Genetic Algorithms for Music Production

Prateek Rath IMT2022017
Ketan Raman Ghungralekar IMT2022058
Rudra Pathak IMT2022081

I. Introduction

Production of 'good' music is largely considered to be a work of art that involves creative thinking methods that are beyond systematic procedures or algorithms. Proponents of computer science and algorithms aim to use heuristics to build systematic procedures and algorithms to solve such problems. A genetic algorithm (GA) is a meta-heuristic inspired by the process of natural selection that belongs to the larger class of evolutionary algorithms (EA). The use of genetic algorithms in particular began in the 1990s when Lee Spector and Alpern Alpern worked on evolved bebop musicians in 1994 and 1995. In 1997, Brad Johanson and Riccardo Poli developed the GP-Music System, which used genetic programming to breed melodies according to both human and automated ratings. Many similar attempts have been made to produce music using evolutionary and other methods such as machine learning. Each specific implementation has its own merits and application constraints.
For interested readers, we recommend reading the 2010 paper by D. Matic [1], which provides an excellent introduction to the subject.

II. Problem Statement

The problem we intend to solve is to generate music while using a single piece or song as a reference or base. This song is considered to be the ideal 'good' song and then based on this a new song is to be generated which is close to the ideal song.

III. Solution Strategy

We propose a solution to the above problem that uses genetic algorithms. We first generate individuals of the initial population via random mutations/crossovers on the 'ideal' song. After these individuals are generated we run the genetic algorithm on them for several iterations. The final song outputted is the best individual in the final population.

IV. Formulation

Formally the problem can be stated in terms of the input given and the output needed. The input given is the midi file that represents the ideal song. The output another midi file that contains the final song. This song closely resembles the initial song in certain ways. However, due to the randomness introduced, it is quite different from the original.

V. A small caveat

Normally a song when represented in midi format consists of many tracks each of which usually plays it's own instrument. So a song could have a bass track, an electric guitar track, a drum track, a piano track etc. Here we work with a single track. The genetic algorithm runs on a single track and modifies it's contents to alter the song. This simplifying assumption can be removed in two ways. One could be to run the genetic algorithm on every track independently. This however would lead to a lack of coordination between the tracks. The second way could be to perform parallel operations on the other tracks. This however does not guarantee a good song as music theory for drums isn't the same for guitar and piano. Hence, we focus on a single track.

VI. Implementation Details

A. *MIDI files and The Mido Library*

MIDI(Musical Instrument Digital Interface) is a standard protocol that allows devices to communicate with each other using MIDI messages. MIDI transmits musical notes, timings, and pitch information, which the receiving device uses to play music from its own sound library. Mido is a Python library for working with MIDI ports, messages and files. While working on this project we could have also chosen java's JMusicUtil or python's MIDIUtil library. However, mido seemed to be the easiest to use and hence we stuck with it. The mido library has functions that enable easy reading of midi files that easily puts the data into python structures such as lists. These lists can then be worked on(run the genetic algorithm) and then converted back to the desired format and saved in a .mid form.

B. *Our Own Note Representation*

When mido reads a midi file, it returns an object that contains all the midi data. This data includes meta data about the song such as the tempo(in beats per minute), the clocks per click, the time signature(number of beats in a bar), the key signature, etc.(the scale). Again, readers interested in this terminology are encouraged to read [1]. Each track is a list of messages and meta-messages, with the time attribute of each message set to its delta time (in ticks). This means that the start time of an event(or message) is 't' units after the start of the previous event. The end times for each event are presented through a different message. This representation is not suitable for running genetic algorithms.
In music books, we often see representations based on note-heads where it is intuitively clear where each note begins and how long it lasts.

Thinking along similar lines, we realized that the ideal representation for a note would be one that kept all the information about the note in a single place. A midi representation would contain separate messages for the start and end time of the note. Moreover, the times are not actual times but 'delta' times which makes it difficult for us to deal with them.
Instead, it will be better if all the information pertaining to a single note is stored in one place. So we created a new class called the NoteRep class that is used to store notes. These notes have a start time (called `actual_time` in code), a duration, a pitch value(a number between 0 and 127 stored in the note attribute), a velocity (the force with which the key is pressed), etc. The NoteRep class holds all this information.

C. *How our representation is put to use*

Initially when a midi file is read, the data is stored in an object. This object contains several tracks, each of which is a sequence of messages. These messages, as mentioned earlier, have a time attribute which is actually a delta time. Thus, the time in these messages is the start time of the current message relative to the start time of the previous message. Running the genetic algorithm on these data is almost impossible, as it would not only be slow and cumbersome. Hence, we first need to convert to our note representation, run the genetic algorithm on the new representation, and then convert back to the old representation.
This brings us to the conversion process. Conversion is not an easy task. The data for a single note is present in two different messages. We can understand this using the "brackets" analogy. Similar to a note starting to play, we can visualize a colored bracket opening. Figure 1 shows this analogy. Rather than storing on and off times of a note, we want to store the start time and the duration. The rest of the values, i.e. velocity, pitch (indicated by color) and channel remain unchanged.

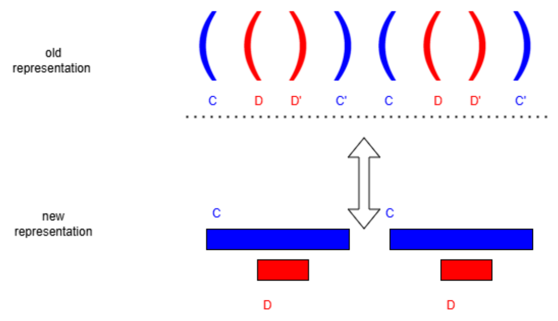

Fig. 1. Our conversion and the brackets analogy

This can be accomplished by simply finding the closing bracket for each opening bracket. The actual start time(in absolute values) is found by accumulating the delta times. However, the messages aren't just 'notes' that produce sounds. There are also messages of other types such as 'control_change', 'program_change', 'pitch_wheel' etc. that play a different role. To deal with this each NoteRep object has a type attribute which can take a value of either 'not_note' or 'note'. For messages of such types, we simply make the NoteRep object encapsulate the data in these messages so that later during the backward conversion the messages can be recreated.
After converting to the desired representation via the `conv_from_midi` function, we run the genetic algorithm. Post this, we convert back to the old notation(the one with delta times in it) by de-accumulating the actual times and getting back the delta times. We also introduce a note_off event for every note in the new representation.

D. *Mutators*

The following mutators are designed to modify properties of a given track, such as note pitch, timing, and simplification. Each mutator operates on the 'track' data structure and applies transformations probabilistically.

1) Pitch Mutator: The `pitch_mutator` function alters the pitch of notes within a specified range. For each note in the track:

- It calculates a range of possible pitch values based on the given range `range1`.
- With a probability `prob_mutation`, it selects a new pitch from the range and updates the note.
- Notes outside the pitch range (0 to 127) are clamped to valid limits.

This mutator allows controlled variation in the pitch of notes, introducing diversity.

2) Actual Time Mutator: The `actual_time_mutator` function modifies the timing of notes based on the greatest common divisor (GCD) of all actual times in the track:

- It computes the GCD of the actual times of all notes.
- For each note, it generates a range of possible timing values based on the GCD and `range1`.
- With a probability `prob_mutation`, it selects a new timing value from the range and updates the note.

This mutator ensures that timing changes are aligned with the track's rhythmic structure.

3) Simplify Mutator: The `simplify_mutator` function reduces the diversity of notes by repeating previously played notes:

- For each note, with a probability `prob_mutation`, it replaces the current note's pitch with the previous note's pitch.
- This creates a simplified and repetitive melodic structure.

This mutator is useful for creating consistent patterns or reducing complexity in tracks.

E. *Raters*

The raters evaluate different aspects of a musical track, including pitch range, melody direction, repetition, and rhythmic diversity. These metrics are combined to produce a final rating for the song. Below is a brief description of each rater and how they contribute to the final score.

1) Neighboring Pitch Range: This rater identifies "crazy notes," which are notes with a pitch value at least one octave away from the previous note. The proportion of such notes to the total number of notes is calculated.

2) Direction of Melody: This rater calculates the proportion of notes that have a higher pitch than the previous note, providing insight into the melody's upward or downward movement.

3) Direction Stability: This rater assesses the consistency of melodic flow by calculating the ratio of pitch direction changes to the total number of notes.

4) Pitch Range: This rater measures the range between the highest and lowest pitch values and computes the ratio of the lowest pitch value to the highest pitch value.

5) Equal Consecutive Notes: This rater determines how often two consecutive notes have the same pitch value by calculating the ratio of such occurrences to the total number of notes.

6) Unique Rhythm Values: This rater evaluates rhythmic diversity by calculating the proportion of unique rhythm values (durations) to the total number of notes.

F. Combining Raters

The individual rater metrics are combined into a final rating using the following process:

1. **Deviation from Targets**: For each rater, the absolute difference between the computed value and a predefined target value (based on the dataset's average) is calculated:

$$\text{Deviation} = |Rater\_Value - Rater\_Target|$$

2. **Weighting by Influence**: Each deviation is multiplied by its corresponding influence weight, which measures the importance of the rater. The influence is computed as:

$$a = \text{Rater\_Target} - \text{Rater\_Min}$$

$$b = \text{Rater\_Max} - \text{Rater\_Target}$$

$$\text{Influence}_{\text{Rater}} = 2 \cdot (0.5 - \min(a, b))$$

3. **Weighted Sum of Deviations**: The weighted deviations are summed:

$$\text{Weighted\_Sum} = \sum_{\text{Raters}} \text{Influence}_{\text{Rater}} \cdot \text{Deviation}_{\text{Rater}}$$

4. **Normalization by Total Influence**: The final rating is obtained by normalizing the weighted sum by the total influence across all raters:

$$\text{Final\_Rating} = \frac{\text{Weighted\_Sum}}{\sum_{\text{Raters}} \text{Influence}_{\text{Rater}}}$$

G. Final Output

The final rating and individual rater metrics are output as a dictionary:

- **Crazy Rating**: Measures pitch outliers.
- **Direction of Melody**: Assesses upward/downward melodic movement.
- **Direction Stability**: Evaluates pitch direction consistency.
- **Pitch Range**: Measures the pitch span in the melody.
- **Equal Consecutive Notes**: Quantifies pitch repetition.
- **Unique Rhythm Values**: Assesses rhythmic diversity.
- **Final Rating**: Combines all metrics into a normalized score.

H. The CrossOver Function

The `crossover_tracks_random` function swaps sections between two tracks of `Note_rep` objects based on a random probability. It works as follows:

- **Time Boundaries**: It calculates the minimum and maximum times for each track, determining the overlapping region between the two tracks.
- **Swap Region**: A random region is selected randomly within the overlap, and the function decides whether to perform the swap based on the specified probability.
- **Valid Regions**: Notes within the selected region from both tracks are identified and swapped.
- **Sorting**: After swapping, the tracks are sorted by `actual_time` to maintain correct timing order.

If the tracks don't overlap or the random check fails, the original tracks are returned unchanged as no crossover is possible for them.

I. How It All Comes Together

In the genetic algorithm, the process of evolving a new song from an input MIDI file involves the integration of mutators, raters, and crossover functions. These components work together in the following manner:

The genetic algorithm proceeds through the following steps in each generation:

1) **Selection:** Select individuals from the current population based on their fitness scores. Higher fitness individuals have a higher chance of being selected.
2) **Crossover:** Apply the crossover function to pairs of selected individuals to produce offspring. This introduces new combinations of genetic material.
3) **Mutation:** Apply the mutators to the offspring to introduce random variations.
4) **Evaluation:** Use the raters to evaluate the fitness of the new population.
5) **Replacement:** Replace the old population with the new population, retaining a certain number of top individuals from the previous generation to ensure the best solutions are carried forward and rest are generated using crossover and mutation.

Dynamic Adjustment of Mutation and Crossover Rates:

As the algorithm progresses through generations, the probability of mutation and crossover decreases. This is done to fine-tune the solutions as they get closer to the optimal solution. Initially, higher mutation and crossover rates help in exploring a wide search space. As the generations advance, these rates are reduced to focus on refining the best solutions found so far.

Handling Population Size:

If the number of individuals in the population is less than required after selection, additional individuals are generated using crossover and mutation. This ensures that the population size remains constant across generations. By combining the genetic material of selected individuals and introducing variations, the algorithm maintains diversity in the population, which is crucial for avoiding premature convergence to sub-optimal solutions.

This iterative process continues for a specified number of generations or until a satisfactory solution is found. By integrating mutators, raters, and crossover functions, the genetic algorithm effectively explores the search space and evolves solutions that improve over time.

VII. Limitations

This algorithm is useful to alter only one track of a song. Altering multiple tracks is possible but it may not lead to desirable results.
The algorithm is designed to run only on MIDI files. It can't deal with vocals or any other formats that midi cannot handle. Usually the zeroth track contains meta information such as the tempo and the key signature while the rest of the tracks contain note information corresponding to different instruments. We assume that all midi files follow this trend.

VIII. Scope for Further Improvement

This project doesn't make use of all possible mutators or raters possible. On one hand it would seem that using all possible raters would lead the algorithm to converge very quickly onto the same song that served as the ideal one. On the other hand using too few raters could lead to the song being utter garbage. There is a fine line between the two ends. Finding this line is extremely difficult and requires human intervention at every stage. Maybe the use of machine learning or some elegant algorithm in this area could be an interesting extension to this project. [2] provide a good list of mutators and raters some of which we have used.
The project also doesn't take into account the 'goodness' of a musical piece in terms of the coordination between various tracks. If one is able to come up with metrics to gauge this, one can also take this score into account while rating a song.

References

[1] D. Matić, "A genetic algorithm for composing music," *Yugoslav Journal of Operations Research*, vol. 20, no. 1, pp. 157–177, 2010.

[2] S. Pavlov, C. Olsson, C. Svensson, V. Anderling, J. Wikner, and O. Andreasson, "Generation of music through genetic algorithms," 2014.