Overview

The idea for my final project is to use the ideas from evolutinary algorithms to create music. Mainly this will be treated as a proof-of-concept. The different effect mutations and population size has on the fitness of the songs will be analysed. The efficiency of using evolutionary algorithms to generate music for human enjoyability will also be explored. This will be done though the enjoyability of listening to the songs; since this is a proof-of-concept project, this will be sufficient.

Methods

Representation of a Song

To work with evolutationary algorithms and fitness functions, a new way of representing songs had to be created. This was done using the idea of bitwise storage of information. Figure 1 shows which bits are allocated to each piece of information. Each note needs a track, channel, volume, duration, and pitch.

Track has a value from 0 to 15 and channel has a value from 1 to 16, thus can each be represented by 4 bits. Volume is any integer value to represent the loudness of the notes. It can be changed at any note, and effects every note after it. Retricted to values from 0 to 127 here for sake of simplicity. Duration can be any number of values, but for the sake of easy implementation, it is retricted to multiples of 1/8 notes, going from 1/8 a beat to 4 beats and thus only needs 5 bits; implemented with 8 bits so that more duration possibilities can be added at a later date. Pitch can be any note in the range of possible values, here the range of a standard piano is used, which is 88 possible pitches spanning the frequency range $27.5Hz$, A0, to $4186Hz$, C8. Now that notes can be represented by a 32 bit integer, a series of these integers makes up a song.

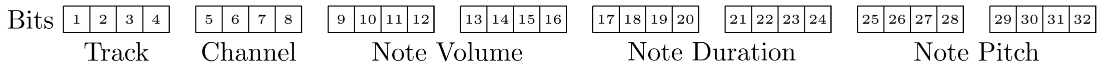

Figure 1: Representation of a single note using 32 bits

Evolving a Song

Algorithm 1 shows the overview for how the songs are evolved. Each step is explained in further detail in the following subsections.

```
Data: runs = 10; popSize = 32; genLength = 500; bestInGen = 150
begin
    for each iterate in runs do
        Initialize a new population of size popSize
        for each generation in genLength do
            Compute the fitness of each member
            Store the top bestInGen members
            Use mutations to create members until there are popSize members
        end
        Convert the top bestInGen members into MP3's
    end
end
```

Algorithm 1: Algorithm for evolving songs; created by Eric Jalbert, implemented in Python.

Initialize Population

The initialization of the population is depicted in Algorithm 2. The channel number and volume are hard coded to simplify the evolutions. Also, the real purpose of channel number is not known at this point.

```
Data: numTracks = 2; numNotes = 100
begin
    Set channel number to 0
    Set volumn to 127 always
    for each i in numTracks do
        Set track number to i
        for each of the numNotes notes do
            Randomize a duration and pitch
            Write the note to file
        end
    end
end
```

Algorithm 2: Algorithm for initializing the population; created by Eric Jalbert, implemented in Python.

Computing Fitness

The algorithm for computing the fitness is described in Algorithm 3. It is based on the compilation of multiple fitness functions that each combine into a robust fitness function. These can be taken externally as arguments to the program so that different features in a song can be added. Extra control is given by the ***fitWeight*** of the function, creating a weighted average of each fitness function.

```
Data: numFuncs = 2
begin
    for each fitness function of numFuncs do
        Check song against fitness function
        Multiple the fitness by fitWeight
        Add this value to totalFitness
    end
    Return totalFitness as the fitness of song
end
```

Algorithm 3: Algorithm for computing the fitness of a song; created by Eric Jalbert, implemented in Python.

Mutations

The mutations for this follow algorithm 4. The crossover is done by taking a random section of notes from one parent and then replacing a random section of notes in the other parent. Here the size of each section is forced to be equal. This could probably be changed so as to encourage more dynamic members in the population, but this was avoided here for simplicity.

Converting to MP3

The conversion of a string of bits into an MP3 is described in algorithm 5. The main thing here is the fact that the series of 32-bit values actually gets converted to a MIDI file first, which can easily be converted to MP3.

```
Data: bestInGen = 5; mnm = 3; popSize = 32
begin
    Store top bestInGen members
    while there are less then popSize members do
        Pick two randoms from the bestInGen members
        Create child by crossing over between two members
        Do mnm single point mutations on child
        Add child to population
    end
    Return new population
end
```

Algorithm 4: Algorithm for mutating the population; created by Eric Jalbert, implemented in Python.

```
Data: songFileName; bitFileName
begin
    Write preamble for conversion to songFileName
    for each note in bitFileName do
        Convert 32-bit into a single note
        Write note to songFileName
    end
    Write postamble for conversion to songFileName
    Convert songFileName to MIDI file
    Convert MIDI file to MP3
end
```

Algorithm 5: Algorithm for converting a series of 32 bits into an MP3; created by Eric Jalbert, implemented in Python.

Results

The results of my work are extremely preliminary. Still, the effect of typical evoluationary parameters had a bit of an effect on the resulting song and fitness. The change in average and maximum fitness for different parameter values can be seen in Figure 2.

The variabilty of the fitness between runs was examined in Figure 3. This was done so that variability could be ruled out as an explaination of the results in Figure 2.

The resulting songs are not completely enjoyable. But the songs with high fitness tend towards a repetative pattern that is bearable. Listen to *repeatative.mp*3. They are also, understandably, dependent on the choice of fitness functions.

Conclusions and Discussion

The results from Figure 2 show that the parameter set does have an effect on the fitness. Mainly that using only one single point mutation and storing a smaller number of top songs per generation is best and that having too many single point mutations and storing too many top songs is bad. These make sense. Because of the nature of the fitness function and how each note needs to work with one another, having too many single point mutations destroys the synergy between notes and lowers the fitness. The storing of a small number of top songs is also sensible. Since the top songs are used as a precursor for the next generation, having a smaller selection forces the children to have sections of notes that have higher fitness. The only issue with this is that the population of songs begin to sound very similar.

The use of evolutinary algorithms to generate songs appears to work. There are a few issues that need to be sorted out before it becomes efficient though. The variability of the parameter set change the fitness of the population of songs, but this does not effect the enjoyability of the music as much as the choice in fitness function. In fact this choice of fitness function is completely what creates

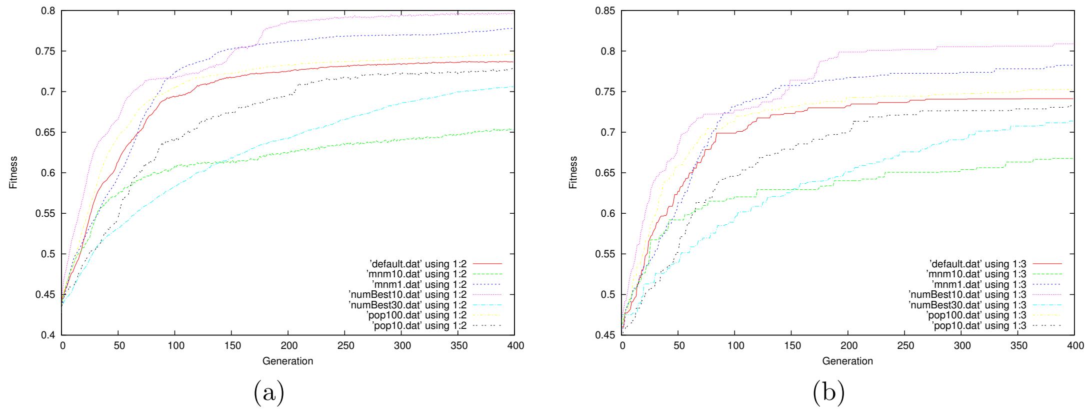

Figure 2: Different parameter settings gives different (a) average fitness and (b) maximum fitness. 'mnmXX' is the number of point mutations; 'numBestXX' is the number of top songs stored each generations; 'popXXX' is the population size.

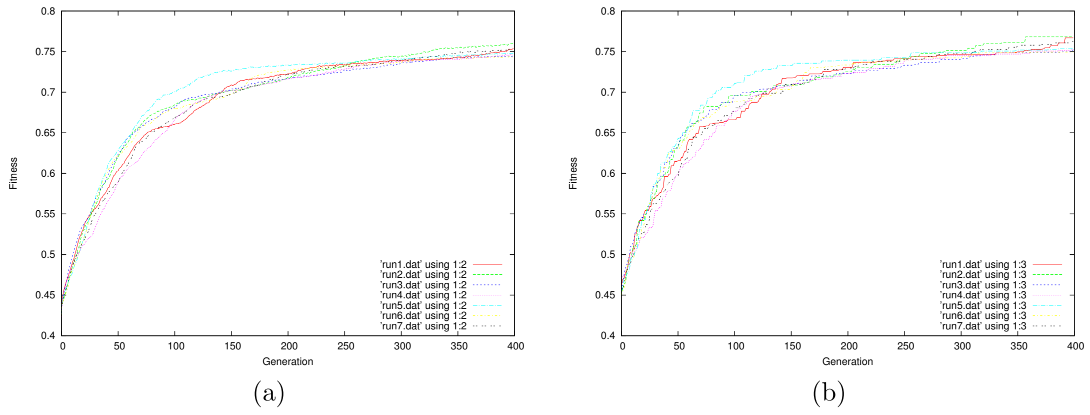

Figure 3: Different runs give similar (a) average fitness and (b) maximum fitness

enjoyable music. Because even though a parameter set may give a high fitness in the population, this does not mean much if the high fitness if the result of a fitness function that rewards poor composition of notes. The main issue with this is that choosing the fitness function is a very difficult ordeal. Since to get a song with many different characteristics would require many fitness functions, which would likely lead to contradictory allocations of fitness giving unsatisfactory results. That being said, the songs that have very high fitness tend to be extremly repeatative. This is because the evolutionary algorithm finds the easiest pattern that can repeated to abuse the fitness function and passing into the next generation. This show of repeatative song with high fitness can be heard in $repeatative.mp3$.

Given the constraint of time, there was only so much work that could be done on this implementation. Some future plans that could be done to improve results would be to look into developing a more robust way of having multiple fitness functions. The idea here would be to have a repository of fitness functions, each encouraging a different aspect in the population of songs. At that point, super fitness functions could be made by grouping up the groups. For example, having a fitness function to force songs in C scale, and another to have low pitch notes play arpeggio's could be smaller fitness functions. Together the two fitness functions could, with other fitness functions, make a super fitness function that creates songs in a jazz style. This kind of variability should be possible, just difficult to actually find the right combination of fitness functions to result in what is needed. One major issue with the current implementation is that the fitness function evaluates the song as a whole, where it should instead compute the fitness for each individual track. This would allow each track to take a particular role in the song (bass, melody, harmory, etc.). This would also open the possibility of instrument specific fitness functions. Here the piano instrument was forced for simplicity, but different instruments could be played on each track, with a specific fitness function for each to match with the instruments playing style.

There are many things that can be done to improve on the current implementation of this evoluationary algorithm for generating songs, but it seems promising that it works to a certain degree.