MelOR

Overview of work completed

We have implemented a melody generator that takes in input using a appJar GUI and generates a midi file that was constrained by the user input.

Overview of each file

* `interface.py` - contains our GUI interface
* `euclidian_rhythms.py` - contains our rhythm generator
* `melody.py` - contains our melody generator and defines the `Melody` class.

The euclidean_rhythms.py file generates rhythms in a few different ways. First there is the E() function, which generates just a traditional euclidean rhythm with k hits and n intervals. This gives an output that resembles an output of the bjorklund algorithm outlined in our pitch paper. The Bjorklund algorithm runs in O(n) time, which seems to render out algorithm useless. However, our ORTools implementation gives us extra flexability that the deterministic Bjorklund algorithm does not have. For example, we can soft constrain the beats to be closer or farther away from each other, even if it does not change the minimum maximum distance of the rhythm.
The other rhythmic tool in euclidean_rhythms.py is the fill_interval program. Essentially it takes a list of lists where each inner list outlines a given chord in a chord progression. For each chord it generates a random number of intervals and creates a unique euclidean rhythm for each chord in the progression. The resulting rhythm of the entire progression is a conglomerant of chained euclidean rhythms, this gives the melody an off-tempo, more organic feel that is hard to generate with computers.

An usage example of the `Melody` class is:

```python
melody = Melody()

melody.generate('E', 'major',
  bars=18,
  allowed_intervals_last_first_neighbours_chords=([3, 4, 5], 'min'),
  chord_progression=['I', 'IV', 'ii'],
  total_notes_per_chord=[2, 3, 2, 5, 4, 5, 2])
melody.show()
```

The `show()` method outputs a list of lists, where each list is a chord (or notes from a chord).
The `generate()` method receives the following parameters:
(PS: for all the parameters that expect an array, the array must have size `7`, given that a scale has `7` diatonic chords).

* `key quality`: the 12 possible keys, followed by the quality which is either major or minor, such as 'A major', 'B- minor', 'C# major'. It is a string.
* `#chords`: the number of chords (that is, the length of the list return by `show()`). It can be any int (but the minimum value is 3)
* `chord_progression`: the chord progression you want the melody to have. It expects either `None` or an array of strings, where each string is a chord in roman numeral (for example, `I`, `iv`, etc). If no chord progression is passed, the first and last chords are always the `I` (in case of `major` scale) or `i` (in case of `minor` scale) chord.
* `min_max_total`: if you want to minimize or maximize the sum of the notes in the entire melody. Possible values are None, 'min', or 'max'
* `min_max_neighbour_chords`: if you want to minimize of maximize the sum of a note and its neighbor. Possible values are None, 'min', or 'max'
* `no_repetition_every_X_chords`: if you want that every `X` chords to be mutually different. This has no effect if `chord_progression` is passed. It must be an integer greater than or equal to 1.
* `allowed_intervals_last_first_neighbours_chords`: the allowed intervals for the last note of a chord and the first note of the next chord, and if you want to minimize of maximize (or neither) the distance between the notes. It expects a tuple, where the first element is an array of ints (the intervals) and the second element is either `None`, 'max', or 'min'.
* `allowed_notes_for_chords`: the allowed notes for each chord. It expects an array, where each element is either an array or 'all'. For example, if a chord is compose of the notes `1, 2, 3`, then if you pass 'all', all the three notes will be used when generating the melody; if you pass `[1, 2]`, only those notes are going to be used.
* `total_notes_per_chord`: the number of notes that must appear for each chord. It expectes an array, where each element is either an int or 'all'. For example, if the chord `I` is compose of the notes `1, 2, 3`, then 'all' would mean that the `I` chords in the final melody would have size `3`; if you pass `2`, all `I` chords in the final melody will have size 2, and they can be any permutation of the possible 3 notes for the chord; if you pass `5`, then all `I` chords in the final melody will have size 5, but since the chord has only `3` possible notes, there will be repetition of notes.
* `soft overlap`: takes values 'true' or 'false', toggles on an off soft overlap. If true, notes will try and spread out as much as possible, even if it doesn't change the min max distance of the rhythm.
* `duration`: duration in seconds of the midi file. This can be converted to beats by multiplying by bpm/60

How to run the project
Packages required
You must have `ortools`, `music21`, and `appJar` installed.

How to run
```
pipenv run python3 interface.py
```