) == true) )
        {
        }
}
public int GetQuality()
{
    Chord chord = null;
    int quality = 0;

    for (int i = 1; i < progression_.size(); i++)
    {
        quality += Chord.GetQuality( progression_.get(i-1),
                                      progression_.get(i) );
    }

    return quality;
}
APPENDIX F: Implementation of GUI Configuration Panel Population Field

```java
configPopulationSizeField_ = new JTextField(
    Integer.toString(configPopulationSize_), CONFIG_FIELD_LENGTH);
configPopulationSizeField_.setToolTipText("Acceptable range[" +
    GeneticAlgorithm.MIN_POPULATION + "-" + GeneticAlgorithm.MAX_POPULATION + "]");

configPopulationSizeField_.addFocusListener(
    new FocusListener()
    {
        public void focusGained( FocusEvent focusEvent )
        {
            // intentionally empty
        }
        public void focusLost( FocusEvent focusEvent )
        {
            boolean configInputValid = true;
            String configInputValue = configPopulationSizeField_.getText();
            try
            {
                configPopulationSize_ = Integer.parseInt(configInputValue);
            }
            catch( NumberFormatException numberFormatException )
            {
                configInputValid = false;
            }
            if( configInputValid == true )
            {
                configInputValid = GeneticAlgorithm.IsValidPopulationSize(configPopulationSize_);
            }
            if( configInputValid == false )
            {
                JOptionPane.showMessageDialog(configPanel_, "Invalid Population Size value[" +
                    configInputValue + "]. Acceptable range: [" +
                    GeneticAlgorithm.MIN_POPULATION + "]" + GeneticAlgorithm.MAX_POPULATION + "]",
                    "Invalid Input", JOptionPane.ERROR_MESSAGE);
                configPopulationSize_ = GeneticAlgorithm.DEF_POPULATION;
            }
            if( configPopulationSize_ != GeneticAlgorithm.DEF_POPULATION )
            {
                configResetButton_.setEnabled(true);
            }
        }
    });
```
APPENDIX G: Implementation of GUI Objective Panel Start Button

```java
objectiveStartButton_.addActionListener(
    new ActionListener()
    {
        @Override
        public void actionPerformed( ActionEvent actionEvent )
        {
            geneticAlgorithmWorker_ =
                new GeneticAlgorithmWorker( configPopulationSize_,
                                            configCrossoverRate_,
                                            configMutationRate_,
                                            configElitismCount_,
                                            configMaxGenerations_,
                                            objectiveColor_,
                                            objectiveQuality_,
                                            statusGenerationField_,
                                            statusAvgPopulationFitnessField_,
                                            statusFittestIndividualField_,
                                            outputChordProgressionField_,
                                            objectiveStartButton_,
                                            objectiveStopButton_,
                                            outputPlayButton_);

            configResetButton_.setEnabled( false );
            objectiveStartButton_.setEnabled( false );
            objectiveStopButton_.setEnabled( true );
            objectiveResumeButton_.setEnabled( false );
            outputPlayButton_.setEnabled( false );
            outputChordProgressionField_.setText( "" );

            geneticAlgorithmWorker_.execute();
        }
    });
```
public Individual doInBackground()
{
    Double generation = 0.0;
    Double populationFitness = 0.0;
    Double fittestIndividual = 0.0;

    if( IsGeneticAlgorithmInitialized() == false )
    {
        objective_ = new Objective( colorObjective_, qualityObjective_ );
        geneticAlgorithm_ = new GeneticAlgorithm( populationSize_, crossoverRate_,
                                                  mutationRate_, elitismCount_,
                                                  objective_, maxGenerations_ );
        population_ = geneticAlgorithm_.CreatePopulation();
    }

    geneticAlgorithm_.Evaluate( population_ );
    fittestIndividual_ = population_.GetFittest( 0 );

    while( geneticAlgorithm_.IsTerminationConditionMet(population_) == false )
    {
        population_ = geneticAlgorithm_.Crossover( population_ );
        population_ = geneticAlgorithm_.Mutate( population_ );
        geneticAlgorithm_.Evaluate( population_ );
        fittestIndividual_ = population_.GetFittest( 0 );

        try
        {
            Thread.sleep( S_secureRandom_.nextInt(5) );
        }
        catch( InterruptedException interruptedException )
        {
            return fittestIndividual_;  
        }

        generation = (double)geneticAlgorithm_.GetGenerations();
        populationFitness = population_.GetFitness()/population_.GetSize();
        fittestIndividual = population_.GetFittest(0).GetFitness();

        publish( generation, populationFitness, fittestIndividual );

        if( isCancelled() == true )
        {
            break;
        }
    }

    return fittestIndividual_;
APPENDIX I: Interface between GUI Play button and MIDI Subsystem

```java
outputPlayButton_.addActionListener(
    new ActionListener()
    {
        @Override
        public void actionPerformed( ActionEvent actionEvent )
        {
            objectiveStartButton_.setEnabled( false );
            objectiveStopButton_.setEnabled( false );
            outputPlayButton_.setEnabled( false );

            speKculatorPlayer_ = new SpeKculatorPlayer();
            speKculatorPlayer_.Play(
                geneticAlgorithmWorker_.GetFittestIndividual().GetChordProgression() );

            outputPlayButton_.setEnabled( true );
            objectiveStartButton_.setEnabled( true );
        }
    }
);
```

```java
//========================================================================
//= SpeKculatorPlayer::MidiInitialize()  
//= Wrapper method for initializing MIDI player with MIDI sequence
//========================================================================

public void MidiInitialize()
{
    midiPlayer_.Initialize( sequenceCreator_.GetSequence() );
}
```

```java
//========================================================================
//= SpeKculatorPlayer::MidiPlay()   
//= Wrapper method for playing a MIDI sequence
//========================================================================

public void MidiPlay()
{
    midiPlayer_.Play();
}
```

```java
//========================================================================
//= SpeKculatorPlayer::Play( ChordProgression )    
//= Provide audio representation of passed in ChordProgression. Play()  
//= method processes the ChordProgression by iterating through chord  
//= progression and creating MIDI events for each chord in the progression.  
//= The MIDI sub-system is then initialized and the sequence is played.  
//========================================================================

public void Play( ChordProgression chordProgression )
{
    Chord chord    = null;
    int    chordCount = chordProgression.GetCount();
```
```cpp
ResetTimers();

for( int i = 0; i < chordCount; i++ )
{
    chord = chordProgression.GetChord( i );

    if( i == 0 )
        tick_ = start_time_sequence_;
    else
        tick_ += start_time_relative_;

    SeqAddEvent( ShortMessage.NOTE_ON, chord, tick_ );
    tick_ += event_duration_;    
    SeqAddEvent( ShortMessage.NOTE_OFF, chord, tick_ );
}

MidiInitialize();
MidiPlay();
```
import java.io.File;
import java.io.IOException;
import javax.sound.midi.MidiSystem;
import javax.sound.midi.Synthesizer;
import javax.sound.midi.Transmitter;
import javax.sound.midi.Receiver;
import javax.sound.midi.Sequence;
import javax.sound.midi.Sequencer;
import javax.sound.midi.MidiChannel;
import javax.sound.midi.ShortMessage;
import javax.sound.midi.MidiUnavailableException;
import javax.sound.midi.InvalidMidiDataException;
import javax.sound.midi.MetaMessage;
import javax.sound.midi.MetaEventListener;

public class MidiPlayer {

    private Synthesizer synthesizer_;    // gathered from static MidiSystem
    private Sequencer sequencer_;       // gathered from static MidiSystem
    private Receiver synthReceiver_;    // gathered from synthesizer_ instance
    private Transmitter transmitter_;   // gathered from sequencer_ instance
    private Sequence sequence_;         // populated by caller
    private int volume_;               // default value, can be set by caller

    public MidiPlayer() {
        synthReceiver_ = null;
        sequence_ = null;
        volume_ = 80;    // default value. TODO: Pass in as constructor arg
        try {
            synthesizer_ = MidiSystem.getSynthesizer();
            synthesizer_.open();
        }
    }
}
try
{
    sequencer_ = MidiSystem.getSequencer( false ); // false: do not connect to
}                                                 //   default
synthesizer.

try
{
    transmitter_ = sequencer_.getTransmitter();
}

catch( MidiUnavailableException e )
{
    e.printStackTrace();
    System.exit(1);
}

//-------------------
// Register listener with sequencer_listen for end of track, and if
// encountered close sequencer_ and synthesizer_ instances.
// TODO: Review is this is necessary
//-------------------
sequencer_.addMetaEventListener(
    new MetaEventListener()
    {
        public void meta( MetaMessage metaMessage )
        {
            if( metaMessage.getType() == Const.MM_TYPE_END_OF_TRACK )
            {
                sequencer_.close();
                synthesizer_.close();
            }
        }
    });

//========================================================================
//= MidiPlayer::Initialize() //=
//= Opens the sequencer_ object, and if successful, assigns to
//= it a sequence taken from data member sequence_. //=
//========================================================================
private void Initialize()
{
  try
  {
    sequencer_.open();
  }

  catch( MidiUnavailableException e )
  {
    e.printStackTrace();
    System.exit(1);
  }

  try
  {
    sequencer_.setSequence( sequence_ );
  }

  catch( InvalidMidiDataException e )
  {
    e.printStackTrace();
    System.exit(1);
  }
}

//========================================================================
//= MidiPlayer::Initialize( Sequence )
//= -----------------------
//= Add MIDI sequence to player
//========================================================================
public void Initialize( Sequence sequence )
{
  if( sequence == null )
  {
    throw new IllegalArgumentExcep

MidiChannel[] channels = null;
ShortMessage shortMessage = new ShortMessage();

try
{
    synthReceiver_ = synthesizer_.getReceiver();
    transmitter_.setReceiver(synthReceiver_);
}
catch(MidiUnavailableException e)
{
    e.printStackTrace();
}

sequencer_.start();
channels = synthesizer_.getChannels();

for(int channel = 0; channel < channels.length; channel++)
{
    try
    {
        shortMessage.setMessage(ShortMessage.CONTROL_CHANGE, channel, Const.CC_VOLUME, volume_);
    }
    catch(InvalidMidiDataException e)
    {
        e.printStackTrace();
    }

    synthReceiver_.send(shortMessage, -1); // -1 disables time-stamp
    // TODO: review previous
}
    // call to receiver.send()
public class Const
{
    //===========================================================================
    //==
    //Musical related constants
    //==
    //----------------------------------------------------------------------------
    public static final long AUG13  = 1L;    // position:  1, index: 37
    public static final long MAJ13  = 2L;    // position:  2, index: 36
    public static final long MIN13  = 4L;    // position:  3, index: 35
    public static final long AUG11  = 8L;    // position:  4, index: 34
    public static final long MAJ11  = 16L;   // position:  5, index: 33
    public static final long MIN11  = 32L;   // position:  6, index: 32
    public static final long AUG10  = 64L;   // position:  7, index: 31
    public static final long MAJ10  = 128L;  // position:  8, index: 30
    public static final long MIN10  = 256L;  // position:  9, index: 29
    public static final long AUG9   = 512L;  // position: 10, index: 28
    public static final long MAJ9   = 1024L; // position: 11, index: 27
    public static final long MIN9   = 2048L; // position: 12, index: 26
    public static final long OCTAVE = 4096L; // position: 13, index: 25
    public static final long MAJ7   = 8192L; // position: 14, index: 24
    public static final long MIN7   = 16384L; // position: 15, index: 23
    public static final long MXM7   = 32768L; // position: 16, index: 22
    public static final long AUG6   = 65536L; // position: 17, index: 21
    public static final long MAJ6   = 131072L; // position: 18, index: 20
    public static final long MIN6   = 262144L; // position: 19, index: 19
    public static final long MXM6   = 524288L; // position: 20, index: 18
    public static final long AXA5   = 1048576L; // position: 21, index: 17
    public static final long MAJ5   = 2147483648L; // position: 22, index: 16
    public static final long MIN5   = 4294967296L; // position: 23, index: 15
    public static final long MXM5   = 8589934592L; // position: 24, index: 14
    public static final long AUG4   = 17179869184L; // position: 25, index: 13
    public static final long MAJ4   = 34359738368L; // position: 26, index: 12
    public static final long MIN4   = 68719476736L; // position: 27, index: 11
    public static final long MXM4   = 137438953472L; // position: 28, index: 10
    public static final long AUG3   = 274877902048L; // position: 29, index: 9
    public static final long MAJ3   = 549755804096L; // position: 30, index: 8
    public static final long MIN3   = 1099511627776L; // position: 31, index: 7
    public static final long MXM3   = 2199023255552L; // position: 32, index: 6
    public static final long AUG2   = 4398056411104L; // position: 33, index: 5
    public static final long MAJ2   = 8796112822208L; // position: 34, index: 4
    public static final long MIN2   = 17592225644416L; // position: 35, index: 3
    public static final long MXM2   = 35184451288832L; // position: 36, index: 2
}
public static final long ROOT = 137438953472L; // position: 38, index: 0

//===========================================================================
//= Enumeration for every possible note. Value passed into constructor in
//= declaration is the NoteType MIDI value.
//===========================================================================
enum NoteType {
    A_FLAT(56), A(57), A_SHARP(58),
    B_FLAT(58), B(59), B_SHARP(60),
    C_FLAT(59), C(60), C_SHARP(61),
    D_FLAT(61), D(62), D_SHARP(63),
    E_FLAT(63), E(64), E_SHARP(65),
    F_FLAT(64), F(65), F_SHARP(66),
    G_FLAT(66), G(67), G_SHARP(68);

    private final int MIDI_value;

    //= Constructor
    //= Constructor
    NoteType(int MIDI_value) {
        MIDI_value = MIDI_value;
    }

    //= Return NoteType MIDI value
    //= Return NoteType MIDI value
    protected int MIDI_value() {
        return MIDI_value;
    }
    ..
}
APPENDIX L: Test Results

C: Color Objective  
Q: Quality Objective

<table>
<thead>
<tr>
<th>C</th>
<th>Q</th>
<th>Chord Progression</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>[BbMaj(no5), E#13(no11), D#7#9, C6/9(no5), C7sus2(b5), A#sus4(b2b5b6), G#7#11, D7#9]</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>FbMaj7(no5), Bb7sus4, B11#5, Bbm#5, B#add9(#11), B#m9</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>C#13(no5), A#m7(no3), F#+(maj13), B13(no9), Ab7b5, Am(maj), A6(no5), Ebhalfdim(b9)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>C#7sus2(b5), B#7b13, EMaj6(no5), Ch9, EMaj13(no9), Abm(maj9)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Eadd4, Cm(maj), G7b13, D(mystic), Fb7b13, Fbaug7b9, Abm</td>
</tr>
<tr>
<td>-30</td>
<td>-30</td>
<td>Db6(no5), G#(petrushka), E#7b5, F#11#5, C#:Cmaj6, Gb(petrushka), F7sus4(b5b9)</td>
</tr>
<tr>
<td>-30</td>
<td>-30</td>
<td>C#m(maj), D13(no5), C#:petrushka, Eb7(no5), G(petrushka), Ab(electra), G+(maj11), Bb(petrushka)</td>
</tr>
<tr>
<td>-30</td>
<td>-30</td>
<td>BMaj13(no5), Cdim(m9), D9(no5), E#:petrushka, Gbmhalfdim(b9), Fb(electra), F#9(no5), E#:petrushka)</td>
</tr>
<tr>
<td>-30</td>
<td>-30</td>
<td>Ch7#9, Gb(petrushka), F#Maj13(no5), Gb(petrushka), Csus4(b2b5b6), G#(petrushka), E#dim(m9), D(mystic)</td>
</tr>
<tr>
<td>-30</td>
<td>-30</td>
<td>Ghalfdim13, C#:petrushka, Fbsus4(b2b5b6), Fb(petrushka), Fb13#5, Db(petrushka), Dbm13</td>
</tr>
<tr>
<td>-30</td>
<td>30</td>
<td>Em(maj7), Abdim(Maj7), C#dim, D#dim7, D#dim7, E#m(no5), A#m7b5, A#m7b5</td>
</tr>
<tr>
<td>-30</td>
<td>30</td>
<td>Abm7(add4), Fbm7b5, Bdim, D#dim(m9), Dbdim11, Gbdim11</td>
</tr>
<tr>
<td>-30</td>
<td>30</td>
<td>Dm(maj11), D#:m(maj11), Dbm(maj), E(petrushka), E(petrushka), Ab(petrushka)</td>
</tr>
<tr>
<td>-30</td>
<td>30</td>
<td>Cm(maj11), Fm6#9, Fm6/9, G(petrushka), G(petrushka), Gb(petrushka)</td>
</tr>
<tr>
<td>-30</td>
<td>30</td>
<td>Fm(maj7), Dm(maj9), Dm(maj9), DbMaj(no5), DbMaj(no5), A(petrushka), A(petrushka), A(petrushka)</td>
</tr>
<tr>
<td>30</td>
<td>-30</td>
<td>E11#5, Db(tristan), DbMaj7#11, Db(tristan), Eb+(maj13), Gb(mystic), Gbmaj13(no5), D#(tristan)</td>
</tr>
<tr>
<td>30</td>
<td>-30</td>
<td>B#aug, Db+(maj13), D(tristan), D13sus4, Db(tristan), Amaj13, E(#tristan), D#Maj13</td>
</tr>
<tr>
<td>30</td>
<td>-30</td>
<td>CMaj9(no5), F#(tristan), Abmaj13(no11), F#(tristan), E#+(maj9), Db(tristan), Bmaj13, Db(tristan)</td>
</tr>
<tr>
<td>30</td>
<td>-30</td>
<td>FbMaj9(no5), D13#5, G(tristan), Cmaj13(no9), G(tristan), E+(maj13), G(tristan), Gbmaj13(no5)</td>
</tr>
<tr>
<td>30</td>
<td>-30</td>
<td>G6/9, D#(tristan), D#Maj13, D#(tristan), Cbm(maj13), D#(maj9), E(#tristan), D#Maj13</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>E7#9, G#Maj11, Gmaj13, A#+(maj13), Bb13#5, Gmaj13, A#+(maj13), D#6/9</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>EMaj11, Dbmaj13(no9), E#13#5, A#Maj7#11, Gbmaj13, Gbmaj13, D#Maj13(no9), DMaj13</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>C#Maj6, D#Maj13, E#Maj13(no9), G#Maj13, D+(maj11), D6/9(no5), D#Maj13, G+(maj13)</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>Cbm(maj9), C+(Maj7), F+(maj13), B#Maj13, EMaj13(no9), F+(maj13), G#Maj13(no9), G+(maj13)</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>Gbm6/9, EMaj13(no9), CMaj13(no5), C#+(maj13), Fbmaj7, D#+(maj13), G+(maj13), D#+(maj13)</td>
</tr>
</tbody>
</table>

Table 8: Test Results for Elitism 0, Crossover Rate 0.95, Mutation Rate 0.1
C: Color Objective  
Q: Quality Objective

<table>
<thead>
<tr>
<th>C</th>
<th>Q</th>
<th>Chord Progression</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Bbm7(no5), B6, G7b13, G#m9b5, D#13#5, F#5, A#7#11, A7b5#9</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Ebaug7b9, E#5, Eb+(maj13), G#(mystic)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>F#5, Bdim(Maj7), B#11, BbMaj6, Am9, G#7#5, Gb11#5</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>C#6/9(no5), Fm(b6), Bfaug7b9, BMaj13(no11), C#m13</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Cb6(#11), E#m7, E#ad9(#11), Ddim9, Fb7#9, D6(no5)</td>
</tr>
<tr>
<td>-30</td>
<td>-30</td>
<td>Db6/9, Chm7, Ab(petrushka), ChHalfdim13, F#(electra), E(petrushka), Cb+(maj9), Eb(petrushka)</td>
</tr>
<tr>
<td>-30</td>
<td>-30</td>
<td>E7sus2(b5), A(petrushka), Fb9b5, Dbm(maj13), Cb(petrushka), F5, Eb(petrushka), G#13(no5)</td>
</tr>
<tr>
<td>-30</td>
<td>-30</td>
<td>Cm7(add4), Cbdim(Maj7), G#(petrushka), Dadd9, D#(petrushka), E#13sus4, G#(petrushka), Cb13sus4</td>
</tr>
<tr>
<td>-30</td>
<td>-30</td>
<td>Abadd9(#11), EbbHalfdim11, A#(petrushka), Ab6(#11), Cb(petrushka), G#m9b5, Gbb7b9, A#(petrushka)</td>
</tr>
<tr>
<td>-30</td>
<td>-30</td>
<td>Gb+(maj13), Ab(petrushka), A#7sus2(b5), B(petrushka), F#7sus2(b5), Csus4(b2b5b6), B#+(Maj7), E(petrushka)</td>
</tr>
<tr>
<td>-30</td>
<td>30</td>
<td>BbTritone(add6), Cbdim, Ddim7, Dbm7b5, G#dim(Maj7), Fbdim(Maj7), Dbm7b5, C#m7b5</td>
</tr>
<tr>
<td>-30</td>
<td>30</td>
<td>C#m(maj11), Gm6/9, Gm6/9, Cb(petrushka), B(petrushka), C(petrushka)</td>
</tr>
<tr>
<td>-30</td>
<td>30</td>
<td>Ab7#11, Bbdim11, Gbdim11, Gbdim11, Gbdim11, Fbdim11</td>
</tr>
<tr>
<td>-30</td>
<td>30</td>
<td>C#7#5, D#m(no5), Fbm(maj9), Fbm(maj9), Fbm(maj9), Cb(petrushka), Cb(petrushka), C#(petrushka)</td>
</tr>
<tr>
<td>-30</td>
<td>30</td>
<td>F#9(no5), E#dim11, Em7b5, Fbm7b5, F7sus4(b5b9), F7sus4(b5b9), F7sus4(b5b9)</td>
</tr>
<tr>
<td>30</td>
<td>-30</td>
<td>B#+(maj11), C(tristan), D#13#5, F#(tristan), Db(mystic), D#13#5, F#(tristan), D#13#5</td>
</tr>
<tr>
<td>30</td>
<td>-30</td>
<td>Cb11#5, G(tristan), C#(+)(maj13), B(tristan), C#(+)(maj13), G(tristan), C#(+)(maj13)</td>
</tr>
<tr>
<td>30</td>
<td>-30</td>
<td>E7#11, C#Maj6, E#(tristan), D#+(maj13), Cb(tristan), C#13, E#(tristan), D#+(maj13)</td>
</tr>
<tr>
<td>30</td>
<td>-30</td>
<td>Bbaug, B#Maj13, E(tristan), Gb(mystic), CbMaj13, A(tristan), B#Maj13, E(tristan)</td>
</tr>
<tr>
<td>30</td>
<td>-30</td>
<td>Daug7#9, B#+(maj13), Gb(tristan), E#+(maj9), C#(tristan), Am13, B#(tristan), B#+(maj13)</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>G+(maj13), D13(no11), A13#5, C+(maj13), F#+(maj13), Bb11#5, E#Maj13, Db6/9</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>FMaj13(no9), E13sus4, AMaj13(no11), BMaj13(no9), D13#5, Gb+(maj13), Db+(maj11), Bb+(maj13)</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>C13#5, E#+(maj13), GbMaj13, E#+(maj13), E#+(maj13), C#aug7#9, E#+(maj13)</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>CMaj13, Daug9(#11), FbMaj13(no5), DMaj13, CMaj13(no9), Bm(maj13), Gb+(maj13), DMaj13(no9)</td>
</tr>
<tr>
<td>30</td>
<td>30</td>
<td>Caug7b9, Bb+(maj13), G+(maj13), ChMaj13, A#Maj13(no9), B#13#5, GMaj13(no9), G+(maj13)</td>
</tr>
</tbody>
</table>

Table 9: Test Results for Elitism 1, Crossover Rate 0.95, Mutation Rate 0.1
C: Color Objective
Q: Quality Objective

<table>
<thead>
<tr>
<th>C</th>
<th>Q</th>
<th>Chord Progression</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>C#Maj7(no5), D#add9, Eb9(no5), D#5, D13(no11), Gbdim(m9), A(tristan)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Cb+(maj11), Bbaug, Em9b5, Dbsus4, BbMaj6(no5), Caug7b9</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>C#m(maj7), DMaj9, F+(Maj7), Eb9b5, Ch7sus4, Fbm(b6), Fbaug7b9, G6</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>B7b13, E7sus4, Abm13, Cb7#5, G#sus2, E#m(maj13)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>Fhalfdim13, D#m(no5), Fadd4, E9(no5), CbMaj6, B#aug, Dbfrankenstein, DMaj(no5)</td>
</tr>
</tbody>
</table>

-30 -30 A7sus2(b5), A#(petrushka), GMaj13(no9), Cb(petrushka), F#5, C#aug7b9, Cb(petrushka), F#5 |
-30 -30 C(petrushka), G#13(no9), F#7sus4(b5b9), CbMaj(no5), G#m9, Gb(petrushka), Db13#5, Eb(petrushka) |
-30 -30 C13(no9), Fb(petrushka), Bm(maj13), C7sus4(b5b9), Eb(petrushka), G7#5, Eb(petrushka) |
-30 -30 BbMaj6, Gb(petrushka), C#+(maj13), G#dim11, Bbdim9, Ab(petrushka), Caug7b9, Gb(petrushka) |
-30 -30 D+(maj9), Gb(electra), B9, Db(petrushka), C#+(maj11), C(petrushka), G#(electra), C(petrushka) |

-30 30 G#, G#m(no5), G#m(no5), Gdim11, Gdim11, Gdim11, Gdim11, G#m(no5) |
-30 30 C#add9, B5, B5, B5, Fb(petrushka), Fb(petrushka), E(petrushka) |
-30 30 Db5, Ab9, Ab9, Ab9, E(petrushka), E(petrushka), A#(petrushka) |
-30 30 G#(electra), Bbhalfdim11, Bbhalfdim11, Bbhalfdim11, Bbhalfdim11, Bbhalfdim11, Bbhalfdim11, B#(petrushka) |
-30 30 G#dim, Fbdim, B#dim7, Am7b5, Fsus4(b2b5b6), Fsus4(b2b5b6), Fsus4(b2b5b6) |

30 -30 Cb+(Maj7), C(tristan), F#(maj13)(no9), F#(tristan), GMaj13(no5), A(tristan), CbMaj13(no11), C(tristan) |
30 -30 BbMaj13(no9), Eb(tristan), Bb+(maj13), Eb(tristan), Cm7(no3), E(tristan), Eb+(maj13), E(tristan) |
30 -30 F#11#5, Eb#Maj13(no9), Cb(tristan), G#Maj13, Eb(tristan), G#Maj13(no5), B#(tristan), G#Maj13(no5) |
30 -30 G#11, B#(mystic), B#(tristan), A#Maj13, B#(tristan), A#Maj13, B#(tristan), A#Maj13 |
30 -30 Gb6(#11), D(tristan), EMaj13(no5), A(tristan), G#m(maj13), D(tristan), G#(mystic), E#+(maj13) |
30 30 B(b5), A#Maj13, C#+(maj13), C#+(maj13), E#13, C#+(maj13), FbMaj13(no11), C#+(maj13) |
30 30 Eb+(maj13), CbMaj13(no11), G#Maj7(no5), A+(maj13), Dm(maj13), GMaj13, CMaj13(no11), Fb+(maj13) |
30 30 Bb6#9, Cb(maj13), E#13(no5), Cb+(maj13), C#Maj11, Fb13, GMaj13, Ab+(maj13) |
30 30 C#Maj13(no11), Db+(maj13), D#Maj13(no11), BbMaj13(no5), D#Maj13(no11), Cb+(maj11), AMaj13(no9), D#Maj13(no11) |
30 30 C#Maj13, DMaj13(no11), Fb+(maj13), G#Maj13, F#+(maj13), AMaj6(no5), FbMaj7#11, A+(Maj7) |

Table 10: Test Results for Elitism 0, Crossover Rate 0.7, Mutation Rate 0.3
<table>
<thead>
<tr>
<th>C</th>
<th>Q</th>
<th>Chord Progression</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0</td>
<td>0</td>
<td>Cbdim(m9), F9(no5), A#sus4, FMaj7(no5), Bb9b5, G6(no5), DbMaj13(no9), Bb13</td>
</tr>
<tr>
<td>0 0</td>
<td>0</td>
<td>Bb7#9, FMaj6, F#sus2, Gb7b9, A7sus4(b5b9), E#+(maj11)</td>
</tr>
<tr>
<td>0 0</td>
<td>0</td>
<td>BbMaj7(no3), Cb13, C#m7(no5), B#Maj9, DMaj7(no5), Db9b5, A6</td>
</tr>
<tr>
<td>0 0</td>
<td>0</td>
<td>Gb+(maj9), G13, D#Maj7, C11#5, B#(electra)</td>
</tr>
<tr>
<td>0 0</td>
<td>0</td>
<td>Ab9b5, C#m(maj7), G#aug7b9, Caug7#9, F#7b5, Ab6, Dfrankenstein, DMaj7#11</td>
</tr>
<tr>
<td>-30 -30</td>
<td>-30</td>
<td>C13(no9), Gbsus4(b2b5b6), A#(petrushka), B6(no5), D#dim(Maj7), Ab(petrushka), Gb13#5, D#(petrushka)</td>
</tr>
<tr>
<td>-30 -30</td>
<td>-30</td>
<td>A#(petrushka), C#13#5, B#(petrushka), D9, Gbsus4(b2b5b6), A#sus4, Ab(petrushka)</td>
</tr>
<tr>
<td>-30</td>
<td>-30</td>
<td>A5, C(petrushka), A#7#11, A#(petrushka), Cb13sus4, AbMaj7(no3), F#(petrushka), Db7sus4(b5b9)</td>
</tr>
<tr>
<td>-30 -30</td>
<td>-30</td>
<td>CMaj9(no5), G(petrushka), Fbhalfdim13, C(petrushka), Bb13(no5), Fbdim(petrushka)</td>
</tr>
<tr>
<td>-30 -30</td>
<td>30</td>
<td>G#m7b5, Ahalfdim(b9), Fbdim, B#(petrushka), A#(petrushka), C#7b5</td>
</tr>
<tr>
<td>-30 -30</td>
<td>30</td>
<td>ChbMaj(no5), E6(no5), Gb6(no5), G#(petrushka), G#(petrushka), C(petrushka)</td>
</tr>
<tr>
<td>-30</td>
<td>-30</td>
<td>G#dim7, A7sus4(b5b9), A7sus4(b5b9), A7sus4(b5b9), A7sus4(b5b9), B#halfdim11, A7sus4(b5b9)</td>
</tr>
<tr>
<td>-30</td>
<td>-30</td>
<td>E(mystic), A11, A13#5, F#13(no5), Fb(petrushka), Bf(petrushka), Fb(petrushka)</td>
</tr>
<tr>
<td>-30</td>
<td>-30</td>
<td>E#+(Maj7), D#m7(add4), D#m7(add4), D#m7(add4), D#m7(add4), D(petrushka), Gb(petrushka)</td>
</tr>
<tr>
<td>30 -30</td>
<td>-30</td>
<td>Fb+(Maj7), Cb(mystic), GbMaj13(no9), B(tristan), GbMaj13(no5), F(tristan), Gb+(maj13), C#(tristan)</td>
</tr>
<tr>
<td>30 -30</td>
<td>-30</td>
<td>D#Maj13(no11), D#(tristan), A(tristan), G#(petrushka), G#(petrushka), C#(tristan)</td>
</tr>
<tr>
<td>30 -30</td>
<td>-30</td>
<td>G#m7b13, F#Maj13, D(tristan), FMaj13, D(tristan), Abm(maj13), A#(tristan), G#+(maj13)</td>
</tr>
<tr>
<td>30 -30</td>
<td>-30</td>
<td>F9, D#(tristan), Cb+(maj13), B+(maj13), B#(tristan), Bbbad9(#11), A#+(maj13), Gb(tristan), G13</td>
</tr>
<tr>
<td>30 -30</td>
<td>-30</td>
<td>D13#5, Cb(tristan), AMaj13, B#halfdim13, Cb(tristan), AMaj13, Db(tristan), AMaj13</td>
</tr>
<tr>
<td>30 -30</td>
<td>-30</td>
<td>C13(no9), Gbsus4(b2b5b6), A#(petrushka), B6(no5), D#dim(Maj7), Ab(petrushka), Gb13#5, D#(petrushka)</td>
</tr>
<tr>
<td>30 -30</td>
<td>-30</td>
<td>F+(maj11), B+(maj13), F8, E#13, B+(maj13), B+(maj13), DMaj13(no11), EbMaj13(no9)</td>
</tr>
<tr>
<td>30 -30</td>
<td>-30</td>
<td>Bbaad9, D+(maj13), E#+(maj13), F#Maj9, E#Maj13(no9), G#m(maj13), E#+(maj13), CMaj13</td>
</tr>
<tr>
<td>30 -30</td>
<td>-30</td>
<td>F#Maj11, Eb+(maj13), Gb6, FbMaj13(no11), FbMaj13(no11), EbMaj13, Db+(maj13), FbMaj13(no9)</td>
</tr>
<tr>
<td>30 -30</td>
<td>-30</td>
<td>Db+(maj13), E7#11, FbMaj13(no11), D#Maj13(no9), CbMaj13(no9), EbMaj13(no9), Eb+(maj13)</td>
</tr>
</tbody>
</table>

Table 11: Test Results for Elitism 1, Crossover Rate 0.7, Mutation Rate 0.3
APPENDIX M: Open Issues

[P1 defect]: Frankenstein chord showed up in completed chord progression.

[P1 defect]: Fitness of fittest individual is -0.1 in objectives where color = 0 and quality = 0. This value should never fall below 0.

[P1 defect]: In GUI, displayed chord count sometimes different than played chord count. Seems related to stop/resume feature.

[P1 defect]: Frankenstein chord may allow different notes of the same pitch (B_SHARP & C). Check for this potential issue.

[P2 defect]: Commandline usage incorrect for class without unit tests. Example: SequenceCreator has no unit tests yet reports options testRuns= and verboseTest. Also, command-line option "help" results in missing argument message. Help should only result in help information.

[C:\Users\jkeeling\projects\school\CSc_502\speKculator\src>java CommandLine count=7 type=Garg]

Test Summary:
run count: 1
test count: 0
-------------------
tests run: 0
tests passed: 0
tests failed: 0

[P2 defect]: Chord progressions of type key= mode= have no allegiance to the key= note. This note should probably have a higher probability of occurrence in the first chord.

[P2 feature]: Need to add ChordType entries for chords with optional notes. See:

Consider how to weight the probability of random chord selection so that optional note versions of chords are appropriately lessoned.

[P3 feature]: Need to address symmetric in chords (i.e diminished)
[P3 feature]: Consider/implement chord inversions

[P3 feature]: Should a Chord Progression include an entry whose audio value is silence? This may provide a transition towards application of Meter class.
[P3 defect]: GUI: chord progression field not wide enough

-----------------------------------------------------------------------------

[P4 feature]: Note class. TODO comment to get away from hard-code MIDI value. Replace with dynamically derived value.

[P4 feature]: Implement enumeration of ProgressionType. Include all standard progressions (i.e. I-II-V). NOTE: This feature requires timing information to provide accurate execution of standard chord progressions.
WORKS CITED


