Evo Composer

CO659: Computational Creativity - Project Report (2019)

Daniel Krastev
School of Computing
University of Kent
dkk6@kent.ac.uk

Abstract.

Using a genetic algorithm (GA) in order to produce melodies, I am entering a discussion, where I will try to justify why the system I am proposing includes creative processes in its foundation and why I have chosen to implement exactly this method of music composition. I am also doing a detailed analysis of the core of every GA, its fitness function and the steps I am taking in each individual cycle in order to improve the output in every iteration.

Introduction.

Music as an art is definitely one of the most prolific areas to do research in the context of computational creativity. It is very interesting how something so arty could at the same time obey the strict laws of physics and mathematics. Exactly this common ground between science and music could be exploited by the computers in order to try to reverse-engineer similar or novel content.

In this project, I will try to produce new melodies using GA, and the relationship between mathematics and music. I have chosen to use a GA, since it is a primary way in the generation of novel content, it is very efficient in exploring vast areas such as the composition of music and is a good way to test the idea of the *edge of chaos* presented in the lectures of Computational Creativity [1].

What we need to know about the GA in order to understand the remaining of this report is that this is a search methodology, and is a subset of the evolutionary algorithms. GAs are inspired by Charles Darwin's theory of natural evolution and as such, they contain ideas such as crossover, mutation and natural selection of the fittest individuals or also called candidate solutions. The most interesting part of the algorithm, however, is the fitness function. This is the way of evaluating each of the candidate solutions based on some predetermined criteria. Exactly these predetermined criteria are what will guide the evolution in our population of individuals. The formulation, complexity and the coverage of our fitness function will also to a great extent predetermine the quality of the end product.

In my work, I have emphasized on the mathematical approach towards the music generation and I have barely touched any music theories. I am mostly exploring the relationship between the notes in a song and I am trying to build balanced tunes with almost steadily rising or falling pitch.

The technological stack I am using is pure Java together with a music library called JFugue. The library is used mostly to review the generated melodies and is not strongly coupled to the idea of the project.

In the remaining sections of this report, I will look at previous works that have inspired me and will continue with more detailed explanations of how my program and particularly the GA works. At the end, I will try to evaluate the creativity of the project and will conclude with some thoughts about future works and improvements.

Background.

Music generation using creative computing has been a long-standing area of interest to computer scientists. One of the reasons for that is exactly the fact that it is a multidisciplinary area and one can take a variety of approaches in order to accomplish its goal. Currently, scientists' creative approach includes the usage of cellular automata, parallel derivation grammars and evolutionary algorithms ([3], [4], [2]) in order to generate novel content.

The relevant projects that have most inspired me are using GAs. The program AMUSE (A MUSical Evolutionary assistant) [3] for example (Ender Özcan, Türker Erçal) develops its fitness function based on music theories concerning melody and harmony. The function is evaluating each candidate solution based on two groups of features. The first group includes core features, which should be valid to all musical pieces and in the second group the features are adjustable. The later includes parameters that could vary and are dynamically passed to the fitness function by the user. The purpose of AMUSE is to create improvised music based on a given piece of music.

In [4], Waschka II is trying to define basic criteria for a fitness function generating music so that it could have the bare minimum in order to produce aesthetically good content and to serve as a building block for other more concrete evaluation methods. The function is also coupled highly with music theory and like AMUSE uses an inspiration set of musical pieces.

A significantly different approach for its candidate solutions evaluation process is taken by M. Alfonseca et al. in [2]. For their fitness, they are using a function coming from the Algorithmic Information Theory: the Normalized Information Distance (NID). The theory suggests that if a small distance between two objects of any type is measured then the NID is also small between these two objects. In their project, the songs are represented as strings and their distances to other songs are measured through the computable version of the NID, the Normalized Compression Distance (NCD):

$$NCD(x,y)=\frac{max\{C(xy)-C(x),C(yx)-C(y)\}}{max\{C(x),C(y)\}}$$

Here *x* and *y* are the string representations of the songs and *xy* and *yx* are their concatenation. In the core of the GA used in [2] is the idea of measuring distances between the candidate solutions and an inspiring set of musical pieces used. Thus by minimizing the NCD, Alfonseca et al. argue that a given individual will sound similar to the guiding set of songs used.

Inspired from the reviewed projects ([3], [4], [2]), the Evo Composer is trying to accommodate both approaches in its fitness evaluation, using a mathematical function as in [2], to try to capture some of the music theories presented in [3] and [4]. And as well as the presented here projects, I am also using a set of inspiring musical pieces, which guide the evolution of the population in the GA.

Design and Implementation.

Limitations.

In my project, I will evaluate just one element of the music [2] – the melody. And although I will give each note a length, this would not be used in the evaluation of the fitness. In other words, I will leave harmony and rhythm out of the scope of my project. The music that will be generated will be monophonic: one note at a time will be played.

Notes Representation and Encoding.

Two methods have been used for encoding and representation of the songs. The first method comes from the JFugue's *MusicString* representation (1), which includes all the twelve notes written in English notation. This notation is used in the project in order to pass the generated song to JFugue's player and the second reason is to provide a visualization of the generated music.

(1) *Notes = {C, D, E, F, G, A, B};*

Octaves = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

{Sharps, Flats} = {#, b};

Durations = {w (Whole), h (Half), q (Quarter), i (Eighth), s (Sixteenth), t (Thirty-second), x (Sixty-fourth), o (One-twenty-eighth)};

A snippet of a song using this notation will look like this (2).

(2) *"E4i G#4h G#4h G#4i A#4q C5i C5q B4h G#4i G#4i G#4q G4q"*

The other way used to encode the songs is by using integer arrays. Each note is represented by two numbers, one for the pitch and one for the length. The note's pitch number is the number used in JFugue and the integer for the duration is a number from 0 to 7 (3).

(3)

Octave	**C**	**C#/Db**	**D**	**D#/Eb**	**E**	**F**	**F#/Gb**	**G**	**G#/Ab**	**A**	**A#/Bb**	**B**
0	0	1	2	3	4	5	6	7	8	9	10	11
1	12	13	14	15	16	17	18	19	20	21	22	23
2	24	25	26	27	28	29	30	31	32	33	34	35
3	36	37	38	39	40	41	42	43	44	45	46	47
4	48	49	50	51	52	53	54	55	56	57	58	59
5	60	61	62	63	64	65	66	67	68	69	70	71
6	72	73	74	75	76	77	78	79	80	81	82	83
7	84	85	86	87	88	89	90	91	92	93	94	95

8	96	97	98	99	100	101	102	103	104	105	106	107
9	108	109	110	111	112	113	114	115	116	117	118	119
10	120	121	122	123	124	125	126	127				

Notes.

Duration	**Integer**
Whole	0
Half	1
Quarter	2
Eighth	3
sixteenth	4
thirty-second	5
sixty-fourth	6
one-twenty-eighth	7

Durations.

This notation is used in the GA, so we can easily calculate distances between notes. The same song from (2), encoded with integers will look like this (4).

(4) [[52, 3] [56, 1] [56, 1] [56, 3] [58, 2] [60, 3] [60, 2] [59, 1] [56, 3] [56, 3] [56, 2] [55, 2]]

Fitness Function.

As stated in the *Background* section, the fitness function of the Evo Composer is trying to evaluate a mathematically computable parameter (characteristic) of a nice sounding melody. Influenced by my own understanding of music and later confirmed by [3] and [4] I have chosen this parameter to be the distance between the consecutive notes in the melody.

What we often see in music is that consecutive notes often have small tonal distance. That is why by reducing the distance between consecutive notes we can hope to increase the likelihood of the generated melody to sound good. However, directly manipulating the music, just to decrease this distances seems too short-sighted and not at all enough for a system to claim to be computationally creative. That is why I am using a guiding set, which is a collection of songs, used to push distances to go in the direction of the ones in the guiding set, rather than systematically favouring lower distances.

The steps done inside the fitness function are as follows:

1. For each of the songs in the guiding set $G = \{g_i, ..\}$, create an integer array containing the tonal distances between each consecutive notes. The song encoded with $g_i = [[n1, l1], [n2, l2]…, [n48, l48]]$, where n is the pitch and l is the length of the note, will produce the int array of distances $\Delta g_i = [[n2 – n1], [n3 – n2]…, [n48 - n47]]$. This is done for each of the songs in the guiding set and then an average distance for each position of the int array in all the songs in the guiding set is computed $\Delta G = [av.$

G.dist. 1, av. G.dist. 2…, av. G.dist. 48]. For this reason. All of the songs are with a fixed number of notes 48. (This step is done only once since we have a single guiding set)

2. The second step inside the fitness function is to compute the tonal distances between consecutive notes from the candidate solution (the same strategy as the first part in step1. above). *ΔC = [dist. 1, dist. 2…, dist 48]*

3. In the third step, we calculate the distance between *ΔG* and *ΔC* by calculating the absolute value for each element (*av. G.dist. 1 - dist. 1*) … (*av. G.dist. 48 - dist 48*). And since we aim to reach the distances from the guiding set. We would want this to be smaller.

4. This is why for the last step, we need to put the sum of the elements from step 3. in the denominator of the final fitness function.

(5) $$Fitness = \frac{100.0}{\sum_{i=0}^{48} (|av.G.dist_i - dist_i|)}$$

The number 100.0 is not of particular importance here. Higher values of the fitness function will mean a reduction of the distance between the guiding set and the candidate. In other words, higher values of the fitness will mean that the relative distances between the notes in the candidate solution will be closer to the relative distance between the notes in the guiding set.

Genetic Algorithm – steps.

As noted in the previous section, the songs from the guiding set and the individuals from the population all have the same amount of notes, 48. Another restriction I have applied is for the range and durations of the notes. All notes are from octaves 4 and 5, which are notes with numbers between 48 and 71. As regards the duration the restriction is to use either 1, 2 or 3 (half, quarter or eighth).

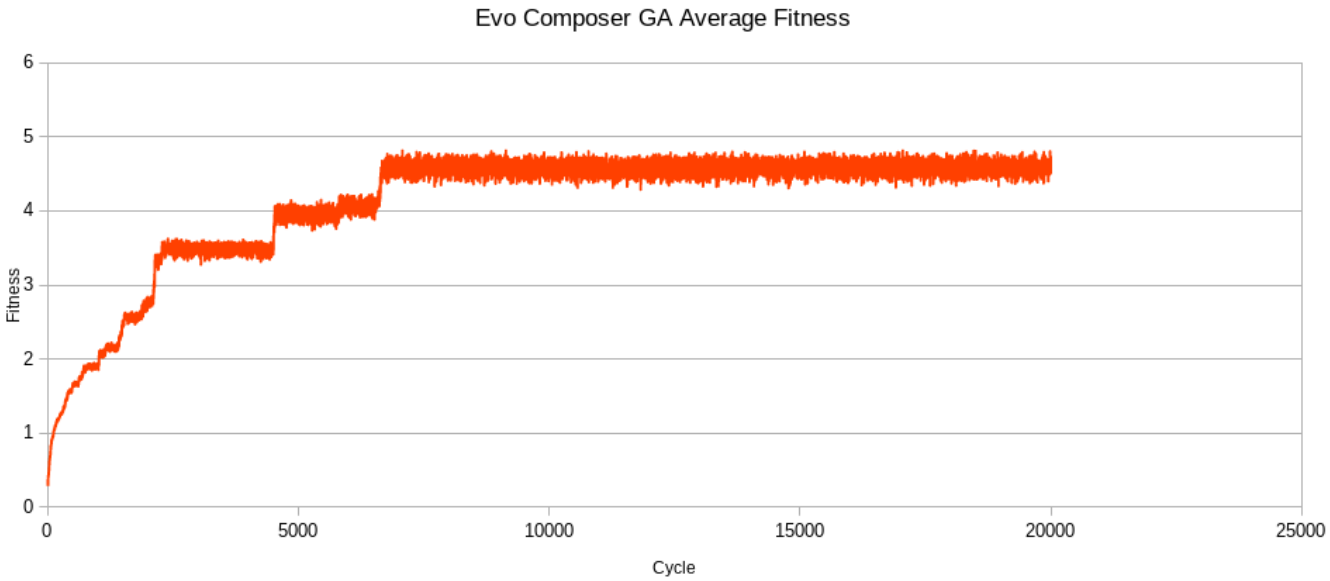
In our GA the population is stored in a 3-dimensional array. Respectively, each candidate solution (individual) is a 2-dimensional array. This is the representation shown in (4). The initialization of the population is done by assigning random integers for the pitch and the duration of the notes in the respective ranges for each of the songs. After this, the population is evaluated against the fitness function discussed above and the algorithm continues with the following cycle:

1. We select the new population by either doing a crossover from the selected candidates, using a tournament selection technique or by adding the best candidate from the tournament selection directly to the new population. With the current parameters, about 75% of the new candidates undergo crossover. The crossover is done by swapping halves of songs between two candidates.

2. After this, a small number of songs are mutated. About 12% of all the candidates. This is done by randomly selecting 2 new notes to take the place of randomly selected 2 notes in the songs, chosen for mutation.

3. The cycle completes by evaluating the fitness of each of the newly created individuals and comparing the fitness to the current best individual. If we have a better one, we keep it and then we go to step 1. again.

The current GA is implemented so that we keep just one individual with the best fitness. The algorithm finishes when all the cycles are completed and at the end, we return the song with the highest score. In (6) we can see a graph showing the average fitness of all the candidates of every cycle of an example run of the program. We can observe that the average fitness value is increasing steadily with each cycle, which means that the algorithm is working well as regards to the current implementation of the fitness function. Another thing we notice is that 20000 cycles might be a bit more with the current setup. After around the 8000th cycle, nothing significant happens to the average fitness, which means that we can probably reduce the total number of cycles to about 10000.



(6) GA Average Fitness graph.

Here are some of the generated songs (7).

(7)

Fitness: 3.85
Song: “G4i B4h A#4q A#4h B4h C#5i C5h B4i G#4i G#4h G#4h G4q A#4i A#4i A4q G4i A4i A4i A#4q B4i D#5q E5q E5h D5i C#5q B4q B4h C#5h C5i C#5h A4i A4q G#4q G#4q G#4i F#4h F#4q F4i E4h D4i F#4h F4q F4q D#4q F4h F4q F#4q F4h”

Fitness: 5.00
Song: “E4q G4i C#5q D5h E5q F#5q F#5h F5i D#5q E5q F5i E5h G5i F#5i F5i D#5q F5h F5i F5i E5i F#5i G5q F#5h E5h D#5h C#5h C#5q D#5h C5h C#5i B4i C5i C#5q D#5q D#5h D#5i E5i E5h D5i C5h D#5h D5i C#5i A#4q C5i C5i C5i B4q”

Evaluation.

For the purpose of the evaluation of the product and the process, I will use Colton's Creative Tripod framework discussed in the lectures [1] and will also add other users' feedback to the evaluation. I will try to justify why at least the process used shows qualities in the three aspects of the framework – Skill, Appreciation, and Imagination.

Evaluating the process will directly link to the stages and principles used in the GAs. Even before presented in the lectures [1] as a way to produce a creative output I already considered the idea to use evolutionary algorithms, since I had used them before and I was always surprised by their effectiveness and unexpected results. It is important, however, that GAs works well only when given a relevant for the task fitness function. The fitness function is the skill required for the GAs to produce interesting output for the user. And if we accept that the fitness function I have used is a relevant Skill (understanding) in the domain of music, than this Skill being part of the code has become part of the process itself.

As regards to the Appreciation, which translates to "*the ability of the system to critically assess its output"*, we can see that this is the core idea of the GAs in general. We know that the output of the GA is generated by repeating multiple steps a certain amount of cycles. At the end of each cycle, the algorithm is assessing its output and determines whether it is valuable or not, based on the fitness. That is why the process I have used have not only the ability to appreciate its output but is also doing a feedback loop and improves its generated content in every cycle until a satisfactory result is reached.

Imagination is the hardest of the three aspects to satisfy. One can argue that there can be no true imagination in any of the processes that include computers, however we should admit that the output is extremely hard to be predicted and is also in my case guided by an inspirational set of musical pieces. Furthermore, the idea of GAs is to solve a problem, that has a very large search area, which makes it even harder to be predicted, otherwise, we would not have used GAs in the first place.

As for the produced songs, I have tried to assess them by comparing the outputs from different stages of the project. Since I was initially using not very successful ideas of fitness functions, the output was not very pleasant to me and to other random listeners. As I was progressing with the idea and the implementation of the fitness function, so was the quality of the songs. Other listeners have confirmed that with the latest outputs I have managed to achieve basic melodies characterized by at least a stable, rising or falling pitch. With that said, it is important to point out that the quality of the melodies will reflect the fitness function, which is directly linked to the programmers' knowledge about music theory. In other words if you are capable of describing and knowledgeable about concepts in music you will most likely produce a much better algorithm.

Conclusion.

Future Works.

The possibilities to elaborate on this idea are limitless. Furthermore, the main problem with my project was to focus on a single aspect of music composition. I have chosen to work mostly on the mathematical aspect, however, I feel that if more time was available to work on this project I would

have researched more music composition theories and would have tried to implement them together with the current concepts used. Such theories might be the ones presented in the referenced papers [3] and [4], but it is possible to add other different aspects, such as evaluation of the rhythm, or working without guiding set at all, just with the rules from the fitness function. This is a vast topic and I believe that the work I have done is scalable in almost endless directions.

Final Thoughts.

Throughout the project I was very inspired to work in this area since I wanted to learn more about music, I wanted to implement one more GA, and mostly I wanted to find out if a true computational creativity existed. And even though I was a little sceptical about the output of my system, with time I understood that this project was not only about computational creativity, but it was also about our own creativity. We are the ones who need to do something and make the computer *think* and *act* like it is creative, that is why the computational creativity is strongly coupled with our own. In this context, if we look more philosophically about the project we might say that it has been an excellent way to provoke us and take us out of the stereotypes of computer programmers and gives us a taste of what is like to be an artist.

Reference

[1] Computational Creativity Lectures on “Music, Sounds, Computers” (1 & 2) and “Bio-Inspired Art and Creativity,” University of Kent, 2019

[2] M. Alfonseca, M. Cebrian and A. Ortega, "A simple genetic algorithm for music generation by means of algorithmic information theory," *2007 IEEE Congress on Evolutionary Computation*, Singapore, 2007, pp. 3035-3042.
doi: 10.1109/CEC.2007.4424858

[3] Özcan E., Erçal T. (2008) A Genetic Algorithm for Generating Improvised Music. In: Monmarché N., Talbi EG., Collet P., Schoenauer M., Lutton E. (eds) Artificial Evolution. EA 2007. Lecture Notes in Computer Science, vol 4926. Springer, Berlin, Heidelberg
doi: https://doi.org/10.1007/978-3-540-79305-2_23

[4] Waschka II, Rodney “Minimal Fitness Functions in Genetic Algorithms for the Composition of Piano Music” ICMC 2015 – Sept. 25 - Oct. 1, 2015 – CEMI, University of North Texas

[5] “The Complete Guide to JFugue: Programming Music in Java™” Book 1st Edition.