Exploring Finite-State Methods and Context-Free Grammars in Composition of Music

Introduction

Music has an inherent structure that can be amenable to composition and analysis using formal language models such as context free grammar and finite-state automata. The research presented here explores the use of such computational methods for music composition, with particular interest in context free grammar (CFG) in melody creation.

Background

Finite-state automata (FSA) and context free grammar have been used widely in computational musicology (Study of music using math and statistics) and natural language processing. FSAs can be used to model simple musical patterns, while CFG allow hierarchical structures in melodies like motifs and phrases. Such methods have been used in prior work to automate composition and music analysis.

Methodology

The project consists of two parts:

Survey of existing approaches: Literature review of FSAs and CFG applications to music composition, structure, harmony, and melody generation.

Implementation: Python code for melody generation using CFG.

Specify a CFG to represent the structures of melodies.

Employ a parser that generates melodies based on the grammar.

Convert generated melodies to MIDI format.

Inspect the output and compare it with musical notions.

Implementation: Python-based Melody Generator

The following is Python code for generating simple melodies using context free grammar: This program outputs a random melody and prints it as a MIDI file.

```python
import random  ##for CFG choices
import mido  ##midi file handling
from mido import Message, MidiFile, MidiTrack  ##create midifiles and track object

note_to_midi = { ##dictionary to map midi numbers to notes
    "C4": 60, "D4": 62, "E4": 64, "F4": 65, "G4": 67, "A4": 69, "B4": 71
}


def generate_melody(): ##function to create melody using context free grammar
    grammar = { ##dictionary for CFG language
        "S": ["Phrase Phrase X"],  ##melody will have 2 "phrases"
        "X": ["S", ""], ##either S or empty
        "Phrase": ["Note Note Note"],  ##phrase has 3 notes
        "Note": ["C4", "D4", "E4", "F4", "G4", "A4", "B4"]  ##possible notes


        ##without X, it would result in
        ##[["note","note","note"],["note", "note", "note"]]



    }

```

Originally, the CFG was S -> [“Phrase Phrase”], however I added “X” for more variability and longer melodies

```
    def expand(symbol): ##recursive function for creating melody
        if symbol in grammar: ##if non-terminal symbol
            expansion = random.choice(grammar[symbol]).split() ##randomly chose a rule in the dictionary
            return sum([expand(s) for s in expansion], []) ##calls sum to "flatten" the lists and combine them into one list
            ##return [expand(s) for s in expansion] ##recursivly call expand with new terminal or non terminal symbol
        else:
            return [symbol]  ##return the note if it is terminal symbol

    melody_structure = expand("S") ##begin recrusion from S

    ##melody = [item for sublist in melody_structure for item in (sublist if isinstance(sublist, list) else [sublist])] ##"flatten
    ##[['note'], ['note'], ['note'], ['note'], ['note'], ['note']]


    return melody_structure



def create_midi(melody, filename="melody.mid"):
    midi_file = MidiFile()  ##create midi file
    track = MidiTrack()  ##create track object
    midi_file.tracks.append(track)  ##add track object to midi file
    time_per_note = 480  ##speed of melody

    for note_name in melody:  ##loop through the notes in the melody list
        midi_note = note_to_midi.get(note_name, 60)  ##convert note to midi number (C4 or 60 as default)

        track.append(Message("note_on", note=midi_note, velocity=64, time=0))  ##start the note
        track.append(Message("note_off", note=midi_note, velocity=64, time=time_per_note))  ##stop the note after the time for tha

    midi_file.save(filename)  ##save melody as midi file
    print(f"MIDI file saved as {filename}")
```

```
##run the code


melody = generate_melody() ##call generate_melody function
print("Generated Melody:", melody)
create_midi(melody) ##call create_midi to create midi file

```

Results and Discussion

The generated melodies exhibit structured patterns based on the context free grammar rules:

More advanced CFGs or probabilistic grammar may introduce further musical variety.

Comparison with existing computational music approaches shows that:

CFG-based approaches work well in representing simple melodic patterns.

Deep learning could add depth and musical variety to melodies.

Conclusion

This project demonstrates that context free grammar is applicable in melody generation. While the approach captures basic musical structures, possible extensions such as probabilistic CFG or the addition of harmony would render it more realistic. Future work would involve integrating deep learning models to enhance melody generation.

References

Steedman, M. J. (1984). A Generative Grammar for Jazz Chord Sequences. Music Perception, 2(1), 52-77.

Rohrmeier, M. (2011). Towards a Generative Syntax of Tonal Harmony. Journal of Mathematics and Music, 5(1), 35-53.

Chomsky, N. (1956). Three Models for the Description of Language. IRE Transactions on Information Theory, 2(3), 113-124.

Cope, D. (1996). Experiments in Musical Intelligence. A-R Editions.

Mido. Mido Documentation. Read the Docs, n.d., https://mido.readthedocs.io/en/stable/

Time Spent:

Luke Herron : 6-12 hours

3-4 hours researching music and math in music
4-6 hours on coding/writing report
I have never done anything with Mido Library, so actually going through and learning about what is contained within the library and how the objects work, such as track how the notes are actually converted from their numbers into actual sound to be played on the track object which acts almost as a list.

First attempted to use other libraries before Mido, such as piano and music21, but other complications would not allow me to use them.