CS361: Evolutionary Computing and Artificial Life
Yanhan Lyu, Hazel Que
Final Project - Final Report

Introduction

Problem Definition:

We want to find a suitable genetic algorithm for generating short musical compositions. We follow the rules and instructions that human composers use to compose music. By using genetic operators and taking a multi-objective approach to the problem, we enable improvement of the population while maintaining its diversity.

For evolutionary-music (the code provided on GitHub), we set the input parameters in GeneticMusicChorale.java. The parameters include population size, chromosome (genotype) size, number of generations and selection method. The weights of rules that we use to calculate the fitness function is set in ChoraleFitnessFunction.java. The final fitness is calculated as the sum of weight_i * finess_i. (This is used for our baseline tests)

For our own codes, we set the input parameters in EvoMusicRun.java. The parameters include population size, chromosome (genotype) size, number of generations, mutation rate, crossover rate and the selection parameter used for tournament selection. (This is used for results and analysis)

Infrastructure:

Our project is built upon an open-source program on GitHub. You can access the original evolutionary-music here:https://github.com/davidenunes/EvolutionaryMusic
To access our project, you can access it in Yanhan's github:https://github.com/YanhanLyu/EvolutionaryMusic.

Since we are working on a music-related project, we used jmusic extensively. We also took advantage of the existing classes in jgap for general GA components and jfree for drawing the chart. Although our final EvoMusicRun.java is more hand-made than package-reliant.

Approach:

We discovered the evolutionary-music program rather late and decided to incorporate our findings from the papers into the program, instead of modifying the program to reflect the papers. Therefore, our approach has changed significantly as compared to the final proposal.

Representation -

The author of evolutionary-music has a more sophisticated way of representing musical notes. We retained the approach instead of using integers from 0 to 15 to represent individual notes and converting an int string to a melody (the approach we got from Matic's paper[1]).

Each Note is composed of pitch, octave, alteration and duration:

- Pitch: A, B, C, D, E, F, G for note pitches, R for pause.
- Octave: from 0 to 8.
- Alteration: simple pitch alteration; S for sharp, F for flat and N for natural.
- Duration: 1, 2, 4, 8, 16, 32… which starts at whole note.

Each letter representation in the Note (i.e. pitch and alteration) also has a value associated with it (i.e. pitch A is 0, alteration S is 0.5). The values will be used to calculate the distance between two Notes later. The distance calculation is crucial in our genetic algorithm.

Individual Note then is represented as pitch:alteration:octave/duration. For example, **E:N:5/4** is a Note on major E and octave 5, with natural alteration and lasts for a quarter.

However, in evolutionary-music, a melody is not only composed of individual Notes. Instead, it's composed of chords. Each chord has 4 parts: Soprano, Alto, Tenor and Bass and each part corresponds to a Note. Therefore, every single sound (chord) we eventually hear has 4 Notes in it, represented in the following way:

(A:F:4/4 | B:N:3/4 | D:N:4/4 | A:N:3/4)

If we have a chromosome size of 8, our individuals (melodies) will have 8 lines.

Genetic Operators -

Because our representation is rather unique, all genetic operators are hand-made.

- Mutation operator (Notes): Loop through each chord. There are 16 components that can be mutated (4 Notes x 4 components per Note). For a small probability, the component would take on another valid value (current value included).
- Crossover operator (crossing over Notes in chords): Given that each chord consists of 4 Notes, two-point crossover would not be very meaningful. One-point crossover is implemented.
- Switching adjacent Notes in a melody: Loop through a given melody. For a small probability, the current chord switches position with the next chord.

Fitness -

[1] http://elib.mi.sanu.ac.rs/files/journals/yjor/39/yujorn39p157-177.pdf

Evaluating the quality of a melody can be a very subjective task. To build a successful GA program that evolves melodies, we need to move away from the subjectivity and ask ourselves: “are there any quantitative features or melodic properties that define a good melody?” Unsurprisingly, there are multiple objectives we can look out for while evolving a good melody. Evolutionary-music already implemented 10 rules in its fitness function. In Towsey’s paper[2], there are another 21 quantitative features concerning pitch, tonality, melodic contour, rhythm and repeated pattern of motif that can be used to evaluate a melody.

- Fitness for individual rules

In evolutionary-music, each rule can give a fitness score from 0 to 1. The components in a rule share the fitness score equally.

For example, in Avoid Dissonance Rule, we want to compare the distances between the Soprano Notes and Alto Notes, Alto Notes and Tenor Notes and Tenor Notes and Bass Notes separately. Each distance then determines ⅓ of the fitness score.

- Overall fitness function

We can piece as many rules as we want together and use them for the overall fitness evaluation. As a starting point, each rule is assigned the same weight, calculated by (1/# rules). But of course, certain rules are more important than others. For those rules, the weight is really (weight * constant). Therefore, the maximum fitness score often exceeds 1.

Baseline Tests

Evolutionary-music compiles and runs readily when it’s cloned from the GitHub repository. The default values for the GA are:

- Population size: 40
- Chromosome size (number of chords in a melody): 8
- Number of generations: 500
- Selection operator: tournament selection
- Tournament size: 3
- Tournament probability: 0.7

It is worth noting that in the original test, there is no genetic operators in place, i.e. mutation and crossover are not available. Therefore, in the average fitness evolution graph, the fitness only shows an upward increasing trend. Also, the population is filled

[2] http://eprints.qut.edu.au/169/1/towsey.pdf

with randomly generated individuals, although there are certain constraints - each Note can only take on a certain octave range and last for a quarter Note.

After a run, we obtained a fittest individual with a fitness value of ~1.29 (out of a full score of 1.42). The individual's alleles are as follows:

[(F:F:4/4 | C:N:4/4 | G:N:3/4 | C:N:3/4),
(G:N:4/4 | B:N:4/4 | G:N:3/4 | D:N:2/4),
(G:N:4/4 | D:N:5/4 | G:N:4/4 | B:N:4/4),
(G:N:4/4 | E:N:5/4 | C:N:4/4 | C:N:3/4),
(F:N:4/4 | C:N:4/4 | A:N:4/4 | E:S:2/4),
(E:S:4/4 | D:N:4/4 | B:N:3/4 | B:N:2/4),
(F:N:4/4 | F:N:3/4 | A:N:3/4 | C:N:2/4),
(G:N:4/4 | E:N:3/4 | C:N:3/4 | C:N:2/4)]

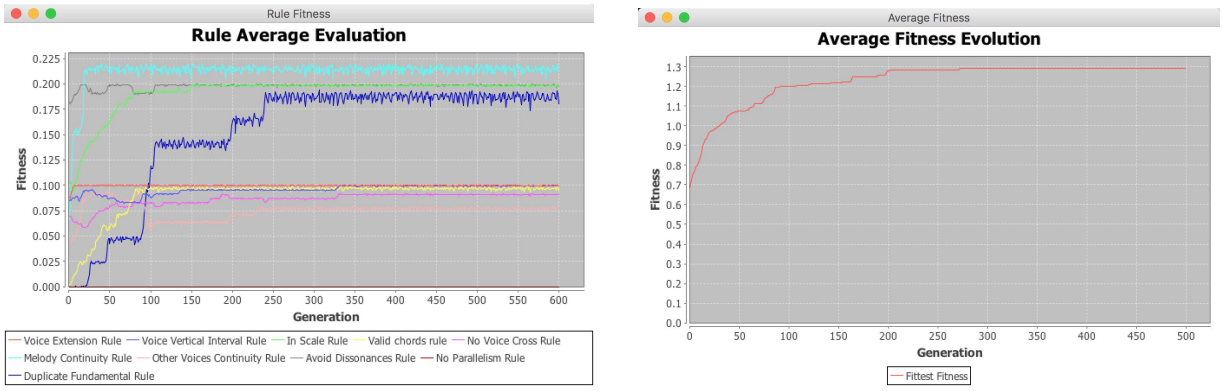

Figure 1: graphs for the default run

The resulting melody, although has a high score, does not actually sound like a melody human beings are familiar with, which means that the existing project does not work so well. Before implementing the genetic operators and advanced GAs, we decided to play with the project semi-manually and see how we should tailor our mutation/crossover methods to improve the melodies.

Experiment 1: Duration

Currently, the melody sounds monotonous because each Note lasts for the same amount of time (and a quarter Note is pretty long!) and there's no variation in pace. I changed the range of duration values allowed to be taken by the Notes to powers of 2s from 4 to 32 inclusive. As a result, the melodies generated sound closer to a real melody. Changing the duration value while evolving the population is probably going to have a significant effect on the quality of the melodies.

Experiment 2: Octave

We tried to spread out the octave ranges different Notes could take, hoping to generate melodies with more variations in tone. However, when the range's bigger, we are more likely to have two Notes in the same chord that share a huge pitch gap, which makes the chord sound less harmonious. We will not want to mutate the octave ranges significantly in a run.

Results and Analysis

Different Fitnesses

Currently, fitness function looks at 10 rules, and each rule is assigned a weight rather arbitrarily. We cannot guarantee that the current composition of rules provides optimal, or even just decent performance. In Ozcan's paper[3], several experiments were conducted to analyze the extent to which different musical rules affect how the audiences perceive how good a melody is. According to the paper, chord note, relationships between notes and duration change objectives seem to be the most important objectives and are worth prioritizing. Therefore, we decided to assign the rules in evolutionary-music that are related to these three aspects higher weights than the rest. For example, Avoid Dissonance and Voice Cross, among other rules, were given weights that are two times than usual.

However, with reference to Figure 1, the different rules were already approaching their respective maximum scores by the end of a run. Since all rules are having optimal performances, assigning different weights would have minimal impact on the overall fitness function.

We also tried to remove less significant rules and add hand-made rules. Both did not work well.

Investigating the Effects of Changing Genetic Operators

Changing the Mutation Rate -

We implement single mutation according to the paper we read (refer to Matic's paper), but it does not make the evolution much better.

The GA configuration we use is, tournament selection was used:

- numGenerations = 200
- chromosomeSize = 8
- populationSize = 100
- mutationRate = 0.1
- crossoverRate = 0.001

The final result are:

[3] https://link.springer.com/chapter/10.1007/978-3-540-79305-2_23

- initial average fitness = 1.1492857142857138
- final average fitness = 0.41071428571428575

We ran the code several more rounds with the same parameters, and the performance was consistent - they all showed a significant decrease from the initial average fitness to the final average fitness. We then changed our mutation rate to 0.01 and 0.001, with the following results-

When mutation = 0.01:

- initial average fitness: 1.1990625
- final average fitness: 0.6875

When mutation rate = 0.001:

- initial average fitness: 1.2008482142857142
- final average fitness: 2.133928571428575.

Again, having few more rounds led us to similar results. The trends were intact. We could therefore conclude that our mutation only made the evolution worse.

A possible cause for mutation's disappointing performance is the nature of of the Chorale. Composing a Chorale is much more complicated than evolving an integer. Since for example, if we want to evolve a random number to 100 and use a bitstring 01010101001... to represent it, if we flip one bit, it may not affect the fitness very much. For the individual that does not equal to 100 but is close to 100, although it has more probability to have a bad mutation, there still exists probability to perform a good mutation. However, since the rules of composition and fitness calculation are too complicated, there may exist such a situation that although the individual is close to the global optimal, there is no probability for a good mutation. If our guess is right, the probability of a bad mutation is extremely high.

Changing the Mutation Function -

As mentioned in the previous section (and also a little bit from Ozcan's paper), duration seems to be an important aspect that should be altered to a greater extent, whereas octave should be confined in a smaller range. We modified our mutation function with the given observation, making the probability of duration being mutated 2*mProb, whereas octave will not be mutated at all. We obtained the following results (all the other input parameters were kept the same):

mutationRate = 0.1:

- initial average fitness: 1.2096428571428566
- final average fitness: 0.25

mutationRate = 0.01:

- initial average fitness: 1.1452232142857137
- final average fitness: 0.625

mutationRate = 0.001:

- initial average fitness: 1.1420089285714292
- final average fitness: 1.8169642857142854

Modifying the mutation function, therefore, did not necessarily improve the effectiveness of mutation.

Crossover -

The cross over in our method performs pretty well. With all the other input parameters kept the same, when mutationRate = 0.001 (almost turned off because a smaller mutation rate leads to better performance) and crossoverRate = 0.01:

- initial average fitness: 1.2008482142857142
- final average fitness: 2.133928571428575

With other parameters unchanged, if we set crossover rate = 0.1, the results are:

- initial average fitness: 1.1731696428571425
- final average fitness: 1.9598214285714253

Overall, crossover almost guarantees an improvement in average fitness.

Fitness Sharing

We tried to implement fitness sharing but failed. By definition of fitness sharing, we need to calculate the hamming distance between two individual. Since the representation of each individual (melody) is not only not a bitstring, but also way more complicated than common representations in Genetic Algorithms, we struggled a lot with coming up with a way to calculate hamming distance (or something of a similar kind) directly. We tried to convert its pitch (A-G&R), note, octave and duration into integers and then calculate it, but it does not seem to be intuitively correct. Since if so, the distance between pitches can be at most 7, but the biggest distance of other part may be more than 7. We can not find an efficient weight to balance them, so we finally gave up the idea.

Recursive Reflection

Representation

To begin with, we spent a lot of time choosing a good representation of musical Notes. The papers we read have simpler, easier-to-implement representations. On the other hand, the project we found on GitHub has the more complicated representation we are using right now - we were very impressed with how professional the individual Note presentation (pitch:alteration:octave/duration) looks like; the construction of a melody filled with chords, which in turn consist of four different Notes, seemed to promise prospects of evolving better melodies. Besides, a whole project was already

built on this representation; we could simply tweak some aspects or add more GA mechanisms to make higher-level tests (MOGA, fitness sharing, etc.) However, as we delved into the project, the decision backfired because: 1. (surprise!) both of us know almost nothing about music, and a significant amount of time was invested into understanding music fundamentals, which took away time we could've spent on real EvoComp materials/tests. (which brings me to a sub-point - maybe do not pick a topic that you know almost nothing of) 2. Because of the unique representation, we had to tailor-make genetic operators, built with some constraints in mind (the constraints are described in Infrastructure: Genetic Operators. Overall, we were too ambitious with the project. If we chose to stick to the simpler representation detailed in the final proposal, i.e. tones are confined to C major scale, with notes represented by a list of int (Figure 3), we could've probably achieved much better results, although the melodies would be musically simple. Two lessons learnt from this experience are pick something you at least have some knowledge of for final project, and start with something small.

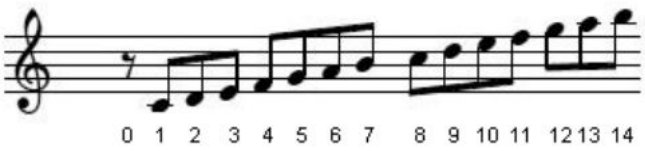

Figure 2: an easier Note representation

Fitness Evaluation

We have successfully evolved melodies with close-to-perfect fitness scores. However, the fact doesn't stop them from sounding terrible. Or, from another point of view, if we convert both a random individual from the starting population (with a fitness score that's ~50% of the perfect score) and the fittest individual at the end of a run to melodies, the starting random individual sometimes even sound better. With our current representation and genetic operators, this problem might be better solved with other approaches and evolutionary computing might not be the best option.

Using External Packages

A huge chunk of the project relies on JGAP, an external Java Genetics Algorithms Package. For example, when populating the first generation of individuals, GeneticMusicChorale.java uses jgap.Chromosome, jgap.IChromosome, jgap.Genotype, etc. Although having existing classes is nice, it requires a significant amount of learning, especially when our program is already moderately complex.

Understanding how classes in JGAP works is however, far less challenging than standardizing the package usage. For example, we could not get mutation and crossover working in JGAP (i.e. IDE complains when we tried to do cfg.addGeneticOperators(new MutationOperator()) and even if it does not, we would not

have information about how mutation and crossover actually works given our unique representation). As we gave up on JGAP, we had to re-write everything without the help of an external package, which made our task more difficult.