Evolution of Music

FINAL YEAR PROJECT REPORT

SHANNON MORAN
11394476
4BCT

Project Supervisor – Colm O'Riordan

Table of Contents

Chapter 1. INTRODUCTION

This project, entitled Evolution of Music, will involve the evolution of music though the use of evolutionary algorithms. The evolution itself will be guided by human feedback and constrained by a number of heuristic based aesthetic measures. The motivation for pursuing this project comes from the inspiring work that has been achieved through evolutionary computation to-date. This work has been carried out in various domains, which only leads one to wonder about the undiscovered possibilities that evolutionary computation possesses in other domains. Due to having a very deep passion for music, evolutionary computational methods will be applied to this domain to see what can be uncovered.

Project Area

The area of evolutionary computation is considered to be a branch of artificial intelligence and has been around for a lot longer than one would expect. The uses that have been found for evolutionary computation are often somewhat surprising, with it being applied to anything from optimising telecommunication network routing to designing highly specialised antennas for NASA. The area has huge potential for further development as it can be applied to almost anything to help find solutions to problems that may not be obvious to humans. Often human logic and restricted thought processes can put the solution to a problem somewhat out of sight. As evolutionary computation is fearless in its pursuit of a solution to the problem at hand, this can result, and has, time and time again, in extraordinarily surprising, effective solutions.

Goals, Aims & Opportunities

The goal of this project is to create a system that evolves music based on a well-defined and well-designed fitness function, which is based upon the theory of music aesthetics and what makes music appealing to listeners. The resulting evolved music should be appealing based upon the definition of the fitness function used and the subjective input of the users. By allowing the system to take into account users' subjective ideas of what makes music appealing to them, the aim is to arrive at a solution, i.e. a piece of music that is appealing to them, sooner than if no user interaction was allowed. To achieve this, the fitness function will need to be able to be tailored to define music as being 'fit' based on each user's subjective opinions of music. The hope is that the system will be an enjoyable experience for the user and that it will be intriguing to witness the system evolve music to match the user's taste based upon minimal input.

Another major goal of this project is to find out some interesting information from the analysis of the system's performance under different conditions and with different users. This will be carried out through the analysis of the system's performance through a number of experiments and through the comparison of results between these experiments. The aim is to arrive at some interesting hypotheses regarding the aspects of what it is that makes music appeal to its listeners. It is hoped that the analysis of the evidence gathered through carrying out these experiments will result in the formulation of a hypothesis regarding the best method of evolving music. The system will cater for these experiments by allowing the user to tweak the genetic algorithm used, such as the number of evolutions of the system, population size and other configuration settings. The system will also allow the user to decide upon the type of fitness function that will be used in the evolution. By allowing these configuration changes the aim is to have a wider scope to base the testing on, which will result in larger amounts of data and thus lead to a more in-depth analysis of the project.

There are possible opportunities for the work carried out on this project to be used in other areas of evolutionary computation, specifically in the area of music evolution. If the analysis of the data collected from experiments results in some interesting hypotheses, these may be of interest to others in the field of music evolution, in particular the area of automated music evolution.

Approach

This project with approached with an open mind, hoping to discover about the power of evolutionary computation as well as the possibilities that lie in the somewhat niche area of music evolution. The project initially began by defining the project direction. It was decided to take the more research based approach, as opposed to the approach of creating a finished, distributable application. This approach was decided on due to being intrigued to find out more about the possibilities that lie in the area of musical evolution – an area previously unknown before being made aware of it. By defining the project direction at the beginning this allowed the project to be approached with a definitive end goal in mind, rather than continuing with a degree of uncertainty as to what the project will entail. Specifically, it allowed the focus to be put on implementing the genetic algorithm to evolve music while gathering data to be used for analysis, rather than putting time and effort into creating a front end GUI that would be of no benefit to the research.

After finalising the decision on the project direction, the designing and planning the project began. This involved splitting the project up into tasks. The order in which these tasks had to be carried out and the links that lie between them then had to be decided. An agile approach of iteratively developing and testing the system was taken to allow for possible changes to the project along the way, such as if new ideas were thought of or any other beneficial additions to the original project specification.

The experimentation and analysis section of the project was approached by first ensuring that the development and testing of the genetic algorithm was fully completed. This ensured that the gathering of data for analysis was not impacted by any ongoing development of the genetic algorithm. After collecting sufficient data from the experimentation stage, the results could then be analysed and concluded based on the findings of the research.

Layout

This report will go through all aspects of the project in detail. First, it will be ensured that sufficient detail is given regarding the area of evolutionary computation and all other associated topics involved in this project. Thus, a full understanding can be had of all subject matter covered throughout the report so that the work can be perfectly communicated. The report will also ensure that the thought process behind of all of the design decisions is clear for the implementation of the genetic algorithm and testing aspects. The results section will require the accumulation all of this knowledge learnt prior to this for the presentation of the results and notable findings of the project. The report will finish with a summary and conclusion of the project as a whole, with some possible work items noted that deserve future work.

Chapter 2. RELATED WORK

Biological Evolution

The motivation behind evolutionary computation itself stems from the theory of the biological evolution of organisms, as proposed by Charles Darwin and others. This is where the DNA of organisms evolves through reproduction in an effort to allow the organism to adapt to their surroundings and changing circumstances with the aim that the organism will be able to survive and reproduce.

The DNA of an organism evolves through the amalgamation of the genes of the parent organisms. At the chromosome level, this happens through recombination, also known as crossover, and random mutations. Crossovers occur when two parent chromosomes exchange sub-segments of themselves to form an offspring chromosome that is made up of genes from both parent chromosomes. Random mutations can then occur in offspring chromosomes, where genes are copied from parent to offspring, which in nature can result in 'copying errors'.

Genetic Algorithms

One area of evolutionary computation that is particularly of interest in this project is the area of genetic algorithms. Genetic algorithms were invented by John Holland in the 1960s; unusually, they were not designed to solve specific problems. The original basis for genetic algorithms was to develop a system with the ability to computationally evolve in a way that is similar to that seen in biological evolution, as outlined above. In other words, it aimed to mechanise and formalise the biological phenomenon of natural selection in computing. [1] [2]

Genetic algorithms are based upon the evolution of “chromosomes”, which are sequences of bits, called “genes”, with a value, or “allele”, usually either 0 or 1. The initial chromosomes form a “population” of size n, all of the same length, which are often initially randomly generated. At each iteration of evolution, the chromosomes in the population may be subjected to either, or all, of the following operators which are used to mimic natural selection: crossover, mutation and inversion. Crossover happens between two chromosomes and results in randomly choosing a locus (position along a chromosome) and exchanging the sequences of bits before and after that locus. This produces two offspring chromosomes. Mutation changes the value of a bit, or allele, in a chromosome at a locus. This operator uses a given probability for a mutation to occur at each locus along the chromosome. Lastly, the inversion operator works by reversing an adjoining string of bits in a chromosome. Inversion is another form of mutation. It is worth noting that although the typical allele value is either 0 or 1, this can differ depending on the design decisions taken when implementing a genetic algorithm, with allele values being chosen depending on its particular use. For example, each allele could instead be any integer value.

At each iteration of evolution, each chromosome is tested against a ‘fitness function’ to determine which chromosome of the population is the best or ‘fittest’, in a process called ‘selection’. A fitness function needs to be well defined and may differ depending on the application of the genetic algorithm. This process is repeated until a specified fitness is reached by a chromosome or if the specified number of evolution iterations has been reached.

There are some limitations to genetic algorithms, however, and they are often used to help decide if a genetic algorithm is the right type of optimisation algorithm for the task at hand. One of the main limitations is the fitness function. The fitness function needs to be able to determine the fitness of the chromosomes based upon some set of specified rules as defined by the genetic algorithm creator. This can be a huge challenge in certain domains and is very hard to get right, with trial and error often being one way of creating a reasonable fitness function. With a poor fitness function comes a poor genetic algorithm, so this is why such care has to be taken to design an effective fitness calculation method. In complex genetic algorithms, the fitness function can involve very computationally expensive simulations that can take a long time to complete. Instead an approximation is often taken to help reduce the computation and time expenses, thus adding another degree of estimation to the algorithm.[3]

Due to the nature of the task of finding a solution to a problem, the current best solution is only the best in comparison to previously found solutions. The same goes for genetic algorithms. Knowing when to stop searching for a ‘better’ solution is not always clear, so creating stop criteria can not only be a major challenge, but it can be a huge limitation, due to not knowing if a ‘better’ solution exists, thus possibly limiting the potential of the genetic algorithm.

Genetic algorithms have a tendency to converge to local optima rather than the global optima when searching for the solution to a given problem. The reason for this is that genetic algorithms cannot sacrifice short-term fitness to gain longer-term fitness. The aim of the algorithm is to find the fittest solution to the problem, so the notion of bringing chromosomes of poorer fitness onto the next stage of evolution goes against its only principle of evolution. However, by only carrying the fittest chromosomes onto the next stage of evolution, the algorithm is limiting its search space for solutions to the problem. Thus, mutation is often used as a potential solution to this problem. By mutating a chromosome it is hoped that if it is stuck at a local optima, the mutation could help move it on nearer to reaching the desired global optima by once again expanding its search space.

Genetic Programming

There has been extensive research into the area of using genetic algorithms to write computer programs, known as automatic or genetic programming, where a computer program is used to write another computer program. The first successfully applied and widely recognised genetic programming system was developed by Koza in 1992, which formed the basis of conventional genetic programming systems. Koza’s approach represents programs by a tree structure that captures the ordering of the components within a program, allowing the trees to be easily evaluated in a recursive manner. The tree is created such that the output appears at the root node, functions are internal nodes with their arguments given by its child nodes, with leaf nodes representing terminal arguments. The evolutionary processes of crossover and mutation are carried out by crossing over subtrees (see figure 1.2) and mutating nodes. Thus, genetic programming creates new programs from existing programs. Although research is still ongoing, this once again shows the potential power that the field of evolutionary computation possess as this could be applied to a vast amount of problem areas. [5] [6]

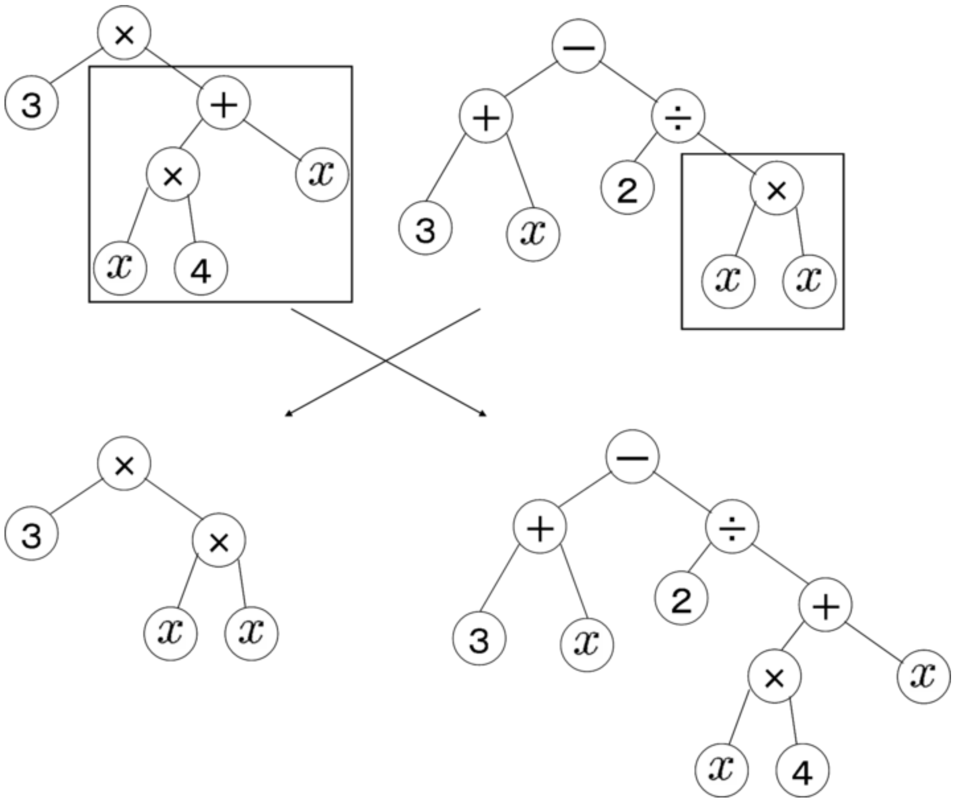


Figure 1.2
(by U-ichi, CC SA 1.0)

Real World Examples

A famous example of the powerfulness of evolutionary computation is how, in 2006, NASA discovered the design for a highly complicated spacecraft antenna (see figure 1.1). The ultimate design goal for the genetic algorithm was to create an antenna with the best radiation pattern, which was to be used in communications between ground stations and satellites in space. Through the use of genetic algorithms, the following antenna was evolved as a solution and became the world's first artificially evolved object to fly in space. [4]

Figure 1.1

Genetic algorithms have been used in the field of security, in both data encryption and code deciphering. When applied to code breaking, the task is to search the large search space of ciphers for the correct decryption cypher.

Music Evolution

Music evolution can be simply described as the creation of music using evolutionary algorithms, most commonly using either genetic algorithms or genetic programming. Music evolution begins with a population of individuals that each have some musical or audio representation. In the case of genetic algorithms, a chromosome would have a representation that maps to some form of audio, such as a chromosome that maps to a number of beats and their individual timings, for example. The initial population may be random or seeded with existing music – human generated or otherwise. Music is evolved through the application of biologically influenced evolutionary processes on the population, such as mutation and crossover, with the aim of evolving the population to become more musical. [7]

As music is very subjective, the evolution of music is often influenced by some user interaction with the evolution. The idea is that by allowing the user to have an input into the evolution of music, the resulting evolved music should be tailored to their own specific subjective opinions of what it is that makes music appealing to them. This is usually implemented in the fitness function of the evolutionary algorithm, where the user can determine the fitness of the individuals in the population. As it is very difficult to compute the aesthetic qualities of music, this type of user interaction helps the evolutionary algorithm to determine these qualities based on the provided user fitness values.

Although computationally defining the aesthetic qualities in music is very difficult, there is ongoing research into the automation of music evolution, i.e. evolving music without human involvement or input. This type of evolution is based upon the principles of musical aesthetics, which are a theoretical set of traits that tend to define music that appeals to most humans. Due to the subjectivity of music, this is not an exact science, but instead a theoretical basis to which researchers hope to build upon with computational measures.

There has seemingly been very little commercial work done in the area of music evolution, surprisingly. Most of the commercial work in the area seems to be in the creation of friendly user applications to create simple melodies and rhythms. However, there seems to be steady research being carried out in the area. *Melomics* is an artificial intelligence group that uses evolutionary algorithms to compose music of all types. They have the largest collection of computer-composed music in the world, available in audio formats, music scores for performers and in XML files for personalisation. [8] In 2012, they created the first album that was composed by a computer and performed by human musicians. [9] *DarwinTunes* is another research experiment that has been running since 2009. Their aim is to try to understand the underlying mechanisms of Darwinian evolution by natural selection through the evolution of music – "the oldest and most widespread form of culture". [10]

As a huge music lover it was surprising to see the lack of publicised exploration being carried out in this area of huge potential, even though the two research projects mentioned had impressed. Although it is easy to understand the difficulties of trying to create a mould by which appealing music can be created from, it is believed that this area deserves more commercial investment. As the music business is such a thriving area, it was surprising that there hasn't been more interest in the area of music evolution. With this project the hope is to make some interesting findings in the analysis as regards the evolution of music, in both assisted and automated evolution, which may be of benefit to those working in the area.

Technology

Finding the most suitable technologies to use for this project was an important task that took some time to decide upon. Due to the project's aim of evolving music, a music technology was needed that was flexible and configurable, that was fast at producing music and that was lightweight. Due to having to work with genetic algorithms, a way of implementing a genetic algorithm that would be not limited in its evolution possibilities was needed. By ensuring the technologies used had these desired properties, it was believed that it would give the freedom to implement the project without restraint, with the ultimate aim of evolving an appealing piece of music according to the notion of musical fitness.

JGAP

Java Genetic Algorithms Package (JGAP) is a Java framework that provides an API for working with genetic algorithms and genetic programming. It abstracts the basic setup and configuration of a genetic algorithm through its API. JGAP only requires the user to define the fitness function to be used in the evolutionary process to determine the fitness of solutions relative to other solutions. Although JGAP is designed to be very easy to use 'out of the box', it caters for more advanced and adventurous users that wish to add extra functionality or customisation to their genetic algorithm. It allows for this due to its highly modular design, thus allowing users to mix and match components as they desire. JGAP also allows for the creation of custom genetic operators and custom genes, as it provides a number of interfaces that can be implemented. It also gives users the freedom to add any extra functionality to existing classes by allowing the user to extend classes and possibly override methods if necessary. [15]

For this project, JGAP was chosen as the framework of choice for the implementation of the genetic algorithm. It will allow for getting up and running quickly with a basic genetic algorithm to begin with, before continuing on to customise the genetic algorithm as desired after becoming accustomed to working with the JGAP framework. The most comforting benefit is that JGAP will assure the freedom to be able to implement all desired functionality in the genetic algorithm due to the modularity and configurability of this framework.

MIDI

Musical Instrument Digital Interface, or MIDI, is a protocol that was developed to allow electronic musical instruments, computers and other digital music tools to communicate with each other. MIDI itself does not make the sound - the MIDI protocol is actually a set of instructions. MIDI data is sent between communicating devices, which simply contains a list of events or messages that tell the electronic device how to generate a certain sound. MIDI has the file extension .MID and these files are saved as a list of messages and instructions. When an electronic device plays MIDI files they use their internal synthesiser software to follow the instructions to play the piece of music according to the messages and instructions contained in the MIDI file. Any electronic device with synthesiser software will be able to play MIDI files, including computers and smartphones that can use their sound cards to produce the sounds as specified in the MIDI file instructions. It is worth noting that with MIDI, the sound will vary slightly depending on the device it is being played on due to the different audio sources. This point highlights another big advantage of using MIDI, which is that seen as the .MID files contain no actual music or sound, this means that the playback devices can modify the sound to their needs without having to re-record any audio. For example, this property is taken advantage of in some karaoke machines where the machine can easily change the pitch of the song being played to suit different vocal ranges of the singers. Tempo, duration and other parameters can be also be changed just as easily. Lastly, another fundamental property of the MIDI protocol is that the files it produces are very compact. As .MID files do not contain any sampled audio like the common audio formats MP3 or WAV, they are much smaller due to containing just simple instructions. Due to .MID files being so small, this has increased the appeal of using the protocol on devices with limited memory, such as smartphones and tablets, and in video games. On the other hand, seen as MIDI only contains instructions on how to play audio, it is incapable of containing speech or sound effects, the way WAV or MP3 can. [11] [12] [13]

For this project MIDI will be used as the audio technology of choice. The reason for choosing MIDI is mainly due to its flexibility, as outlined above, but also due to its simplicity as a music protocol. MIDI will give the freedom to generate a chromosome representation that will map the chromosome values to simple MIDI messages and events to easily create pieces of music.

Chapter 3. DESIGN

Genetic Algorithm

The design process of this project was critical to its success. The designing of the different aspects of this project took a lot of time and thought to ensure that the correct approach was taken and that the resulting application was of a high standard. As with any work with genetic algorithms, a lot of time goes into their design as this is fundamental to the quality of the result of the evolution process. The genetic algorithm at the heart of this project needed to be able to create an initial sample population of chromosomes that would evolve based upon a fitness function. This fitness function acts as a measure of how 'good' each chromosome is, and hence how 'good' each piece of music is. This genetic algorithm needed to result in the evolution of a very fit chromosome that represents a very 'good' piece of music based upon the fitness function used during the evolution process.

Representation

Another critical aspect of the design process was the designing of the chromosome representation. It was decided that each chromosome would represent an individual piece of music. This meant that through the evolutionary processes of mutation and crossover that 'fit' parts of different chromosomes could be combined with the aim of resulting in a fitter chromosome. By allowing mutation, this ensures that the genetic algorithm does not get stuck at local maximums, rather it allows for the genetic algorithm to aim for reaching global maximums.

Genotype to Phenotype

The chromosome is created initially based upon a template chromosome. The design decision behind this is to ensure that all chromosomes are of the same structure at the genotype level, while still allowing for individuality at the phenotype level as the individual chromosome values will vary greatly resulting in huge variations. For example, even though all chromosomes are the same length and structure, the resulting phenotypes, which are pieces of music, will be of varying length with varying musical characteristics.

Fitness Function

When it came to the fitness function design, its importance was clear as it is the only point of user interaction during the evolutionary process. As well as this, as the main constraint on the project is the fact that music is very subjective, this constrains the project in the sense of determining the 'fitness' of music. When designing the fitness function, this notion of subjectivity was overcome by allowing the user to impact the evolution based upon the fitness source used. The fitness function was designed to consist of two main types of fitness calculation: explicit fitness and implicit fitness.

Explicit fitness is obtained by asking the user to provide a fitness rating within a specified range for a given piece of music created from the representation of the chromosome being evaluated at any given stage of evolution. The design decision for this was that by taking into account the user's subjective input into the evolutionary process, this means that the resulting piece of music at the end of the evolution process will be tailored to the user’s opinion of a ‘good’ piece of music based upon the fitness ratings given. This works due to the ability of the user provided fitness ratings to influence which chromosomes are selected to be brought onto each subsequent iteration of evolution. If a user provided fitness rating of a piece of music is high, this will improve the chances of the genetic algorithm bringing this chromosome onto the next iteration of evolution and thus bringing with it its distinct features and traits through mutation and crossover. If the user provided fitness rating of a piece of music is low, this increases the likelihood that the genetic algorithm will discard this chromosome from the evolutionary process.

Implicit fitness is based upon the design decision that regularity in music is appealing. Implicit fitness is obtained through the calculation of the regularity of a piece of music generated by the genetic algorithm. Through the mapping of the compression ratio of a piece of music to a fitness value, the genetic algorithm can then decide if the chromosome from which the piece of music was generated from will be brought onto the next iteration of evolution or if it will be discarded. This feature of the design of this genetic algorithm significantly increases the speed of the evolutionary process due to it taking away the need for human intervention.

Apart from the two main types of fitness, i.e. explicit and implicit, there are two other types of fitness that make use of these main types: hybrid and combined fitness. Hybrid fitness, as the name states, is a mix between explicit and implicit fitness. The fitness function decides whether or not to ask the user to provide an explicit fitness rating for each chromosome based upon the value obtained from the implicit fitness calculation for that chromosome. If the implicit fitness rating is outside of a specified range, the user is asked to provide an explicit fitness rating, which is then used as the fitness for that chromosome. Otherwise, if the implicit fitness rating is within the specified range, the implicit fitness rating is used.

Combined fitness calculates a fitness rating for a chromosome based upon *both* the explicit and implicit fitness ratings of that chromosome. Combined fitness gives the fitness function the functionality of being able to apply weightings to the two different sources of fitness used. The user can determine the weightings that will be given to the two sources of fitness values and thus, for example, could give greater weighting to human feedback over implicit fitness, and vice versa, or even give them the same weighting.

Depending on the intentions of the user, i.e. depending on certain testing scenarios, for example, a different fitness source can be used. Giving the user the option to decide on the source of fitness to be used for the evolution process was another very important design decision. This design allows for greater analysis to be carried out on the performance of the genetic algorithm due to allowing for comparisons to be made between the different sources of fitness (see in *Chapter 6. Results*).

Seeding the Evolution

This genetic algorithm has the added functionality that it can accept a seed chromosome that will be added to the initial population before the evolution process begins. The initial design decision behind this functionality was to help speed up the evolution process. By introducing a seed chromosome at the beginning of the evolution process this will result in the genetic algorithm arriving at chromosomes that appeal to the user more quickly as opposed to having no seed chromosome.

This works by seeding the genetic algorithm with a user selected seed chromosome that is considered to have a high fitness rating by the user, i.e. it is a chromosome, and in turn a piece of music, that appeals to the user. In the first iteration of evolution, the population will thus contain this seed chromosome. Due to this seed chromosome being initially selected by the user, the user is then very likely give it a very high fitness rating if an explicit fitness value is required (depending on the configuration of the genetic algorithm). The genetic algorithm will then be much more likely to bring this seed chromosome onto the next iteration of the evolution process due to it being very fit, resulting in this chromosome being used in crossover operations with other chromosomes, as well as being mutated along the way.

This design means that the characteristics and features of the initial seed chromosome get spread through the population, as well as getting mutated along the way. The ultimate design goal is to evolve chromosomes that appeal to the user earlier in the evolution process, which it is hoped will lead to the evolution of a very appealing piece of music based upon the user's tastes in a shorter amount of time.

Chapter 4. IMPLEMENTATION

For the implementation of this project, I decided to code it in Java. The main deciding factor when deciding on the language of choice was that I had great experience working with Java and felt that I had a very comprehensive understanding of the language. I had built up my knowledge and experience of Java from regularly using it throughout my college course, throughout my work placement and for personal use. It is also a very popular language worldwide, for both personal and enterprise standard applications, which is a testament to its powerful capabilities.

Secondly, I looked for available APIs for working with genetic algorithms that might be of use to implement this project. I soon discovered JGAP, which is a Java based API for working with genetic algorithms, explained in detail earlier (see *Chapter 2. Related Work*). JGAP had all of the sufficient functionality that I desired to implement the genetic algorithm, so the decision was made to use this framework.

Finally, I needed to come up with a way of creating pieces of music based upon chromosomes, so a music framework with the freedom to do this was needed. After some research, I decided that working with the MIDI protocol would enable me to do this, also explained earlier (see *Chapter 2. Related Work*). As MIDI fundamentally creates audio/music based upon a list of instructions, this meant that a chromosome representation could be created that maps a chromosome's values to a set of instructions in order to create an individual piece of music based on that chromosome.

Configuration Object

As part of the JGAP framework, setting up the genetic algorithm consists of the use of a *Configuration* object. The *Configuration* object is used to set up and configure all of the settings that are desired for the genetic algorithm, prior to running it. As there are numerous settings that can be configured, JGAP provides a *DefaultConfiguration* class that comes preconfigured with the most common settings. However, every genetic algorithm must specify three other fundamental pieces of information: the fitness function to use, the chromosome setup and the number of chromosomes in the population. [19] [20]

For this genetic algorithm, the *Configuration* object was initialised with a *DefaultConfiguration*:

```java
// Start with DefaultConfiguration, which comes setup with the most common settings
conf = new DefaultConfiguration();
```

Fitness Function

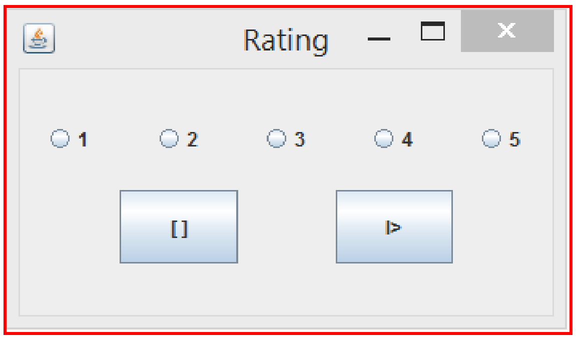
To implement the fitness function for the genetic algorithm a fitness function class called *ChromosomeFitnessFunction* was created, which extends JGAP's *FitnessFunction* class. This fitness function was then set on the *Configuration* object:

```
// Set the fitness function to use
myFitnessFunc = new ChromosomeFitnessFunction(crThresholdLower, crThresholdHigher);
try {
    conf.setFitnessFunction( myFitnessFunc );
} catch (InvalidConfigurationException e) {
    e.printStackTrace();
}
```

By overriding the *evaluate* method of *FitnessFunction*, this enabled the implementation of a project specific method to calculate the fitness for each chromosome, which gets called by the genetic algorithm every time a chromosome needs a fitness value. This method returns a positive double which the genetic algorithm uses as a measure of how fit the chromosome being examined is. The higher the returned value, the more fit the chromosome.

The *ChromosomeFitnessFunction* has the ability to calculate the fitness of a chromosome from one of a four of different fitness sources or calculations, which the user decides upon before they run the genetic algorithm. The four types are: explicit, implicit, hybrid and combined.

For **explicit** fitness, the chromosome is converted into a MIDI file and played to the user. The user then gives the piece of music a fitness rating out of 5. The rating given is then divided by 5 to get the fitness value used, as 5 is the max possible rating. The GUI for obtaining the explicit fitness rating was implemented using the *javax.swing* package, with added playback controls available to the user to stop and play the MIDI file:



For **implicit** fitness, the regularity of the chromosome is used as a measure of the fitness. The regularity is calculated as the compression ratio of the MIDI file generated from the chromosome, with the compressed size of the MIDI file calculated after compressing it into a zip file using *java.util.zip*: [21]

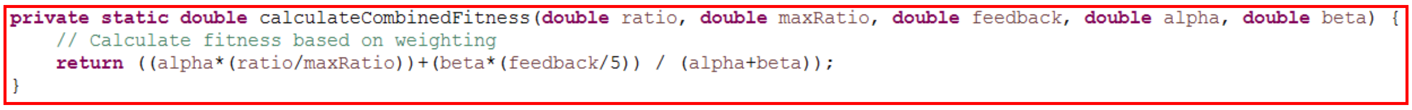
```
// Calculate CR by getting the ratio between the uncompressed size and compressed size
double compressionRatio = uncompressedSize/compressedSize;
```

The above compression ratio is then normalised using the maximum compression ratio for that iteration of evolution:

```
private static double calculateImplicitFitness(double ratio, double maxRatio) {
    // Calculate fitness based on weighting
    return ratio/maxRatio;
}
```

For **hybrid** fitness, the compression ratio of the MIDI file is calculated first. Then, depending on whether or not the compression ratio is within a user specified range, the chromosome will either be given an explicit or implicit fitness rating. This range is specified by the user prior to running the genetic algorithm by specifying the upper and lower thresholds for the range.

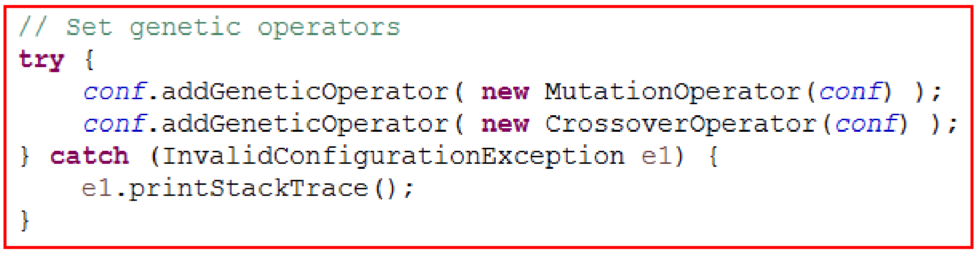
Finally, **combined** fitness is calculated using both an explicit and implicit fitness rating. With user specified weightings, these two fitness values are combined to calculate the combined fitness of the chromosome:



```
private static double calculateCombinedFitness(double ratio, double maxRatio, double feedback, double alpha, double beta) {
    // Calculate fitness based on weighting
    return ((alpha*(ratio/maxRatio))+(beta*(feedback/5)) / (alpha+beta));
}
```

Mutation and Crossover

To implement mutation and crossover, the genetic operators to be used by the genetic algorithm were set by adding a *MutationOperator* object and a *CrossoverOperator* object to the *Configuration* object:



```
// Set genetic operators
try {
    conf.addGeneticOperator( new MutationOperator(conf) );
    conf.addGeneticOperator( new CrossoverOperator(conf) );
} catch (InvalidConfigurationException e1) {
    e1.printStackTrace();
}
```

For the *MutationOperator,* the mutation rate is expressed as a denominator that divides 1. For example, if 100 was set as the desired mutation rate, this would result in 1/100 genes being mutated on average. When initialising the *MutationOperator* I decided not to pass in a specified mutation rate. Instead, by not passing in any specified mutation rate, this results in dynamic mutation being turned on. Dynamic mutation works by automatically determining the mutation rate based on the number of genes present in the chromosomes. [16]

For the *CrossoverOperator,* the crossover rate is determined by a crossover rate calculator, which calculates the required dynamic rate. When initialising the *CrossoverOperator* I chose not to pass in a crossover calculator. By doing this, it results in the dynamic crossover rate being turned off. This means that the crossover rate will be fixed at populationsize/2. I believed that this was a sufficient rate. [17] [18]

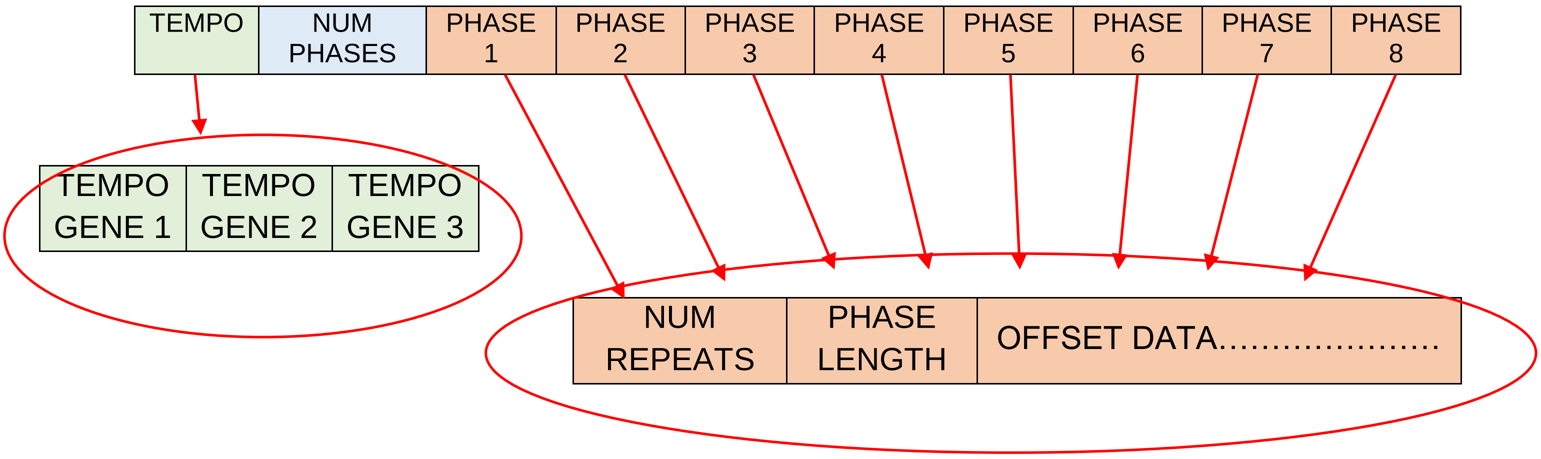
Chromosome Representation

Another essential part of the genetic algorithm setup that needs to be specified is the chromosome representation or structure. JGAP uses a *Chromosome* object that is initialised by passing in an array of individual *Gene* objects that make up the chromosome. JGAP provides subclasses of the *Gene* class for specifying the type of the individual *Genes* that make up a chromosome, such as *IntegerGene*, which was used in this implementation. *IntegerGenes* can be initialised with upper and lower bounds that restrict the range of legal values allowed by the *Gene*. Thus, when mutated, the value of a *Gene* is restricted to within its specified range. [22]

To create a sample *Chromosome* object for JGAP to use to create the initial random population of chromosomes, a method needed to be created to initialise the array individual Gene objects that make up a Chromosome object based on a template. This chromosome template is based on the creation of a MIDI file, initialising different *Genes* as *IntegerGenes* with different upper and lower bounds depending on that *Gene's* value in the *Chromosome* representation of the MIDI file. The chromosome representation will have *Genes* such as tempo *Genes*, the *Gene* that represents the number of phases in the MIDI file and phase data that gives the beat information in terms of the offset of each beat from the previous one. Depending on the number of genes in the sample chromosome, as specified by the length of the uninitialised *Gene* array passed as a parameter to this method, the template first calculates the maximum number of genes per 'phase'. This template uses the concept of a phase, which is a variable number of sequential *Genes*, i.e. a segment of *Genes* in the *Chromosome*, which can be repeated more than once when getting mapped to a MIDI file, with the number of repeats specified by a *Gene* before each phase. The template iterates through the uninitialised *Gene* array, initialising each *Gene* to be an *IntegerGene* with different bounds, as follows:

```java
private static Gene[] initialiseGenesFromTemplate(Gene[] sampleGenes) {
    // Calculate max num of genes per phase
    // Minus 4 (3 tempo genes, 1 gene for number of phases); Divide by max number of phases allowed
    int maxGenesPerPhase = (sampleGenes.length-4)/8;

    // Create template IntegerGene objects to represent each of the Genes
    // Set upper and lower bounds on each gene depending on their representation in the chromosome
    // Create the chromosome template with the maximum allowed values for each part of the chromosome
    // Create max of 8 phases for template, with each phase having a specified max length (maxGenesPerPhase)
    // This may result in 'dead DNA' depending on the Gene values specified by each chromosome
    // Dead DNA will be ignored when creating MIDI file
    int count = 0;
    for (int i=0; i<sampleGenes.length; i++) {
        // For each gene, set its bounds depending on its representation in the chromosome
        try {
            if (i<3) {
                // First 3 genes: tempo
                sampleGenes[i] = new IntegerGene(conf, 1, 20);
            } else if (i==3) {
                // Fourth gene: number of phases
                sampleGenes[i] = new IntegerGene(conf, 1, 8);
            } else {
                // Remaining genes: phase data
                if (count%maxGenesPerPhase == 0) {
                    // number of times phase is repeated
                    sampleGenes[i] = new IntegerGene(conf, 1, 6);
                } else if (count%maxGenesPerPhase == 1) {
                    // length of phase (# of Genes)
                    sampleGenes[i] = new IntegerGene(conf, 1, maxGenesPerPhase-1);
                } else {
                    // offset values
                    sampleGenes[i] = new IntegerGene(conf, 1, 30);
                }
                // Increment counter
                count++;
            }
        } catch (InvalidConfigurationException e) {
            e.printStackTrace();
        }
    }
    return sampleGenes;
}
```

A graphical explanation of the template chromosome representation is shown below. Note that this implementation allows a maximum of 8 phases:



Thus, a sample *Chromosome* was then generated, passing in the array of sample *Genes* initialised from the chromosome template above. The sample chromosome and the population size was then set on the *Configuration* object:

```
// Set number of genes in each chromosome
// Initialise all genes based upon template
Gene[] sampleGenes = initialiseGenesFromTemplate(new Gene[numGenes]);

// Create sample chromosome
Chromosome sampleChromosome = null;
try {
    sampleChromosome = new Chromosome(conf, sampleGenes);
} catch (InvalidConfigurationException e) {
    e.printStackTrace();
}

// Set it on the Configuration object as a template for the chromosomes to create
try {
    conf.setSampleChromosome( sampleChromosome );
} catch (InvalidConfigurationException e) {
    e.printStackTrace();
}

// Set population size
// The bigger the popn the more potential solutions but takes longer to evolve each iteration
try {
    conf.setPopulationSize(populationSize);
} catch (InvalidConfigurationException e) {
    e.printStackTrace();
}
```

Seeding the Evolution

The user has the option to add a seed chromosome to the initial population before the evolution begins. This is implemented by replacing a random chromosome from the initial population of chromosomes, which themselves are randomly generated values at the beginning, with the seed chromosome.

```java
// Add seed chromosome to population
private static Genotype addSeedChromosomeToPopn(IChromosome seedChromosome, Genotype population) {

    // Gets initialised with seeded population
    Genotype seededPopulation = null;

    // Add seed chromosome to population if specified
    if (seedChromosome != null) {

        // Get initial random population of chromosomes
        IChromosome[] chs = population.~~getChromosomes~~();

        // Replace random chromosome in array with seed chromosome
        int randomPos = (int) (Math.random()*100)%chs.length;
        chs[randomPos] = seedChromosome;

        try {
            // Create new Genotype with seeded population
            seededPopulation = new ~~Genotype~~(conf, chs);
        } catch (InvalidConfigurationException e) {
            e.printStackTrace();
        }
    }

    // Return seeded population
    return seededPopulation;
}
```

The seed chromosome itself is created by reading from a text file that contains the integer values for all the genes of the chromosome in the form of a comma separated value (CSV) file. These gene values are read in sequentially and are set as the allele value for the initialised array of *Gene* objects created based upon the chromosome template, as shown:

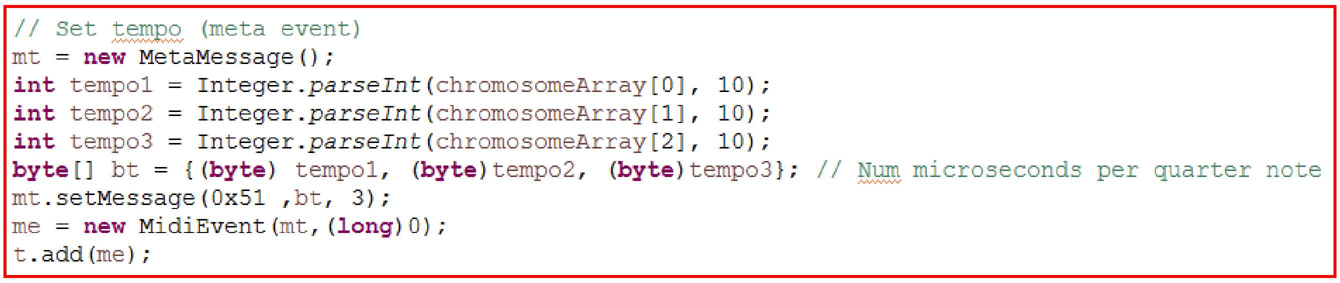
```java
br = new BufferedReader(new FileReader("seedChromosome.txt"));

// Read line
String line = br.readLine();

// Split the genes, using comma as separator
String[] genes = line.split(",");

// Set number of genes in the chromosome
// Initialise all genes based upon template
Gene[] seedGenes = initialiseGenesFromTemplate(new Gene[genes.length]);

// Set each gene's allele value according to seed chromosome values
for (int i=0; i<seedGenes.length; i++) {
    seedGenes[i].setAllele(Integer.parseInt(genes[i]));
}

seedChromosome = new Chromosome(conf, seedGenes);
```

MIDI

This project creates music in the MIDI format (extension *.mid*). Thus, working with MIDI in Java was implemented using the *javax.sound.midi* package. This package provided access to the Java Sound API's MIDI architecture, which is a very useful API to make working with MIDI much more manageable. [23] [24]

To work with MIDI, all methods were grouped into a helper class called *MidiActions*. This made MIDI actions available to the classes in the genetic algorithm package that needed to access them in order to work with MIDI. *MidiActions* contains the fundamental methods to create and play a MIDI file, as well as others for playback control and to get the compression ratio of a MIDI file.

The implementation of creating a MIDI file involved working directly with the *javax.sound.midi* package. This package provides classes for the creation of *MidiEvent* objects that are added to *Tracks*. Note that this implementation contains only one *Track*. These *MidiEvent* objects are fundamentally what make up the list of instructions that are contained in a .MID file, with each *MidiEvent* containing a *MidiMessage* and a timestamp for the event. Therefore, in order to create a MIDI file from a *Chromosome*, its *Genes* were used to create different *MidiEvent* objects depending on the *Gene's* representation from the chromosome template, explained above. Tempo was implemented through the creation of one *MidiEvent*, which required the first three Genes in the chromosome to set the tempo for the *Track*:



```
// Set tempo (meta event)
mt = new MetaMessage();
int tempo1 = Integer.parseInt(chromosomeArray[0], 10);
int tempo2 = Integer.parseInt(chromosomeArray[1], 10);
int tempo3 = Integer.parseInt(chromosomeArray[2], 10);
byte[] bt = {(byte) tempo1, (byte)tempo2, (byte)tempo3}; // Num microseconds per quarter note
mt.setMessage(0x51 ,bt, 3);
me = new MidiEvent(mt,(long)0);
t.add(me);
```

The remaining chromosome data is converted to a MIDI file by reading the phase data, as described by the template above, creating *MidiEvents* to generate a beat at the times specified by the offset data of chromosome. The number of times all the *MidiEvent* objects for a phase get written to the MIDI file depends on the number of phase repeats specified by the chromosome.

This implementation uses two *MidiEvent* objects to create a beat, one event with a "Note On" message, and the other with a "Note Off" message. These control signals, as well as the instrument to use, are set using MIDI protocol codes when creating the message that is set with a *MidiEvent* object. [30] The "Note Off" message is created with a timestamp slightly offset from the "Note On" message to ensure that each beat is played fully and not overlapping with the next beat:

```
// Note on - Electric Snare - 40
mm = new ShortMessage();
mm.setMessage(0x99, 0x28, 0x40);
me = new MidiEvent(mm,(long)beatTime);
t.add(me);

// Increment beatTime for the note off event
// Ensures each beat is played and not overlapping with next beat
beatTime+=20;

// Note off - Electric Snare - 40
mm = new ShortMessage();
mm.setMessage(0x89, 0x28, 0x00);
me = new MidiEvent(mm,(long)beatTime);
t.add(me);
```

The end of the MIDI file is specified with another control signal, in the form of a *MidiEvent* with a message and timestamp to indicate this:

```
// Set end of track (meta event)
mt = new MetaMessage();
byte[] bet = {}; // empty array
mt.setMessage(0x2F,bet,0);
me = new MidiEvent(mt, (long)beatTime);
t.add(me);
```

To play a MIDI file, the *javax.sound.midi* package makes use of a *Sequencer* object that is initialised in this implementation with the default sequencer for the device. The *Sequencer* is then set with the MIDI file to play in the form of an *InputStream*. The *Sequencer* also provides a number of methods for playback control. [26]

```
// Get default Sequencer connected to default device
sequencer = MidiSystem.getSequencer();

// Opens the device acquiring any system resources it requires
sequencer.open();

// Create input stream to read in MIDI file data
InputStream in = new BufferedInputStream(new FileInputStream(midiFile));

// Set the sequence to operate on, pointing to the MIDI file data in the input stream
sequencer.setSequence(in);

// Start playing the MIDI file data
sequencer.start();
```

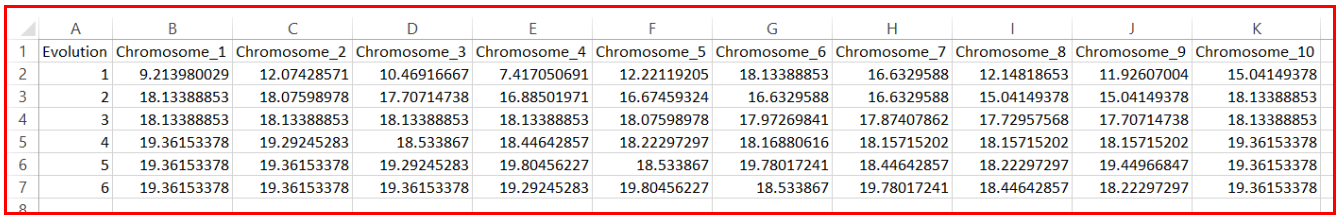
In the implementation of the creation of MIDI files, some of the precise system settings used for configuring MIDI with the *javax.sound.midi* package were based upon numerous sources of information due to their high level of technicality from a MIDI perspective. For example, the settings used to initialise the *Sequencer* object before obtaining a *Track* from it when creating a MIDI file, and the settings to turn on the General MIDI sound set are very precise, technical settings that go beyond the scope of moderate expertise needed for this project. [25] [27] [28] [29]

Gathering Data for Analysis

The gathering of compression ratio data was implemented by creating a 2D array of doubles to collect the data before writing it to a file at the end of the evolution process. The first dimension of the array stores the iteration of evolution and the second dimension stores the compression ratio values for the chromosomes in each iteration of evolution.

```
// 2D array of doubles to store all compression ratios for each iteration of evolution and final population
double[][] compressionRatios = new double[numEvolutions+1][populationSize];
```

The data gets written to a CSV file to allow for easy analysis when the data is opened in Microsoft Excel:



	A	B	C	D	E	F	G	H	I	J	K
1	Evolution	Chromosome_1	Chromosome_2	Chromosome_3	Chromosome_4	Chromosome_5	Chromosome_6	Chromosome_7	Chromosome_8	Chromosome_9	Chromosome_10
2	1	9.213980029	12.07428571	10.46916667	7.417050691	12.22119205	18.13388853	16.6329588	12.14818653	11.92607004	15.04149378
3	2	18.13388853	18.07598978	17.70714738	16.88501971	16.67459324	16.6329588	16.6329588	15.04149378	15.04149378	18.13388853
4	3	18.13388853	18.13388853	18.13388853	18.13388853	18.07598978	17.97269841	17.87407862	17.72957568	17.70714738	18.13388853
5	4	19.36153378	19.29245283	18.533867	18.44642857	18.22297297	18.16880616	18.15715202	18.15715202	18.15715202	19.36153378
6	5	19.36153378	19.36153378	19.29245283	19.80456227	18.533867	19.78017241	18.44642857	18.22297297	19.44966847	19.36153378
7	6	19.36153378	19.36153378	19.36153378	19.29245283	19.80456227	18.533867	19.78017241	18.44642857	18.22297297	19.36153378
8											

Similarly, the gathering of fitness data was implemented by writing it out to a CSV file to allow for easy analysis in MS Excel. However, instead of storing the data in memory until the end of the evolution, a static *BufferedWriter* object and a static *FileWriter* object were created on the initialisation of the *ChromosomeFitnessFunction* object. This facilitates writing out to the file during the evolution process as the fitness values are calculated. Each time the *evaluate* method in the *ChromosomeFitnessFunction* object is called, the newly calculated fitness value is written to the *BufferedWriter*. The *BufferedWriter* is flushed after each iteration of evolution which writes the fitness values for that iteration to the file.

```
// Constructor
public ChromosomeFitnessFunction(double crThresholdLower, double crThresholdHigher) {
    // Set CR threshold bounds on initialisation
    CRTHRESHOLDLOWER = crThresholdLower;
    CRTHRESHOLDHIGHER = crThresholdHigher;

    // Initialise writers to write to fitness data file
    fitnessFile = new File("FitnessData.csv");
    fw_fitness = null;
    bw_fitness = null;
    try {
        fw_fitness = new FileWriter(fitnessFile);
        bw_fitness = new BufferedWriter(fw_fitness);
    } catch (IOException e) {
        System.out.println("Error creating writers for FitnessData.csv");
        e.printStackTrace();
    }
}
```

A very similar approach is used to implement the gathering of data for the hybrid fitness source and for the analysis of the mapping of the implicit fitness to the explicit fitness, which writes error rates to a file.

How to Evolve Music

At a higher level, the user begins the process of evolving music by selecting the desired configuration settings, such as number of evolutions, population size, the option to seed the chromosome, the fitness source and other configurations for certain fitness sources. The genetic algorithm is then run.

The application will begin by configuring the genetic algorithm according to the user specified configuration settings outlined above. It will then evolve a population of chromosomes the specified number of evolutions, possibly involving some user input in the fitness calculations throughout the evolution depending on the fitness source used. The fittest chromosome of the evolved population is then selected, a MIDI file is created from its representation and is then played back to the user.

Data will be gathered throughout the evolution and written to files to allow for analysis by the user. The data that is gathered will vary depending on the configuration settings chosen by the user.

Chapter 5. EXPERIMENTS

For all types of fitness sources that can be used with this genetic algorithm, which are used in the experiments listed below, compression ratio data and fitness data are *always* gathered. This gives the user the option to analyse this data to check regularity of the music throughout the evolution and also facilitates the analysis of the fitness of the chromosomes in each iteration of evolution. However, it may not be necessary to analyse the compression ratio data and fitness data for every experiment, depending on the aim of the experiment, as it may already have been dealt with in a previous experiment.

Experiment 1 - Explicit Fitness: Human Feedback

This experiment aims to analyse the trends and results of the explicit fitness function, where the user is asked to provide a fitness rating in the range of 1-5 for each chromosome in the evolutionary process. Each chromosome is converted to a MIDI file, played to the user and a fitness rating is taken before going on to the next, until the evolution is complete. It is expect to show variances between different users during this experiment to reflect the subjective differences of opinions of the users.

Experiment 2 - Implicit Fitness: Regularity (No Human Feedback)

This experiment aims to analyse the trends and results of the implicit fitness function, where each chromosome in the evolutionary process is implicitly given a fitness rating based upon the regularity of the piece of music it produces. The measure of regularity that is used is the compression ratio of the piece of music. This is calculated as the uncompressed file size divided by the compressed file size. The implicit fitness value given is normalised according to the maximum compression ratio for all chromosomes, per evolution. This experiment involves no human involvement.

This experiment will be ran with both a seeded and unseeded population respectively. This will allow for further analysis and comparison on the effect of seeding the genetic algorithm.

Experiment 3 – Analysis of the Mapping of Implicit Fitness to Explicit Fitness

This experiment will use an implicit fitness function for its entire evolution. In order to analyse the mapping of the implicit fitness to the explicit fitness, this experiment will ask the user for an explicit fitness rating for some of the chromosomes at each iteration of evolution. Note that these explicit fitness ratings will not influence the evolution, they are purely for analysis purposes.

This experiment will store the top 3 chromosomes with the best implicit fitness values at each iteration of evolution. The user will then be asked to give these top chromosomes an explicit fitness rating. The implicit and explicit ratings for these chromosomes will be stored separately for analysis.

Experiment 4 - Compression Ratio Analysis

This experiment involves gathering data regarding the compression ratio over time, to see if there are any trends that exist in the data. For fitness sources that in some way use implicit fitness, it is expected to see that the compression ratio levels out over time, going from low regularity to higher regularity, due to the genetic algorithm favouring high compression ratios. This experiment will gather compression ratio data for each type of fitness source and analyse the results of each.

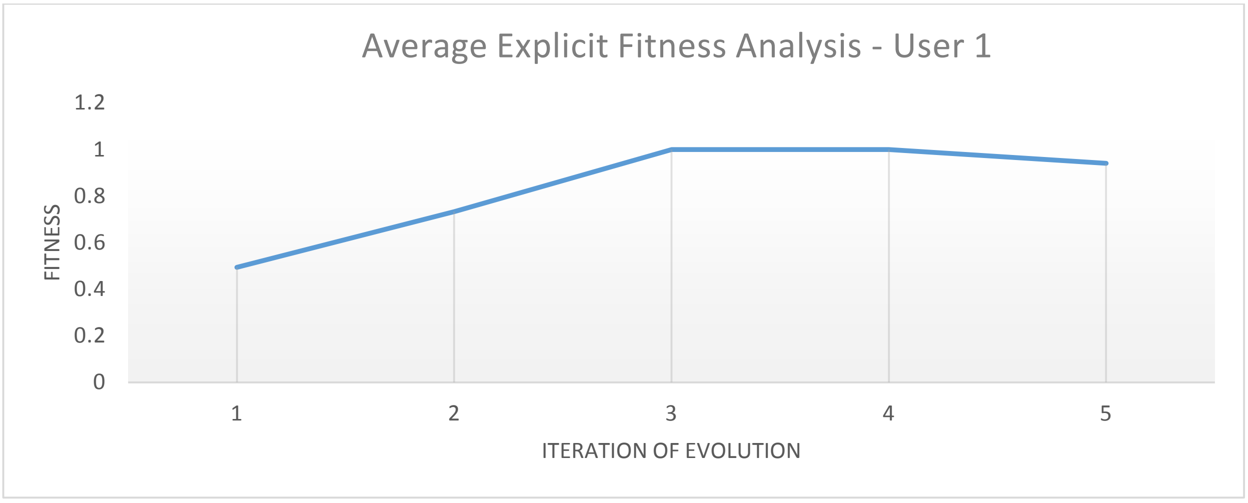
Experiment 1 - Explicit Fitness: Human Feedback

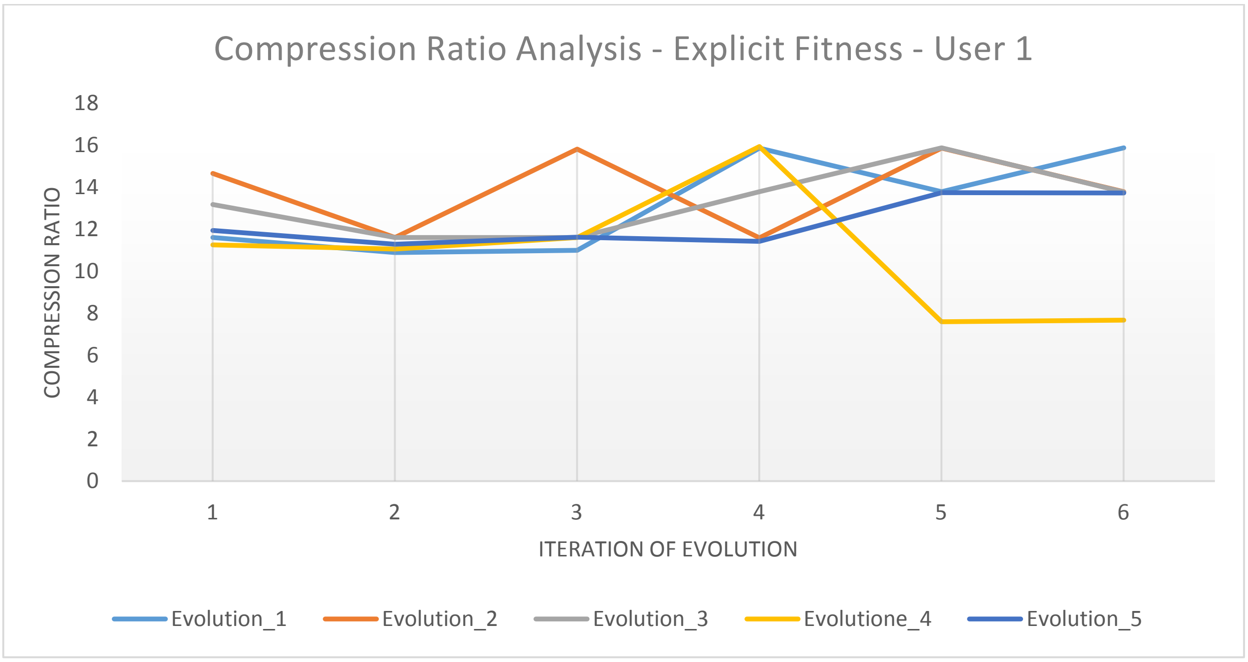
This experiment was ran with two different users.

- Configuration Settings:

Configuration Variable	Setting
Population Size	5
Number of Evolutions	5
Seed Evolution	TRUE

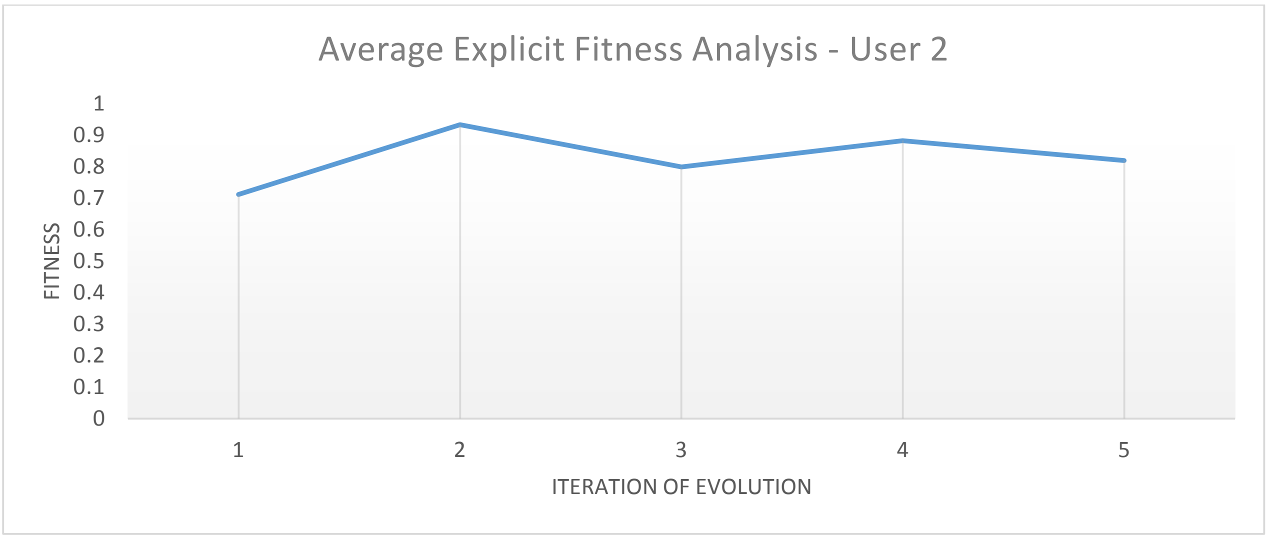
- User 1:

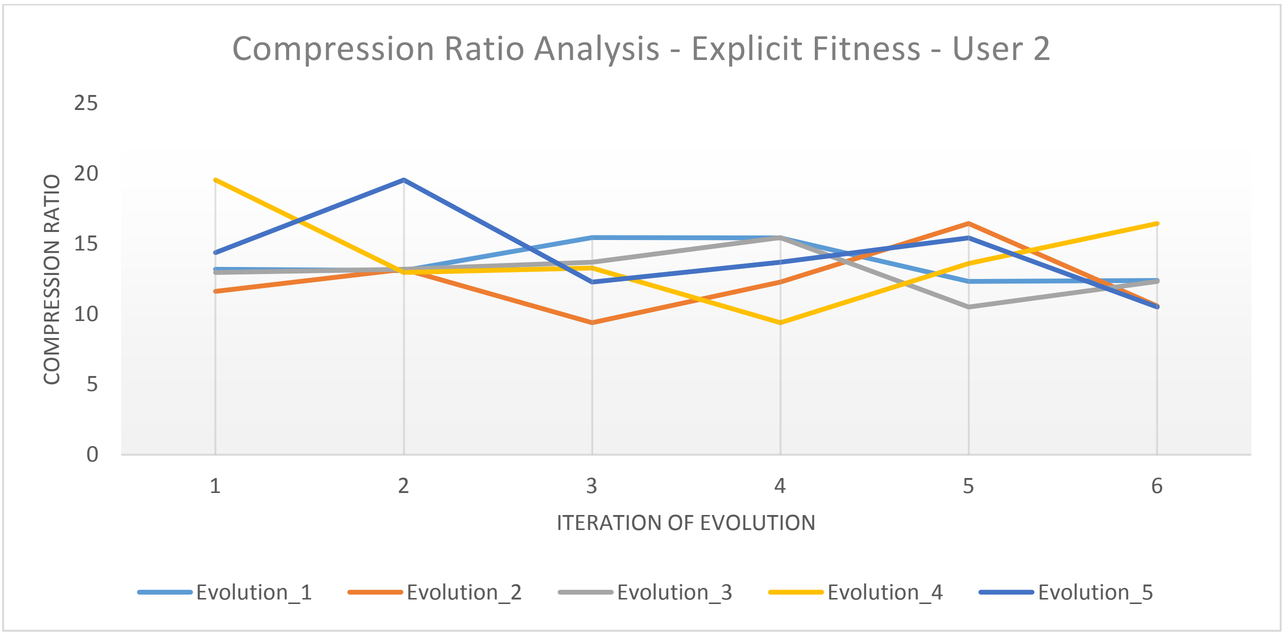




From the above graphs for User1, it is clear that the average fitness rose over the first two iterations of evolution before stabilising and slightly tapering off towards the end. The data in the compression ratio analysis graph shows the clustering of the compression ratio values for the chromosomes mostly between 10 and 16.

- User 2:





From the above graphs for User2, the average fitness rose over the first iterations of evolution and began fluctuating quite a lot for the rest of the. The data in the compression ratio analysis graph shows the clustering of the compression ratio values for the chromosomes mostly between 8 and 20.

When comparing the two users, it is obvious that there exists some differences. From the comparison of the average fitness graphs, it can be seen that User1 favoured the pieces of music more as the evolution went on, with slightly less favouring towards the end of the evolution. User2 differed slightly as the average fitness began to fluctuate after the second iteration of evolution. From the comparison of the compression ratio graphs, the compression ratio data for User1 seems to be within a narrower clustering than the compression ratio data clustering for User2 which seems to be wider. These results reflect the expected outcome of this experiment, as these variances between the different users reflect the subjective differences of opinions that people have about music

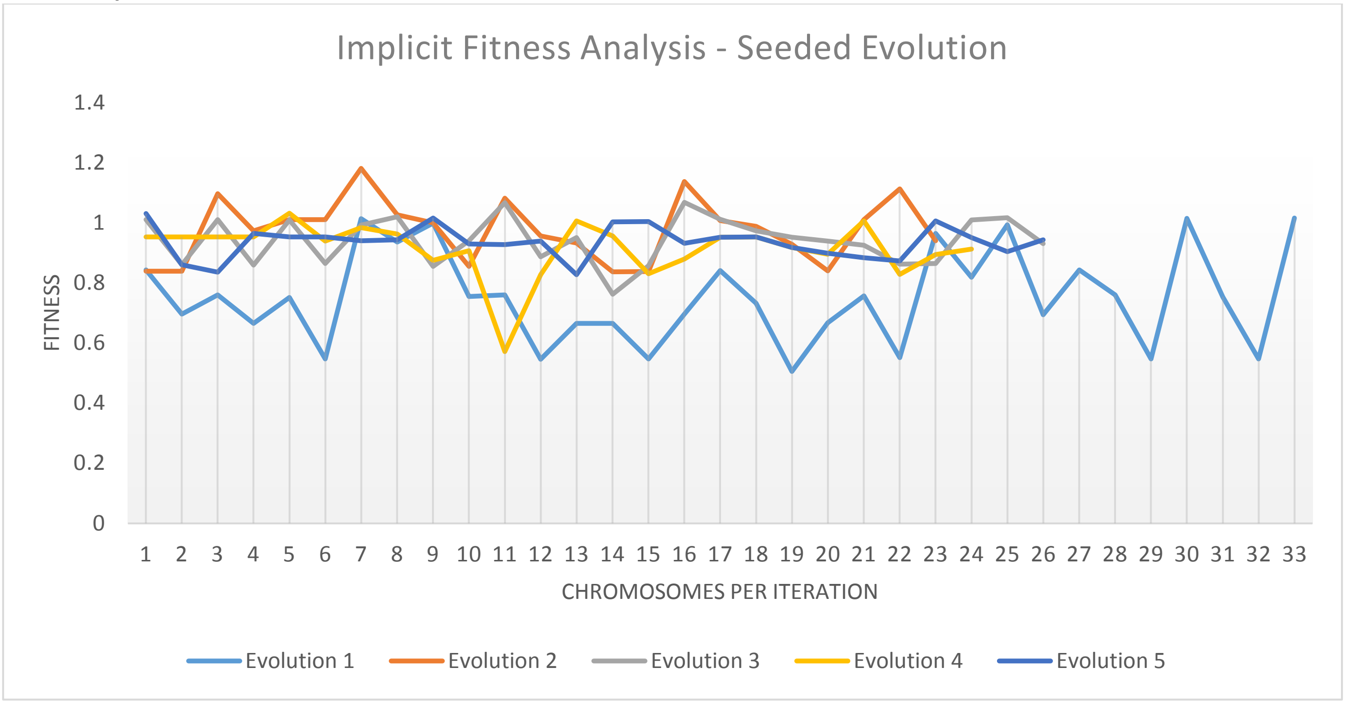
Experiment 2 - Implicit Fitness: Regularity (No Human Feedback)

a) Seeded

- Configuration Settings:

Configuration Variable	Setting
Population Size	10
Number of Evolutions	5
Seed Evolution	TRUE

- Graph:

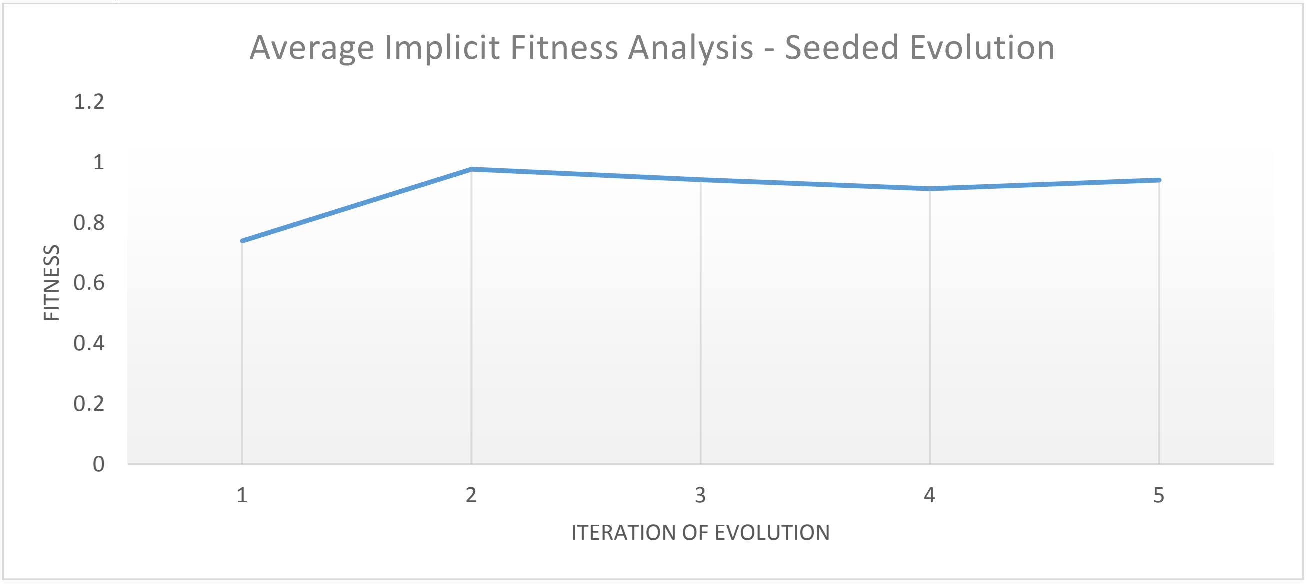


(Note: Although there are only 10 chromosomes per iteration of evolution in this experiment, that the above graph shows many more fitness calculations per iteration. The possible reasoning behind this is the way in which JGAP evolves the population of the chromosomes, calling the evaluate method numerous times during this process. It is believed that it calls it numerous times to evaluate the fitness of the children chromosomes created from mutation and crossover. Thus, as this project implements the gathering of the fitness data by saving every fitness value calculated in each iteration of evolution, this has led to the gathering of all of this data. However, this does not impact the analysis, as the analysis is based upon the overall traits of the data as a whole.)

Implicit fitness is based on regularity, so the chromosomes with a higher fitness are those that are more regular. From this graph of seeded implicit fitness, it is clear that as the evolution iteration increases the fitness of the chromosomes in that iteration also increases. This is to be expected as the aim of the genetic algorithm is to evolve chromosomes based upon their fitness, with high fitness chromosomes being desired at the end of the evolution. Here, it can be seen that the line on the graph for the final iteration of evolution approaches a fitness of 1. Note that the line is not very smooth as it contains some chromosomes with fitness values that are further away from 1 than the majority of others in that iteration.

It is believed that the cause of these deviations from the fitness of 1, i.e. the reason for the jaggedness of the line, is due to the seeding of the evolution with a seed chromosome. As this seed chromosome was chosen by the user, this chromosome may not have been a highly regular piece of music, due to the tendency of users to prefer music that is not overly regular. Thus, as this was introduced to the evolution it may have influenced the outcome of the regularity of the rest of the chromosomes if its traits were brought onto subsequent iterations of evolution through mutation and crossover. Therefore, it is believed that the implicit fitness was impacted by the lower regularity of the seed chromosome resulting in more diversity of implicit fitness in the final evolution. Although that this would mean that seeding the evolution reduces the average implicit fitness, as implicit fitness favours regularity, it still ends up evolving chromosomes that are regular but with some diversity, which tends to be what appeals to users.

- Graph:



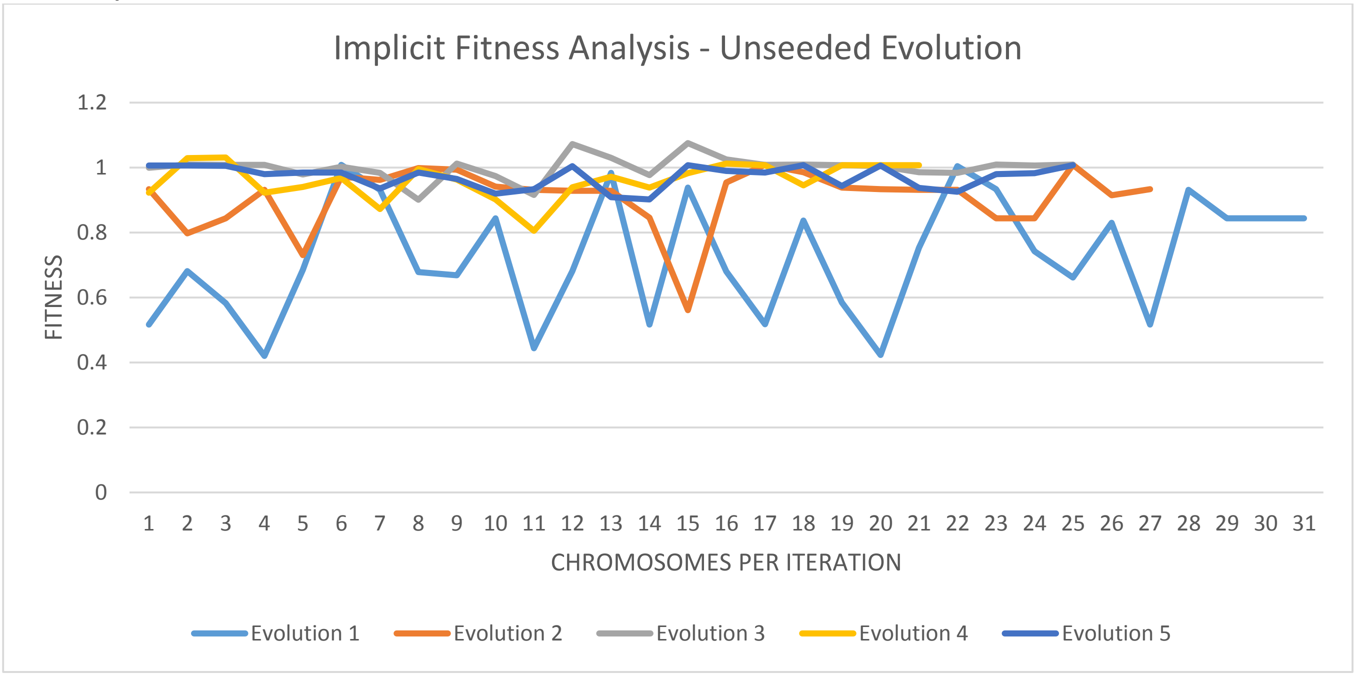
From this graph of the average implicit fitness, it is clear that the fitness increases dramatically after the first iteration of evolution before levelling out with just minor fluctuations. This sudden increase in fitness after the first iteration of evolution may be due to the seed chromosome, as its intended use was to help speed up the process of evolution.

b) Unseeded

- Configuration Settings:

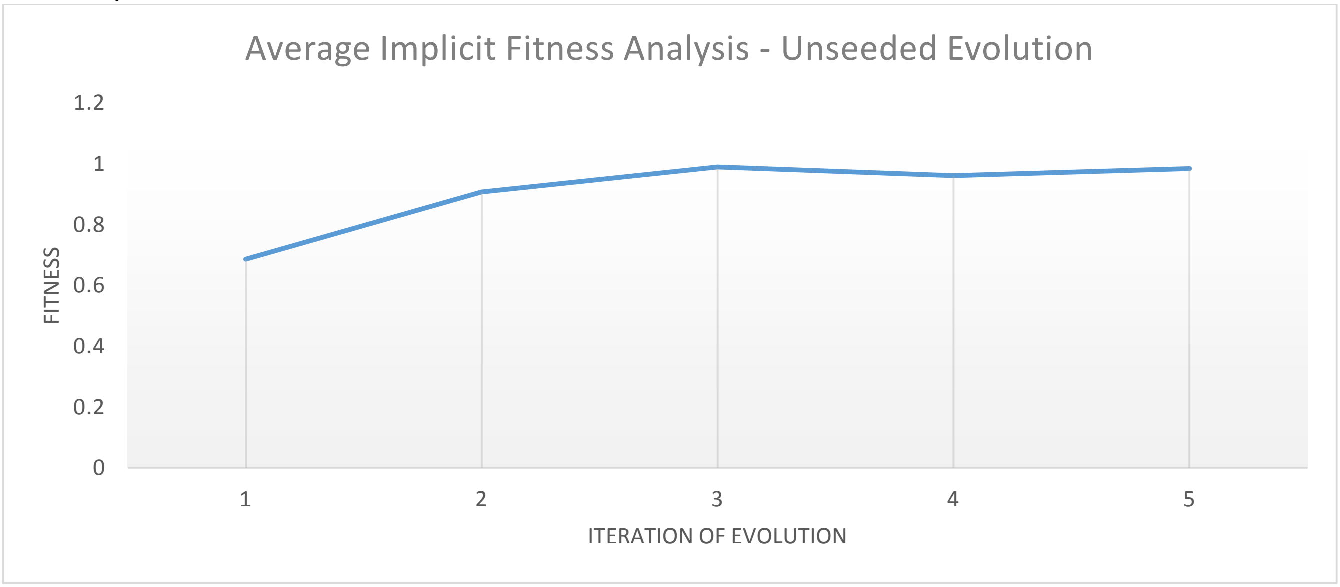
Configuration Variable	**Setting**
Population Size	10
Number of Evolutions	5
Seed Evolution	FALSE

- Graph:



Once again, from the above graph of unseeded implicit fitness, it is clear that as the evolution iteration increases the fitness of the chromosomes in that iteration also increases. Here it can be seen again that with each iteration of evolution the lines smoothen out and approach the fitness value of 1. Noticeably, in this graph of unseeded evolution, the lines are much smoother with every iteration in comparison to the lines in the seeded evolution graph above. This backs up the reasoning proposed for the cause of the more jagged lines in the seeded evolution above. Due to having no seed chromosome, this implicit fitness function can select the fittest chromosomes to bring onto subsequent stages of evolution as normal and evolve the best solutions, i.e. the most regular chromosomes. Without having to deal with a possibly more irregular seed chromosome than the rest of the population, there is less risk of the regularity of the chromosomes being impacted due to the traits of the less regular seed chromosome being spread throughout the population by mutation and crossover.

- Graph:



From this graph of the average implicit fitness, it is clear that once again the fitness increases with each iteration of evolution before levelling out. In comparison to the graph of the average implicit fitness in the seeded evolution above, the fitness takes slightly longer to increase before levelling out. This may be due to having no seed chromosome to help speed up the process of evolution.

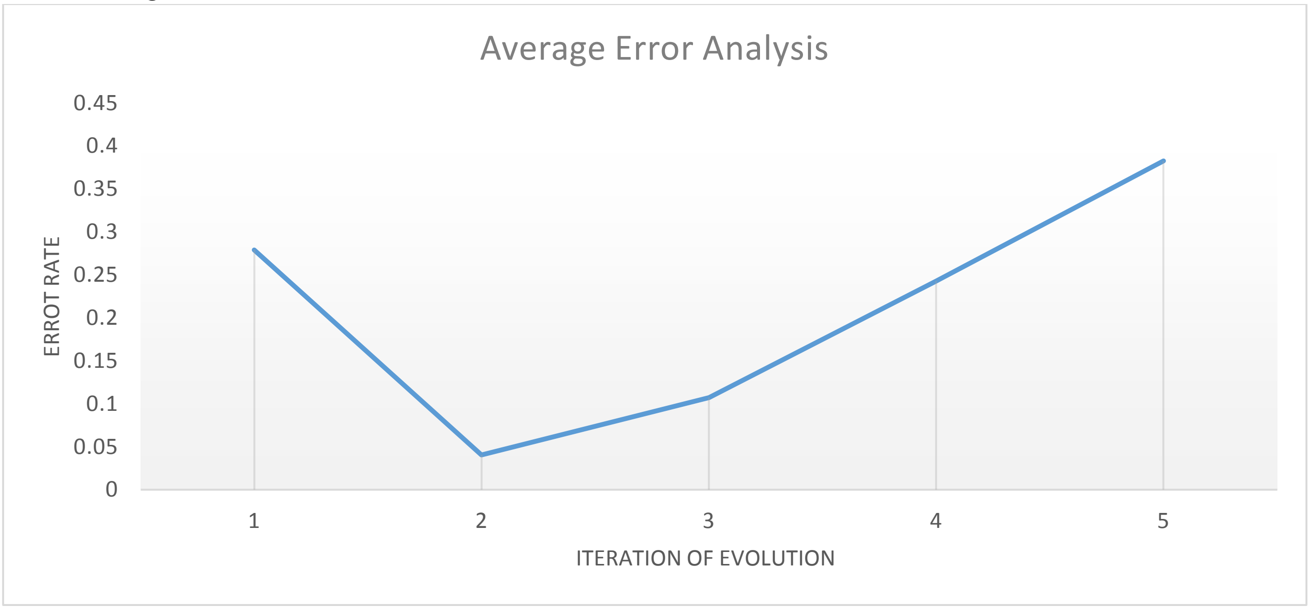
Experiment 3 – Analysis of the Mapping of Implicit Fitness to Explicit Fitness

For this experiment, the genetic algorithm was ran using implicit fitness. After getting the explicit fitness ratings for the top 3 fittest chromosomes per iteration of evolution, this data was stored as the error between the implicit and the explicit fitness, i.e. implicit fitness – explicit fitness. The average error per iteration of evolution was also calculated. Graphs were generated from the gathered data and the results were analysed below.

- Configuration Settings:

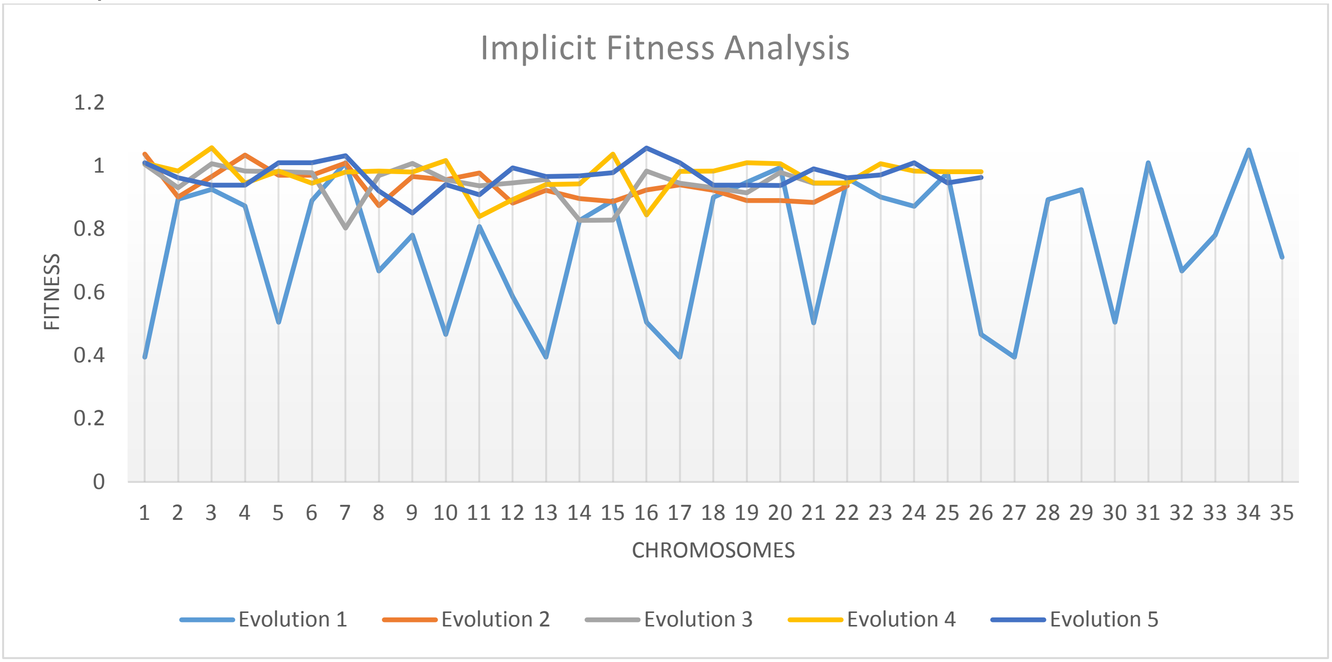
Configuration Variable	Setting
Population Size	10
Number of Evolutions	5
Seed Evolution	TRUE

- Average Error:



The above graph shows the average error over each of the five iterations of evolution. The user's opinion of fitness differs to the implicit fitness mostly at the beginning and towards the end of the evolution process, with the middle of the evolution process being the only period where the user's opinion was somewhat similar to that of the implicit fitness.

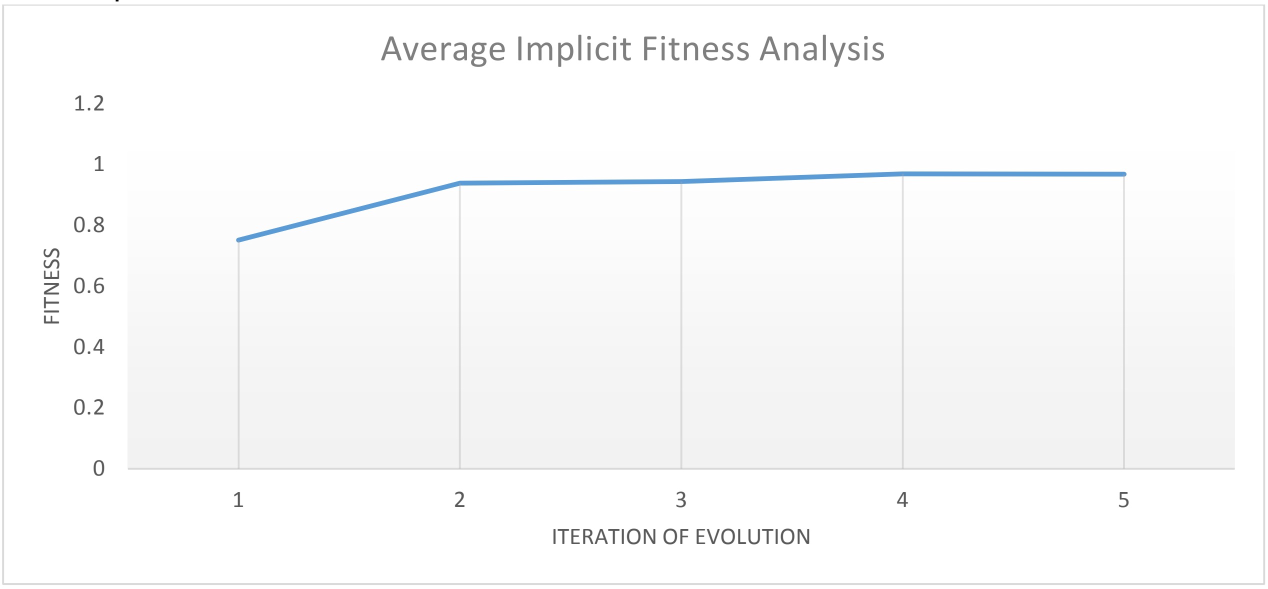
- Implicit Fitness:



From the above graph of the fitness data in this experiment, it is clear that the genetic algorithm evolves chromosomes to be fitter with each iteration, as expected. The lines on the graph go from being somewhat erratic in the first iteration of evolution to a more smooth line that approaches a fitness of 1 in the final iteration of evolution.

As implicit fitness is based on and favours regularity, the chromosomes with the highest fitness are those that are very regular. As can be seen in the graph of average error above, the difference in the user's explicit fitness ratings of the chromosomes and the implicit fitness ratings increase from the second iteration of evolution onwards. It is believed that this is due to the pieces of music becoming too regular for the user's subjective taste as the evolution goes on. This proposed reasoning is backed up when at the second iteration of evolution the average error drops significantly from the first iteration. Thus, it is believed that the pieces of music produced at the second iteration were of optimal regularity to the user in this case and that the first iteration produced chromosomes that were too irregular. Once again, the results seem to back up the notion that regularity is favoured, but only favoured with a limit as to the amount of regularity.

- Graph:



Once again this graph of the average implicit fitness, shows that the fitness increases with each iteration of evolution before levelling out, as expected. This graph shows very little fluctuation in the average fitness between the final four iterations of evolution, as well as showing that the average fitness dramatically increased after the first iteration of evolution. This seems to reinforce the concept that the seed chromosome helps to speed up the process of evolution, as a fit chromosome is introduced to the population before the evolution beings.

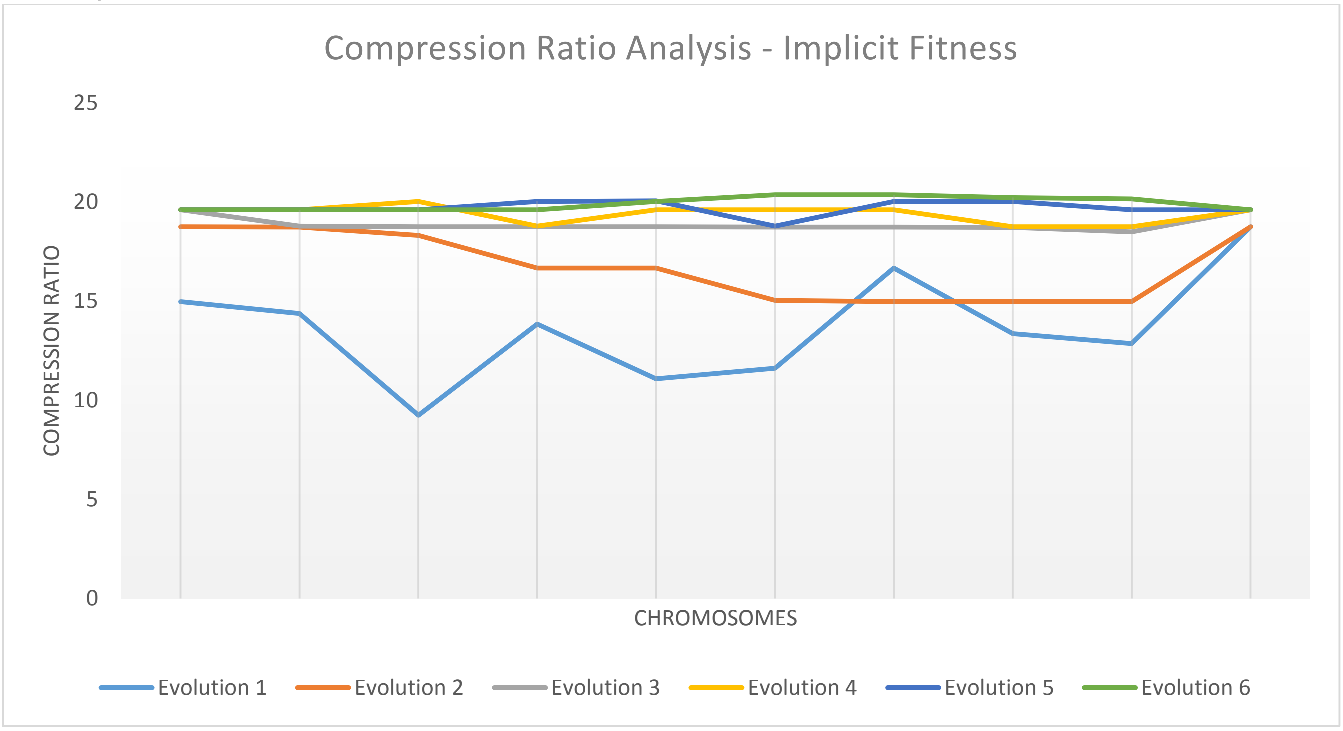
Experiment 4 - Compression Ratio Analysis

For this experiment the genetic algorithm was ran with all types of fitness sources respectively. Graphs were generated from the gathered data and the results analysed for each below. (Note: in the graphs below, “Evolution 1” is the initial population of chromosomes before evolution begins)

- Configuration Settings:

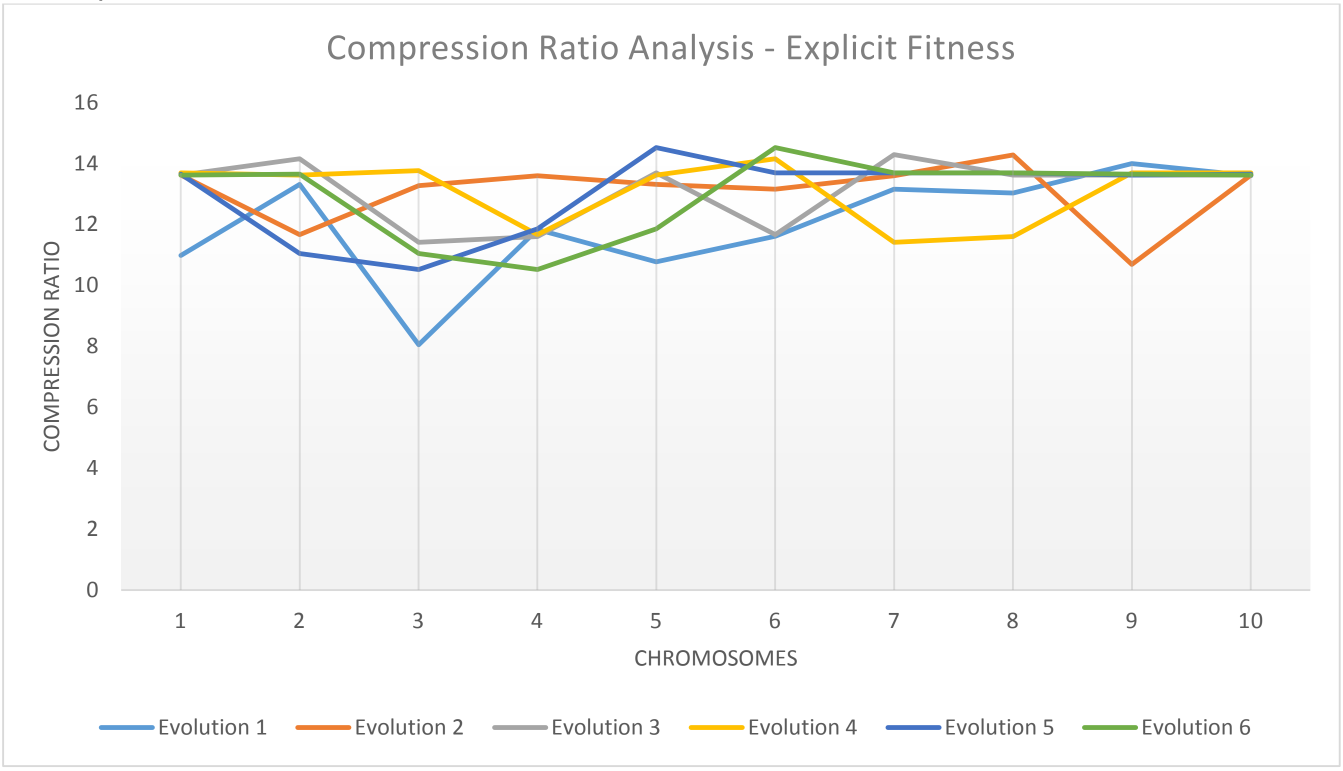
Configuration Variable	Setting	Used by Fitness Source
Population Size	10	All
Number of Evolutions	5	All
Seed Evolution	TRUE	All
CR Threshold Lower	8	Hybrid
CR Threshold Higher	20	Hybrid
Implicit Weighting	1	Combined
Explicit Weighting	2	Combined

- Implicit Fitness:



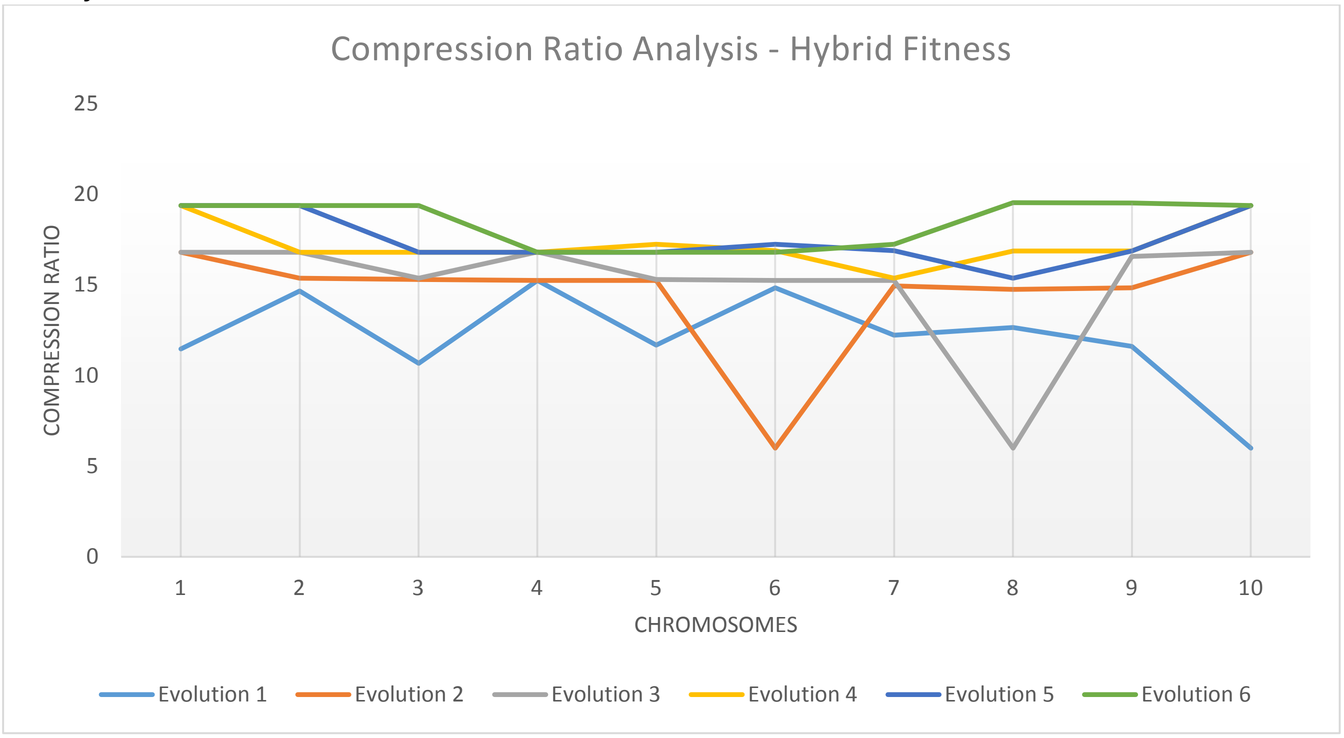
As expected, it is clear from the above graph that the compression ratio levels out over time. This is due to the genetic algorithm favouring higher compression ratios, hence it will try and evolve a solution that has a high compression ratio as quickly as possible. From the above graph it is obvious that as the evolutions of iteration increase, the line to signify the compression ratios of the chromosomes at each iteration smoothens out as it approaches a very high compression ratio near 20. Thus, it is true to say that implicit fitness favours regularity, evolving chromosomes to become as regular as possible.

- Explicit Fitness:



From the above graph, the explicit fitness does not show any obvious signs of favouring regularity. The lines that signify the compression ratios of the chromosomes for each iteration have no overly apparent trends, apart from their broad-scoped clustering in the range of 10-15. However, on further inspection it can be seen that the first iteration of evolution seems to have chromosomes of lower regularity than those in the evolved population. It is believed that the reasoning behind this is that some amount of regularity appealed to the user, but the user did not like music that was very regular. This graph shows that the evolved population's regularity is well below its potential maximum regularity, as gauged from the implicit graph above which showed that the chromosomes examined approached a maximum compression ratio around 20. Thus, this reinforces the belief that users like regularity in music, but not too much.

- Hybrid Fitness:

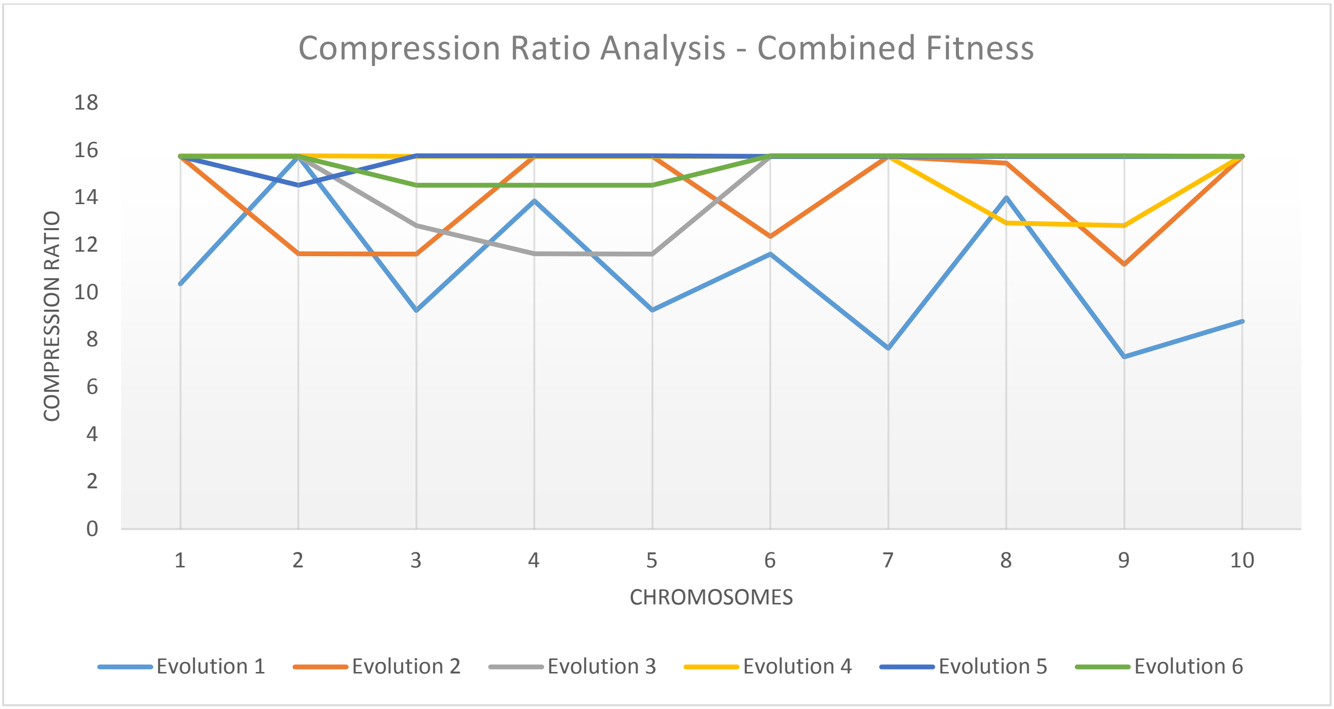


The trend is clear from the above graph that the lines smoothen out as they approach a higher compression ratio with every evolution. As was the case with the implicit fitness compression ratio analysis, it was clear that the evolution favoured the most regular pieces of music. Thus, as implicit fitness is used as part of hybrid fitness calculations, the fitness function will favour regularity; however, that regularity is limited by the thresholds that hybrid fitness uses to determine whether or not to ask the user to provide an explicit fitness rating. This can be seen on the graph as hybrid fitness tends to limit the lines on the graph to be kept below the upper compression ratio threshold of 20.

As this type of fitness source gets injected with human feedback, i.e. explicit fitness, when the compression ratio is deemed to be outside of the specified range, the explicit fitness values can have a major impact on the evolution as their values may change the direction of the evolution. In this experiment, the user was asked for an explicit fitness rating twice in the first iteration of evolution (results to be seen in “Evolution 2”), four times in the second iteration (results to be seen in “Evolution 3”) and three times in the third iteration (results to be seen in “Evolution 4”). There are some correlations to be seen on the graph above, as the compression ratio drops significantly in comparison to that of the other chromosome in that iteration in both Evolution 2 and Evolution 3. It is believed that this is due to the impact of the human feedback differing from the trend of the implicit feedback, i.e. if the implicit feedback favoured more regular chromosomes than the user, thus possibly causing the carrying forward of less regular chromosomes to the next iteration of evolution which caused the drops in compression ratio. In Evolution 4, there are no noticeable influences by the human involvement on the compression ratio. This may be due to the explicit fitness being similar to the trend of the implicit fitness values for the chromosomes in that iteration of evolution, i.e. the regularity of the chromosomes in that iteration of evolution were appealing to the user, so no drastic changes can be seen in the compression ratio.

Thus, from this evaluation of human involvement on the compression ratio, it can be shown that human involvement can cause a change in the compression ratio if the user prefers music with a different amount of regularity than is favoured by the implicit fitness values. In this case, the user seemed to prefer slightly less regularity in earlier iterations of evolution than the implicit fitness function, but was in agreement near the end of the evolution process.

- Combined Fitness:



Combined fitness uses both explicit and implicit fitness ratings to get an overall fitness value for each chromosome. It allows for weighting of explicit and implicit fitness ratings to bias a certain type of fitness over the other, if required. For this experiment, it was decided to give explicit fitness a higher weighting than implicit fitness. The reason for this is to help capture the user's subjective music preferences in the evolutionary process. By giving explicit fitness a higher weighting this increases the user's impact on the evolution, which is believed to be beneficial to evolve a piece of music that appeals to the user.

From the above graph, it is obvious that as the evolutions of iteration increase, the line to signify the compression ratios of the chromosomes at each iteration smoothens out as all chromosomes in the population approach a similar regularity. With each iteration the chromosomes approach a reasonably high compression ratio that is seemingly bounded around 16. As seen in the compression ratio analysis of implicit fitness, the chromosomes examined approached a maximum compression ratio of around 20. Thus, it is thought that the regularity of these chromosomes is being bounded slightly lower than this by the explicit fitness ratings. Due to the explicit user ratings having a higher weighting than the implicit fitness ratings, their impact on the overall fitness is greater. This means that if the explicit fitness for a chromosome was lower than the implicit fitness, it would bring down the overall fitness for that chromosomes. Thus, if this was the case throughout this experiment then this would account for the seeming bounding of the regularity of the music produced, as the user once again tended to favour music that was regular, but not too regular.

Chapter 7. CONCLUSIONS

Summary

As a whole, I believe this project was a success. My main goal was to create a system to evolve music that appealed to the user based on a well-defined and well-designed fitness function. I feel that I have fulfilled this goal to the best of my ability as this project allows the user to configure the fitness function to choose from a number of different fitness sources. As the user has a lot of freedom when choosing the configuration of the system, especially the fitness function, this allows the genetic algorithm to be used to their needs. Another key goal of this project was the analysis of the evolved music to try and arrive at some interesting hypotheses regarding music appeal. After carrying out a number of experiments and thorough analysis of each, I believe I fully accomplished this goal also.

Learned

Before beginning this project, I was completely unaware of the existence of evolutionary computation. As I began researching into the area I was intrigued by the proven performance of evolutionary computation, time and time again, in a seemingly endless amount of possible application areas. I specifically learned a lot about the power of genetic algorithms, becoming very familiar with them through the implementation of this project. Aside from evolutionary computation, the area of music aesthetics was not very familiar to me, although I have a passion for music. Through this project, my knowledge of the area has grown through the analysis of the experiments carried out.

Discovered

The main discovery of this project is the relationship between regularity in music and its appeal to users. Through the experimentation and analysis above, it is clear that regularity is a key to creating an appealing piece of music. The difficult part of the analysis was deciding how much regularity was the optimal amount to increase user appeal. As, music is very subjective, this varied slightly. However, as a whole, it became clear that too little regularity in music is unappealing to users, as reflected in the explicit fitness analysis. On the contrary, it was also clear that too much regularity was unappealing to users. Thus, the main discovery is that regularity is a good trait in music, but some spontaneity and irregularity is good to keep the user interested, as overly regular music reduces its appeal.

Future Work

There are a lot of possibilities for future work on this project. One of the main areas in which further work could be carried out is in the area of user interaction. Currently the project is not focused on user interaction, with only a simple GUI when interacting with the user when obtaining the explicit fitness ratings. I believe that there is potential for this project to be transformed into a more user-friendly application, including a high quality GUI, based upon the principles of human computer interaction (HCI), through which the user interacts with the application. This would give the project a more polished feel and I believe it would make the experience of using the application to evolve music more enjoyable as it would be easier to use.

Through the creation of a GUI in which the user could select the pieces of music to be used to seed the genetic algorithm, the user would get the sense that they are having a big impact on the evolutionary process right from the get-go. By having a number of pieces of music available to the user, each of which are playable in order for the user to make their decision, they could then simply click to select the piece/pieces of music they wish to see the evolution with. These playable MIDI files would be created from sample chromosomes available to the application, which, on selection, would be passed into the application in their chromosome form and added to the initial starting population ready for evolution.

Similarly, the application GUI could have the options available to the user to select which instrument they wish to use in their evolved music, with the possibility of previewing the instrument before selection.

The creation of a playback visualizer for when the MIDI files are getting played would be another area of possible development. I believe this would make the evolutionary process - more enjoyable through the simple visualisation of the piece of music being played, as the visualizer's graphics would be a direct mirroring of the beats being played. This visualisation would also be a way of representing the regularity of a piece of music graphically, as regularity would be very noticeable to the user due to having very uniform graphics appearing, as opposed to somewhat erratic graphics for more irregular music.

A more technical possibility for future work on this project is the creation of MIDI files with more than one track. As this implementation only uses one MIDI *Track* object, this leaves potential to explore the area of working with numerous tracks to create more advanced pieces of music. This would be an interesting area of development as it may lead to some other unknown hypotheses regarding the appeal of music to listeners. It would open up numerous other routes for analysis, such as taking into account the data from each track in the MIDI file and, essentially, help to try create a template for what it is that makes a piece of music appealing.

Conclusion

The task of evolving music through the use of evolutionary algorithms to generate appealing music, based on computational measures of music aesthetics, is not a straightforward task, simply due to the fact of human individuality. However, this project's success has led to interesting results regarding the very real possibilities in the area of music evolution. Through the analysis of the experiments carried out, this project has reinforced the existence of a link that lies between regularity in music and its appeal to users.

REFERENCES

[1][2] Mitchell, M. (1996). *An Introduction to Genetic Algorithms*. Cambridge, Mass.: MIT Press.
[3] *Genetic Algorithm*. In Wikipedia. Retrieved April 3, 2015, from http://en.wikipedia.org/wiki/Genetic_algorithm
[4] *Evolved Antenna*. In Wikipedia. Retrieved April 2, 2015, from http://en.wikipedia.org/wiki/Evolved_antenna
[5] Koza, John (1992). *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press.
[6] *Genetic Programming*. In Wikipedia. Retrieved April 3, 2015, from http://en.wikipedia.org/wiki/Genetic_programming
[7] *Evolutionary Music*. In Wikipedia. Retrieved April 4, 2015, from http://en.wikipedia.org/wiki/Evolutionary_music
[8] Melomics (2013, January 10). *One Minute Video*. Retrieved from https://www.youtube.com/watch?v=kmrw9him9Oc
[9] News Scientist (2012, 5 July). *Computer composer honours Turing's centenary*. Retrieved from http://www.newscientist.com/article/mg21528724.300-computer-composer-honours-turings-centenary.html#.VSAEX_nF8aw
[10] DarwinTunes. *DarwinTunes – Survival of the funkiest*. Retrieved April 4, 2015 from http://darwintunes.org/
[11] Ghassaie, Amanda. *What is MIDI?* Retrieved April 4, 2015 from http://www.instructables.com/id/What-is-MIDI/
[12] *MIDI*. In Wikipedia. Retrieved April 4, 2015, from http://en.wikipedia.org/wiki/MIDI
[13] Ross, Dave. *How MIDI Works?* Retrieved April 4, 2015 from http://entertainment.howstuffworks.com/midi1.htm
[14] Abyssmedia. *MIDI vs WAV.* Retrieved April 4, 2015, from http://www.abyssmedia.com/midirenderer/midi-vs-wav.shtml
[15] Java Genetic Algorithms Package. *What is JGAP?* Retrieved April 5, 2015 from http://jgap.sourceforge.net/
[16] JGAP – Javadoc 3.6. *Class MutationOperator*. Retrieved April 7, 2015, from http://jgap.sourceforge.net/javadoc/3.6/org/jgap/impl/MutationOperator.html
[17] JGAP – Javadoc 3.6. *Class CrossoverOperator*. Retrieved April 7, 2015, from http://jgap.sourceforge.net/javadoc/3.6/org/jgap/impl/CrossoverOperator.html
[18] JGAP – Javadoc 3.6. *Interface IUniversalRateCalculator*. Retrieved April 7, 2015, from http://jgap.sourceforge.net/javadoc/3.6/org/jgap/IUniversalRateCalculator.html
[19] JGAP – Javadoc 3.6. *Class Configuration*. Retrieved April 8, 2015, from http://jgap.sourceforge.net/javadoc/3.6/org/jgap/Configuration.html
[20] Java Genetic Algorithms Package. *Getting Started With JGAP*. Retrieved April 8, 2015, from http://jgap.sourceforge.net/doc/tutorial.html
[21] Oracle. *Compressing and Decompressing Data Using Java APIs.* Retrieved April 8, 2015, from http://www.oracle.com/technetwork/articles/java/compress-1565076.html

[22] JGAP – Javadoc 3.6. *Class IntegerGene*. Retrieved April 8, 2015, from http://jgap.sourceforge.net/javadoc/3.6/org/jgap/impl/IntegerGene.html
[23] Oracle. *Overview of the MIDI Package.* Retrieved April 9, 2015, from http://docs.oracle.com/javase/tutorial/sound/overview-MIDI.html
[24] Oracle. *Package javax.sound.midi.* Retrieved April 9, 2015, from http://docs.oracle.com/javase/7/docs/api/javax/sound/midi/package-summary.html
[25] Brown, Karl. *How to write a java program that writes a MIDI file.* Retrieved April 9, 2015 from http://www.automatic-pilot.com/midifile.html
[26] Tsagklis, Ilias. *Play Midi audio.* Retrieved April 9, 2015 from http://examples.javacodegeeks.com/desktop-java/sound/play-midi-audio/
[27] Richard, Stéphane. *MIDI Programming - A Complete Study*
Part 1 - MIDI File Basics. Retrieved April 9, 2015 from http://www.petesqbsite.com/sections/express/issue18/midifilespart1.html
[28] Recording Blogs. *MIDI Set Tempo meta message.* Retrieved April 9, 2015 from http://www.recordingblogs.com/sa/tabid/88/Default.aspx?topic=MIDI+Set+Tempo+meta+message
[29] Vandenneucker, Dominique. *MIDI Tutorial.* Retrieved April 9, 2015 from http://www.cs.cmu.edu/~music/cmsip/readings/MIDI%20tutorial%20for%20programmers.html
[30] MIDI.org. *MIDI Message Table 1.* Retrieved April 9, 2015 from http://www.midi.org/techspecs/midimessages.php