Using an Interactive Genetic Algorithm and Markov Chain to Compose Melodies Relevant to the User's Taste

By

Ben Gelb

A Project Proposal Submitted in Partial Fulllment of the Requirements for the Degree of Master of Science in Computer Science

Supervised By Dr. Joe Geigel

November 24, 2013

Contents

1 Abstract

Music can be created by anything; it is the melody, which leaves a lasting impression. Simulating the process of evolution on a computer allows one to witness and impact the creation of melodies. I propose a technique to have the computer compose simple, but euphonic melodies by using a Markov chain and an Interactive Genetic Algorithm (IGA). A Markov chain is created from a user selected corpus, and is used to generate the initial population where individuals resemble musical segments the user enjoys. As a result, the initial population will consist of individuals relevant to the user's musical taste. Coupling the Markov chain with an IGA that allows the user to make modifications to each melody by re-ordering the notes, decreases the total number of generations the IGA needs to run for. Additionally, the type of interaction that the fitness function provides is much more engaging and fun for the user than previous approaches, which results in a decreased user burnout rate.

2 Introduction

Artificial intelligence (AI) aims to create systems capable of solving problems in various domains. One such domain is melody composition. By simulating evolution on a computer, one is capable of witnessing and impacting the creation of melodies. However, "creating melodies which human listeners would appreciate has shown to be difficult with current AI techniques" [8]. Although melody composition can be modeled using mathematics, "it distinguishes itself by also involving human emotions and aesthetics, which are domains not fully understood and also difficult to describe mathematically" [8]. Thus, teaching the computer to compose appealing melodies proves to be a challenging problem as composers have their own, unique styles, which are not confined to existing rules. If composers cannot fully model the process when they create melodies then how can it be adapted to the computer?

Genetic algorithms (GA) have been around for decades and have proven "to be a powerful [tool] for searching in complex domains too difficult to tackle using analytical methods" [8]. While using this technique to compose melodies may appear to offer promising results, the biggest challenge will be the design of a fitness function. An automatic fitness function, which relies on mathematics to evaluate melodies is not dependent upon a human component, and therefore is more efficient as it directs "evolution towards pleasant melodies" [8]. However, this system will not fully capture the user's preferences as it is unable to translate subjective trends. A more suitable approach to capture the user's preferences is the utilization of an interactive fitness function, referred to as an interactive genetic algorithm (IGA). While less efficient and more strenuous on the user, the likelihood that the melody reflects the user's preference is much greater. By manually ranking each melody, the user will influence how future pieces sound, thus ensuring it aligns with the user's taste. Hence, composing a melody is not just about placing characteristics in their technically correct positions, it involves using emotion as a way to guide the composer through the complex domain that is music.

Coming up with a technique to algorithmically compose melodies that are relevant to the user's taste will require a couple of modifications to the typical IGA.

One of the most influential components of an IGA is the generation of the initial population. The initial population serves as the IGA's starting point, and plays a large role in how many generations an IGA needs to run for before seeing decent individuals. In order to minimize the number of bad generations, subsets from a Markov chain are used to represent individuals in the initial population. The Markov chain is based on a user selected musical corpus. Typical IGA evaluations require a user to listen to a melody, and then provide it a score based on some internal, subjective rubric. It is a very linear process and doesn't truly capture how a user feels because all the IGA can see is a number. Instead, the IGA will require a user to listen to the melody, and then allow the user to actually play with and make modifications to the individual. Modifications include re-ordering the melody's notes. The types and number of changes made have different impacts on the individual's fitness score, and more importantly, provide the IGA with information on what the user liked and didn't like about the original individual. By using standard genetic operators, but altering how the initial population is generated and the type of feedback the user supplies, the IGA should produce interesting, relevant, and unique melodies.

3 Problem

IGAs are prone to a fitness bottleneck when used for melody composition making it difficult to compose a compelling solution. Specifically, users experience fatigue when required to evaluate individuals over an extended period of time. Fatigue affects many stages of evolution resulting in under-evolved populations. Additionally, users are often inconsistent with their evaluations during the early and later generations [3]. It is certainly possible to create a compelling solution under these circumstances, however this requires a certain amount of finesse due to the amount of influence the bottleneck has on the design and quality of the system [3].

Typically, IGAs run for a limited number of generations as a user is unable to sit and evaluate millions of individuals in a reasonable amount of time. However, placing these constraints on an IGA limits how quickly a population evolves from generation to generation, often resulting in under-evolved individuals. Although

the elimination of the human component can produce "intellectually stimulating noise" [2] by measuring the technical merit of an individual, it does not account for the user's musical preference, and lacks a compelling dimension, which can only be achieved through subjective feedback [3]. Furthermore, a large portion of the user's time is spent evaluating populations dominated by noisy data. This is largely influenced by the quality of the initial population and consistency in user evaluations. Inconsistent user feedback slows down the evolutionary process. Specifically, the amount of random noise found in early generations results in the user losing focus and assigning random scores [3]. This confuses the IGA and causes bad individuals to influence the next generation instead of being filtered out.

Fatigue is the only constant in this problem that is guaranteed to happen after the n^{th} generation. It is a time dependent issue that poses a major threat to the IGAs performance. While limiting the size of the population and the number of generations reduces the amount of time the IGA runs, it also hinders the IGAs performance and the quality of the solutions. Early generations tend to be filled with noise representing random, unstructured individuals with a minimal number of relevant features. Even when a considerable amount of the noise has been filtered out, and the individuals start becoming decent, the rate of growth from one generation to the next is gradual. When time is of the essence it becomes essential to either replace the bottleneck or build around it. Therefore, it is imperative to design a system that accounts for human fatigue so that it does not dramatically alter the results.

4 Previous Research

Eliminating the fitness bottleneck prohibits the user from projecting their musical preference into the melody. Therefore, attempts have been made to minimize the bottleneck by studying the user's rating habits over a period of time [12]. Neural networks (NN) have been able to replicate the user's taste to a degree allowing the IGA to run for a longer number of generations. While the auto-rater is able to maintain a consistency in its scoring, it is minimally successful in mimicking the user's taste, a result of an under-trained NN [2].

The process of evaluating an individual typically require the user to listen to the melody and then assign it a score based on a linear scale representing the user's opinion of the overall piece. The inherent flaw is that even if a user enjoyed bits and pieces of the melody, the section that was enjoyed will typically be overlooked as the individual's evaluation is based on the entire piece, and not segment by segment. This type of evaluation is often the slowest to receive decent individuals due to its vague nature.

The concept of quality and quantity often go hand-in-hand in IGAs; supplying more consistent feedback results in better individuals. According to Biles, a population should be represented as a single continuous knowledge base [5]. In doing so, a unity among the population is achieved making evolution more fluid. The user is able to rank each gene in real-time ensuring that the good ones aren't overlooked. Additionally, the IGA knows specifically what the user liked and didn't like, and as a result, has reduced the number of bad generations in the beginning [5].

An IGA needs a starting point, which can be generated randomly. However this can be problematic as it increases the amount of noise found in early generations, which in turn increases the amount of time the user is required to evaluate poor populations. The concept of randomly generating individuals is similar to pressing random keys on a piano and hoping for a pleasing melody. Surely an intriguing sound may occur on occasion, but relying on such an inconsistent approach is not ideal given the strict time constraints.

Seeding the initial population has been a promising approach given its ability to set boundaries in the otherwise infinite and random space that music offers [4]. Providing a sense of structure, even minimally, to each of the individuals reduces the amount of noise and decreases the gap between the starting population and the optimal solution. Determining the boundaries for generating the initial population are generally user defined or based on statistics. A common approach is to generate a Markov chain using user defined constraints, e.g. only minor chords. Doing this has shown to enhance the initial population's starting state. According to [10], larger text often results in more choices at each transition, providing higher quality

results. Evolution, even in its natural domain, is a slow process [11]. Any structure or guidance will have a positive impact on the quality of early generations.

5 Overview

5.1 Interactive Genetic Algorithm

An IGA is a search algorithm that has been influenced by Mendel's laws of classical genetics and Darwin's theory of evolution [8]. It has successfully solved problems in natural, social, and formal sciences, while sometimes generating unexpected and creative solutions. Formal IGAs structure themselves around the elements found in natural evolution, meaning there's a "population, diversity within the population, a selection mechanism, and genetic inheritance" [8]. The process begins by creating a pool of individuals to use as the starting point. Each individual, known as a genome, represents a chromosome. Genomes can be presented in many formats, but the most common one is a "set of binary integers such as 1001011" [9]. Each member of the population is assigned a fitness score, which is a ranking that determines how fit it is. In an IGA, the fitness score is determined by a human, who evaluates each individual based on a personal, subjective rubric. Once the entire population has been evaluated, each individual is then passed to another function, which selects individuals from the population to reproduce [9]. The selected chromosomes then crossover generating a set of children, which are used in the next generation. The children then have a chance of being chosen to undergo mutation in order to help keep the population diverse by randomly altering chromosomes. This process is repeated until the termination requirements are met.

5.2 Markov Chain

A Markov chain is a Markov process, which has a discrete state-space, and is used to represent a system that changes over time. It models transitions from one state to another for random, independent events, i.e., "the next state depends only on the current state and not on the sequence of events that preceded it" [13]. It works by creating a model from a collection of random variables. The model is a set, S, of states, $S = s_1, \ldots s_r$, and the transitional probabilities $P_{i,j}$ between

the states. $P_{i,j}$ "is the probability that the Markov chain is at the next point in state j given that it is at the present time point at state i" [1]. Thus, the Markov chain can be represented as $\Pr(X_{n+1} = x | X_n = x_n)$. The word "chain" is used to describe the output of this process because the results are often "chained" together.

This is similar to using a Markov chain to generate sentences where the output may not make complete sense, but grammar, sentence structure, and dialogue from the original text will be followed to a degree [6]. By setting a prefix size, a model can be built by gathering statistics about the frequency of which words follow certain prefix groups of other words [6]. For example, if N=2, the next generated words are based on the previous two words.

6 Hypothesis

This research will focus on creating a system that can compose a compelling melody. In doing so, I will not only gain an understanding of how IGAs can be applied to subjective domains, but also a deeper understanding of music structure. While IGAs are often popular choices for solving creative problems, previous implementations require minimal human feedback due to user fatigue. By altering the type of feedback the system requires, user fatigue becomes less of an issue and allows the IGA to arrive at an optimal solution faster than implementations that require the user to manually enter in a score. Minimizing the fitness bottleneck, and thus speeding up the rate an IGA converges, requires a system designed to exploit the most influential components in the evolutionary process; the fitness function and the initial population.

The first step is to alter the type of feedback required by the fitness function. The user becomes the composer and can make a limited number of adjustments to the melody. In doing so, I hope to improve upon the rate at which an IGA evolves, while also decreasing the amount of time it takes for undesirable individuals to die out. This is because the IGA learns what the user liked, didn't like, and ultimately, the preferred structure of the melody. Additionally, a Markov chain will be used to generate the initial population. It will be built off of user selected melodies with the goal of reducing the occurrence of weak individuals found in

early populations. As a result, the individuals will have some structure embedded in them and contain a musical style that the user is already familiar with and likes.

6.1 Questions

Will this keep the user engaged and reduce burnout? How many individuals should be displayed at once? How many individuals from the generation should the user be required to listen to? Is it possible to bypass ones that are very similar? How many adjustments should the user be allowed to make per individual? How many influencers should be used?

7 Synthesis

7.1 System Architecture

The IGA is composed of three parts: a web-server (music server), a GA, and a redis database. The music server is responsible for interpreting and directing web requests. It is composed of various endpoints, some of which are RESTful and others which are used to progress the state of the GA. The GA is a standalone library, which can be run independently from the music server. The redis database is responsible for caching user options, individuals from each generation, and song features from the original MIDI file, which includes the notes, chords, and durations as well as the Markov chain generated for the specific song.

User experience (UX) is a crucial factor in the success of this system. Traditional software GUIs are not only tedious to program, but cant match the seamless experience provided by a web-application. Using the web triforce (HTML5, CSS3, and JavaScript) allows for a significant level of control over the UX and user interface (UI), which in turn are responsible for user burnout.

7.2 Breakdown of Music Server

The music server acts as a middle-man between the GA and redis. The RESTful component of the music server supports GET and PUT requests for querying and

updating a population from any generation and speicifc individuals. As a result of this architecture, the GA can be written in any language as long as it sends the required information to the music server about its current state. The RESTful endpoints can be seen below.

1. Generate a population
    - GET /spawn
    - URL query parameters:
        - psize (population size)
        - isize (individual size)
        - gsize (number of generations)
        - mcsize (Markov chain size)
        - mcnodes (Markov chain number of nodes)
        - artist
        - song
2. All individuals and their metadata up until the current generation
    - GET /population
3. All individuals and their metadata for the requested generation
    - GET /population/:generation
4. Metadata for the requested individual
    - GET /population/:generation/:id
5. Update a specific piece of metadata for an individual
    - POST /population/:generation/:id
    - URL query parameters:
        - key (fitness, notes, artist, song)

7.2.1 Extracting Metadata from MIDI files

Music21, a Python library for music analysis from MIT [7] is used to decode the MIDI file and convert the low-level structure into a human-readable format. A sample of converted output can be seen in Table 1. For this project, the file's notes, chords, and durations are extracted and stored in redis.

Table 1: Components of a Music21 Stream

0.0 music21.stream.Part object at 0x2a85d70
0.0 music21.instrument.Instrument P1: Soprano: Instrument 1
0.0 music21.stream.Measure 0 offset=0.0
0.0 music21.meter.TimeSignature 4/4
0.0 music21.clef.TrebleClef object at 0x2ca7e50
0.0 music21.key.KeySignature of 2 sharps
0.0 music21.note.Note E
0.5 music21.note.Note F#
1.0 music21.stream.Measure 1 offset=1.0

The output, referred to as a Stream, is converted to a python dictionary and stored with the song's metadata in a redis hash set. The hash set stores the original notes/chords that were extracted as well as the Markov chain that was generated for that song. An overview of the structure can be seen in Figure 1.

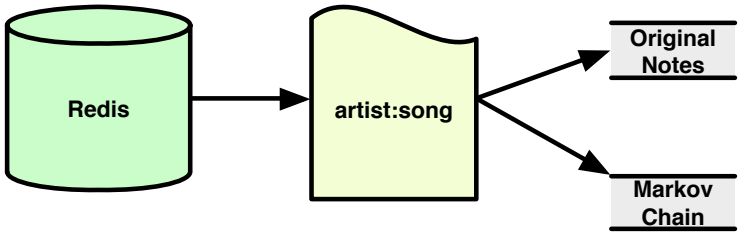

Figure 1: Extracted MIDI Structure

A Stream is organized by parts where each part contains either a nested Stream or a Measure. A Measure is a dictionary of concurrent music features where each key is a Music21 object and each value is the available music feature at a certain time segment. As a result, it is easy to extract and analyze specific features from

the Stream, e.g. finding a song's melody from timestamp x to y, harmony from timestamp i to j, most common note in parts m through n, etc.

7.3 Breakdown of Interactive Genetic Algorithm

7.3.1 Initialization Settings

The user is responsible for determining the initialization settings of the IGA. This is done by presenting the user with an initialization web-page and requiring all the form fields, which represent the required settings, to be filled out before the IGA can begin. These fields include: the influencer, the Markov chain size, the Markov chain history size, the number of notes per melody, the population size, the number of generations, and the mutation rate. This is can be seen in Figure 2.

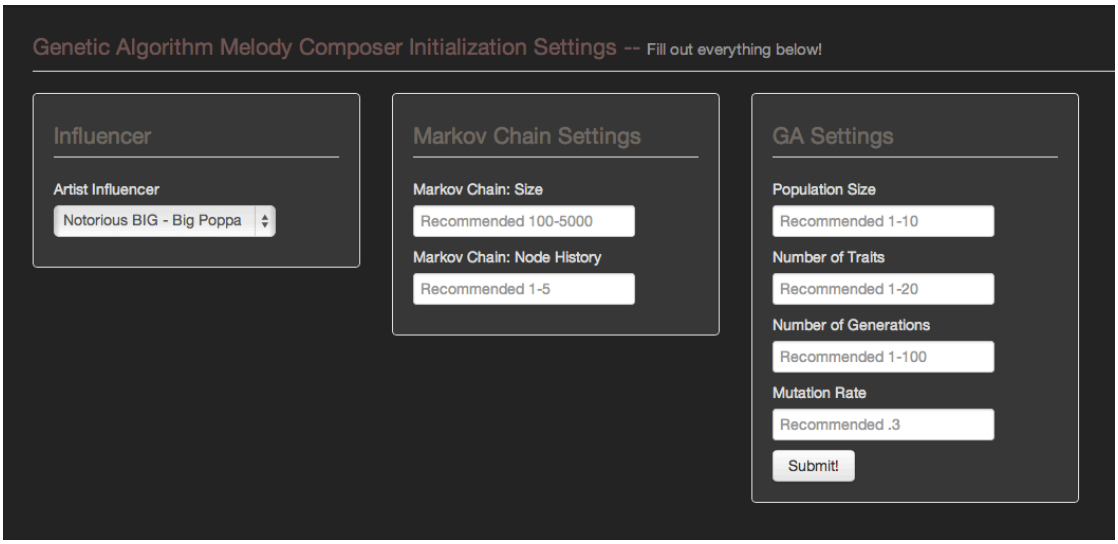

Figure 2: Initialization Settings

7.3.2 Influencer

The influencer's role is to provide the IGA with structure and guidance when picking individuals from the search domain. This is a crucial component because it impacts the number of noisy individuals typically found in the initial population.

Suppose that the search domain is generated randomly and individuals are also

picked at random. For example, imagine that the search domain, S, only contains $[[A], [B, C, D], [E]]$ where a single element in a nested list represents a note and multiple elements in a nested list represent a chord. If an individual is an array of size 2 containing any combination of elements in S, possible solutions include, but are not limited to $[[A], [A]]$, $[[A], [B, C, D]]$, or $[[B, C, D], [E]]$. The immediate problem is that theres no order or understanding of how to go about picking solutions; it is a free-for-all. This isnt much of an issue when dealing with an automatic fitness function because the IGA has more time to evolve and filter out the noise due to the increased number of generations. However, this is not as simple when using an interactive fitness function because the number of generations are limited due to user burnout. As a result, earlier generations have a greater influence on individuals in later generations.

Now, consider a search domain that is still generated randomly, but uses a rule-set to pick individuals, e.g. restricting the possible combinations to only allow major chords. Even if the rule-set is simple, the added structure yields better individuals because there's now a pattern to picking. However, this research is interested in generating melodies relevant to the user's musical taste, not on technical correctness. Such a rule-set would need to be generated dynamically in order to incorporate the user's preferences. The required rule-set to do this would be extremely complex making this an unattractive possibility.

To solve this problem, a Markov chain is used to generate the search domain. It eliminates the need to embed rule-sets in the software and also provides the IGA with a search domain that represents the user's music preferences. In other words, the search domain is no longer polluted with random noise, and instead contains relevant data. This is similar to using a Markov chain to write sentences where the sentence may not make complete sense, but grammar, sentence structure, and dialogue from the original text will be followed to a degree [6]. By setting a prefix size, a model can be built by gathering statistics about the frequency of which words follow certain prefix groups of other words [6]. For example, if n=2, the next generated words are based on the previous two words. When applied to music, this results in new melodies that contain the style and hints of the original.

Once the Markov chain generates a search domain, the IGA must create a pool of individuals for the initial population. In this instance, individuals can be created using random sampling or random continuous sampling. A random sampling is when random elements are selected from a list, e.g. if the search domain, S, only contains notes $[[C, C, A], [B, C, D], [A, B], [C\#, D], [A4, B5], [C6]]$ and an individual is an array of size 3, then a possible solution would be $[[A, B], [C6], [A, B]]$ or $[[A, B], [A, B], [A, B]]$. However, this process essentially negates the work of the Markov chain because all it is doing is dynamically restricting the type of notes available in the search domain. Instead, random continuous sampling is used because it ensures that the Markov chain output is used correctly and it also adds a level of diversity to the population. Random samples are still taken from the search domain, but instead of randomly picking a single element, a consecutive series of elements are picked, e.g. $[[C, C, A], [B, C, D], [A, B]]$ or $[[C\#, D], [A4, B5], [C6]]$. An important question is how much impact replacement has on the initial population, i.e. should a series of notes be discarded from the pool once picked? This is concept is known as independent vs. dependent events.

7.3.3 Markov chain Workflow

First, the user is responsible for both selecting an influencer from the list of available songs and setting the Markov chain constraints *size* and *order*. The Markov chain *size* represents the number of requested notes and the *order* is the desired number of states. The requested song is then sent to the music server where the title is used to find the song's corpus. Once the corpus is found, the notes are extracted and used to genereate a N^{th} order Markov chain. The Markov chain is then cached in redis as a hash set. Next, the IGA uses random sampling to generate the initial population.

7.3.4 Evaluation

The goal of this fitness function is two fold; improve the UX and improve the type of feedback. These factors are responsible for user burnout and the rate of evolution. The rate of evolution determines how well the current generation has evolved from the previous and can be measured by comparing the average fitness

score between the $N^{th}-1$ generation and $N^{th}-2$ generation. In this IGA, 0 is the best score an individual can receive, thus it is important that the average fitness score of the current generation is closer to 0 than the previous generation's.
The Euclidean distance, as seen in (1), is a technique used to determine the distance between 2 vectors, p and q.

$$d(\mathbf{p},\mathbf{q}) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2} \tag{1}$$

The IGA uses this to calculate an individual's fitness score by finding the amount of change that has occurred between the original melody, p, and the adjusted melody, q. The amount of change is determined by comparing the index locations of the notes in the original melody with the index locations of the notes in the adjusted melody. In order to do this, the note's original index location as well as the note's value are stored in a list of tuples as $[(idx_0, note_0), ..., (idx_n, note_n)]$. If the user swaps the first note with the second note, then the index of the first item in the adjusted melody list will have an index of 1, while the index of the second item in the adjusted melody list will have an index of 0. An example can be seen in Figure 3. Additionally, there needs to be a limit on the number of adjustments the user can make to ensure that the fitness scores aren't all over the place. Currently, the number of allowed adjustments are 30% of the total number of notes. Decimals less than 5 are rounded down, and decimals greater than 4 are rounded up. For example, if a melody contains 10 notes, then the user can only make 3 changes (30% of 10 = 3).

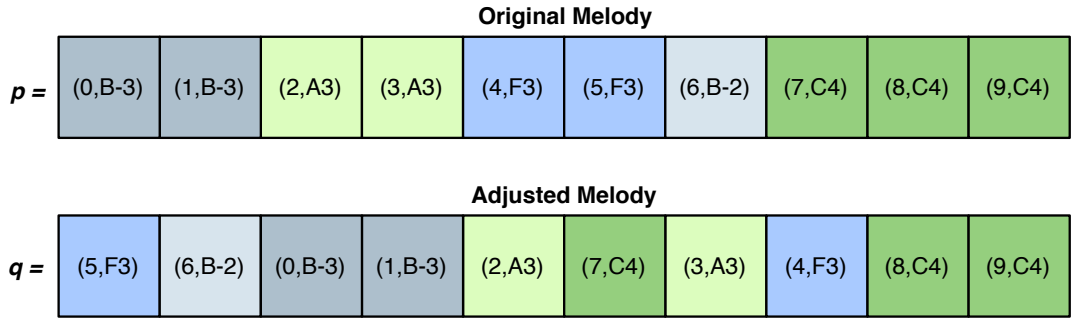

Figure 3: Euclidean Distance

Based on the individual in Figure 3, the maximum number of allowed adjustments are 3 since there are 10 notes, the fitness score is 24, and the pairs used to calculate the Euclidean distance can be seen in Table 2.

Table 2: Euclidean Distance Coordinates

(0,5)	(1,6)
(2,0)	(3,1)
(4,2)	(5,7)
(6,3)	(7,4)
(8,8)	(9,9)

User burnout is inevitable, but the onset of it is directly influenced by the quality of the UX, i.e. how engaging the process is. Typically, an IGA requires the user to examine each individual and assign it a score, i.e. the user listens to the melody and then provides a score representing how the melody was as a whole. An important question this approach raises is what happens if the user really liked parts of the melody, but as a whole didn't enjoy it? Assuming that the user provided a low fitness score the good parts will end up getting lost. This is an issue because it decreases the chance that future individuals will see the traits the user liked. A remedy for this is to modify the type of feedback that is supplied. However, before that can be done, the common approaches need to be analyzed.

Suppose a population contains 5 individuals and runs for 5 generations. That means the user has to listen and evaluate 25 individuals. As a comparison, it is

not uncommon for a GA using an automatic fitness function to contain hundreds of individuals and run for thousands of generations. Typically, the user assigns a fitness score through either a textbox, a slider, or pressing a good or bad button [3]. The last approach requires the least amount of physical work from the user, but results in slower rates of evolution because the fitness score is binary. The user either liked it, or didn't like it.

Instead of using binary evaluation to assign an entire individual a score, suppose the user could assign scores to sub-segments of the individual, e.g. the user may have liked the first x notes and the last y notes, but hated everything in-between. In addition to introducing a variety of possible fitness scores, i.e. scores are no longer just 0 or 100, the IGA is also receiving meaningful feedback; the user is basically able to say what segments were liked and disliked. This concept was introduced by Biles for GenJam [3] and allows the user to listen to and see the melody evolve in real-time. Doing this requires the population to be modeled differently, though. Instead of a population consisting of n individuals, GenJam's populations consist of a single, continuous individual.

As demonstrated by Biles in GenJam, there is a strong connection between the UX and the type of feedback supplied [3]. However, instead of having the user listen and say "yes, I like this", "no, I don't like this", I believe that the user should be able to demonstrate how the melody should be structured. This lets the user simultaneously evaluate and mutate the individual, allowing the user to become a temporary composer. The user listens to the melody and makes adjustments to the note order. Once the user submits the melody, the Euclidean distance between the original and adjusted melody is taken. So, if the user liked the melody as is, then the fitness score becomes 0 since no changes were made. This type of feedback also enhances the rate of evolution because the adjustments, if any, overwrite the original melody. If the adjusted individual ends up getting picked during the selection process, then the child it creates will contain properties the user likes, which wouldn't happen if the adjustments didn't overwrite the original.

Ideally, the IGA will produce a melody that requires the user to make no changes. However, suppose that the melody is horrible sounding, but the user is able to

salvage it by making adjustments, and once the adjustments are made the user actually really likes the way the newly adjusted melody sounds. In this instance, the fitness score would be relatively high (based on the number of adjustments made), and thus this individual would have an extremely small chance of being picked during tournament selection. To account for this, the user is able to override the fitness score and reduce it in order to increase the probability that it gets selected. The number of times the user can do this per generation is limited and based on the number of individuals in the current generation.

In addition to the auditory component, the fitness function also contains a visual component. Each note in the melody is a different color. Notes with the same pitch, but different octaves and accidentals are a different shade of the base color, e.g. if an individual contains [...,C, C#4, C8,...] and C is green, then C#4 might be light green and C8 might be dark green. This is done for both aesthetics and learning. It allows the user to pick up on visual patterns that may occur across individuals. An additional feature is that whenever the user makes an adjustment, the fitness score is immediately updated and displayed in the stats widget. This is so the user can see the impact the adjustment has on the score.

The key components this fitness function improves on are the type of feedback and the UX. Allowing the user to make adjustments provides more meaningful feedback because the fitness score is not calculated by how much the user liked the individual, but by how the user would have structured the melody given the same notes. Allowing the user to act as a composer builds on the overall experience because the user has a sense of ownership and can see how the adjustments impact future generations. Additionally, being able to experiment with music provides a fun factor. The fitness function is essentially providing the user with a template or a suggested melody, and allows users, even ones with no musical knowledge, to experiment and learn about music in a very straightforward, non-overwhelming way. An example of the fitness function can be seen in Figure 4.

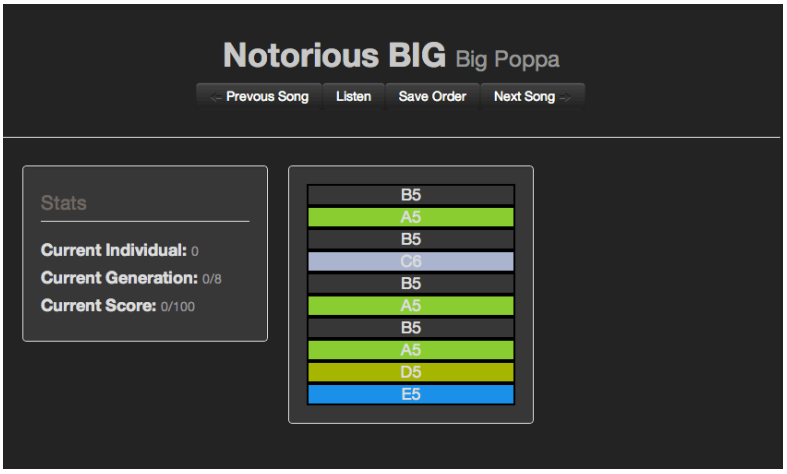

Figure 4: Fitness UI

7.3.5 Fitness Workflow

The user listens to a melody and either likes it as is and moves on to the next one, or doesn't like it. In the former, the melody is given a fitness score of 0, meaning that it was perfect; the user really liked it. In the latter, the user can make a change and preview it. The user can take as much time as needed to make adjustments to the melody; the only constraint is the number of allowed adjustments. The user can listen to the melody at any time by clicking the Play button allowing the user to hear how the adjustments sound. The user is also able to revert back to the original, pre-adjusted, melody if the adjustments aren't liked.

7.3.6 Genotype

Typically, genotypes are encoded in either string or binary format because it's easy to store, process, and is fast. However, this project encodes an indivudal's genotype as a python dictionary making it easy to send as JSON and store in redis hash sets.

The dictionary contains a set of keys representing an individual's metadata. Because this project focuses on generating melodies, many musical features can be ignored since a melody is a combination of notes, rests, and a duration. A note is a 1:1 mapping to a frequency and is made up of three components: a pitch, an accidental, and an octave. The pitch is the degree of the tone, i.e. how high or low

the frequency is, and is represented by letters A-G. An accidental is optional and is denoted by either # or ♭. It raises or lowers the key signature of a pitch. An octave is represented by an integer $0-9$ and is the "interval between one musical pitch and another with half or double its frequency" [15]. One modification has been made to the representation of a melody. Instead of a melody containing a list of notes, the melody contains a list of lists, where each sub-list can either be a single note, or a chord, e.g.
$\{ score : 0, id : 1, generation : 1, notes : [[A, B, C], [B], [D, D, D]]\}$

A note is formed once the components are combined, which can then be converted to its respective frequency, and listened to. "Any note is an integer of half-steps away from middle A (A4) where the distance is denoted by n. If the note is above A4, then n is positive; if it is below A4, then n is negative. For example, one can find the frequency of C5, the first C above A4. There are 3 half-steps between A4 and C5 (A4 $\rightarrow$ A#4 $\rightarrow$ B4 $\rightarrow$ C5), and the note is above A4, so n = +3" [14]. As a result, the frequency of C5 is roughly 523.2Hz when calculated using the equation found in (2).

$$f = 2^{n/12} \cdot 440Hz \tag{2}$$

7.3.7 Genetic Mapping

Genetic mapping transforms the genotype into a phenotype, which means taking a series of notes and turning them into a melody that the user can listen to. In this instance, the genotype is a list of dictionaries and the phenotype is a melody.

Mapping the genotype to the phenotype is done by first, searching through the list of dictionaries. For each dictionary, the notes key is extracted and for each note in the list of notes, the pitch frequency is generated and stored in a tuple.

This process is performed on the client's machine using JavaScript. The IGA sends the browser the genotype encoded as a JSON object containing a list of

dictionaries. Each note is displayed in its own div element on a HTML page where the div ID is set to the note's value, e.g. div id="C5". Whenever the user makes an adjustment and wants to hear it, the JavaScript reads the div elements in order, converts each note to its frequency, and outputs the audio in the browser using the web-audio API.

7.3.8 Interactive Genetic Algorithm Operators

IGA operators are done server-side using Python. This includes selection, crossover, and mutation.

The IGA uses tournament selection to decide which individuals will be selected for crossover by randomly selecting k, the tournament size, individuals from the current population. The individual with the best (lowest) fitness score is picked and placed in the crossover pool. There are n rounds in a tournament and the process allows for an already selected individual to be picked again, i.e. independent events. An example can be seen in Figure 5.

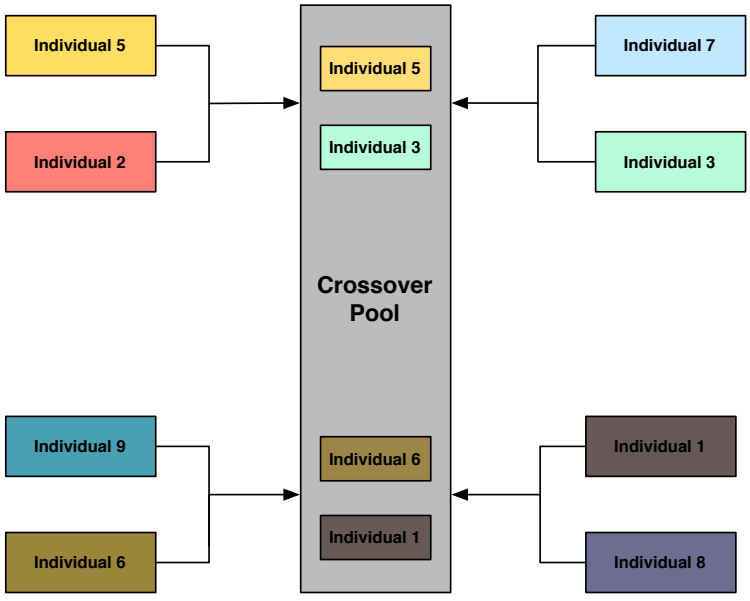

Figure 5: Selection: k=2, n=4

It is possible to modify the probability of weaker individuals (higher fitness scores)

being picked by adjusting k. Each increment of k increases the probability of selecting only the best individuals from the population, while decrementing k increases the probability of selecting weaker individuals from the population, i.e. as k grows or shrinks so does the probability of selecting certain types of individuals. In this research, some good individuals may exist in the population with a weak fitness score, thus k will be set to 2 resulting in binary tournament selection during each round, n. The value of n determines the size of the next generation's population. In order for the population size to remain consistent from generation to generation, $n = \frac{1}{2} * p$ where p is the population size. In order to account for the other half of the population, each iteration of crossover yields 2 children. The general algorithm can be seen in Algorithm 1.

$counter = 0$;
while $counter < N$ **do**
    Randomly select k individuals from the current generation;
    Select the individual with the lower fitness score and add it to the crossover pool;
end

Algorithm 1: Selection Algorithm

Once the selection process has been completed, the IGA undergoes crossover and creates the next generation of individuals. The IGA uses single point crossover yielding two new children per iteration. First, two parents are randomly selected from the crossover pool. A random split point is generated to determine what traits from which parent the child will be made up of. An example can be seen in Figure 6.

Figure 6: Crossover: *splitpoint*=2

There is a 50% chance that there will be an odd number of parents in the crossover pool. If n is an odd number then one parent will have to crossover twice resulting in some commonalities among 4 individuals of the next generation. However, since the crossover point is randomly selected this will not be much of an issue. For this reason, it is important to keep track of which individuals crossover with each other to ensure that there are no duplicates, otherwise there is a chance that the same two individuals might crossover with each other twice. To account for this, parents are stored in a tuple with the lower parent id always being at index 0 in the tuple. Each pair is then appended to a set to ensure that there are no duplicates.

After crossover is complete and the next generation has been created, it is necessary to mutate random individuals in order to diversify the population. The mutation rate, m, is a decimal between $[0, 1]$ that is decided upon during the IGA's initialization. For each individual in the population, a random decimal n between $[0, 1]$ is also generated. If $n > m$ then the current individual undergoes mutation. A random mutation point, rmp between $[0, individualSize]$, and a random range,

rr between $[rmp, individualSize]$ are generated to determine which notes will be mutated. A new Markov chain is then created using the same influencer, and a subset of consecutive notes in the Markov chain are used to replace the notes in individual's mutation range. The general algorithm can be seen in Algorithm 2.

$mutationRate = .3$;
for *each individual* **do**
    rnd = randomDecimal(0, 1);
    if $rnd > mutationRate$ **then**
        $start, stop$ = randomSubset(individual);
        $newGenome$ = markovChain($start - stop$);
        $individual[start : stop] = newGenome$;
    end
end

Algorithm 2: Mutation Algorithm

7.3.9 Interactive Genetic Algorithm Workflow

The IGA workflow is extremely straightforward. The user is presented with a settings webpage, which is used to fill out the initialization parameters. These include: population size, number of traits, number of generations, mutation rate, Markov chain size, Markov chain node history, and the influencer. These parameters are processed and a request is sent to the music server. The music server then calls the GA with the users preferences. The GA generates the initial population and caches it in redis. Next, the user is presented with a web-page for the fitness function where the user evaluates each individual, one by one. After the user is satisfied with the individual, the fitness score is calculated by taking the Euclidean distance between the pre-adjusted melody and the post-adjusted melody. The fitness score and the adjusted melody are sent to the IGA with an AJAX request. Once an individual has been evaluated and sent back to the server, the GA needs to know how to proceed. Essentially, the GA needs to check if the termination requirements have been met. First, the GA checks to see if there are more individuals in the current generation that need to be evaluated. If there are, then the GA returns to the fitness function. If there aren't any more individuals, then the GA checks to see if there are any generations remaining. If there aren't, then the GA terminates, otherwise the GA moves onto selection, crossover, mutation, and

then back to the fitness function where this whole process is repeated. The general workflow can be seen in Figure 7.

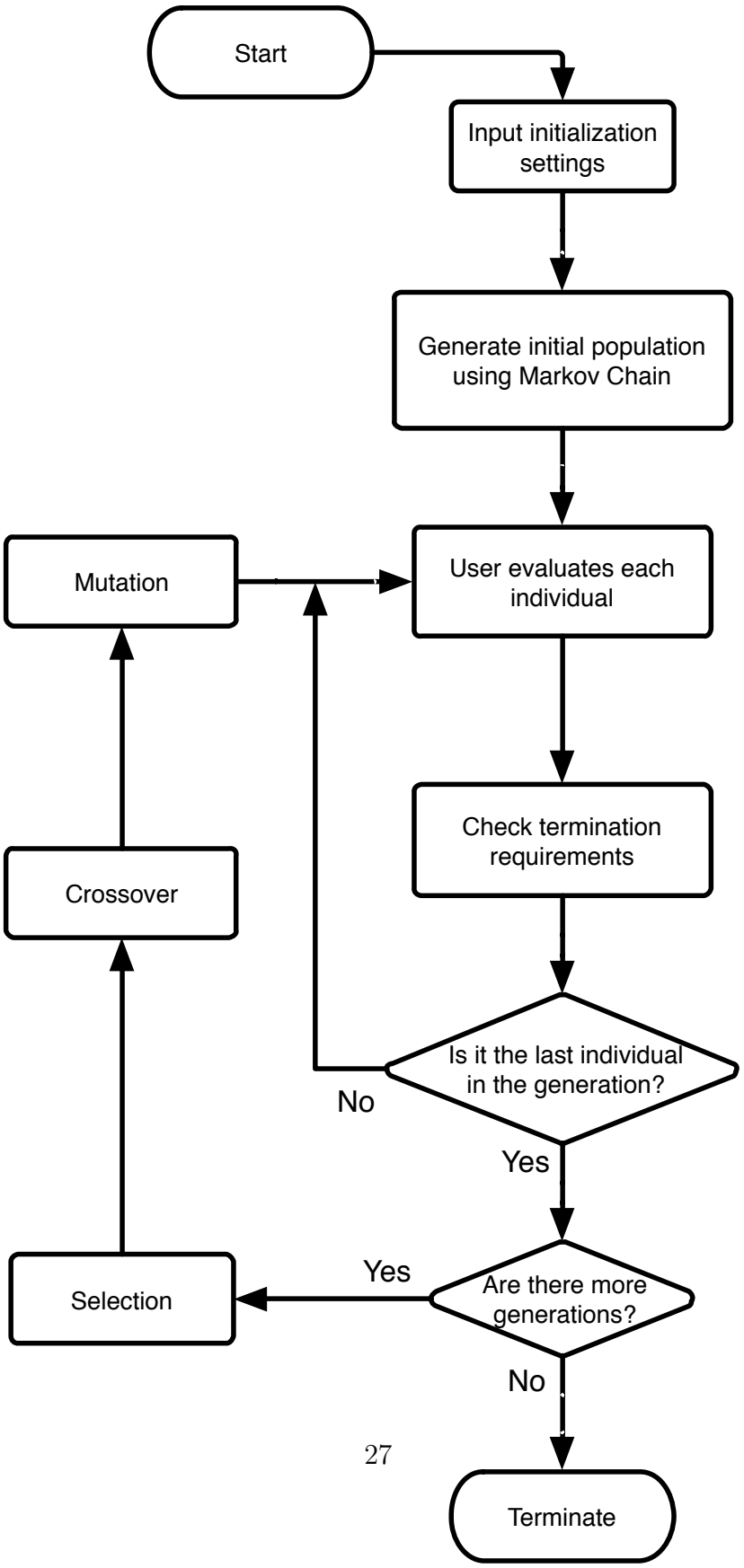

Figure 7: General Workflow

8 Results

8.1 Overview

Tests were designed around determining which combination of the initial population and fitness function yielded better individuals in the least amount of time. Each GA was initialized with the same settings, which can be seen in Table 3.

Table 3: Initialization Settings

Influencer	Antonio Vivaldi - Winter: Allegro (Non Molto)
Markov Chain Size	2500
Markov Chain Node History	3
Population Size	5
Number of Notes	10
Number of Generations	5
Mutation Rate	.3

8.2 Test 1: Random Population and User Scoring

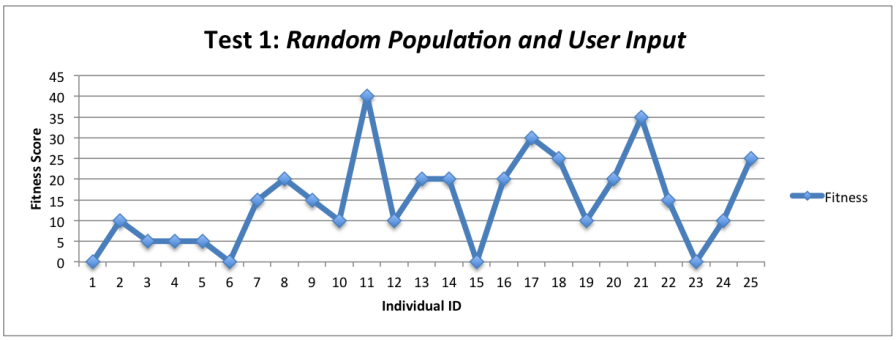

Figure 8: Test 1: Random Population and User Scoring

Table 4: Fitness Score Stats for Test 1

Mean	14.6
Median	15
Best	4

This combination generated terrible sounding individuals. The minimal number of generations in conjunction with an unintelligent seed function produced results comparable with someone hitting random keys on a piano. This was to be expected, though. The leading issue with this combination was the number of generations. 5 generations did not provide enough time for all of the poor individuals typically found in the beginning to die off. Additionally, this combination lacks chromosome repair so there was no way for the user to influence an individual. As a result, the growth of individuals was extremely slow, unstructured, and often regressed from generation to generation. User fatigue was minimal, although users reported boredom due to lack of interesting melodies and repetitive nature of the fitness function. User never found a "favorite" melody and considered them to all be of poor quality.

8.3 Test 2: Random Population and Trait Re-ordering with Euclidean Distance

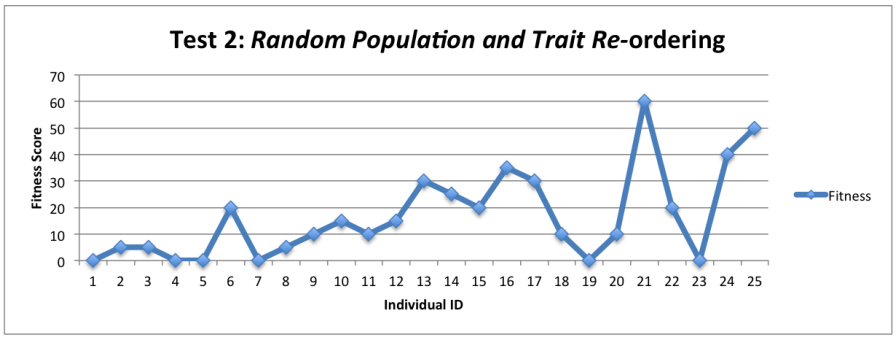

Figure 9: Test 2: Random Population and Trait Re-ordering with Euclidean Distance

Table 5: Fitness Score Stats for Test 2

Mean	16.6
Median	10
Best	60

This combination generated extremely poor individuals in the first generation due to using a random population, however this was quickly remedied in generation

2 due to chromosome repair, i.e. the user's ability to re-organize each individual. While the user made the maximum number of re-orderings allowed per individual, the user was still able to make part of the individual have a funky/fun sound. This can be attributed to the fact that the random population provided a large spectrum of possible notes per individual allowing the user to create interesting patterns. As a result, the growth of individuals started off slowly, but quickly sped up and produced much more interesting melodies in later generations due to the user's help. Additionally, the user was less bored because the user felt more involved in the process. The user reported that boredom and fatigue began to set in during generations 4 and 5 because it began to get very tedious and repetitive. It also took a while for the user to find a favorite melody.

8.4 Test 3: Markov Chain and User Scoring

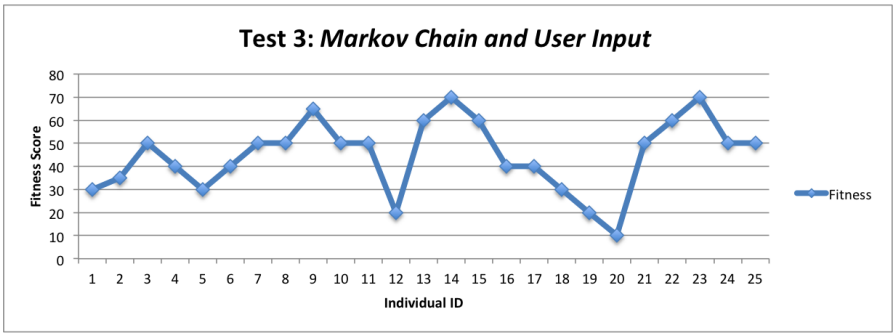

Figure 10: Test 3: Markov Chain and User Scoring

Table 6: Fitness Score Stats for Test 3

Mean	44.8
Median	50
Best	70

This combination provided decent results in earlier generations since the initial population was intelligently generated based off of a song the user already liked. In terms of individual growth from generation to generation, this test was comparable with test 4. The quality of each individual in generation 1 was much better

than in tests 1 and 2. Additionally, the fitness score of the best individual grew significantly from generation to generation. The issue with this test was the lack of chromosome repair resulting in some pretty poor patterns. User fatigue was minimal like in test 1, and it took longer for the fatigue and boredom to set in since the melodies produced were entertaining.

8.5 Test 4: Markov Chain and Trait Re-ordering with Euclidean Distance

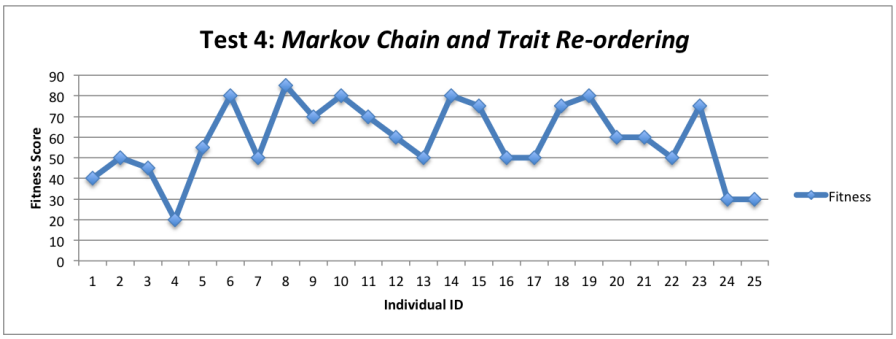

Figure 11: Test 4: Markov Chain and Trait Re-ordering with Euclidean Distance

Table 7: Fitness Score Stats for Test 4

Mean	58.8
Median	60
Best	85

This combination generated the best sounding individuals as well as reduced fatigue and boredom when compared to test 3. This can be attributed to the fitness function. It appears that a more involved fitness function reduces the onset of boredom and fatigue since the user feels more involved in the process. Additionally, most of the poor individuals were seeded out after the first generation.

8.6 Analysis of Tests

Tests 1 and 3 took less time to complete compared to tests 2 and 4. This was because it was much easier for the user to just enter a number in and click next opposed to re-arranging the traits. In contrast, the user had a much more enjoyable experience in tests 2 and 4 opposed to tests 1 and 3. The user felt involved in the process, and even compared it to solving a puzzle. As a result, the fatigue level was minimal for the first 3 generations in tests 2 and 4. The last 2 generations in tests 2 and 4 were much more tiring and difficult for the user to put in as much as effort as was done in the first 3 generations, which explains the slight dip in fitness scores. However, the user enjoyed test 4 more than test 3, and stated that it took longer for boredom and fatigue to set in since the evaluation process was much more enjoyable. The user did not enjoy tests 1 and 3 mainly because the evaluation process was boring.

9 Conclusion

This research has provided a lot of insight into what needs to be done to generate melodies using an IGA. Specifically, it showed how reducing the size of the search space and then intelligently drawing from it influences the quality of individuals found in the initial population. Additionally, the results provided insight into how a more detailed fitness function can help increase the quality and rate of growth of individuals from generation to generation. The goals of this project were met, however, there is still much work that can be done to further reduce user fatigue levels while also increasing the quality of individuals.

References

[1] Ivo Adan. 3 markov chains and markov processes, 2008.

[2] Ryan Becker. Genetic music. Master's thesis, Rochester Institute of Technology, 2005.

[3] John A. Biles. Genjam: A genetic algorithm for generating jazz solos. In *International Computer Music Conference*, 1994.

[4] John A. Biles. Autonomous genjam: Eliminating the fitness bottleneck by eliminating fitness, 2001.

[5] John A. Biles, Peter G. Anderson, and Laura W. Loggi. Neural network fitness functions for a musical iga. In *Soft Computing Conference*, 1996.

[6] Curt Clifton. Markov chain algorithm, November 2011.

[7] Michael Scott Cuthbert and Ben Houge. Music21 documentation, November 2012.

[8] Johannes Hoydahl Jensen. Evolutionary music composition: A quantitative approach. Master's thesis, Norwegian University of Science and Technology, 2011.

[9] Eric Krevice Prebys. The genetic algorithm in computer science. *MIT Undergraduate Journal of Mathematics*, 2007.

[10] Shabda. Generating pseudo random text with markov chains using python, 2009.

[11] Peter M. Todd and Gregory M. Werner. Frankensteinian methods for evolutionary music composition, 2001.

[12] Nao Tokui and Hitoshi Iba. Music composition with interactive evolutionary computation, 2000.

[13] Wikipedia. Markov chain, October 2012.

[14] Wikipedia. Note, November 2012.

[15] Wikipedia. Octave, November 2012.