Synthetic Instrument Generation: Generating novel instrument timbres with a genetic algorithm

By
John Knowlton

A THESIS

submitted to

Oregon State University

Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Scholar)

Presented December 8, 2023
Commencement December 2023

AN ABSTRACT OF THE THESIS OF

John Knowlton for the degree of Honors Baccalaureate of Science in Computer Science presented on December 8, 2023. Title: Synthetic Instrument Generation: Generating novel instrument timbres with a genetic algorithm

Abstract approved:

Patrick Donnelly

The use of genetic algorithms to compose music and generate sounds is an area of interest in the artificial intelligence field. Music and instrument sounds have known rules and structures that can be followed which make them well-suited for genetic algorithms. However, genetic algorithms still struggle to generate sounds comparable to human-designed instruments largely due to the subjective nature of music and sounds. Genetic algorithms also struggle to produce a unique variety of sounds due to issues like premature convergence where the sounds become more similar as the algorithm runs. To address these issues, this project combines sound synthesis, genetic algorithms, crowdsourcing and the population model to generate more complex and enjoyable sounds. In our approach, sounds are synthesized by mixing randomly generated simple sine waves using additive synthesis. The genetic algorithm applies mathematical fitness rules and human evaluation through crowdsourcing to evaluate the quality of the generated sounds. Crowdsourcing evaluators conduct A−B testing through a web application to pick the sounds they prefer which guides the algorithm to produce more complex and interesting sounds. The population model creates subpopulations of sounds that develop separately and periodically trade sounds which encourages a greater variety of sounds and introduces greater genetic diversity each time subpopulations trade sounds. These components work together to take random noise and transform it into sounds that mimic real instruments or sound interesting enough that they could be utilized as an instrument by programming it as a mode to an electronic keyboard.

Key Words: Genetic Algorithm, Crowdsourcing, Population Model

Corresponding e-mail address: knowltjo@oregonstate.edu

©Copyright by John Knowlton
December 8, 2023

Synthetic Instrument Generation: Generating novel instrument timbres with a genetic algorithm

By
John Knowlton

A THESIS

submitted to

Oregon State University

Honors College

in partial fulfillment of
the requirements for the
degree of

Honors Baccalaureate of Science in Computer Science
(Honors Scholar)

Presented December 8, 2023
Commencement December 2023

Honors Baccalaureate of Science in Computer Science project of John Knowlton presented on December 8, 2023.

APPROVED:

Patrick Donnelly, Mentor, representing Computer Science

Justin Preece, Committee Member, representing Agricultural Science

---

Prasad Tadepalli, Committee Member, representing Computer Science

---

Toni Doolen, Dean, Oregon State University Honors College

I understand that my project will become part of the permanent collection of Oregon State University Honors College. My signature below authorizes release of my project to any reader upon request.

John Knowlton, Author

Contents

1 Introduction **3**

2 Background **5**
- 2.1 Genetic Algorithms . . . 5
  - 2.1.1 High Level Overview . . . 5
  - 2.1.2 AI vs Machine Learning . . . 10
  - 2.1.3 Population Model . . . 10
  - 2.1.4 Crowdsourcing Techniques . . . 11
- 2.2 Music and Sound Principles . . . 12
  - 2.2.1 Attributes of a Sound Wave . . . 13
  - 2.2.2 ADSR Envelopes . . . 14
  - 2.2.3 Additive Synthesis . . . 16

3 Related Works **19**
- 3.1 Literature Review on the History of Genetic Algorithms . . . 19
- 3.2 Evolutionary Music Composition Approaches . . . 21
- 3.3 Genetic Algorithms Variants . . . 23
- 3.4 Sound Synthesis . . . 25
- 3.5 Summary . . . 28

4 Methods **31**
- 4.1 Representation . . . 31
- 4.2 Premature Convergence and Diversity . . . 32
- 4.3 Genetic Algorithm Operators . . . 33
  - 4.3.1 Evaluation . . . 33
  - 4.3.2 Selection . . . 40
  - 4.3.3 Crossover . . . 43
  - 4.3.4 Mutation . . . 44
  - 4.3.5 Computation . . . 45
- 4.4 Hybrid GA Implementation . . . 46
  - 4.4.1 Tuning parameters and weights . . . 46
  - 4.4.2 Population Model . . . 46
  - 4.4.3 Crowdsourcing Fitness Evaluation . . . 47
  - 4.4.4 Database and Backend Infrastrucutre . . . 48
- 4.5 Validating and Verifying Results . . . 49

5 Exemplar Walkthrough **52**
- 5.1 Initialization . . . 52
- 5.2 Evaluation . . . 53
- 5.3 Selection . . . 54
- 5.4 Crossover . . . 54

5.5 Mutation 55
5.6 Population Model 55
5.7 Stopping the Genetic Algorithm 56

6 Discussion **58**
6.1 Contributions 58
6.2 Limitations 59
6.3 Future work 60
6.4 Applications 61
6.5 Conclusion 62

1 Introduction

Music is a uniquely human experience. Music is an important piece of culture across all different time periods. Humans naturally create and enjoy music, so as technology rapidly developed in recent years, more tools to create music also developed. The latest technology that is on everyone's mind in this area is artificial intelligence (AI) [43]. The recent breakthroughs in AI beg the question of whether it is possible to teach computers to understand something as uniquely human as music.

Although an appreciation and understanding of music is inherently human, composing and creating music is still a challenging task that requires many years of practice and training. Modern computers provide a lot of conveniences that make this process easier than ever before, but artificial intelligence can provide even more power [3]. As long as humanity collectively agrees to still support human artists, artificial intelligence with music composition will primarily serve as another tool to aid them in their creations. Applications of AI range from helping composers flesh out a musical idea to envisioning new instruments for them to employ.

The goal of this project is to imagine and synthesize new instruments for musicians. Notably, the goal is not to recreate existing instruments but instead search the set of all possible instruments timbres that exist. This space is known as the sound space. To achieve this, we combine several existing algorithmic and AI techniques to create a hybrid algorithm designed to find new instrument timbres.

The primary techniques in the hybrid algorithm are a genetic algorithm (GA), crowdsourcing and the population model. All three techniques are combined to create randomly generated noise and synthesize them into new instruments.

A genetic algorithm is an evolutionary-based algorithm inspired by Darwinian theories of evolution, natural selection and the survival of the fittest [24]. Genetic algorithms are often applied towards optimization problems by creating a "population" of solutions and simulating natural selection by selecting the "fittest" solutions to "reproduce" and create improved solutions in a new generation of solutions [23]. In this case, we want to thoroughly explore the sound space and task the GA to make a wide variety of interesting sounding instruments. What is and isn't considered interesting is extremely subjective, which makes this problem all the more difficult to tackle. However, our approach takes advantage of the fact that instrument sounds are highly structured and have many patterns to create heuristics that determine which solutions are the fittest to narrow down the infinitely large sound space.

We also utilize subjective evaluation through crowdsourcing so humans rate the instrument sounds. Crowdsourcing assigns a large number of people with simple tasks

to quickly collect a large amount of data and input [52]. This project crowdsources to collect a large number of human evaluations on the generated instrument sounds. Human evaluation is more complex and insightful than mathematical heuristics and is more fitting for the subjective problem at hand. Human evaluation contributes to a more thorough exploration of the sound space thanks to the additional insight.

The population model also contributes to an intensive exploration of the sound space. A genetic algorithm normally creates a single population of solutions that all compete with each other [23]. Storing the solutions in a single pool quickly results in premature convergence which is when the solutions of newer generations become homogenous. Premature convergence restricts how thoroughly the GA explores the sound space. The population model mitigates premature convergence by creating more genetic diversity. Instead of a single population, the population model stores several smaller subpopulations to allow smaller groups to develop independently. To introduce more genetic diversity, these subpopulations periodically trade solutions [41]. This system creates a greater variety of solutions and introduces more genetic diversity to slow down premature convergence.

In this project, we create a system that enables the exploration of the sound space. Our approach set the foundation for a hybrid algorithm that combines a genetic algorithm, crowdsourcing and the population model to synthesize instrument sounds. We prove the feasibility of the approach and provide the groundwork for future experiments and exploration. Future exploration could create a large-scale experiment to leverage numerous internet users to collect a large volume of evaluations to encourage a wider variety of unique sounds. Large-scale experiments could generate great quantities of unique instrument sounds that could be stored on a website that serves as a library of downloadable sounds for all users. A large collection of sounds like this could expand sound design in movies and video games or inspire composers to create new pieces and genres thanks to the new tools from the algorithm at their disposal.

2 Background

This project is multi-disciplined with many complex layers. This section provides context for the most relevant fields of research involved in this project. The information covered in this section can be broadly split into genetic algorithms (Section 2.1) and sound synthesis (Section 2.2) and both these sections have several subsections that cover specific chunks of information relevant to those topics.

2.1 Genetic Algorithms

The principal component of this project is a genetic algorithm (GA). Genetic algorithms have the ability to analyze and manipulate large amounts of data. Given the complexity of the approach, there is a lot of prerequisite knowledge to understand. To gain a better understanding, we review a high-level overview to break down the main stages of genetic algorithms as well as some of the ideas that inspired them.

2.1.1 High Level Overview

Genetic algorithms are a technique in artificial intelligence that is primarily inspired by ideas from Darwinian evolution [36]. At a broad level, they attempt to mimic the "survival of the fittest" process popularized by Darwin. This theory states that animals mutate over time and better adapt to their environment to increase their chance of survival. Genetic algorithms apply this framework to a variety of problems by simulating a population of "animals" or candidate solutions that reproduce and mutate over time to create solutions that are better suited to their "environment" or the specific problem of interest. The most common use cases for genetic algorithms are optimization problems with a single answer. They are well suited for these types of problems because of how they tend to quickly find a reasonable solution, but, by design, they are not well-suited to more open-ended problems [15].

Darwin's theories provided a high-level conceptual groundwork. The other components of the algorithm were concerned with how to represent Darwin's ideas programmatically and mathematically. Regardless of the type of problem, genetic algorithms often employ lists of numbers or letters to represent their population of "animals" or candidate solutions [17]. Sometimes, this representation is extremely straightforward, like in this case where the programmer is interested in solving a math equation. These numbers undergo several stages that analyze and find the best solutions. The general flow of these stages of a genetic algorithm is shown in Figure 1.

Figure 1 is a flowchart of the genetic algorithm. We review these steps in greater detail in the 4.3. It is important to note that the local and global loop quotas in the figure are simply parameters set before running the algorithm. These parameters can

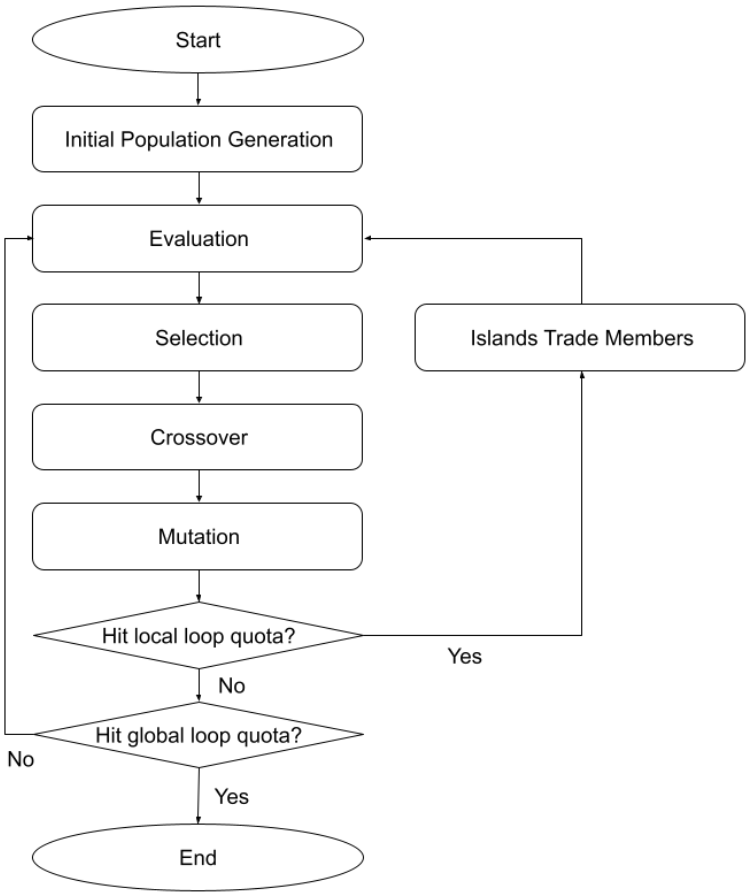


Figure 1: A broad overview of genetic algorithm in the form of a flowchart

vary between different runs of the GA to vary the solutions.

A genetic algorithm generates successive generations until a stopping criteria defined by the user is met. The most common stopping criteria is stopping after a fixed number of generations have been created by the GA, but the criteria can be whatever the user defines. Each generation undergoes four distinct stages [16].

1. **Evaluation**: Scores each solution on certain criteria
2. **Selection**: Pick solutions based on their fitness scores
3. **Crossover**: Combines the data of the selected solutions to create a new generation
4. **Mutation**: Randomly changes a small portion of one solution or multiple solutions' data

This process repeats indefinitely or until the genetic algorithm meets its stopping criteria. These core stages form a typical genetic algorithm which is the primary algorithm used in this project. Now that the basic stages of the algorithm have been introduced, we next discuss some lower-level details.

As mentioned earlier, a genetic algorithm tries to broadly simulate real-world evolution. It takes many of the concepts Darwin is famous for and attempts to apply them to solve a problem algorithmically. A large number of potential solutions are randomly generated, are stored collectively as a population, and are transformed by the four stages of the algorithm: Evaluation, Selection, Crossover and Mutation [19].

Evaluation

The Evaluation stage assigns scores to each candidate solution. This idea comes from the concept of the survival of the fittest in which the animals that exhibit the characteristics best suited to their environment have the best chance of survival. A better chance of survival means they also have the best chance of having offspring and propagating their genetics. With a GA, these advantageous characteristics are measured by a collection of functions called **fitness functions** [26]. We will cover the specific fitness functions built for this project in the 4 section, but a basic example of a fitness function for our task would be a function that rewards louder sounds that have a volume above a certain threshold.

Not every attribute a fitness function rewards or punishes will have the same level of importance, so to compensate, each fitness function can be assigned different weights relative to how important it is in the evaluation step as a whole. Fitness functions can be split into two categories: mathematical evaluation and human evaluation [19]. Mathematical evaluation searches for the desirable patterns common to many acoustic instruments while human evaluation collects ratings from human volunteers.

Selection

The Selection stage picks solutions based on their fitness score to use as parents who reproduce to populate the next generation. Picking the highest-scoring individuals without exception may initially sound like the optimal strategy, but there are other useful options. One such example is Roulette selection [35]. Instead of simulating a bracket, it simulates a giant roulette wheel where each solution has a portion of the wheel corresponding to them. These sections are not equal in size. Instead, they are proportional to their fitness scores. The wheel is "spun" several times until enough solutions have been chosen, such that solutions with lower scores still have a chance to be picked if they get especially lucky. Figure 2 illustrates how the wheel is divided up.

Another approach is a tournament style, in which pairs of solutions are put into a bracket and repeatedly compared [24]. The best of the two moves up the bracket until

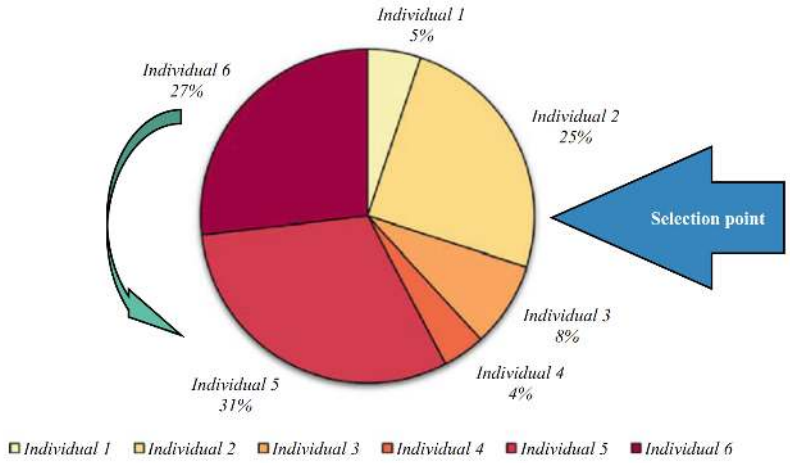


Figure 2: An example of Roulette Selection, reproduced from [35]

the remaining number drops to a desired value of solutions. Figure 3 illustrates the bracket format of tournament selection and depicts the first round of a hypothetical run.

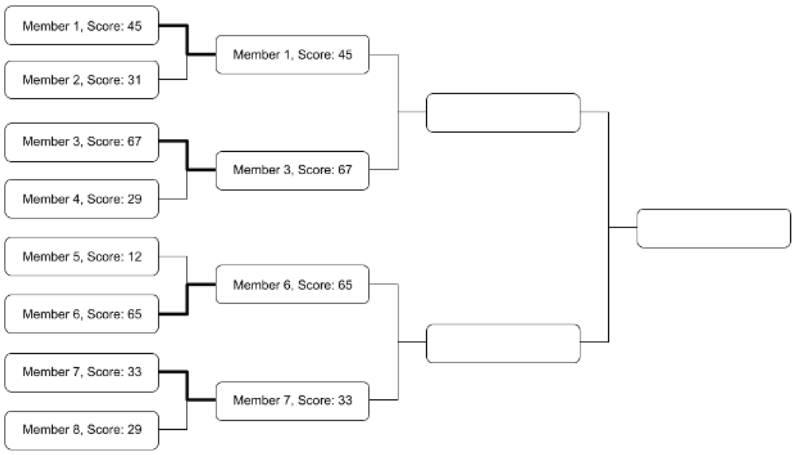


Figure 3: An example demonstrating the first round of a tournament selection using eight solutions

Notice how the solutions that made it to the next round were not the highest-scoring solution. In fact, solution 7 has a mediocre score comparatively, but because of some bracket luck, it was still selected despite several other solutions scoring higher. This encourages more diversity among solutions. Diversity is encouraged because we want to robustly explore the sound space in order to create as many unique sounds as possible. Genetic algorithms converge, which causes the population solutions more

similar over time, therefore there need to be active attempts to maintain a level of diversity in order to produce a better variety of sounds [26]. The solutions selected through this stage are then used as parents in the next step, crossover, to create the next generation.

Crossover

Crossover involves taking the parents and mixing their data or "genetics" to create new "offspring". Figure 4 illustrates two crossover techniques.

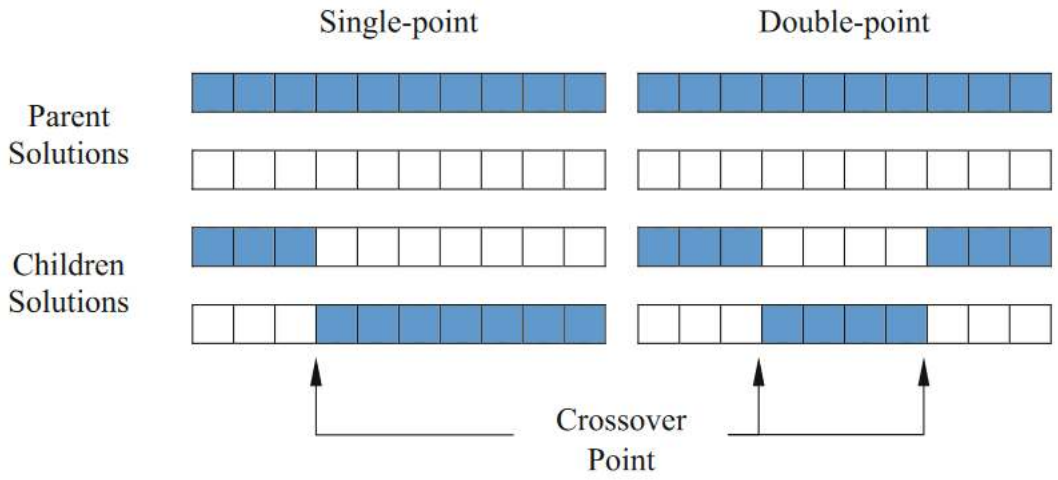


Figure 4: Crossover between two parent solutions to create two children solutions [35]

This figure reprinted from [35] is a visualization of how crossover takes data from two parent solutions and splits them between two new solutions to create offspring. In particular, this figure demonstrates single-point crossover on the left side and double-point crossover on the right side. Single-point crossover is when the parents divide their data or "genes" up by a single point in their array [25]. In this particular example, the single point of crossover occurs between the 3rd and 4th items in the array. Double-point crossover is very similar except it creates two points of division. The example's dividing points are between the 3rd and 4th as well as the 7th and 8th items of the array. There exist even more complicated strategies that involve randomization, but single-point and double-point are common and they adequately demonstrate the basic idea.

Mutation

Finally, the Mutation stage randomly changes a portion of a candidate solution's data. The amount of change that occurs in each mutation can vary. The chance of mutation is randomized, the amount the mutation function changes the values randomized, the number of solutions mutated is randomized, and so on. Regardless of

what is being randomized, mutation's goal is to modify solutions in some form. Otherwise, there would be no new genetic information in the population introduced since all the other methods simply pick from past solutions and build new solutions from those past solutions instead of creating completely original solutions [8]. Like the other operators, there are many different techniques that produce different results, but those will be explored in the 4 section.

These four stages are repeated over and over for each new generation for however long the algorithm runs, and the final generation will be the algorithm's best estimation for the solution to the problem.

2.1.2 AI vs Machine Learning

An important distinction to clarfiy for this project is the difference between artificial intelligence and machine learning. Artificial intelligence is a broad field, but it can be generally defined as the process of designing intelligent machines to mimic human intelligence. Genetic algorithms are one of many artificial intelligence techniques available [8]. Machine learning on the other hand is a branch of AI that designs machines to mimic a human's ability to learn. Note that this project does not involve machine learning because machine learning algorithms require large amounts of labeled or unlabeled training data [11].

There are machine learning algorithms that wield genetic algorithms, but genetic algorithms are not inherently a machine learning technique. One way to distinguish the two is that with AI, the focus is more on mimicking human intelligence so machines can accomplish complex tasks while with machine learning, the focus is more on a machine mimicking a human's ability to learn and pick up on patterns [37]. We will discuss further Machine learning applications in the 3 section.

2.1.3 Population Model

There are numerous extended techniques that can be implemented into a genetic algorithm to add more variation and adapt the genetic algorithm to different problems. The two that we discuss here are the population model and crowdsourcing. First, we will focus on the population model.

The population model is an expansion of the basic genetic algorithm that aims to achieve a higher level of diversity across the numerous solutions [41]. With a typical GA, all the solutions exist in one large population pool. In the population model, the GA creates and isolates several subpopulations so these smaller populations can develop and run separately. The one exception to this isolation is when these islands periodically share solutions to enable the infusion of new genetics. A small number of solutions, sometimes even as few as one, are removed from each population and

then sent to a different island. This strategy encourages a greater variety of solutions to develop because each island develops independently as a distant population that differs from the other islands.

Each population acts as its own island that is isolated from the other population the majority of the time. Each island has its own rules or weights for the various fitness functions to encourage different types of solutions. Storing solutions in one gigantic pool of that all follow the same rules contributes to solutions becoming very similar. The population model is inspired by Darwin's discovery of different birds on the various Galapagos islands and how the different environments and food sources on each island caused their beaks to differ significantly. For the most part, the different birds stay on their own islands and develop separately, but occasionally a few of the birds will fly to different islands and then contribute to the genetic pool of the bird population on this new island. The population model can also be thought of as the solutions acting like early Polynesians, occasionally traveling to distant islands across the ocean and introducing their own genetics into the population of wherever they happen to land.

This model prevented the genetic algorithm from converging on a solution too quickly, which is especially important for this project [26]. Genetic algorithms converge quickly, which means they make the candidate solutions more similar over time. Convergence is especially problematic for this project because the goal is to widely explore the very subjective sound space. If the candidate solutions become similar too quickly, that severely limits the results the algorithm produces and never explores many other viable solutions. This is why the population model is a critical aspect of our design [41]. The population model forces variety by maintaining many subpopulations that develop independently and differently.

2.1.4 Crowdsourcing Techniques

Crowdsourcing is the process of getting a large number of people to share knowledge or perform a certain task [13]. A common example of crowdsourcing is the Waze navigation app[1]. Waze users can report incidents like accidents and police vehicles in specific spots so that other users can be aware of those things. Other examples include Amazon Mechanical Turk[2], which outsources tasks that range from surveys to identifying content in videos and images, and Open Food Facts, a free online database of food products.

Crowdsourcing is a broad topic so there are several sub-categories of crowdsourcing. The subcategory we utilize in this project is called human-in-the-loop crowd-

[1]https://www.waze.com/live-map/
[2]https://www.mturk.com/

sourcing. Normal crowdsourcing can be done offline or separately from the main application while human-in-the-loop crowdsourcing requires human input to be able to run [6]. Waze would not be considered human-in-the-loop crowdsourcing because it still functions as a navigation app even if users do not report accidents or police officers. Our approach is a human-in-the-loop system because we rely on human evaluation as part of the Evaluation stage.

Many AI applications use crowdsourcing to collect data and human evaluations because of how much human input improves data and decisions used in training the algorithms. Both machine learning and artificial intelligence attempt to achieve equal or better judgment than humans. To teach and fine-tune these programs, humans are often used to refine and evaluate their performance so they can approach the same level of judgment.

Crowdsourcing is particularly useful because of how it involves a large volume of people for this evaluation process to happen quickly while simultaneously soliciting a diversity of subjective opinions [51]. Some may argue it would be better to operate a small group of experts to evaluate or contribute to the evaluation to ensure the insight provided is valuable and augments the algorithm instead of muddying it. While it is true that crowdsourcing potentially runs a higher risk of getting incorrect evaluations from unmotivated or mischievous users, this risk is often worth the efficiency that crowdsourcing has over using a small dedicated team. A debilitating bottleneck in this process occurs with only a small number of evaluators and can severely interfere with the human-in-the-loop system's ability to produce meaningful results. Additionally, incorrect entries can be somewhat circumvented by employing rewards for completing evaluations, usually in the form of monetary rewards, as with Amazon's Mechanical Turk.

Ultimately, crowdsourcing can provide additional insight by using human judgment in the evaluation of candidate solutions while avoiding significant slowing down in the Human in Loop system [22]. This project uses crowdsourcing with the hopes of generating a greater range of interesting sounds buoyed by human feedback. The mathematical fitness functions that also pick sounds are much more rigid. These functions attempt to pick up subjective elements that make a sound interesting or enjoyable, but it is far easier for a human to pick up on these elements. A human's evaluation is more nuanced compared to the fitness functions, which makes subjective human evaluation very beneficial for this project.

2.2 Music and Sound Principles

This project uses sounds as candidate solutions for the genetic algorithm. To better understand how the GA evaluates these sounds, this section discusses the various

musical ideas and principles that concern this project.

2.2.1 Attributes of a Sound Wave

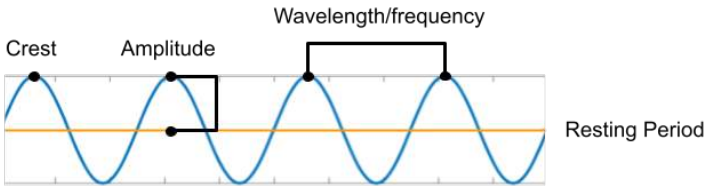


Figure 5: The crest, amplitude and frequency of a sound wave visualized

Figure 5 is a visualization of a simple sound wave and some of its basic attributes. It indicates the crest, amplitude and wavelength of the soundwave.

Broadly speaking, the visualization of a sound wave shows a pattern of up and down waves. The highest part of the wave is called the *crest* [10]. A *cycle* or *wavelength* of a wave describes the distance between two crests. The *period* of a wave is the amount of time it takes for one cycle to complete. *Amplitude* determines the volume of a sound wave. The higher the amplitude the louder the volume. When visualized, amplitude relates to the height of the waves.

Frequency measures the time it takes for one crest to travel back down and then back up to the next crest [1]. It is measured with Hertz. It has an inverse relation with the period of a wave meaning that if the frequency doubled, the period would be halved. Wavelength is also inversely related to frequency. Instead of being a measure of time, wavelength is a measure of distance. Specifically, it is the distance between two crests. The longer the distance, the more time it takes for the crest to come up again. Lower frequency signals have longer waveforms. The frequency also relates to the psychoacoustic concept of pitch. The higher the frequency, the higher the pitch.

Not everything is immediately obvious through visualizing waveforms. A clear example is the timbre of a sound. Timbre can be broadly defined as the quality of sound that makes musical instruments sound different from each other [21]. The timbre of a sound is hard to define specifically since it involves perceptual and objective attributes, so there are multiple valid definitions. Another definition for Timbre is a set of auditory attributes that combine aspects like pitch, volume and duration. The

timbre of an instrument is often used to describe the quality an instrument has as a whole. In reality, each note of an instrument has a distinct timbre, meaning an instrument has a collection of timbres rather than a single, all-encompassing timbre [28]. For example, a low note on a clarinet is typically characterized as dark and rich sounding while a higher note is typically described as bright.

However, there are also certain acoustic traits that are common across an instrument's range. A clarinet across all notes is typically described as reedy, which makes sense considering it is a reed instrument. The intricacies and contradictions surrounding timbre are a large part of why the term is vaguely defined. While many terms can describe timbre, the more common terms relate to perceptual properties like auditory brightness, richness or hollowness. [32]. Timbre is especially important in this project because it contributes a lot to a sound's distinctiveness[39]. For this reason, finding lots of unique timbres is closely linked to our goal of finding a unique variety of instrument sounds.

Most people perceive a note played by an instrument as a single sound. In reality, that single note can be analyzed by smaller components, each with its own frequency and amplitude values. The process of analyzing a sound as smaller components is called Fourier analysis [14]. These smaller components are called *harmonics*. Harmonics are integer multiples of a fundamental frequency. This fundamental frequency often has a high amplitude and is what we perceive as the pitch. The fundamental frequency and the harmonics contribute to the perception of a more rounded, or harmonic, sound.

Since harmonics are direct multiples, harmonics are described by which multiple they are of the fundamental frequency. If a harmonic was twice the frequency of the fundamental frequency, it would be the second harmonic. We perceive this as a musical octave higher [4]. Most instruments are comprised of a series of harmonics with some inharmonic exceptions like a drum that sounds much sharper and snappier. Harmonics often create a rounded, pleasant sound, so it is a useful attribute to manipulate to make the candidate solutions sound more like human-designed instruments.

2.2.2 ADSR Envelopes

The ADSR envelope stands for attack, decay, sustain and release envelope. These are the four components the envelope uses to determine how a sound's amplitude changes over time [49]. Technically, you could have a sound with constant amplitude over time but that is usually not how most complex sounds behave, so ADSR envelopes are used to create and describe more complex sounds.

- The **attack** influences how quickly a sound goes from nonexistent to the peak

of its volume at the start.

- The **decay** influences how long it takes to drop from the peak attack volume to the quieter sustain volume.
- The **sustain** affects how long the volume remains at that volume before it starts dropping
- The **release** influences how long it takes for the sound to sink from the sustain volume to nonexistent

The way these values are represented is not identical. For attack, decay and release, the value describes the amount of time those stages will last. On the other hand, the sustain value must be from 0 to 1 because it represents the percentage of the attack amplitude the sound will drop and sustain [27]. Figure 6 demonstrates how each stage changes the amplitude of a sound over time.

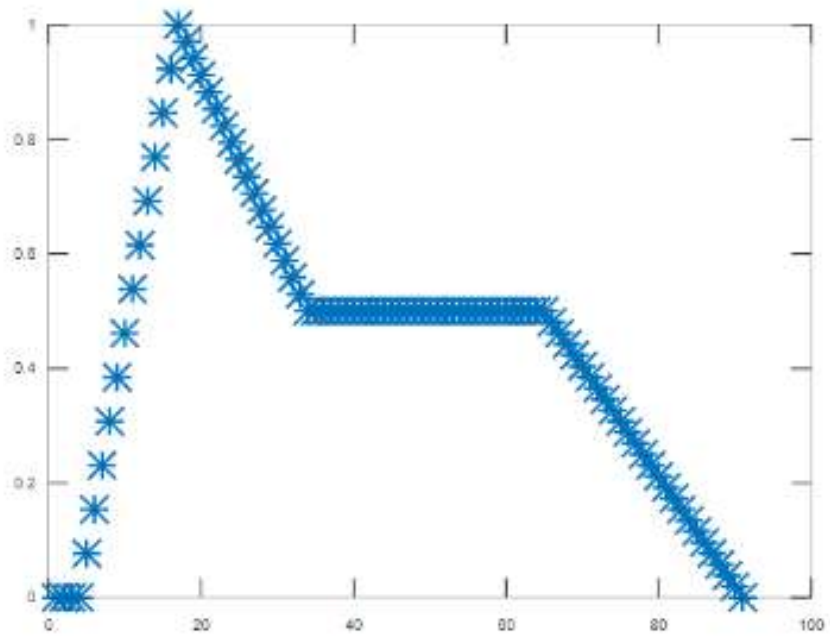


Figure 6: The four main components of an ADSR Envelope. The initial rise is the attack, the following drop is the decay, the flat section is the sustain and then the subsequent drop is the release [44].

The ADSR envelope influences the overall length of certain phases of a sound. By shortening or lengthening the time it takes to increase volume, lower volume or sustain volume, the sound takes on different qualities such as sounding explosive,

plucky or gentle [46]. For example, a short attack and release value create a very quick sound, making it sound like a staccato while a short attack and long release tend to create more percussive sounds like a marimba or cymbal. These are just a few examples of how the ADSR values dramatically affect a sound's timbre. They add a layer of complexity that cannot be achieved by adjusting the frequency and amplitude alone, especially since each harmonic of a sound has its own unique ADSR envelope [40]. The added depth makes ADSR envelopes a valuable aspect that this project uses to create more interesting sonic results.

2.2.3 Additive Synthesis

Additive synthesis is the process of mixing sine waves together to create a more complex waveform. Organs are unique in that they employ a form of additive synthesis [9]. With a pipe organ in particular, each pipe corresponds to a particular harmonic depending on the pipe's length and multiple pipes can be played in combination to create a new sound. Additive synthesis is reliant on the Fourier series which suggests that any audio signal can be broken down into a series of sine waves with different harmonics and amplitudes[7]. The harmonics relate to the frequency of the sound while the amplitude relates to the volume of the sound. The sine waves are repeating waves that individually are very simple. In fact, sine waves are the only waveform in which all of the energy of the wave is focused at a single frequency value, making them well suited to being stacked to create more complex sounds [1]. Additionally, they are very easy to measure and understand, making them well-suited to sound synthesis in general.

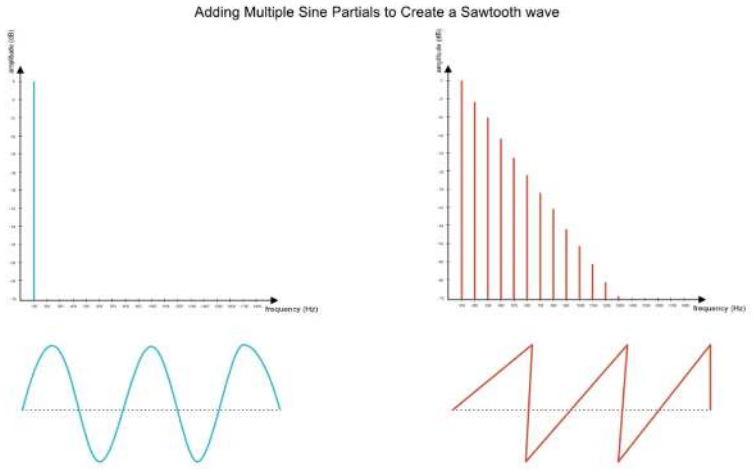


Figure 7: A sine wave made of one partial and a sawtooth waveform made with 13 partials

Figure 7 is from the perfectcircuit [3] website and illustrates two examples of how sine waves are combined into more complex waveforms. It may not be immediately obvious, but the sawtooth waveform is formed from the 13 sine partials. Adjusting the volume or changing the frequency of any of the partials would change the shape of the waveform.

There are many other ways of synthesizing sounds, but additive synthesis is one of the more straightforward approaches. This makes it well suited to manipulation and evaluation by our GA approach and also makes the process of generating sounds quick, making it ideal for this project because of the large volume of sounds the genetic algorithm generates each run [49]. In most implementations, the additive synthesis has envelope control over the volume and pitch of each sine wave or it has control over groups, such as every odd or even harmonic.

The field of sound synthesis has grown and developed over the decades. Initially, the field was focused on simulating the acoustics of rooms but it eventually expanded thanks to advancements in other fields such as interactive computer graphics, parallelism on commodity GPUs, and physics-based simulation [44]. Visualizing sound has played an important role in analyzing the sounds produced by the genetic algorithm and identifying what rules should be changed, removed or added to produce more different and complex sounds. Harmonic synthesis is very similar to additive synthesis [29]. Both are fast and effective at mimicking musical instruments. There is also the distinction between procedural-based methods, that uses signal processing which is fast and easy to implement but requires manual control over parameters, and physics-based methods, which uses auditory physics-based equations that can produce high-quality sounds but usually involve more computational overhead.

There are a number of tools that streamline the process of sound synthesis. Within the programming language Python, the `Pyo` library allows users to easily play and record generated sounds by feeding in values related to a sound's frequency partials, amplitude partials and ADSR envelope partials. Similar tools exist in other languages, such as `Tone.js` for JavaScript. This library provides similar functionality and is especially useful since it can be easily implemented with website programming, a huge advantage when techniques like crowdsourcing are implemented alongside the genetic algorithm.

Visualization is a useful method to analyze a sound into more elementary components. It is extremely useful for verifying the sound synthesis process and can also make analyzing the sounds much easier by displaying information in a way that is hard to perceive when only listening to a sound [9]. We used the Python libraries `NumPy`, `Matplotlib` and `Librosa` were to facilitate visualization. `Numpy` is designed

[3] https://www.perfectcircuit.com/signal/what-is-additive-synthesis

for storing and performing operations on the data, but it is also useful for sound synthesis and visualization. `Matplotlib` is primarily used for plotting graphs and information while `Librosa` is primarily used for music and audio analysis. We used these three libraries to visualize the sounds generated from the genetic algorithm to inform appropriate adjustments and improvements to the algorithm.

3 Related Works

This project combines several approaches in a way that, to our knowledge, has not been previously explored. However, there have been related studies that focus on some of the individual components are used in this project. In this section, we review some of the most relevant novel research. This section will provide additional background by briefly reviewing closely related studies.

3.1 Literature Review on the History of Genetic Algorithms

In this section, we briefly review the relevant literature on genetic algorithms and their applications.

In "Artificial intelligence in organised sound", Miranda and Williams compare old and modern artificial intelligence techniques [34]. The first attempts at applying artificial intelligence to music composition began in the early 2000s and used a symbolic approach that used a hard-coded set of rules to determine the behavior of a machine. These attempts were technically successful in their goals, but most human listeners could quickly tell that the compositions were not human-made. Additional attempts used symbolic AI to mimic musical styles instead of completely generating them from scratch. This includes numerous biologically appraoches as well. Besides the GA, these also include neural networks and other evolutionary approaches. These approaches were more flexible and can adjust their set of rules, potentially without repeated human intervention. Although these approaches are powerful, the authors still run into the issue of being perceived as computer-generated instead of being composed by a human.

Katoch et al. also explored older and modern genetic algorithm implementations, but instead of focusing on the differences, the authors explore the key components of genetic algorithms that stayed consistent as well as the common problems these algorithims face [20]. Genetic algorithms are metaheuristic algorithms that have been used for economics, engineering politics and several other fields. Metaheuristic algorithms solve optimization problems using metaheuristics which are procedures that find optimal heuristics or rules designed to solve a problem. Additionally, the authors have often been used to tackle NP-hard problems and have found success.

Metaheuristic algorithms can be broadly split up into single-solution algorithms and population-based algorithms. Single solution algorithms focus on a single candidate and improve it using local search while population-based algorithms use multiple candidates when searching to maintain more diversity. Genetic algorithms fall under the population-based category. Like most metaheuristic algorithms, GAs are primarily concerned with balancing intensification and diversification. Intensification relates

to thoroughly searching for a specific subset of solutions of a search problem while diversification deals more with finding other subsets of solutions within the search problem.

Population-based GAs are often better at performing global searches, but there are also plenty of issues that GAs struggle with which requires additional attention [26]. One example is the problem of encoding. In many cases, some alterations need to be made to the problem so that it can be represented in the GA in the computationally-friendly form of strings or arrays. There is not a standard method of encoding as the representation will depend on the specific problem being solved. Another crucial issue that most GAs encounter is premature convergence, which is when a GA reaches a local optima too quickly instead of properly exploring the search problem and finding the global optima. These types of problems can be mitigated by using different variants of GAs and different types of operators within a GA.

For example, parallel GAs attempt to improve computational time by using multiple devices to provide extra computational power [26]. Multiple devices lend themselves to the population model with enough devices. Each device can run its own subpopulation and then have one of the devices act as a controller to trade solutions periodically. Parallel GAs can also use multicore CPUs to achieve parallelism, although splitting subpopulations across cores will not provide as much power as using multiple devices.

Chaotic GAs add substantially more randomness to introduce more diversity [26]. Chaotic GAs are meant to entirely remove premature convergence whereas most genetic algorithms can only slow down premature convergence. Chaotic GAs achieve this by replacing the crossover and mutation operations with chaotic maps to increase randomness. A chaotic map is a data structure made of (key, value) pairs where each key is unique and maps to a value. This approach tends to encourage more diversity but is also susceptible to the parameter initialization problem that is discussed in the 4.4.1 section.

Hybrid GAs combine elements from several types of GAs and other optimization methods [26]. Combining methods through the hybrid algorithm contributes to better solution quality with better efficiency compared to many other methods. Hybrid GAs still struggle with common issues such as parameter initialization, especially the population size.

Another method used to increase diversity is the island model (see Section 2.1.3), which is one of the components of the hybrid GA that this project uses. Island models increase diversity across separate populations with different heuristics and evaluations so they can develop more independently and avoid premature convergence. Then oc-

casionally, a small number of solutions from each island are introduced to a different island to further encourage diversity.

GAs can be very difficult to balance and parameterize, as there are several factors that can drastically influence the results such as population size, fitness functions and control parameters. When training a GA, a substantial amount of time is dedicated to find the appropriate degree of crossover and mutation and reducing premature convergence by experimenting with different values for these parameters.

Mirjalili et al. [35] also review the history of genetic algorithms, but the authors primarily focused on the core operators. Genetic algorithms were one of the first population-based stochastic algorithms proposed and primarily used selection, crossover and mutation . This study discussed how mutation rate can affect the algorithm's performance. Too low of a mutation rate and the solutions tend to score worse due to the low amount of variance introduced. Too high of a mutation rate essentially turns the algorithm into a random search algorithm, nullifying the impact of the selection and crossover steps and having a significantly worse performance. The authors also analyzed different crossover rates, but the differences were ultimately negligible, with a slight advantage going to a higher crossover rate.

3.2 Evolutionary Music Composition Approaches

Although not many studies have focused on AI and sound synthesis, there have been many experiments focused on AI with music composition. In this section, we review some recent notable experiments that focused on applying AI to music composition using evolutionary approaches.

In "A genetic algorithm for composing music [31], Matic used a genetic algorithm to compose short passages of music [31]. The author was able to use a set of musical principles to guide their generation. This project has a similar goal of searching for sounds of the set of all possible music outside of the domain of human activity. While the author used some pre-programmed rules, he mainly relied on human evaluators. The problem he ran into is that many of the evaluators said the music sounded too similar and incomplete, so their criticisms and comments did not vary much between the different samples and were not able to meaningfully change the results.

The authors also ran into the fitness bottleneck issue since they had a limited number of evaluators, significantly slowing down the evaluation process. Unusually, their genetic algorithm did not use crossover at all. Instead, the authors relied on mutation and elimination selection where they removed a large portion of the generated passages. By the end, the authors were able to create passages with interesting rhythms but were not able to have positive evaluations because they ultimately sounded too

unusual and unpleasant. The authors notably suggested a hybridization of methods as suggested future research which is exactly what our project aims to do.

Kaliakatsos [19] also used an evolutionary approach but added a particle swarm optimization algorithm to interpret user evaluations [19]. Kaliakatsos referred to the pre-programmed rules as feature-based evaluation and the human evaluators as interactive-based evaluation. The music composition was broken into two primary levels. The higher level used a particle swarm optimization algorithm and based musical feature changes on user ratings. The lower level focused on the feature-based composition using a genetic algorithm. The overarching goal of this project was to see how both techniques performed when it came to convergence. This approach is similar to our project as it uses both genetic algorithms and user evaluations.

Interestingly, users voted on the rhythm and tone of each musical composition instead of having a single rating/vote. The authors also mention the concept of user fatigue, which describes how evaluators tend to lose focus after voting on a large volume of entries. This is another issue that crowdsourcing mitigates. The concept of noise, in terms of noisy annotations not sound noise, is mentioned. Specifically, between user rating, the particle swarm algorithm and the genetic algorithm. To circumvent this, the authors employed artificial raters to make the processes more deterministic and less noisy. The overall conclusion the paper offers is that feature evolution performed better than melody evolution, demonstrating how representation of the data being evaluated can have a significant impact on how the algorithm evaluates and selects solutions.

Figure 8 depicts a rough flowchart of the overall algorithm, combining both the particle swarm and genetic algorithms.

de Azevedo et al. [2] also used an evolutionary music composition approach, but the authors added another layer of complexity by focusing on evoking specific moods and emotions. The problem of capturing the mood and emotion of automatically generated music was tricky considering the vague and subjective nature of the problem. This project used template pieces to work around this and better direct the mood in each composition. Specifically, the authors generated several templates that were either a happy template or a sad template. To get these templates to truly match their respective mood, all the templates were evolved by a genetic algorithm.

Each template had a number of musical characteristics including scales, melody, harmony and tempo. After running the genetic algorithm for a number of generations, the authors used human evaluators to rank the accuracy of the moods of a small sample of templates. One problem the authors encountered was deciding on the weights certain features should have. Specifically, the authors struggled to balance

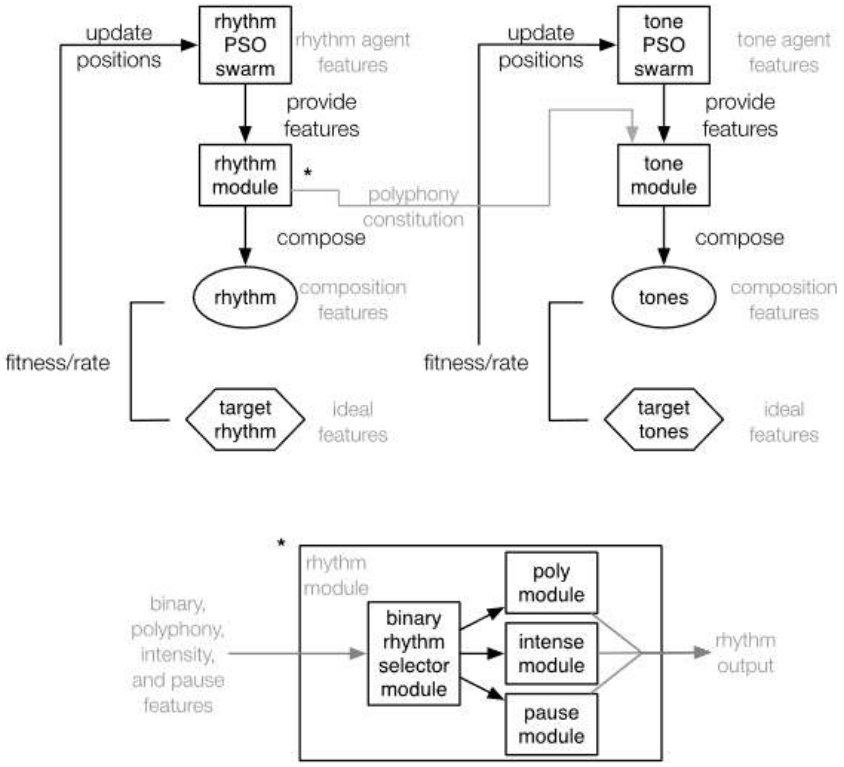


Figure 8: A genetic Algorithm and Particle Swarm Algorithm flowchart [19]

the weights between sounding natural or not computer-generated and matching the mood template. Ultimately though, the authors succeeded in their goal of generating pieces using a certain mood template.

3.3 Genetic Algorithms Variants

The studies previously mentioned primarily focused on just using a genetic algorithm for music composition. This section covers some experiments that implemented a hybrid GA by implementing additional techniques into their algorithm.

Human evaluators are very common in implementations using genetic algorithms. They usually provide better evaluation than pre-programmed rules, but they are also significantly slower than the pre-programmed rules. This speed problem is known as the fitness bottleneck. To fix this bottleneck, Keup [22], focused on solving combining crowdsourcing with their genetic algorithm. The authors were initially worried that human biases and poor short-term memory would muddle with the genetic algorithm, but they attempted to circumvent this issue by providing extrinsic motivation to each volunteer to increase the chance they were focused and engaged with their evaluations.

The authors compared a genetic algorithm that was crowdsourced to a genetic algorithm that was trained by a small team. In the end, there was not a statistically significant difference between the two genetic algorithms. The authors concluded that future research should focus more on adjusting mutation and also let the generations run for longer so the melodies could have developed more. The authors were also curious about purely pre-programmed rules versus crowdsourcing as well as giving different weights to the pre-programmed rules. This project adopted the different weights to the pre-programmed rules to try and create more diversity. This was done to try and reward different attributes between the various populations to decrease the chance of all the sounds sounding identical.

While Keup [22] focused on crowdsourcing to augment their genetic algorithm, Rafael [41] focused the island model, the other technique this project's hybrid algorithm combines alongside a normal GA. Rafael applied the island model or population model to genetic algorithms and music composition.

Rafael was motivated by the fact that algorithmic music composition at the time could only use a single instrument or an ensemble of instruments, unable to switch between either approach smoothly. Most studies stuck to one or the other because it was much simpler to separate the two, but Rafael sought to allow fluid switching between both approaches in their genetic algorithm. Using the island model, the algorithm developed both local and global solutions. Most of the local solutions were single instrument segments of a track and were compared to the global compositions to see which ones fit the best in the global track. The island model is well suited to this problem since it is better at finding more robust and complex solutions. In addition to developing subpopulations as the "islands", the other main component of the island model is the migration operator.

For this experiment, the islands initially sent their best solution to an adjacent island, as the migrating solutions rotated around in a circle. For subsequent migrations, the migration operator changed the pairings of islands and shifted them so that eventually, each island would have sent and received a solution from each island. This strategy is called unidirectional ring migration. Rafael also tested a migration strategy known as circulating ring migrator which turned out to be more effective. Instead of changing migrating pairs by shifting pairs to the next adjacent island, the circulating ring migrator randomly picked pairs each time. The extra randomness this method added proved to be more useful than the more structured unidirectional ring migration.

Rafael also explored the importance of different migration rates. Migrating too often prevents each island from developing enough to properly conduct its local search

while migrating not often enough tends to reduce the impact of migrating by having the global information that comes with the migrating solutions be lost if too much time passes. Rafael's results suggest the island model genetic algorithm was promising, but only when the local segments of the tracks shared some of the structures of the global tracks, signifying that a lot could still be improved in future research.

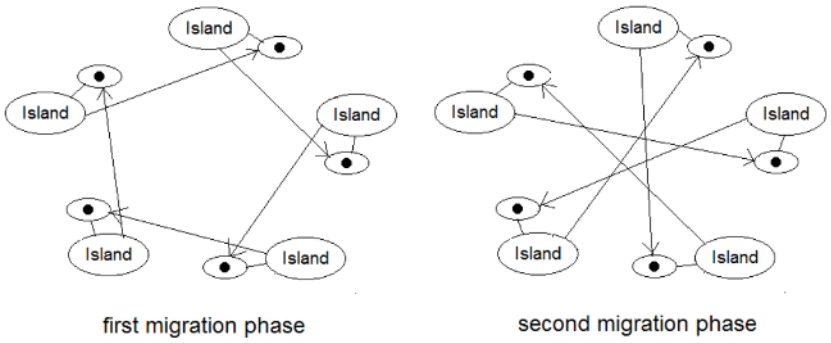


Figure 9: An example of island model migration [41]

Figure 9 illustrates two steps of the unidirectional ring migration technique. The dots in the smaller bubbles represent the migrating solutions.

3.4 Sound Synthesis

Many experiments have focused purely on implementing and analyzing various sound synthesis techniques. This section reviews some of the most relevant and recent sound synthesis experiments.

Woodhouse et al. [50] explored what makes a string instrument sound more like a guitar versus a banjo. The researchers recreated string instruments using sound synthesis to try and identify what characteristics create similarities to a banjo sound. Initially, the authors used spectrograms and loudness profiles to search for differences and they were able to find that certain overtones were much louder and decayed much faster on a banjo. Strangely, the authors found that banjos shared many characteristics with drums, showing that instruments can vary wildly and that it is difficult to predict an instrument's attributes on its sound. Notably, the authors used inversed fast Fourier transformation and a physics-based sound synthesis where as this project uses procedural synthesis.

Figure 10 presents two spectrograms of a banjo and guitar from top to bottom respectively. They are playing the same note, but because of slight timbre differences,

the spectrograms have some slight differences.

As we mentioned earlier, timbre is an elusive term that has many vague definitions. Wallmark and Kendall [48], investigated why timbre lacks a clear definition by analyzing the various definitions as well as the key attributes of timbre. Timbre describes the quality of a sound. It is notoriously hard to describe largely because it has both objective or physical characteristics and subjective or perceptual characteristics. Timbre contributes a lot to what makes a sound or instrument distinctive, so understanding and creating different timbres is a major goal of this particular project. With this research, the authors were more focused on describing timbre and the language that is commonly used to describe timbre. Early definitions described sound by its harshness and its brightness [48].

Later definitions defined timbre as an attribute of sound that allows a listener to distinguish different sounds that have the same volume and pitch or they described it using tone or color. These all fall under perceptual definitions which means they come with biases such as the reification and post hoc problems. Reification relates to how we perceive physical stimuli logarithmically when things like loudness are on an additive scale where strength progresses multiplicatively. To get around this, present-day researchers have chosen it best to separate physical and perceptual frames.

The post hoc problem results from how many of the descriptive terms are given by those running the study and not words that people volunteer themselves, which could potentially lead to framing bias instead of organically allowing participants to pick words themselves. There is a mediating influence at play that overlaps with several other frames of reference that contribute to timbre semantics being mixed. On the other side, you have physical definitions that focus on aspects like a sound's spectrum, temporal envelope and spectral deviation.

The overall conclusion of this study was that although there are many inconsistencies in language, there are some common descriptions used for timbre. Common themes were texture and mass as well as onomatopoeias. A key observation Wallmark and Kendall[48] made was that descriptions could vary significantly between different musicians, especially if they played different instrument groups. Common metaphors for music were light and texture which likely relates to synesthesia, or how one sense can be routed to other senses, causing the observer to experience multiple senses. The light and texture metaphors suggest that most experience sound with a mild mix of auditory, visual and tactile modalities of sound. The researchers concluded that timbre is not purely subjective as suggested by the common terms used and weak synesthesia. The authors also suggested that the vagueness of the terms could be viewed as a semantic shortcut that provides a network of interdependent connotations and implications instead of referring to any established definition.

One option for sound synthesis is to carefully adjust parameters and then combine sounds together as with additive synthesis, but there are also synthesis techniques that can happen almost instantaneously by taking in some form of input in real-time. The example this section will cover comes from "Sound synthesis and evaluation of interactive footsteps for virtual reality applications" [38]. In this experiment, the researchers created a system to synthesize the sound of footsteps in real-time.

The system accounted for different materials being walked on using microphones next to the user's footwear. This differed to other systems at the time which usually relied on shoes with special sensors to synthesize the sounds. The microphones detected the footstep sounds from the user and then estimated the ground reaction force based on the samples. This force was used to control a sound synthesis engine that was based on physical models. The system used the decomposition of complex sound phenomena into more simple elements that could be understood by the system. The authors also covered how the system was able to break down and understand several sonic interactions. It focused on friction and impact to synthesize each sound. After the system was built, it was tested by humans to see if the sounds were convincing enough that humans could distinguish and identify the collection of footsteps sounds across different surfaces.

Nordahl et al. found that interactive systems compared significantly better than prerecorded sounds, but the accuracy for both were similarly high. Participants had more mixed results with recognition. Surfaces like snow, creaking wood and metal were identified consistently while others like high grass and surfaces with pebbles were not. The results indicated that if subjects were wrong, they tended to erroneously classify a surface as another belonging to a similar category such as snow and frozen snow. The addition of reverberation to the sounds also improved accuracy. The researchers also realized that the footstep sound was highly dependent on the floor the test was being conducted in as well as the shoes the subjects were wearing. Subjects also commented that they felt like they were wearing a different shoe, suggesting the ability of auditory feedback to affect the sense of physical touch. This project reflects the success of a physics-based model of synthesis. This project however uses additive synthesis which is a procedural-based model instead of a physics-based model.

Figure 11 depicts the footstep simulation setup. The four microphones take in the input used by the synthesizing system while the special surface identifies other characteristics like friction and force that the microphones could not easily detect.

"State of the art in sound texture synthesis" [42] serves as a review of the most popular sound synthesis methods. Schwarz defines sound texture as having a local structure and randomness where the characteristics of the fine structure must remain

constant overall. Sound textures are formed from basic sound elements or atoms that are periodic, random or both. The characteristics of these atoms must remain the same over long periods of time and must be completely exposed within a few seconds. Randomness is fine as long as there are enough occurrences of randomness so the random properties can be identified. By their definition, they exclude things like sounds resulting from objects interacting, soundscapes and sound design. The author also point out that sound texture is meant to distinguish the generated sound material from tonal and percussive sound, which means that they are primarily interested in the timbre and not the pitch or rhythm of the sound.

Next, the author goes through the various sound texture synthesis methods. Broadly, the author split the methods into physical based models and signal based methods. Noise filtering or subtractive synthesis involves using oscillators to break down and isolate partials of existing sounds. Additive synthesis is in many ways the opposite of subtractive synthesis. Instead of taking apart an existing sound, it builds up a new sound by creating simple sine waves and stacking them to combine them into a more complex sound. Physical modeling is the one physics-based based model the author mentions and it uses mathematical equations based on principles of physics to construct sounds. It can create more in-depth sounds but is often more complicated since each type of sound needs its own class of equations to synthesize them.

Granular synthesis is similar to subtractive synthesis in some ways conceptually as it also breaks down sounds into smaller components. With granular synthesis though, the components are much smaller and are referred to as particles. These particles are then resampled to create new sounds. The rest of the review is spent analyzing these different methods, where the author concluded that physical-based models are more flexible but have attributes that are not directly related to the perceived sound.

3.5 Summary

The studies and methods discussed in this chapter highlight some of the contributions to AI and music that are most relevant to this project. This project involves both a feature and interactive based genetic algorithm much like [19] and relies heavily on human evaluation much like [31] except this project uses crowdsourcing in a similar approach to [22] does instead of a smaller group. The other key additional technique this project used was the island model, similar to [41]. This project also focuses on the attributes and differences between certain instruments to try and replicate their sound much like how [50] did. To achieve this, the signal based technique of additive synthesis is used as mentioned in [42].

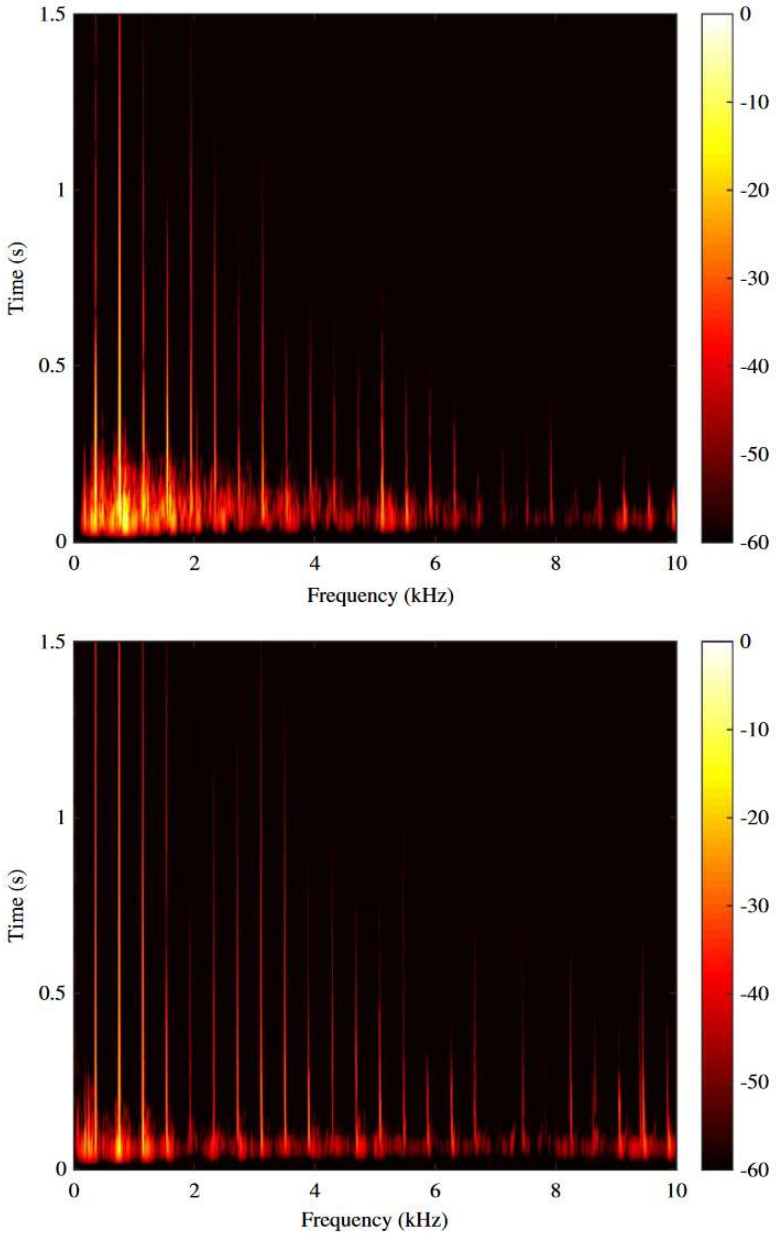


Figure 10: The spectrograms of a banjo and guitar playing the note G4 [50]



Figure 11: The setup for the footstep simulation [38]

4 Methods

In the 2 section, we discussed the general idea and purpose of the four main genetic algorithm operators. This section will explain the specific techniques and variations of these operators implemented in this project. First, we will cover the representation this project uses as well as some of the most notable issues this project encountered and addressed.

4.1 Representation

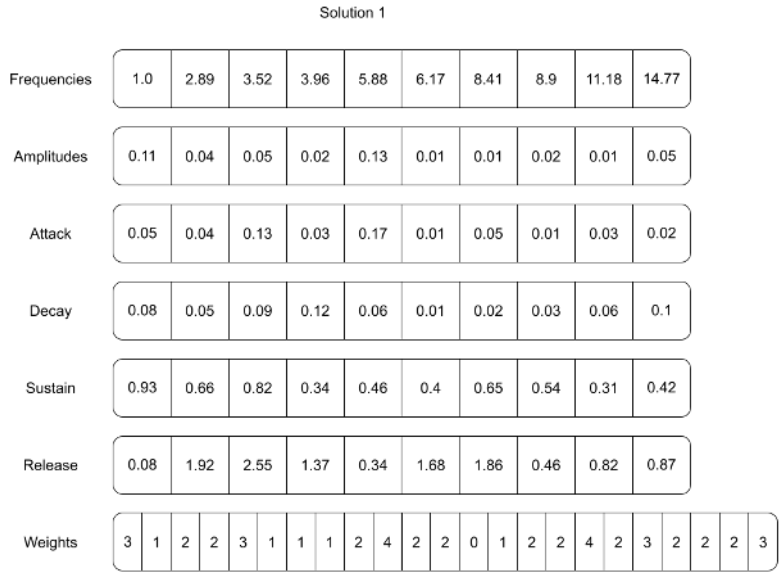


Figure 12: The frequency, amplitude, weight and ADSR arrays that are stored in Solution 1

Figure 12 illustrates the representation of a sound for this project. The representation of a sound consists of arrays of frequencies, amplitudes, weights and the ADSR envelope values for the attacks, decays, releases and sustains. Each of the data types has a length of ten since each sound has ten partials. We picked 10 partials because it balanced providing enough depth without making it too dense and complicated. The arrays are ordered by harmonics, meaning the first element of the array corresponds to the fundamental frequency and the second element corresponds to the second harmonic. The weights array has 23 elements since there are 23 fitness functions in total.

Initially, we tried to represent the frequencies with the raw Hertz values, but we changed the representation to relative ratios compared to the fundamental frequency.

This means that the first partial always has a ratio of 1.0 since it is the fundamental frequency. The other harmonics are represented as randomly generated ratios to the fundamental frequency. The amplitudes are treated as values and not ratios. The attack, decay and release values of the ADSR envelopes are treated as values while the sustain values are treated as ratios.

The array of weights consists of weights for all the fitness rule functions used in the evaluation step. Each individual fitness calculation has a global weight to normalize the effect of each fitness calculation on a candidate solution's fitness score. However, each candidate solution also has its own set of randomized weights so that each solution does not receive identical fitness scores which should have the benefit side effect of creating more diversity. The frequency, amplitude and ADSR arrays are passed into functions from the `Tone.js` library to play the generated sounds using multiple synthesizers for each partial.

Each sound is part of a population. Our project stores eight sounds in a population. We decided on a relatively small number of sounds in a population since we wanted the algorithm to store a large number of populations. Our project rotated between 50, 100 and 500 populations within a single run of the algorithm. A larger number of populations encourages more diversity and the relatively low number of sounds in each population is unproblematic since the algorithm tends to make sounds in a population homogenous over time. Increasing the number of sounds per population does not drastically change this tendency.

This project aims to develop randomly generated sounds into sounds that mimic real instruments. A genetic algorithm evolves the random sound into more complex and interesting sounds, meaning the genetic algorithm takes in and evaluates the representations of the sounds. To augment the genetic algorithm, this project implements the population model and crowdsourcing alongside the GA. Even with these additions, the genetic algorithm still converges quickly which is counterproductive to our goal of exploring the sound space and finding as many interesting instruments as possible [5]. The rest of this section explores the most difficult challenges that we encountered with our representation.

4.2 Premature Convergence and Diversity

Premature convergence occurs when the candidate solutions of a genetic algorithm become extremely similar as a result of the high-scoring solutions and their descendants consistently passing their genes without allowing any of the other candidate solutions to pass their genes [19]. Premature convergence is one of the common downsides of a genetic algorithm and is particularly problematic for this project. This project's desired results are more subjective in nature compared to most optimization prob-

lems, which leads to a wide range of problems that are not usually issues when using a genetic algorithm. Ideally, our genetic algorithm will settle on a wide variety of potential solutions since there is not a single solution. In this work, we desire as many different results as possible, which is one of the main reasons why this genetic algorithm combines a number of different techniques in order to avoid the typical tendencies of a genetic algorithm to optimize a single candidate solution [26]. The perceptual or subjective nature of sounds makes this especially tricky.

Evaluating the quality of sounds using mathematical fitness scores will limit the creative goals of this project, which seeks to create a diverse set of interesting instrument sounds. The process of recreating specific music instruments using additive synthesis is a relatively straightforward process and previous research has explored this endeavor while far fewer have attempted to create a unique variety of instruments [25], [30]. This project seeks to randomly generate sounds in hopes of discovering new and unusual sounds that are unfamiliar but still interesting or enjoyable.

For this reason, we crowdsource to collect a diverse set of subjective human evaluations [52]. If we only used mathematically derived fitness evaluations, the sounds generated would likely remain similar and homogeneous because of the nature of using only pre-programmed rules for a task that is very subjective in nature. To further foster increased diversity, we utilize the population model or island model. This has the effect of a local and global search and generally allows greater genetic diversity among the candidate solutions, making it an ideal approach for a creative endeavor such as ours. We employee both strategies to increase the diversity of the sounds generated from the genetic algorithm.

4.3 Genetic Algorithm Operators

This section will discuss the four main genetic algorithm operators, evaluation, selection, crossover and mutation and their use in this project.

4.3.1 Evaluation

The two types of fitness evaluation our GA uses are mathematical and crowdsourced evaluation. Mathematical evaluation uses several calculated fitness functions to analyze and check each sound. Each fitness function either gives a positive score to the candidate sound if they meet certain criteria or gives them a negative score if they have undesirable traits [12]. The score from each function is weighted using global weights and individual weights in each solution's weights array. The scores from each function are then summed to assign a single fitness score for a solution. The individual weights has the effect that even if two solutions have identical frequency, amplitude and ADSR values, they would receive different scores because of the different weights they have for each fitness function. The weights can also be zero, which means that

some fitness functions may not influence certain solutions at all, adding an additional layer of randomness. Mathematical evaluation eliminates the worst sounds so that we best utilize the crowdsourced evaluation.

Crowdsourced evaluation aggregates votes from an interactive website to determine how "fit" a sound is according to the subjective interpretation of human evaluators [51]. The website conducts A−B testing such that only two candidate solutions are compared for each vote.

In a database, we track how many voting opportunities each sound participated in as well as how many votes the sound won out of those voting opportunities. If the sound has not been evaluated by a human yet, then this part of the evaluation score is not used. Both types of evaluation increase or decrease the fitness score of each solution to determine the "fittest" solutions of the current generation. We designed many mathematical helper functions based on desirable acoustic properties exhibited by many physical and synthetic instruments. Some of these acoustic properties are in direct opposition, which is appropriate for this particular project. The goal of this project is to find a diverse range of interesting sounds, and since each population has its own weights for the fitness functions and also randomly picks a subset of fitness helpers to use for determining the solutions' fitness values, it is not counterproductive to have helpers that reward opposing traits.

The rest of the subsection describes each of the our implementation's current fitness functions. The fitness functions either target a sound's frequencies, amplitudes or envelope, with the one exception being `compare_to_ideal_sound` which checks all of these aspects. First, we will explain `compare_to_ideal_sound`.

compare_to_ideal_sound

`compare_to_ideal_sound` uses a pre-generated sample sound and compares that to each candidate solution produced by the GA. For the best results, the sample sound should be an instrument or exhibit pleasant attributes that are common in instruments.

Since a sound is represented as arrays of integers or float values, we simply compare the values. More specifically, this function keeps track of the absolute value of the difference between each corresponding data value of the sample sound and each solution in the GA. This value is inversed and then added to the solution's fitness score, such that the smaller the difference between a particular solution and the sample sound, the higher the score the solution receives. It has a low global weight because it targets a specific sound when the goal of this project is to find a variety of pleasing sounds. For this reason, we gave it less influence compared to other fitness

functions.

Next, we will discuss the functions that analyze the amplitudes of a sound.

punish_loud_amps
`punish_loud_amps` punishes solutions that produce amplitude values above a threshold set by the user. When generating the sounds, extreme amplitude values can result in the audio clipping, which produces a very harsh, unpleasant noise and is generally problematic for this project. One solution would be to simply limit the upper range of the amplitudes of each harmonic partial when they are randomly generated at the start of the GA. However, since the goal is to produce a lot of diverse sounds, it would be ideal to avoid being too restrictive with the range and limiting the results that could be produced. Many instruments have a few particularly loud partials, and the rest are comparably lower.

As is the case for real world instruments, it is acceptable if certain partials are much louder than others, but it quickly becomes a problem when most of the partials have high amplitudes which combine to produce a loud waveform that results in clipping. The main purpose of this fitness function is to reduce the number of sounds that have clipping audio. It checks each candidate solution and looks at each partial's amplitude. The function increases the solution's fitness score if the amplitude is below a certain threshold. This penalizes sounds in which many of the partials are too loud and the solution's score is significantly lower while still allowing solutions with only a few loud partials to not be punished too severely.

amps_sum
`amps_sum` is another function that tries to minimize clipping, but it engages in a different approach than `check_bad_amps`. Instead of checking individual amplitudes, this function measures the sum of all the amplitudes of the sound. We include this additional metric because it is still possible for a solution to be rewarded by `check_bad_amps` due to not having very loud individual frequencies that still results in clipping of the produced audio waveform. This function was designed to punish one of the more common cases that `check_bad_amps` would not prevent. If the sum of amplitudes is below a certain volume threshold, the solution's fitness score is increased.

error_off_amps
`error_off_amps` also employs root mean squared error, but it applies them to amplitudes instead of frequencies. Another key difference is that the ideal values for the amplitudes of the harmonics are often values between 0 and 1. Instead, the values are derived from an inverse curve, since the amplitudes of the harmonics of many real world instruments tend to decrease compared to the previous harmonic. In this case,

the root mean squared error measures the distance between an inverse curve and the actual amplitude values of each solution. Solutions are rewarded in which a smaller error means the amplitudes are closer to a logarithmic curve and are more likely to produce a pleasant sound.

check_amp_sum
`check_amp_sum` is similar to previous functions related to amplitudes, but it puts a stricter cap on the threshold, only rewarding functions that have an overall amplitude sum that is less than 1.0.

Next, we will discuss the functions that target the frequencies or harmonics of a sound.

check_increasing_harmonics
`check_increasing_harmonics` rewards solutions that have their partials' frequencies in increasing order. This pattern is generally more desirable compared to having a completely random order for the frequencies of each partial. This function checks a solution's adjacent frequencies and rewards solutions for each partial that has a lower frequency than its neighboring partial that is one index higher than it in the frequencies array.

check_true_harmonics
`check_true_harmonics` rewards solutions with partials that are closer to integer values. For most harmonic musical instruments, the loudest partials tend to be integer multiples of the fundamental frequency. This tends to result in a more subjectively pleasing sound, so we want to reward solutions with a partial that is close to a ratio of 2 instead of a non near integer multiple of 2.47. This function uses the base frequency and finds the closest integer multiple for each of its frequencies and then checks if the distance between that multiple and the frequency is within a certain threshold. If it is, then it will increase the solution's score to reward this pattern.

check_wobbling
`check_wobbling` attempts to punish solutions that have frequencies that are too close to the base frequency. When frequencies are too close to each other, it produces dissonance and creates a wobbling effect. To try and minimize this, this function checks each solution and lowers the score for each frequency that is below a certain distance threshold from the base frequency.

check_octaves
`check_octaves` is a function that rewards solutions that contain an octave between partials, which is a very specific condition. In terms of frequency values, an octave is exactly twice of another frequency value. This function takes the base frequency

of each solution and checks if there is an exact multiple of two within the other frequencies. This helper is ultimately too specific and very rarely alters the score but we balanced it by heavily rewarding a solution when the function does actually find an octave since that is a desirable trait.

check_fifths
`check_fifths` uses a very similar idea to `check_octaves` but instead of searching for an octave, it rewards occurrences of a perfect fifth between partials. Where an octave has a frequency ratio of 2:1, a perfect fifth has a frequency ratio of 3:2. This is also a desirable trait that occurs in many human designed instruments, therefore this function heavily rewards solutions that contain this quality. Similar to `check_octaves`, this function changes fitness scores very rarely, so its utility is fairly limited due to how infrequently it affects the scores of the candidate solutions.

error_off_partials
`error_off_partials` is similar to `check_true_harmonics` in that it rewards a very similar pattern but differs because it uses the root mean squared error 15 to calculate each solution's fitness score. Root mean squared error is very useful for measuring the average difference between ideal values and actual values. In this case, the ideal values are exact integer multiples of the base frequency and the actual values are each solution's frequency values. The larger the error, the larger the distance and the worse score that solution receives. Again, this is performed because exact integer partials tend to sound more pleasing and are more common to most harmonic instruments.

$$RMSE = \sqrt{\sum_{i=1}^{n} (\hat{y}_i - y_i)^2 / n} \tag{1}$$

Equation 4.3.1 presents the mean squared error formula. As indicated by the figure, the equation takes the difference between an expected value and the actual value, squares it, divides by the number of values taken and then takes the square root of that value.

check_freq_sparseness
`check_freq_sparseness` is similar to `check_wobbling` in that it punishes frequencies that are too close to each other. The main difference is that it examines each adjacent solution instead of just focusing on the base frequency. Solutions with larger sparseness are rewarded with higher fitness scores.

check_multiple_bands
`check_multiple_bands` rewards sounds are within a specific bandwidth range. Bandwidth is the difference between the upper and lower frequencies of a sound. A larger

bandwidth can potentially result in more noise. Having a smaller bandwidth allows for higher audio quality, so this function sets a low bandwidth threshold, keeps track of the lower and upper frequency ranges of each solution and rewards them if they fit within that bandwidth threshold.

Finally, we will discuss functions that analyze a sound's envelope.

check_decreasing_attacks

`check_decreasing_attacks` focuses on the ADSR envelope instead of the frequencies or amplitudes. Specifically, it rewards solutions that exhibit decreasing attack values as the frequency values increase. This is a generally desirable trait for sounds to have. Solutions who follow this pattern are given a higher fitness score.

check_pads rewards

`check_pads rewards` solutions with long attacks and long releases. This rewards sounds that are more like [4]synth pads, which are long sustained notes. The term comes from orchestra where long notes would "pad" out certain sections of musical pieces. This function checks each solution's attack and release values and rewards the solution if their sum of attacks or releases is above a certain threshold to reward sounds that mimic pads.

check_staccatos

`check_staccatos` rewards sounds with short attacks and short releases. This creates a very quick, explosive note that gives it a staccato quality. It essentially rewards the opposite of what `check_pads_rewards`, which is intentional. The different weights each sound has enables a balance between functions that reward opposing traits. This function checks each sound's attack and release values and rewards the sound if their sum of attacks or releases are below a certain threshold.

check_percussive_sounds

`check_percussive_sounds` rewards solutions with short attacks and long releases. This is a quality exhibited by most percussion instruments. In general, most of the functions thus far have rewarded traits of harmonic instruments, and percussive instruments differ to other instruments in several ways. This function was made to increase the chance of inharmonic sounds like a drum being selected in the GA. Similar to the other envelope related functions, this function checks each solution's attack and release values and rewards them if they are below or above certain thresholds respectively.

check_transients

`check_transients` rewards solutions with short sustains and longer decays. This

[4]https://lunacy.audio/synth-pad/

is another quality is common to many percussive instruments. To create a diverse range of sounds, we created enough functions that also reward more percussive traits. Similar to the other envelope related functions, this function checks each solution's sustain and decay values and rewards them if they are below or above certain thresholds respectively.

check_amplitude_sparseness
`check_amplitude_sparseness` uses standard deviation to reward solutions that have a more consistent blend of amps rather than one central amplitude. Most instruments have at least a few loud partials that primarily dominate the sound, so it is perhaps more desirable to have a few louder amplitudes rather than one with a particularly loud amplitude. The standard deviation is used to quantify how spread out the values are compared to the mean. Those with a smaller spread are rewarded since a smaller spread means more consistency.

avoid_too_quiet checks
`avoid_too_quiet checks` searches for amplitudes that are below a threshold set by the user. Although we do not want clipping, a harmonic serves little purpose if it is too quiet and does not contribute to the overall timbre. This function checks each solution's frequencies and increases the solution's score if a frequency is above a certain threshold.

check_decreasing_amps
`check_decreasing_amps` rewards solutions with decreasing amplitudes as the frequencies become higher. Generally, the lowest partials are the most prominent or loudest. It is usually desirable to have quieter partials as the frequencies become higher. This function checks adjacent frequencies and rewards the solution if their amplitude decreases as the frequency increases.

fundamental_freq_amp
`fundamental_freq_amp` punishes solutions if they have a frequency that is louder than its base frequency. As mentioned earlier, the lower partials tend to be the most prominent, and this is especially true of the fundamental frequency. To reward this, this helper checks each frequency and punishes the solution for each frequency that is louder than the fundamental frequency.

inverse_squared_amp
`inverse_squared_amp` rewards functions that decrease in amplitude as the frequency increases. This function approaches this differently than `check_decreasing amps`. It looks at the index of each amplitude in the array of amplitudes and compares the value to

$$\frac{1}{\text{index in the array squared.}} \tag{2}$$

The difference between these two values is subtracted from the solution's score, meaning the closer the amplitude value is to the inverse square of the index, the better score it will achieve.

4.3.2 Selection

After each solution receives a fitness score, the algorithm shifts to the selection phase to determine which sounds are used as parents for the next generation. The different selection methods generally trade off between consistently rewarding the higher scoring solutions and randomly selecting solutions to create more randomness and diversity. Here are the various selection approaches this project used.

Figure 13 illustrates a rough outline of the genetic algorithm flowchart.

Once the fitness scores are determined for each solution, the population of each island undergoes selection. Intuitively, one might assume that it would be optimal to simply choose the solutions with the highest fitness scores, but this can actually contribute to a lack of genetic diversity of the results which is contrary to the creative goals our project. Selection still needs to reward solutions with higher scores so the fitness helpers have their intended effect, but there ideally is also a level of randomness that maintains some level of diversity [35]. The process of selection is not as straightforward as it might be assumed, so there are a handful of different techniques that implement selection in slightly different ways. Since all of the techniques provide a comparable level of utility, our implementation has the option to run any of the selection methods listed below to modify the solutions selected throughout the genetic algorithm. Here are the selection methods we implemented for this project.

tournament_selection

`tournament_selection` effectively puts all the solutions of an island into a bracket tournament. Sounds are randomly seeded into the bracket and then are compared against each other. The solution with the higher fitness score moves on in the bracket. This pattern continues until a small enough number of solutions are left. This method still rewards higher fitness scores while also allowing solutions with scores on the middle or lower end to potentially be selected based on luck of the draw when it comes to who they go against in the bracket. This encourages genetic diversity in the population.

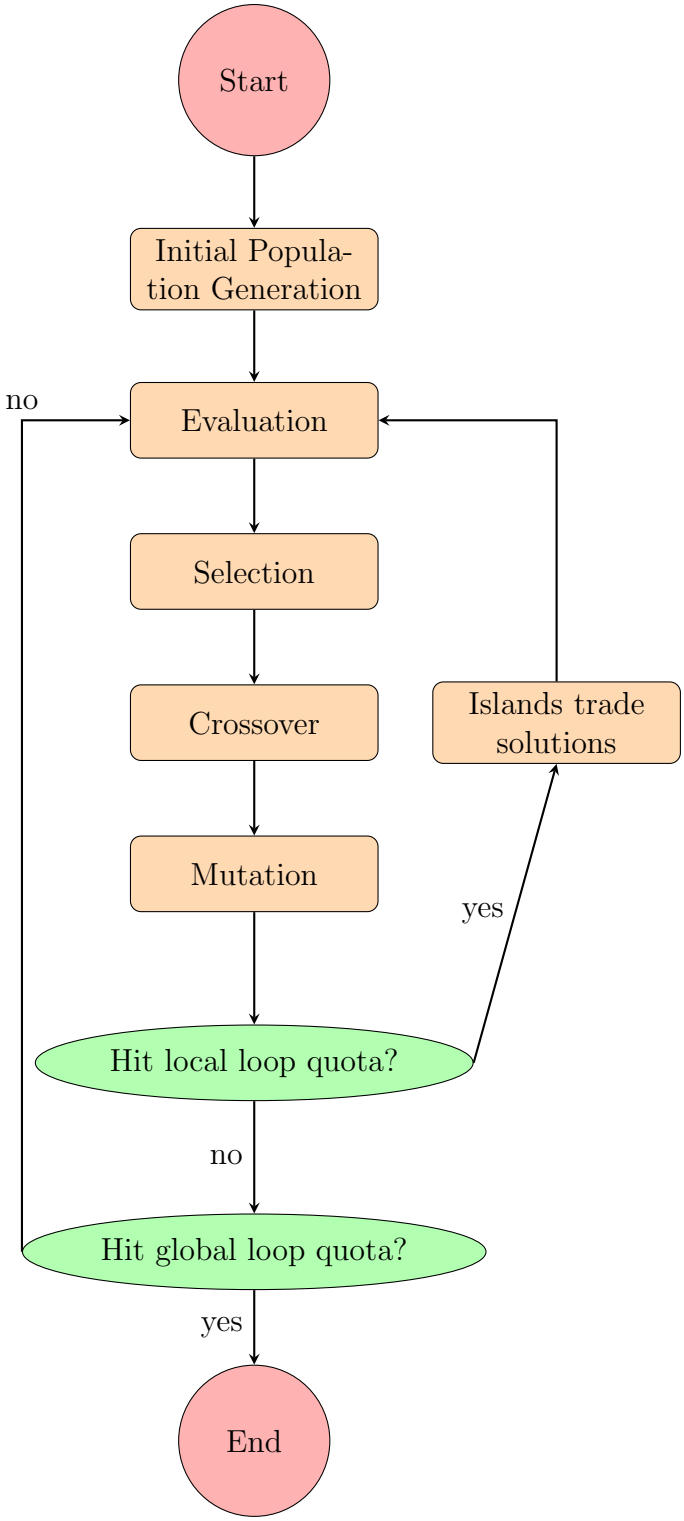


Figure 13: A flowchart of the genetic algorithm

elitism_selection
`elitism_selection` purely picks solutions based on fitness score. It will pick the percentage of the highest scoring sounds without exception. Although it is usually desirable to have some randomness to have diversity, the use of elitism selection can still be very effective because ultimately the selection should be based on fitness helpers to propagate the criteria they rewarded and remove the criteria they punished.

variety_selection
`variety_selection` operates on almost the exact opposite of elitism selection and focuses primarily on maintaining genetic diversity. The goal of variety selection is to pick the sounds that differ most, or the highest scoring solutions and the lowest scoring solutions. The function will still choose some sounds that were rewarded more by the fitness functions, but it will also choose sounds on the lower scoring side to try and allow solutions that do not strictly adhere to the fitness function's criteria to still have some chance at being selected. This function allows for more diversity and to try and avoid premature convergence as much as possible.

roulette_selection
`roulette_selection` bases selection by simulating a giant roulette wheel similar to game shows like wheel of fortune. It achieves this by adding up each solution's score to use as a sum. Each solution's score is used as a portion of the roulette wheel. From there, a random number is generated in between 0 and the sum and the solution that has a score that contains that value is selected. This process continues until enough solutions are selected. Figure 2 demonstrates this process.

Here is an additional example on top of the Figure. Imagine there are five solutions with fitness scores 1.0, 3.0, 5.0, 7.0 and 9.0. The range would be 0-25 and the space that each solution would occupy on the wheel would be 0-1, 1-4, 4-9, 9-16 and 16-25 respectively. if the value were 5.0, then the solution with a score of 5.0 would be selected. To resolve picking solutions if the randomly generated value ends up being perfectly between two solutions (such as if the value ended up being 1.0, 4.0, 9.0 or 16.0), a comparison operator determines which range is included. This method's purpose is very similar to tournament selection in that solutions with higher fitness scores still have a clear advantage, but there is still a chance for solutions with lower scores to be picked.

rank_selection
`rank_selection` is very similar to roulette selection in that it simulates a giant roulette wheel to pick its solutions, but the method it divides up the wheel between sounds is the key difference. Instead of using the scores as the segments of the wheel, the lowest scoring solution is given a chunk of 1, the next lowest scoring solution is given a chunk of 2, and so on and so forth. Every other aspect of the implementation

is basically identical to roulette wheel like with the random value and in between values. This is very useful if the sounds of a population end up having very similar scores or if there are some major outliers in the scores. Rank selection standardizes the range much more than roulette selection. It still follows the pattern of rewarding high-scoring solutions but still gives a much better chance for lower-scoring solutions.

4.3.3 Crossover

Once solutions are selected to reproduce, the next generation is the result of combinations of the representations of two "parent". Similar to the other operators, there are a handful of different techniques that provide different utility [24]. Similar to 4.3.2, all the crossover techniques provide utility, so instead of using a single crossover technique, we implemented a few crossover techniques that we could switch between. Here are the main crossover techniques we designed for this project.

Single-point crossover
`Single-point crossover` is arguably the simplest of all the crossover techniques. It picks a single point of each data array to partition the representation into two halves. The first section of each data array is taken from the 1st parent of a pair and the second half is taken from the 2nd parent. The reverse is done for the second child of that parent pair. All crossover methods contribute to premature convergence and single-point crossover arguably contributes the most out of the crossover techniques used in this project. The way it determines which genes each parent propagates is very simple and repetitive with the least randomness comparatively. Due to premature convergence, sounds tend to become more similar over time which means that splitting and creating future generations in the same, simple way can quickly create many solutions that are virtually identical. Still, homogenous sounds is a common issue across all forms of crossover, so we still use single-point crossover as we switch between crossover techniques.

uniform_crossover
`uniform_crossover` has the potential to add more randomness to the crossover to combat premature convergence. The additional randomness comes from using a coin flip to determine which parent is picked for each array of data for the children. This maintains some level of diversity by allowing certain parents to be significantly more or less prevalent in their offspring's genetics. Similar to single-point crossover, uniform crossover combines and mixes the genetics of parents with less randomness comparatively, but we still use uniform crossover as we switch between crossover techniques.

deep_uniform_crossover
`deep_uniform_crossover` takes the randomness that was present in uniform crossover and scales it up even further. Instead of randomly selecting arrays from each parent,

each individual element in the data arrays is picked with a coin flip. This further increases diversity in the population by having a more random mixture of each parent's genes present in the offspring. Figure 14 illustrates a simple example of deep uniform crossover using two arrays filled with zeros and ones.



Figure 14: An example of deep uniform crossover [33]

4.3.4 Mutation

Many of the methods mentioned in the previous section focus on maintaining diversity. However, the nature of a genetic algorithm will ultimately push solutions in a certain direction, and over many generations, risks converging to a population of nearly identical sounds. Mutation is an important technique to introduce diversity in the population by randomly picking solutions from a population and then randomly altering some of the solution's genetics to introduce something completely different from the rest of the sounds. It allows for completely new genes to form, much like how humans mutated to have completely new traits like blue eyes [12]. Mutation is particularly crucial because without it, the solutions are never truly introduced to new genetics. Without mutation, all the genes of future generations would come from "parent" solutions. Mutation is an important step with the potential to dramatically change the genetics of a solution. These changes can combat premature convergence. Here are some of the mutation methods that we designed for this project.

mutate_gene
`mutate_gene` randomly picks between a solution's array of frequencies, amplitudes or one of the components of the ADSR envelope and multiples each value of a scalar. A coin flip is used to determine if the scalar is greater than one or less than one (non negative). Although mutation is often very helpful for encouraging genetic diversity, overly aggressive mutation strategies can create too much randomness and either introduce noise to the data or create sounds that are penalized by the majority of the fitness functions. Therefore less aggressive methods like this one can still be quite useful.

mutate_solution
`mutate_solution` mutates each array within a solution by multiplying it by a scalar.

This can be thought of as a more aggressive version of `mutate_gene` since it affects the entire solution instead of a specific component. While overly aggressive methods can be detrimental to the genetic algorithm, a blend of aggressive and non-aggressive strategies can maintain diversity without creating a counterproductive amount of randomness.

mutate_individual_weight
`mutate_individual_weight` mutates the weights that each solution has for the fitness helpers used. Solutions are randomly assigned individualized weights for the fitness functions so not every solution is rewarded identically with the same criteria in order to create more diversity in the population. A better mixture is created when the reward criteria is not identical for each solution. This mutation method allows these weights to change over time to encourage more different kinds of sounds to propagate through the GA. A random fitness helper is picked and a coin flip determines whether the scalar value that the weight will be multiplied by will be less than or greater than 0.

4.3.5 Computation

The largest time complexities in the algorithm result from the four core genetic algorithm operators. In this section, we describe the computationally intense operations within the genetic algorithm. Certain functions such as the compare_to_ideal_sound function compare two sounds directly. Specifically, each individual data value that is stored within a solution is checked against a pre-generated sound's data values. To understand the level of computation required for this function, we first need to understand the high-level view to consider all the solutions that could undergo this comparison in a single generation.

Each run of the genetic algorithm contains several populations and each population typically has eight candidate solutions. Each solution has 10 partials for each of its data arrays storing values related to data like frequencies and amplitudes. If n is the number of partials in each solution, t is the number of data types stored in each data array of a solution, m is the number of solutions each population has and s is the number of subpopulations the genetic algorithm keeps track of, then the number of comparisons made in total is $n \cdot t \cdot m \cdot s$. This is implemented in the helper function as a loop to cycle through the solutions of a population, a nested loop within is used to cycle through each array data type (frequency, amplitude, etc.) and then a third nested loop within is used to check each value within each data array, resulting in a time complexity of $\mathcal{O}(n \cdot t \cdot m)$ for a single population.

To make this time complexity manageable for this experiment, we have generally kept, n, t and m as small constants, greatly reducing the time compared to the poten-

tial time complexity if these were larger numbers. In testing, we empirically evaluated these parameters to determine an optimal range instead of randomly assigning them within a single run of the algorithm or between different runs of the algorithm.

4.4 Hybrid GA Implementation

This section discusses some of the extra techniques used that are not the core components of a genetic algorithm. The techniques discussed here range from additional measures meant to improve the genetic algorithm to sanity checks that verified that all the components were working correctly.

4.4.1 Tuning parameters and weights

The values of the parameters given to the GA can significantly affect the results produced. The large number of parameters makes it difficult to properly balance the various parameters used by the genetic algorithm to get the desired results [45]. Although many value ranges can be immediately ignored for consideration, the best way to figure out what values are ideal for each parameter in order to produce optimal results is to simply run the genetic algorithm repeatedly and compare the results. With the many layers of randomization that are inherent to several components of the genetic algorithm, determining appropriate parameters can take a lot of time. In fact, tuning optimal parameter values and weights becomes an optimization problem on its own. In this work, we isolated individual functions and operators to analyze their effects to decide the weights of each function.

4.4.2 Population Model

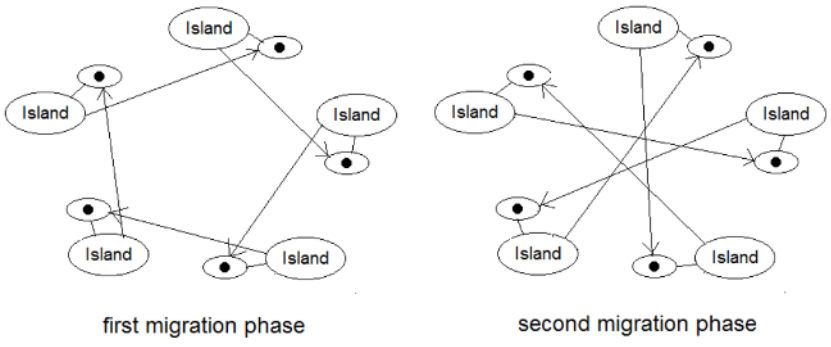


Figure 15: An example of the population model and two migration steps

The population model or island model is one of the most important additions elevated our genetic algorithm into a hybrid genetic algorithm. Its implementation separates the global population into specific subpopulations and swaps sounds between subpopulations throughout the genetic algorithm's runtime. Figure **??** visualizes the representation of the population model as well as the migration operation it periodically performs. The subpopulations or islands are stored in an array. The frequency of the migration operation is a parameter decided by the user. Every time a migration occurs, adjacent islands swap at least one random solution. To ensure that each island trades with different islands each migration, the array that stores the islands shuffles their order so the pairings are different every single migration. This process resembles the circulating ring migration mentioned in [41] but it is not quite identical.

4.4.3 Crowdsourcing Fitness Evaluation



Figure 16: A screenshot of our web-based AB testing

This project utilized crowdsourcing through an interactive website. This component was built by another student named Bradly Landucci as part of his capstone project. Bradly also made contributions to the 4.3.1. Figure **??** is a screenshot of the website's main page. As the screenshot indicates, the website tasks evaluators with A/B testing by generating two sounds to pick between. A database keeps track of how many votings each sound participated in as well as how many of the votes the sound won out of the votings it participated in.

The website and database were crucial for crowdsourcing and adding human evaluations to the genetic algorithm's fitness evaluation. The website was largely built through a combination of Javascript, React and Python. It generates the sounds by taking a solution's information from the database and then synthesizing it using the

`Tonejs` audio library [5] using additive synthesis. Synthesis through `Tonejs` primarily involves plugging the values each solution stores into the audio library's class for sine waves and then playing all the partials at the same time. To make comparisons between sounds easier, the pairs of sounds generated have the same fundamental frequency.

4.4.4 Database and Backend Infrastrucutre

To store the representation, we used a cloud-based database through an API accessed by the client side website. Figure 17 depicts the relations between the user interface, the genetic algorithm and the API. All three components need to communicate with each other to run properly. The database was originally created through MySQL but we eventually moved it over to Amazon Web Services (AWS). The API is still currently run using Flask, which is a web framework built in Python. The frontend uses React to handle the client side.

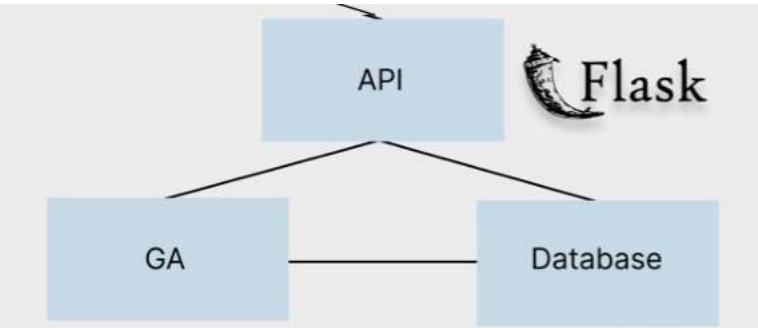


Figure 17: The relations between the backend and frontend of our project

The database was necessary to store solutions, allowing the GA to preserve solutions and resume to advance generations at any time rather than running the genetic algorithm in a single execution. As mentioned earlier, the database is deployed through AWS and communicates with the website in order to share and update solutions as well as to keep track of the crowdsourcing votes each solution receives.

In the initial design, the GA ran locally through several Python scripts. The transistion to the website and database approach forced the representation to change in order to accommodate the new setup. Previously, the GA ran in a single continuous execution and restarted every time the experiment was initiated. The GA is tasked to find novel instrument sounds, so ideally we desire the GA to run indefinitely without a preset number of generations. This required a change in design since we needed to consider real-time interactions between the crowdsourcing aspect of the website,

[5] https://tonejs.github.io/

communication between the GA, website and database to pick and store solutions in tandem.

Ultimately, we redesigned the system to adopt a temporal model to periodically create new populations and produce new generations after a sufficient number of crowdsourcing users voted through the website. Each time a user participates in the vote, the website retrieves a pair of sounds from the database. For each new generation, the GA retrieves solutions from the database and simultaneously adds a new generation of instruments created by the process described in Section 4.

4.5 Validating and Verifying Results

Although we can listen to the instrument sounds in order to judge the quality of the results, listening alone provides only limited insight to the overall quality and diversity of the solution set. Other techniques were instrumental in verifying that the process was working as expected and identifying what needed to be fixed or improved. In this section, we describe the most important verification techniques we used to ensure our system worked as intended.

Unit Tests
Even when considering only basic and small representations of the sounds with the genetic algorithm, the representations are complicated enough that verifying the validity of the representation is still a necessary step. Many errors can be difficult to identify without a systematic approach to identify and debug them. To increase our confidence in the verification step, we designed many unit tests. Unit tests test various functions by allowing users to control the input to functions and verify that the output aligns with what we expected. We were able to identify many potential issues using unit tests that would have otherwise been difficult to identify and correct. This enabled us to improve many facets of our process and ensure the representation was working as intended.

Visualization
As we previously mentioned, the act of listening to the sounds alone is an easy way to check a sound on a superficial level, but most untrained listeners are unable to perceive all the intricacies layered in the sound. For this reason, the use of visualization tools was an important technique that allowed us to identify which frequencies stood out for each sound and to better understand the distribution of the harmonics [18]. During our development, we discovered several other issues and weaknesses of our process only after examining visualizations of the generated instruments. Early in our development process we observed that many of the graphs the GA produced looked strange which helped us identify problems with our implementation. These clues eventually led us to discover that the generated audio often clipped. The various

graphs indicated attributes that should not be possible, such as those with more than ten partials.

We primarily relied on spectrograms and waveforms to brainstorm ideas to make sounds more diverse and to find more hidden issues. A spectrogram visualizes the spectrum of frequencies a sound has across time[39]. This shows which part of the sounds are the most prominent and how the sound varies over time. A spectrogram is a 2D graph where the x-axis represents time, the y-axis indicates the frequencies. An example spectrogram of a trombone sound is given as Figure 18. These images visualize a third dimension in the form of volume/amplitude that is represented by the color or brightness of the graphs. Generally the lighter the color, the higher the amplitude. A graph set on a black background will consist of several peaks that represent the most prominent frequencies over time where the brighter peaks are louder.

Just as a spectrogram provides insight about the frequencies and amplitudes over time, a waveform visualizes the amplitudes of a sound over time [18]. Much like spectrograms, studying the waveforms of our generated sounds helped us to identify ways to diversify the sounds and improve our algorithm. In a visualization of a waveform, a sound that has suffered from clipping was identifiable if the amplitudes ever reached 1.0 or -1.0. It also clearly showed that early versions of our genetic algorithm primarily created bell tones. This indicated that certain fitness functions such as `error_off_partials` dominated the overall fitness calculation and heavily rewarded bell tones. Both techniques revealed errors and indicate what improvements could be made to further diversify sounds.

Figure 18 is a histogram generated from a note played by a Trombone. Here we can clearly see the individual frequencies and their volume.

Figure 19 is a waveform generated from a note played by a Trombone. Here we can clearly see how the volume of the sound changes over time.

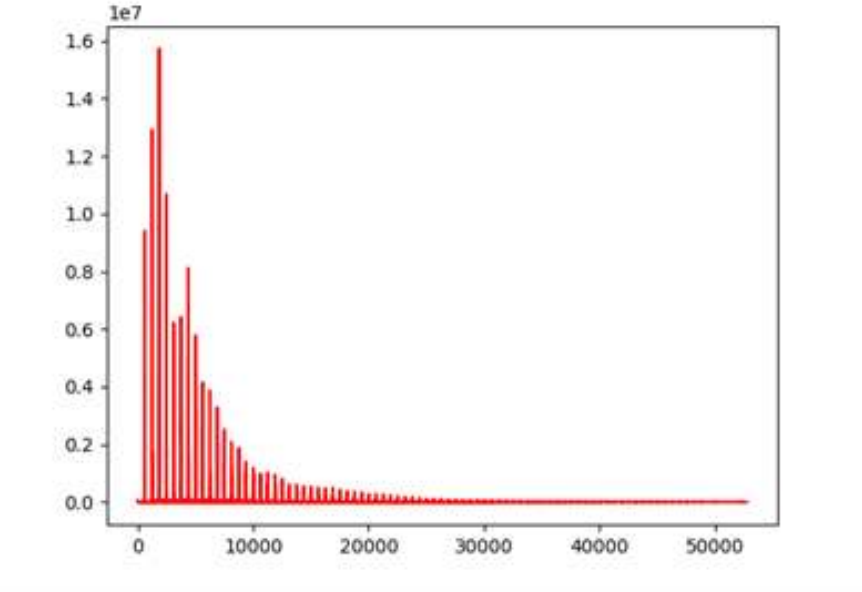


Figure 18: A histogram of a trombone note

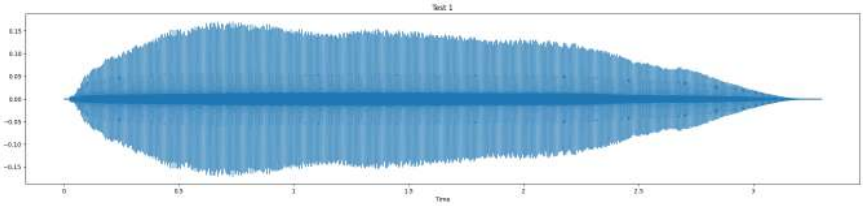


Figure 19: A waveform of a trombone note

5 Exemplar Walkthrough

This section further explains the genetic algorithm in greater detail by walking through each of the important operations with direct examples. To best navigate this complex process and for simplicity, this section will primarily focus on two solutions that will be referred to as `Solution 1` and `Solution 2` throughout this section. There are still eight total solutions within the subpopulation presented in this example.

5.1 Initialization

The start of the genetic algorithm begins with initializing a population. Each candidate solution stores its attributes as value in arrays and all the data is randomly generated within a range set by the user. In this work, each solution has multiple arrays to store all the data of the representation. Specifically, each solution has an array for its frequencies, amplitudes, weights for fitness functions, attack values, decay values, sustain values, and release values. Figure 20 depicts `Solution 1` and all the initialized values in each array.

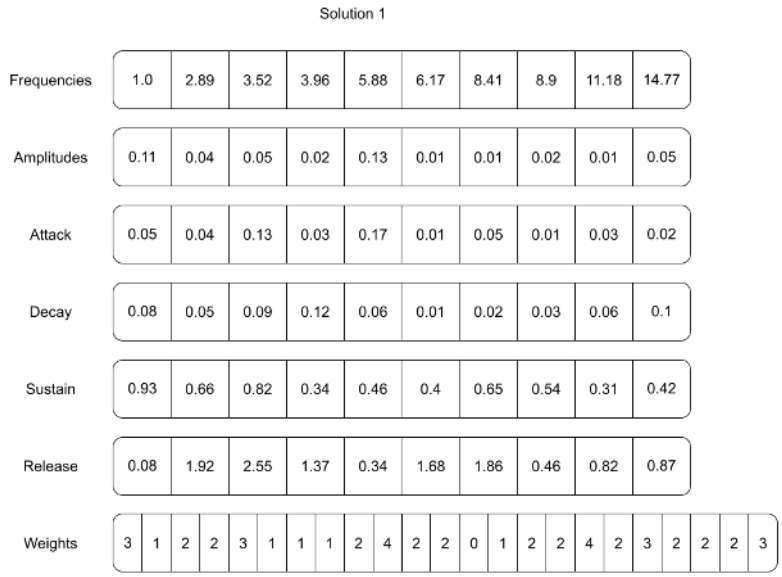


Figure 20: The frequency, amplitude ADSR envelope and weight arrays for `Solution 1`

The current version of the genetic algorithm employs 23 fitness functions which is why the weight array contains 23 values instead of 10 values like the other arrays. The rest of the data arrays have a length of 10 to correspond to the 10 partials stored

by each solution. All the values are randomly generated within a range that was either determined through unit testing and analyzing the fitness functions or inherent restrictions such as only being able to set the sustain value from 0.0-1.0 since it acts as a percentage. For readability, the weights are rounded to whole numbers and the rest of the values are rounded to two decimal places while the actual representation stores all of them with five decimal places. The number of digits of precision can be adjusted by the user.

5.2 Evaluation

Once all the sounds of the subpopulation have been initialized, the next step is the evaluation phase. For this example, we will focus on only one fitness function even though in practice, all 23 evaluate the solutions. Once again, only `Solutions 1` and `2` will have their scores explicitly shown for this walk through. The single fitness function that we will focus on is 4.3.1. To quickly review, this function checks each partial of each sound and lowers their score for each amplitude that is above a threshold set by the user. This function filters out sounds that are too loud to the point that the audio clips, resulting in harsh and garbled sound. For this example, the threshold is set to 0.18 to determine if a partial is too loud. Figure 21 illustrates the amplitude arrays for `Solutions 1` and `2` and their resulting fitness scores after evaluation by `punish_loud_amps`.

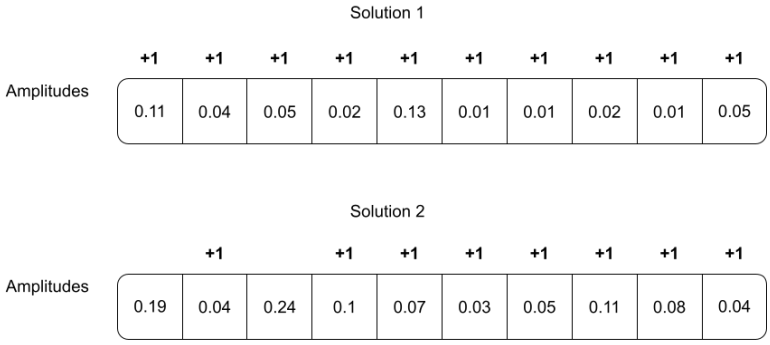


Figure 21: The amplitude evaluation for `Solution 1` and `Solution 2`

As Figure 21 shows, `Solution 1` has ten partials below the volume threshold and `Solution 2` has 8 partials below the threshold. Therefore, their respective fitness scores from this function are 10 and 8 before they are scaled by the weights.

The other component of evaluation is the crowdsourced evaluation. The genetic algorithm queries the database and checks the number of rounds of voting for each sound and how many votes each sound has won. In this case, these are newly generated sounds and they have not yet had the chance to participate in any votes, so the crowdsourced evaluation does not factor into their scores yet. Once they have been voted upon, the crowdsourced evaluation creates a percentage using the votes won and dividing it by the total number of rounds of votes involving this sound. This percentage is then scaled using a user defined weight and then added to each sound's mathematically calculated fitness score. Table 1 provides an example of how crowdsourced evaluation influences the score. Since the sounds have not participated in any votes yet, the numbers are invented for this example.

Sound	Votes Won	Rounds	Win Rate	Weight	Fitness Score
Solution 1	6	8	0.75	10	7.5
Solution 2	5	10	0.5	10	5.0

Table 1: The voting variables used to calculate the crowdsourced fitness score

5.3 Selection

Following the evaluation stage finishing, the next stage is selection. For this example, we choose tournament selection. The scores for the six other solutions will be included despite not being explicitly shown before. Their scores are needed to present the entire tournament bracket. In a subpopulation with eight sounds, only four parents are needed to create the next generation of eight solutions. This means only one round of the tournament bracket is necessary to select the parents. This single tournament round is illustrated in Figure 22.

Note how the best solutions are not all selected. `Solution 1` and 2 had relatively high scores, so they were naturally picked but `Solution 3` had a relatively low score and was still selected as a result of getting paired up with an even lower scoring solution.

5.4 Crossover

After the selection stage finishes, the next phase is crossover. For this example, we will apply a deep uniform crossover. We will also just examine the crossover between `Solutions 1` and 2, but crossover occurs with solutions 2 and 3, solutions 3 and 4, and solutions 1 and 4. Each crossover results in two new solutions created, resulting in a new generation of eight solutions in total. All the frequency, amplitude, weight and ADSR arrays of each solution undergo crossover but Figure 23 only explicitly

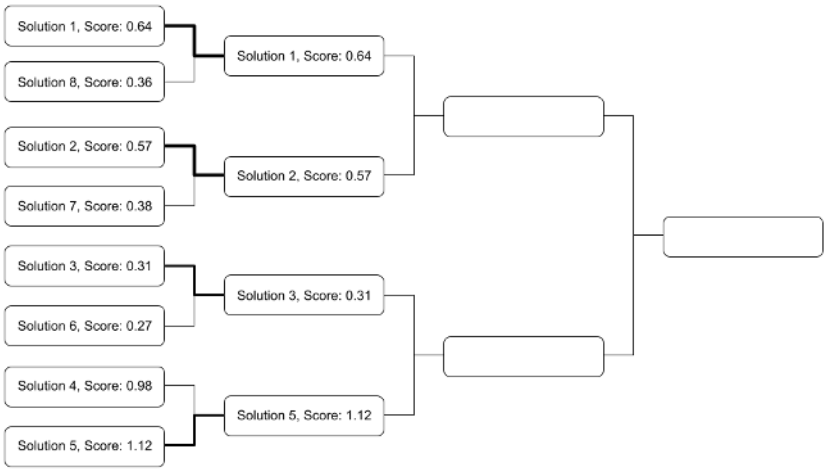


Figure 22: The amplitude evaluation with tournament selection for `Solutions 1` and 2

presents crossover for the frequency array. Figure 23 depicts the two new resulting solutions from the deep uniform crossover between solutions 1 and 2. These new solutions from here on out are referred to as `Solutions A` and `B`.

In Figure 23, `Solution 1`'s data is signified by bolded values while `Solution 2`'s data is signified by italicized values. The partials of solutions A and B are determined through coin flips to decide from which parent each partial should get its data.

5.5 Mutation

After the crossover stage completes, the next stage is mutation. For this example, only solution A and B's mutation is explicitly shown. Mutation occurs randomly, so not every solution is affected by mutation. For the sake of demonstration, both the new solutions are selected for mutation in this example. For solution A, the entire attack value array is chosen and is scaled up by a coefficient of 1.5 while for solution B, the entire amplitude array is scaled down by a coefficient of 0.6. Figure 24 visualizes these transitions and provides the new arrays resulting from the mutation.

5.6 Population Model

Once mutation finishes, there are a few parameters that are checked to determine if additional events occur or if the genetic algorithm can stop running. The first of these events that could occur is migration between populations. So far, the population of 8

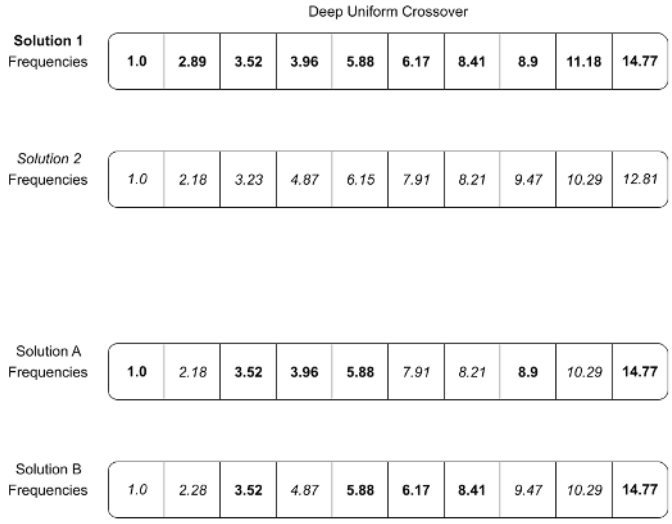


Figure 23: Crossover between `Solution 1` and `Solution 2`

solutions has been completely self-contained, but the genetic algorithm controls other subpopulations. The population model periodically signals to the subpopulations to trade solutions to infuse more genetic diversity into these isolated groups. Once a looping threshold set by the user is met, the populations trade solutions and then resume executing the genetic algorithm. For the sake of this example, there are four other subpopulations that trade one solution from each subpopulation to another subpopulation.

Figure 25 illustrates four subpopulations trading four solutions. In this case, the migration forms pairs of islands and tasks them to swap a random pair of solutions. Each solution is denoted by the island it came from followed by the solution index it had within its original subpopulation.

5.7 Stopping the Genetic Algorithm

The last steps in a generation the algorithm executes are either the genetic algorithm continues looping or it stops running altogether. Much like for the migration event, the stopping criteria is set by the user depending on how many generations they want the genetic algorithm to simulate. For our eventual deployment of this system, the algorithm effectively runs indefinitely as explained in the 4.4.4 subsection. This concludes the example walkthrough.

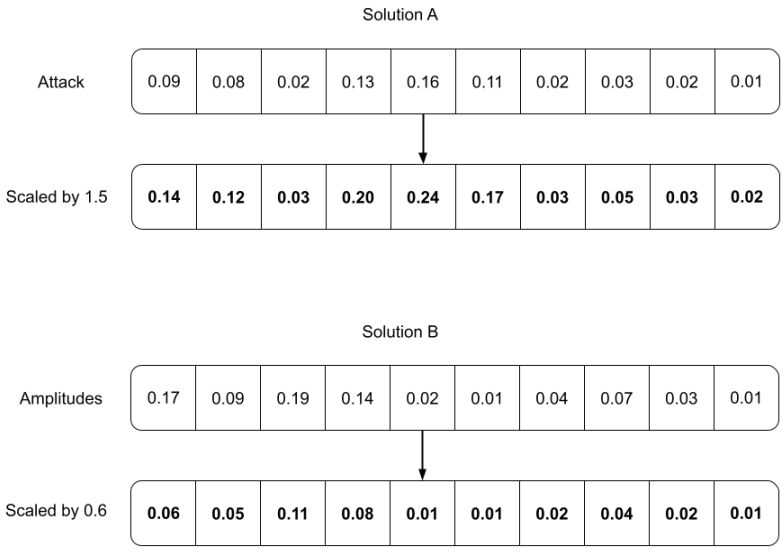


Figure 24: Mutation for solutions A and B

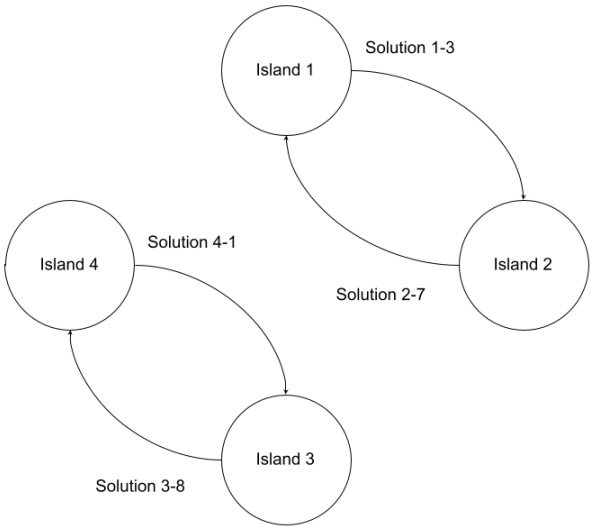


Figure 25: Migration between four subpopulations

6 Discussion

This project designed a novel implementation of a hybrid genetic algorithm by incorporating human crowdsourcing and the population island model. The base genetic algorithm performed evaluation, selection, crossover and mutation to evolve new generations of solutions. A website and database were designed and deployed to conduct A/B testing with human evaluators, tasked to assess two randomly selected solutions in each vote. The population model created subpopulations and periodically traded solutions between different subpopulations to encourage diversity across the total solution set. In this section, we discuss the contributions of this project, the limitations this project encountered, areas for potential future research, and some applications for the findings from this project.

6.1 Contributions

This project presents a hybrid genetic algorithm that generates sounds that mimic synthetic musical instruments. Currently, the generated instruments tend to be very similar to one another and arguably simpler than most acoustic instruments. Nevertheless, the hybrid algorithm's foundation still proved to be more effective than the basic genetic algorithm, demonstrating the merit and promise of this approach.

In the 3.2 section, we observed that there has been far more research focused on applying artificial intelligence to music composition compared to applying it to sound synthesis. This project aimed to explore this research task informed by with some additions commonly used in other music composition studies. Through this project, we have established the foundation for a hybrid genetic algorithm that utilizes the population model, crowdsourcing, and additive synthesis in conjunction to create interesting synthetic instrument sounds. To the best of our knowledge, this approach has not been previously attempted. The hybrid approach used in this project opens up numerous possibilities for exploring the sound space in the sound synthesis field. To provide additional context, we will quickly cover important discoveries stemming from past works **??** to better underscore the expansions we accomplished.

Early experiments like Johnson's [17] set a precedent for genetic algorithms use in sound synthesis. Johnson's system also used an interactive voting interface with several options for evaluators to listen to and pick from at once. Later projects focused on more specific tasks such as Lai [25] and Marcret [30]. These experiments focused on recreating target sounds instead of exploring the sound space like Johnson and were able to reasonably recreate a handful of specific sounds. Other experiments like Chinen [7] explored other synthesis techniques besides additive synthesis to test their effectiveness.

However, most of these experiments were concerned with testing the effectiveness of the baseline genetic algorithm. Building the basic genetic algorithms proved to be difficult on its own considering weighting the parameters could be quite complex and time-consuming by itself. In other fields such as music composition, researchers further developed their algorithms with additions like crowdsourcing as seen in experiments like Casal's [6] once it was established that the baseline algorithm was an effective, replicable method. This project sought to expand the typical sound synthesis genetic algorithm in a similar fashion with the addition crowdsourcing and the population model techniques to investigate if these techniques would achieve similar levels of success with sound synthesis. To the best of our knowledge, there has not been previous research with sound synthesis and genetic algorithms that also employed crowdsourcing and the population model, making this experiment unique.

In our approach, we begin with randomly generated noise and quickly filter it into pleasing instruments such as bells. We were able to confirm our initial hypothesis that we could combine the population model, crowdsourced evaluation, and mathematical heuristics to synthesize instrument-like sounds with a genetic algorithm. These extended techniques used in this project noticeably improved the diversity of the sounds thanks to the additional randomization and refined evaluation. We still encountered similar issues described by other researchers [31] such as premature convergence and homogenous timbre, but the foundation of our approach proved effective enough to warrant further exploration and expansion.

6.2 Limitations

Despite our success in generating novel instrument sounds using a genetic algorithm, our approach has several limitations that we will address in future work. The subjective nature of the problem created several difficulties that our approach was unable to fully overcome. Genetic algorithms are normally very sensitive to initialized and user set values like weights. Finding ideal initial values was an optimization in itself and one that this project was not able to fully address. The high number of parameters meant it was difficult to tune the values. This was made more difficult because of the creative nature of our approach in which it is difficult to objectively determine which setting should be considered optima in this search problem. As such, this project was unable to fully explore the full range of values for the weights and parameters which should be explored in future endeavors to improve the quality and range of the sounds.

Additionally, the maintenance of our database and website required further monetary investment to fully operate the large-scale experiment this project's approach was designed for. Crowdsourcing is ideal for gathering a diverse set of subjective evaluations in a potentially short time, but we need a large number of users to participate and this web traffic increases the costs and resources necessary to host out backend

of the website and the database. The website and database frequently communicate since the website relies on the database to store the data of the sounds. The large cost that would have been required to crowdsource this project to the full extent forced us to scale down the scope and have a smaller pool of evaluators. Both these factors limited our ability to perform a large-scale experiment and constrained our ability to fully explore the potential of our approach.

6.3 Future work

Primarily, this project serves as a foundational experiment and was designed to test the feasibility of an approach that extends a genetic algorithm with the population model and crowdsourcing for sound synthesis. In this work, we provide a solid foundation in which researchers can further investigate and expand upon my approach. In future work, we intend to deploy a large-scale crowdsourcing experiment to produce a more diverse set of instruments.

As mentioned in the 6.2 section, the process of finding appropriate weights and parameters is an optimization problem in itself and in future work we will empirically tune more parameters to better determine which default values improve our process. Furthermore, we intend to explore other population model migration strategies to explore their effect and potentially produce improved results.

Other strategies we intend to explore as future improvements are primarily focused on addressing the bell tones homogeneity we encountered. One strategy is to increase the randomness more by deemphasizing the mathematical rules to have less influence in order to give sounds that score lower from the current fitness functions a higher chance of being selected, resulting in more diversity. Another improvement would be the addition of more rules that reward different attributes by adding rules to counter bell tones and the traits they have.

The opposite change could also improve results. Removing mathematical rules and transitioning to an entirely human-based fitness evaluation would provide greater insight and assessment. We intend the objective fitness rules as a tool to quickly filter garbled and unpleasant noise in early generations and we intend to adapt our system to rely on them less in higher numbered generations. The importance of the built-in fitness functions can be reduced without breaking the functionality of the algorithm.

In addition to the work presented here, there are other sound synthesis models that are worth exploring. Additive synthesis is a simple and straightforward sound synthesis technique, but it is arguably less powerful than other methods like physical modelling. The novel timbres that other synthesis techniques provide are able produce instruments that additive synthesis simply cannot. The use of other evolutionary

techniques such as the particle swarm algorithm used by [42] could be applied to this particular problem and are worth investigating. Additionally, machine learning techniques could prove to be an effective method for sound synthesis. They have been applied to numerous applications in the music composition world, and many of these techniques would be well suited to sound synthesis too.

6.4 Applications

In this work, I was able to produce very simple instrument representation, but after further refinement of our approach and algorithm, we envision a system in which musicians and composers can develop new instruments. Future exploration could expand off the foundation we set to run large-scale crowdsourcing to collect a large volume of evaluations from internet users to generate an extensive collection of unique instrument sounds. This library of sounds could be stored on a website that allows composers and musicians to download any of the unique instruments. Providing electroacoustic composers with more options for their instrumentation is the most direct and relevant application of this work.

However, synthesizing new sounds could be useful in a variety of other fields. Warning signals are critical to inform nearby individuals of dangerous and problematic incidents. Alarms and sirens are some of the more common warning signals because of their ability to quickly get everyone's attention. However, their effectiveness can drop over time due to alert fatigue, the tendency for workers to become desensitized to the alarms such that they fail to respond to the alarm's warning. New siren sounds would mitigate alert fatigue by changing the warning sound which might lower the chance that nearby individuals would tune out the alarm altogether. Police sirens already use this strategy. There are actually several different sirens that police officers can use depending on the context. The main three used are yelpers, wailer and piercers. The chief difference between the three is how quickly they are meant to get the attention of nearby drivers and pedestrians. Having a variety of sirens reduces alert fatigue and ensures that people respond appropriately to the warnings.

There are similar uses in less critical situations such as a prompting jingle in a train station. Past research has explored music and its effect on focus, mood and sleep. Additional instrumentation discovered through sound synthesis could further explore music and its effect in the aforementioned areas. Further exploration of the sound space through sound synthesis could contribute to new discoveries about the impact music has in fields that have not been considered previously. Future exploration could also evolve the sound design in movies and video games by providing designers with more tools and inspiring them to expand their approaches [47].

6.5 Conclusion

In this work we present an approach that generates novel synthetic instrument sounds using a hybrid genetic algorithm. Genetic algorithms have been explored in past sound synthesis research studies, but our novel approach incorporates crowdsourcing and the population model together, forming an approach distinct from any previous work. In this process we revealed new challenges such as convergence to bell-like tones. Despite the built-in randomization and human evaluation, many sounds ended up resembling some variation of a bell tone. This issue likely resulted from overly zealous fitness rules that reward sounds with specific patterns of harmonics. The parameters and weights set likely gave too much influence to certain fitness helpers that overly rewarded the specific patterns. The biggest contributors to these patterns were likely some of the overly specific rules focused on amplitudes and frequencies. The timbre of certain bell tones contain harmonics at precise integer ratios to the fundamental frequency. A lot of the fitness functions that tried to encourage common instrument sounds ended up primarily producing bell tones without further developing them into novel instruments in later generations. To the best of our knowledge, our approach is the first genetic algorithm to combine the population model and crowdsourcing to explore the sound synthesis space. With this work, we contribute a creative and baseline approach to sound synthesis that merits further expansion and investigation.

References

[1] Alina Abraham. "The Zone of Musical Creativity: Harmonic Series Structures-from Pictorial Representation to a Method of Teaching". In: *2012 Seventh International Conference on Knowledge, Information and Creativity Support Systems*. IEEE, 2012, pp. 207–210.

[2] L Rocha de Azevedo Santos, Carlos Nascimento Silla Jr, and MD Costa-Abreu. "A methodology for procedural piano music composition with mood templates using genetic algorithms". In: (2021). Publisher: IET.

[3] Todor Balabanov. "Distributed evolutional model for music composition by human-computer interaction". In: *Proceedings of International Scientific Conference UniTech*. 2015, pp. 389–392.

[4] Juan José Burred, Axel Robel, and Thomas Sikora. "Dynamic spectral envelope modeling for timbre analysis of musical instrument sounds". In: *IEEE Transactions on Audio, Speech, and Language Processing* 18.3 (2009). Publisher: IEEE, pp. 663–674.

[5] Marcelo F Caetano, Jônatas Manzolli, and Fernando J Von Zuben. "Interactive Control of Evolution Applied to Sound Synthesis." In: *FLAIRS Conference*. 2005, pp. 51–56.

[6] David Plans Casal. "Crowdsourcing the corpus: Using collective intelligence as a method for composition". In: *Leonardo Music Journal* 21 (2011). Publisher: MIT Press 55 Hayward St., Cambridge, MA 02142-1315, USA journals-info@ mit. edu, pp. 25–28.

[7] Michael Chinen and Naotoshi Osaka. "Genesynth: Noise band-based genetic algorithm analysis/synthesis framework". In: *ICMC*. 2007.

[8] Iyad Abu Doush and Ayah Sawalha. "Automatic music composition using genetic algorithm and artificial neural networks". In: *Malaysian Journal of Computer Science* 33.1 (2020), pp. 35–51.

[9] Jesse Engel et al. "Neural audio synthesis of musical notes with wavenet autoencoders". In: *International Conference on Machine Learning*. PMLR, 2017, pp. 1068–1077.

[10] Andy Farnell. *Designing sound*. Mit Press, 2010.

[11] Naohiro Fujinuma and Samuel E Lofland. "Physics-Based Human-in-the-Loop Machine Learning Combined with Genetic Algorithm Search for Multicriteria Optimization: Electrochemical CO2 Reduction Reaction". In: *Advanced Intelligent Systems* (2023). Publisher: Wiley Online Library, p. 2200290.

[12] L Haldurai, T Madhubala, and R Rajalakshmi. "A study on genetic algorithm and its applications". In: *International Journal of Computer Sciences and Engineering* 4.10 (2016), p. 139.

[13] Aymen Hamrouni et al. “Toward collaborative mobile crowdsourcing”. In: *IEEE Internet of Things Magazine* 4.2 (2021). Publisher: IEEE, pp. 88–94.

[14] Martin Henz, Shang-Hui Koh, and Samyukta Sounderraman. “Teachable moments in functional audio processing”. In: *Proceedings of the 2021 ACM SIGPLAN International Symposium on SPLASH-E*. 2021, pp. 65–70.

[15] Andrew Horner, James Beauchamp, and Lippold Haken. “Machine tongues XVI: Genetic algorithms and their application to FM matching synthesis”. In: *Computer Music Journal* 17.4 (1993). Publisher: JSTOR, pp. 17–29.

[16] Shiyuan Jin, Ming Zhou, and Annie S Wu. “Sensor network optimization using a genetic algorithm”. In: *Proceedings of the 7th world multiconference on systemics, cybernetics and informatics*. 2003, pp. 109–116.

[17] Colin G Johnson. “Exploring the sound-space of synthesis algorithms using interactive genetic algorithms”. In: *Proceedings of the AISB’99 Symposium on Musical Creativity*. Society for the Study of Artificial Intelligence and Simulation of Behaviour, 1999, pp. 20–27.

[18] Shahrul Kadri et al. “Physics demonstration of sound waves using Visual Analyser”. In: *Lat. Am. J. Phys. Educ.* 7 (2013), pp. 10–15.

[19] Maximos A Kaliakatsos-Papakostas, Andreas Floros, and Michael N Vrahatis. “Interactive music composition driven by feature evolution”. In: *SpringerPlus* 5.1 (2016). Publisher: Springer, pp. 1–38.

[20] Sourabh Katoch, Sumit Singh Chauhan, and Vijay Kumar. “A review on genetic algorithm: past, present, and future”. In: *Multimedia tools and applications* 80 (2021). Publisher: Springer, pp. 8091–8126.

[21] Rainer Kelz, Sebastian Böck, and Gerhard Widmer. “Deep polyphonic adsr piano note transcription”. In: *ICASSP 2019-2019 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2019, pp. 246–250.

[22] Jessica F Keup. *Computer Music Composition using Crowdsourcing and Genetic Algorithms*. Nova Southeastern University, 2011.

[23] Oliver Kramer. “Genetic algorithms”. In: *Genetic algorithm essentials*. Springer, 2017, pp. 11–19.

[24] Manoj Kumar et al. “Genetic algorithm: Review and application”. In: *Available at SSRN 3529843* (2010).

[25] Yuyo Lai et al. “Automated optimization of parameters for FM sound synthesis with genetic algorithms”. In: *International Workshop on Computer Music and Audio Technology*. Citeseer, 2006, p. 205.

[26] Annu Lambora, Kunal Gupta, and Kriti Chopra. "Genetic algorithm-A literature review". In: *2019 international conference on machine learning, big data, cloud and parallel computing (COMITCon)*. IEEE, 2019, pp. 380–384.

[27] Jeppe Veirum Larsen and Hendrik Knoche. "States and sound: modelling user interactions with musical interfaces". In: *New Interfaces for Musical Expression 2017*. NIME, 2017.

[28] Che-Yuan Liang et al. "Musical Offset Detection of Pitched Instruments: The Case of Violin." In: *ISMIR*. 2015, pp. 281–287.

[29] Shiguang Liu and Dinesh Manocha. "Sound Synthesis". In: *Sound Synthesis, Propagation, and Rendering*. Springer, 2022, pp. 7–28.

[30] Matthieu Macret, Philippe Pasquier, and Tamara Smyth. "Automatic calibration of modified fm synthesis to harmonic sounds using genetic algorithms". In: (2012). Publisher: Simon Fraser University.

[31] Dragan Matić. "A genetic algorithm for composing music". In: *Yugoslav Journal of Operations Research* 20.1 (2016).

[32] Stephen McAdams and Meghan Goodchild. "Musical structure: Sound and timbre". In: *The Routledge companion to music cognition*. Routledge, 2017, pp. 129–139.

[33] Usama Mehboob et al. "Genetic Algorithms in Wireless Networking: Techniques, Applications, and Issues". In: *Soft Computing* 20 (June 2016). DOI: 10.1007/s00500-016-2070-9.

[34] Eduardo R Miranda and Duncan Williams. "Artificial intelligence in organised sound". In: *Organised Sound* 20.1 (2015). Publisher: Cambridge University Press, pp. 76–81.

[35] Seyedali Mirjalili. "Genetic algorithm". In: *Evolutionary algorithms and neural networks*. Springer, 2019, pp. 43–55.

[36] Seyedali Mirjalili et al. "Genetic algorithm: Theory, literature review, and application in image reconstruction". In: *Nature-inspired optimizers* (2020). Publisher: Springer, pp. 69–85.

[37] Anastasia Natsiou and Seán O'Leary. "Audio representations for deep learning in sound synthesis: A review". In: *2021 IEEE/ACS 18th International Conference on Computer Systems and Applications (AICCSA)*. IEEE, 2021, pp. 1–8.

[38] Rolf Nordahl, Luca Turchet, and Stefania Serafin. "Sound synthesis and evaluation of interactive footsteps and environmental sounds rendering for virtual reality applications". In: *IEEE transactions on visualization and computer graphics* 17.9 (2011). Publisher: IEEE, pp. 1234–1244.

[39] Panos Photinos. *The Physics of Sound Waves ; Music, instruments, and sound equipment*. 2021.

[40] Edward Pó{\textbackslash}lrolniczak and Micha{\textbackslash}l Kramarczyk. "Comparison of singing voice quality from beginning of the phonation and in the stable in the case of choral voices". In: *2019 Federated Conference on Computer Science and Information Systems (FedCSIS)*. IEEE, 2019, pp. 349–353.

[41] Brigitte Rafael, Michael Affenzeller, and Stefan Wagner. "Application of an island model genetic algorithm for a multi-track music segmentation problem". In: *International Conference on Evolutionary and Biologically Inspired Music and Art*. Springer, 2013, pp. 13–24.

[42] Diemo Schwarz. "State of the art in sound texture synthesis". In: *Digital audio effects (DAFx)*. 2011, pp. 221–232.

[43] Nermin Naguib J Siphocly, El-Sayed M El-Horbaty, and Abdel-Badeeh M Salem. "Top 10 artificial intelligence algorithms in computer music composition". In: *Int. J. Com. Dig. Sys* 10.1 (2021).

[44] Xiaotian Su et al. "Implementation of Digital Synthesis in Functional Programming". In: ().

[45] Augoustinos Tsiros. "Sonic Xplorer: A machine learning approach for parametric exploration of sound". In: *Electronic Visualisation and the Arts (EVA 2017)* (2017), pp. 144–149.

[46] Cyrus Vahidi et al. "Timbre space representation of a subtractive synthesizer". In: *arXiv preprint arXiv:2009.11706* (2020).

[47] Mariano Vargas-Santiago et al. "Complementing solutions to optimization problems via crowdsourcing on video game plays". In: *Applied Sciences* 10.23 (2020). Publisher: MDPI, p. 8410.

[48] Zachary Wallmark and Roger A Kendall. "Describing sound: The cognitive linguistics of timbre". In: *The Oxford handbook of timbre. Advance online publication. New York, NY: Oxford University Press. https://doi. org/10.1093/oxfordhb/9780190637224.* 14 (2018).

[49] Ferry Wahyu Wibowo. "The detection of signal on digital audio synthesizer based-on propeller". In: *Advanced Science Letters* 23.6 (2017). Publisher: American Scientific Publishers, pp. 5472–5475.

[50] Jim Woodhouse, David Politzer, and Hossein Mansour. "Acoustics of the banjo: measurements and sound synthesis". In: *Acta Acustica* 5 (2021). Publisher: EDP Sciences, p. 15.

[51] Xingjiao Wu et al. "A survey of human-in-the-loop for machine learning". In: *Future Generation Computer Systems* 135 (2022). Publisher: Elsevier, pp. 364–381.

[52] Fabio Massimo Zanzotto. “Human-in-the-loop artificial intelligence”. In: *Journal of Artificial Intelligence Research* 64 (2019), pp. 243–252.