Music Generation

Anna Boronina

I. Introduction

ALL games have music playing in the background. Usually, this music also depends on the current environment around a player. If they are going through a rough dungeon, the pitch and tempo would be low and slow, respectively; if they are going through some beautiful valley, the pitch would be on an average or high level and the tempo would be faster. Besides reflecting current situation, soundtracks also can warn a player. A change in music could be a warning to a player, e. g. a dangerous object is close. Also, background music can play a reliever role after a hard level so that a player can calm down a little and feel safe.

Besides affecting only gaming time, music can exist even after the game is not popular. For example, if Mario players hear music from the game now, it would bring up some memories.

Music is an important part of game development, especially for games with long playing time. Collins [1] says music functions include anticipating action, drawing attention, representing current time and place, etc.

First attempts to create generative music included putting IF and ELSE between prerecorded melodies. E.g., music and its changing could depend on the player's and enemy's health or on the number of enemies around a player. This approach was more about music parametrization than a generation. [2]. Collins [2] mentions Guy Whitmore who created a transition matrix for *No One Lives Forever* game. The game had hundreds of small music samples and the matrix gave a transition from one to another.

Nevertheless, if the game is short and has few states, e.g. (a) player is active, (b) screen is loading, (c) player is dead, then background music should not be distracting, hence should be simple and probably repetitive. If levels are simple, there is no need for long melodies as well.

To sum up this chapter, there is no one way to create a perfect music sample using just one tool - it always depends on the game specifics.

II. Idea Explanation

The idea behind this project is music generation using an evolutionary algorithm. It is aimed at simple games which need repetitive music and no chords. The algorithm has a rich configuration file: a user can set scale, scale root, pitch level, number of bars, number of notes per bar, etc.

The algorithm provides two ways of evaluating each sample: manually and automatically. Manual evaluation implies ranking every sample (or genome) during a selected number of epochs. Automatic evaluation does not include interaction with a program, only getting the final output. Even though the first way takes longer, it might be closer to a user's desires, as the automatic evaluation method has a predefined fitness function aimed at repetitiveness.

III. Taxonomy

A way to describe an algorithm for procedural generation is to use an existing taxonomy. One of the most popular ones is by Togelius et al. [3].

A. Online Versus Offline

Can be both.

Online: if the game is stochastic, then the music could be generated during the level creation itself. Since the algorithm takes music descriptors as input, they could be created upon the level descriptors (difficulty, number of enemies, stress level, etc).

Offline: if the game is deterministic, then all the music samples could be generated along with the levels before game time.

B. Necessary Versus Optional Content

Necessary. The importance of background music in games was briefly discussed in the introduction.

C. Random Seeds Versus Parametric Vectors

Both. The algorithm takes melody descriptors (scale, scale root, pitch level, number of bars etc) as well as contains randomness, since the initial melodies are generated randomly. Moreover, the algorithm contains random mutations and random crossovers for the top-ranked melodies.

D. Stochastic Versus Deterministic Generation

Stochastic. If the algorithm gets the same sets of parameters twice, the results will be different due to the reason above.

E. Constructive Versus Generate-and-Test

Generate-and-test. This idea is at the core of the evolutionary algorithm. There is an initial population that is iteratively improving by utilizing a fitness function. The number of iterations can be predefined or bounded by reachable criteria, e.g., generating new samples until the best ranking is greater than 100. In this algorithm, the first approach is used.

IV. Methodology

To run the algorithm, the following parameters should be set:

1) **NUMBER OF BARS** - number of bars in the final melody.
2) **NOTES PER BAR** - number of notes in one bar.
3) **BITS PER NOTE** - number of bits to encode a note. The less bits, the smaller span of notes is available for generation.
4) **BEATS PER BAR** - number of beats per bar; equals to 4 and should not be changed.
5) **SCALE** - scale of the melody ('major' or 'minor')
6) **SCALE ROOT** - scale root of the melody ('A', 'B', 'C' etc).
7) **PITCH LEVEL** - level of pitch of the melody; may sound differently after converting .mid to .mp3.
8) **BPM** - beats per minute; may sound differently after converting .mid to .mp3.
9) **VELOCITY** - velocity of the melody; may sound differently after converting .mid to .mp3.
10) **EPOCHS** - number of epochs for the evolutionary algorithm; in other words, number of generations.
11) **POP SIZE** - population size; recommended range is between 20 and 100.
12) **FITNESS FUNCTION** - fitness function to use; 'manual' will allow to grade the melodies as they are generated manually, 'auto' will mean that melodies will be ranked automatically by the predefined fitness function.

A. New population creation

The initial population is created and immediately evaluated by the chosen way of ranking. Then, all the genomes (or melodies) are sorted by their rank, and the best ones are taken out. The "top genomes" number was tuned during the testing of the algorithm and equals the maximum between 25% of the population size and 4. E.g., if the population size is 15 then the best 4 are selected; if the population size is 50 then the best 12 are selected.

Each of the best genomes goes through random mutations and generates two new genomes from a crossover with a randomly chosen genome from the first 30% of the top ones. To keep the diversity of the population, I create several new random genomes. The size of this group equals 30% of the initial population size.

After the best ones are selected, their mutated versions and a few crossovers between them will go to the next population. The calculations are below. Whenever the result of division or multiplication is a rational number, consider it floored.

1) Population 1:

Initial population size: N
Top genomes: $0.25N$
Mutations: $0.25N$
Crossovers: $0.25N \cdot 2 = 0.5N$
New randomly initialized genomes: $0.3N$
New population size: $0.25N + 0.5N + 0.3N = 1.05N$

2) Population 2:

Initial population size: $1.05N$
Top genomes: $0.25 \cdot (1.05N) = 0.2625N$
Top genomes: $0.25 \cdot (1.05N) = 0.2625N$
Mutations: $0.2625N$
Crossovers: $0.2625N \cdot 2 = 0.525N$
New randomly initialized genomes: $0.3N$
New population size: $0.2625N + 0.525N + 0.3N = 1.0875N$

The population size seems to *slowly* increase. Yet, for the initial population of size 50 and 50000 epochs, the last population had the length of 51. It happens due to floor rounding at steps of taking the best genomes and initializing new ones.

B. Mutation

A mutation is necessary to prevent the same genomes get stuck at the top. I have two main types of mutations: one-note mutation and piece mutation. The former one has six different ways of doing it. A melody goes through either several one-note mutations (the amount of them depends on the melody length) or one-piece mutation.

Type 1.1: changes current note to be as the next one.
Type 1.2: changes current note to be as the previous one.
Type 1.3: changes current note to be as the one before the previous one.
Type 1.4: changes current note to be as the the one after the next one.
Type 1.5: changes current note to be as the one 8 notes before the current one.
Type 1.6: changes current note to be as the one 8 notes after the current one.
Type 2.1: swaps two pieces of a melody.

C. Crossover

There are two crossovers: one-point and two-point.

One-point crossover: randomly chooses one number to split two melodies into two parts and creates two new melodies by swapping first parts of two melodies.
Two-points crossover: randomly chooses two numbers to split two melodies into three parts and creates two new melodies by swapping the melodies' middle parts.

D. Fitness Function

A fitness function consists of two parts: evaluating the repetitiveness of a melody and the beginning and the final parts of the melody.

Repetitiveness: favors repeated notes on even places and penalizes repeated consecutive notes.
Beginning: favors increasing progression with step of one or two notes and penalizes everything else.
Ending: favors decreasing progression with step of one or two notes and penalizes everything else.

V. Results

To evaluate the performance, I have used the following metrics: running time in seconds and the best genome's rank from the last epoch. The two changing parameters were the initial number of genomes and the number of epochs. The first parameter is either 15, 50, or 80. The second parameter is either 5000, 10000, or 50000. The highest rank was achieved by pair $(80, 10000)$.

To download a particular sample, follow the corresponding link in the last column.

The source code is in this GitHub repository.

Initial Number of Genomes	Number of Epochs	Time Spent (s)	The Best Genome's Rank	The Result
15	5000	13.77562	59	link
15	10000	22.68578	60	link
15	50000	106.16036	60	link
50	5000	20.84560	54	link
50	10000	39.67426	65	link
50	50000	188.70546	54	link
80	5000	32.15173	53	link
80	10000	61.03217	79	link
80	50000	291.33743	73	link

Fig. 1. Results: time and best rank

References

[1] K. Collins. *Game sound: An introduction to the history, theory, and practice of video game music and Sound Design*. MIT Press, 2008.

[2] Karen Collins. An introduction to procedural music in video games. *Contemporary Music Review*, 28(1):5–15, 2009.

[3] Julian Togelius, Georgios N. Yannakakis, Kenneth O. Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011.