Generating Music with Answer Set Programming

Arlo Rostirolla, Vijit Kumar

Table of Contents

List of Figures

List of Tables

Abstract

Musical composition is a complex process, with many different perspectives and genres. Although technically there are no hard rules regarding musical expression, there exists a large and complex set of soft rules that have been formalised to define the structure and organization of consonant music. Some are general, such as the ability of repetition to foster liking, while some are very specific, such as the notion that dominant 7 chords should always resolve to the root chord of the key. Given composition is the assignment of discrete sound atoms to a finite number of steps, this problem is very amenable to modelling via answer set programming. The aim of this study was to model musical composition using the Python Clingo API. The first problem to be explored was the simple problem of creating a single section of music, with drums, chords, and melodies, selecting only notes from a single key. The second problem was similar to the first, however the domain was expanded to include all keys , and extra chord types such as suspended 4^{th}. The third experiment involved creating a dynamic drum beat creator, with which tempo and rhythm could be controlled. A Turing test was used to evaluate experiment 1, resulting in 20 correct answers and 3 incorrect. Experiment 3 was evaluated by generating drumbeats and attempting to combine them with recordings from experiment 1 and 2. Albeit being a difficult problem, our programs generated reasonably musical pieces in the styles of low-fi jazz house and classical.

Introduction

Different genres of music can have different default patterns they follow. For example, in funk music, there are more pauses, often with very abrupt note endings. In techno, the bass drum plays on every beat. These intricacies are amenable to modelling with constraints. For the purposes of this assignment, we focussed on music which uses the western tonal system, and generally worked in the same key. We used Clingo and answer set programming to model this problem.

Answer set programming consists of finding stable models, or answer sets, given a database and a program. In the domain of music generation, often a database is made of musical notes and chords with different pitches and durations. The common rules of musical consonance are operationalized in the logical program. After this, a solver generates sets of compositions, all of which draw from the database of chords and notes and follow the constraint rules specified by the human user. There are many heuristics to guide the composition of consonant music. For example, the Pink Floyd guitarist David Gilmour is known for his emphasis on 'chord tones' in his solos. Chord tones are notes contained in the chords being played in the backing harmony. These are by default the most consonant notes that could be played in the melody. The most general rule of consonance is that there should be no dissonances between notes played at the same time. Examples of dissonances include the minor second interval (used in the Jaws theme), the tritone interval (used in sirens) and generally any interval that is not in the key of the chord being played. There have been many previous attempts to use logical programming in aid of musical composition.

Literature Review

Early works include Strasheela [1] and the Bol processor [2]. Strasheela was created in the Oz logic programming language, with the aim of implementing computer aided composition [3], not pure computer composition. Its author modelled musical notation as class objects, with different notes and timings. The user can influence the results by changing integer parameters. For example, one constraint is how large a leap between notes is. This can be modelled with finite domain integers. Computer aided composition can be effective, as it is difficult to model what timbre's sound good. A human can put together the sections given by a computer. This has been used successfully before to generate a Pink Floyd suite [4] using the Musenet transformer model [5]. The Bol processor was designed for apple hardware and could be used to perform inference with expert systems. Logic programming can be very effective at musical composition.

More recent research has attempted to generate twelve bar blues songs [6]. The Anton [7] and Armin [8] systems were designed to generate classical and trance music, respectively, using

ASP. Boenn and colleagues [9][10][11][12] in papers describing their musical composition system Anton detailed many rules to ensure a generation is musical. Interesting examples are that the interval between the highest and lowest note in a piece must be consonant, and that any note in the key can precede or follow the root note. In contrast, not every note in a key can precede another note.

Harmonization constraints require that the secondary instrument plays in the same key as the underlying, repetitive instrument. Within this key, there are certain intervals that should be rarely used, such as the tritone, which is commonly used in emergency services sirens due to its stressing affect. Other intervals, such as the third and fifth, are very consonant. However, when used in excess, can make a piece uninteresting. Boenn and colleagues disallowed use of the tritone entirely, and only allowed octave leaps when it was from the root note of the key to itself. They defined a predicate that was true whenever two notes were played at the same time and made constraints to ensure the two notes didn't clash.

Rhythm adds an extra dimension of complexity. In Anton 2.0 [7], rhythm was modelled using Farey sequences. Generally, rhythm can be modelled by specifying a silence atom, and specifying start and end times for every atom assigned. Rodrigo Pierto's [13] haspie answer set harmonizer used a complicated musicXML parser to read music and then generate harmonies. Their implementation of the musical logic rules was quite simple. The program would try determining whether the tonal centre of the composition in its current state was heading upwards or downwards, and then generate harmonies which followed the trend. They used integrity constraints to avoid assignments that 'clash' such as using the tritone in a chord.

For experiment 1 silence atoms were added, which could be maximized or minimized to control the number of silences. Another more difficult constraint would be to ensure music is repetitive, but not too repetitive. This could be simply done by generating a single repeating bar, say for the bass, and having this repeat while other instruments harmonize with it.

Clingo and Python Methods

The clingo python API [14] was used to ground and solve the logic program. Conversion from logically formatted compositions into actual sound was done through Spotify's interface with VST synthesizers, pedalboard [15][16]. This allowed us to compress the audio and add reverb to make it sound more like a produced piece of music. A python synthesizer [17] which played the basic waveforms (sine, square, sawtooth, triangle) was used for the core sound generation. The playsound module [18] along with multiprocessing, was used to playback results.

There was a downside to doing this, in that there was no guarantee the independent processes would play together in time, leading to some out of time recordings. To get a well timed recording,

each instrument was output to a wav file and then manually put together in audacity. For Experiment 2 we used midi as a more reliable way of outputting playable music.

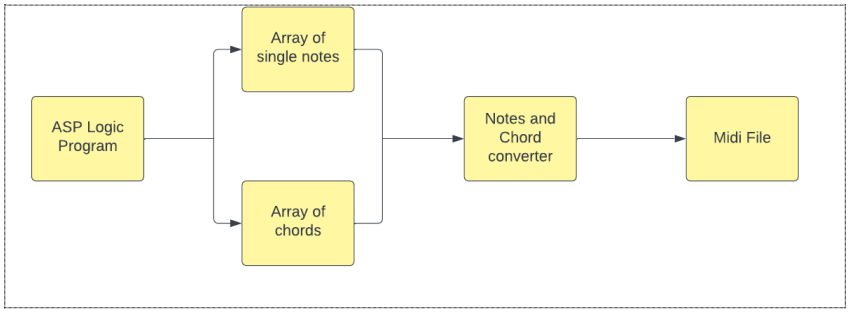
MIDI Methods

MIDI which also stands for Musical Instrument Digital Interface is currently a technical standard used in the music industry that allows a wide variety of digital musical instruments, computers, and respective audio devices to communicate with each other to record, play and edit a musical piece. An important thing to understand here is that MIDI itself does not transit the actual audio but rather it contains the information of each interaction with a key, note, button, knob, or slider of a given instrument with a specified timestamp. A single MIDI file can carry up to 16 channels of MIDI data, each of which can be routed to a separate device.

A basic unit of any music instrument is a note, and each note has a specific frequency that has a sound. For example, each key of piano represents a note and in total there are 88 piano notes. In our ASP logic program implementation, we used chords and scales as the information database and implemented a combination of rules to create a specific chord progression based on scales. Each of these chords originate from a list of 3 to 4 notes depending on the type of chord it is. Our ASP logic program was ground and solved using Clingo, the output model consisted of a grid of chords and notes (More details about this in Experiment 2 section below).

The output from the ASP was parsed into a list of notes and chords as shown in the Figure 1 below. In the next step we implemented a class that combined the two lists into a single 2D-numpy array. This was implemented using a sequence of 15 notes in a row followed by a chord throughout the entire musical piece. Each row of the NumPy array has 88 columns, and the index of each column represents the corresponding piano note with the value at each index representing the velocity at which that note should be played. Furthermore, each row in the NumPy array represents the note or a chord that should be played at given timestamp. After the NumPy array conversion, subsequently the array is converted into a MIDI file, figure 1 shows that process below.

Figure1.
Block Diagram of ASP to Midi file conversion



To convert a NumPy array into a midi file, we used a python package called Mido. The contents of a midi file usually have a list of messages that contains the information of what to play next along with the other parameters such as track details, key signature, instrument details, instrument channel etc. There are two kinds of messages in a midi file namely a meta message and message. In meta message we defined the 'set_tempo' parameter that specifies the number of microseconds in a beat with a value of 500000, which generally controls the pace of a musical piece. In a message, we defined the channel for piano instrument, and then used a list of parameters namely 'note_on', 'note_off', 'note', 'velocity', and 'time'. In figure 2 below we have a generated a sample of chord sequence that contains the information about the three notes 50, 53, 54 that are part of this chord sequence, along with the intensity at which this note should be played using the velocity parameter. The note-on and note-off parameter indicates the information about when to press and release a note respectively.

Figure 2

A chord sequence in a midi file

```
MidiFile(type=1, ticks_per_beat=480, tracks=[
  MidiTrack([
    MetaMessage('set_tempo', tempo=500000, time=0),
    Message('note_on', channel=0, note=50, velocity=100, time=0),
    Message('note_on', channel=0, note=53, velocity=100, time=0),
    Message('note_on', channel=0, note=60, velocity=100, time=0),
    Message('note_off', channel=0, note=50, velocity=120, time=880),
    Message('note_off', channel=0, note=53, velocity=120, time=0),     Message('note_off', channel=0,
note=60, velocity=120, time=0),
```

Experiment 1

Aim

The aim of experiment 1 was to compose in the style of low-fi jazz-house, a genre commonly found on youtube study playlists (See [19] for examples). To dissect this genre, 'low-fi' is due to the use of low fidelity sounds. A common way of obtaining low fidelity audio is playing the piece through an old or poor-quality speaker and recording the sound. Given the python synthesizers used were very basic, they were conducive to a 'low fi' sound. The jazz designation, at least for this genre, is due to the use of extended chords, such as 7^{th}, 9^{th}, 11^{th} and 12^{th} chords. House is a genre known for

repetitiveness, mixing basic chords, extended chords, and ensuring a basic 4/4 repetitive rhythmic structure. Jazz-House is essentially a cross between the two genres, it generally uses longer, more complicated structures with extended chords, but makes them repetitive, such they more resemble dance music. The Melbourne band ZFEX provides a good example of the sound of this genre [20].

Methods

A python script was used to contain all functions for this experiment. Five synthesizers were implemented with a sine wave for harmony, and one of each type of wave for the other 4, which were to be used for melodies. The volume of the synthesizers could be used to mute some when they were not necessary. The Clingo program was input as a string to the Clingo API. It stated all notes and chords in the E Dorian scale, and a basic set of drum symbols. Eight steps were defined for chords, along with a 4 by 8 grid/2 predicate for instruments. The initial assign/2 rule assigned the root chord of the key as the first chord in sequence. This was to centre the progression around the root, which is common in harmonisation rules. The second assigned a chord to every step after. The fill/3 predicate assigned a note/1 to every position in grid/2, and the beat/2 predicate assigned the drums.

Figure 3.
Experiment one constraints.

```
:- fill(Step, Instrument, Note), fill(Step+1, Instrument, Note).
:- fill(Step, Instrument, "b2"), fill(Step, Instrument+1, "cs3");fill(Step, Instrument+2, "cs3");
fill(Step, Instrument-1, "cs3");fill(Step, Instrument-2, "cs3").
:- fill(Step, Instrument, "e2"), fill(Step, Instrument+1, "fs2");fill(Step, Instrument+2, "fs2");
fill(Step, Instrument-1, "fs2"); fill(Step, Instrument-2, "fs2").
:- assign(Bar, Chord), assign(Bar+1, Chord).
:- assign(Bar, dmaj7;dmaj9), assign(Bar+1, dmaj7;dmaj9).
```

The first integrity constraint ensured the same note wasn't assigned twice in a row. The second two made sure that a major second clash did not occur between notes occurring at the same time. The fourth hard constraint ensured that the same chord wasn't played twice, and the last hard constraint made sure chords with the same key didn't play sequentially.

Figure 4.
Experiment 1 optimization statements

```
#minimize{1, fill, Step, Instrument, Note : fill(Step, Instrument, Note), Note != false}.
#maximize{4, fill, Step, Instrument, Note: fill(Step, Instrument, Note), rootnote(Note);
        1, fill, Step, Instrument, Note: fill(Step, Instrument, Note),third(Note);        1, fill, Step,
Instrument, Note: fill(Step, Instrument, Note), fifth(Note);
        1, fill, Step, Instrument, Note: fill(Step, Instrument, Note), seventh(Note)}.
```

The frequency of silences (false notes) and emphasis on chord tones could be specified using the optimization statements. Also, note assignments that were the root, third, fifth or seventh of the current chord were maximized. Drums were hard coded in Clingo to simplify the problem, with a basic kick – hat – snare – hat pattern implemented. The constraints for drums will be explained further in experiment 3.

Symbol outputs were stored in a nested list. For output, a random model was picked from the output, parsed using basic python functions, and stored as note strings which could be played by the synthesizer. Multiprocessing was used to play back output. Alternatively, a commented section at the bottom outlined the process of outputting each channel to a wav file.

Results

The results can be found in the results/Experiment1 folder of our GitHub repo [21]. We generated around 8 pieces using different variants of the Clingo program.

Discussion

It was difficult to properly evaluate the output, as the use of multiprocessing could lead to timing differences between instruments. On some recordings it is barely noticeable, however occasionally it would be completely out of time. This led us to experiment with midi output in experiment 2.

To evaluate the outputs from these experiments was difficult, and the only way of properly quantifying its musicality was through survey. A virtual Turing test was conducted on a local community group. Two songs were presented, one being the result of the CAC version of our output, and another being a song produced by a human. Participants were told that both compositions were mixed and organised by a human, however one of the songs had its sections composed by a logic programming algorithm. Twenty participants correctly identified the AI composed song, whereas three misidentified it. The main reasons given for this identification was that the AI composed song was more repetitive. This result guided our reasoning for experiment 2; classical music would be harder to differentiate given its lack of repetitiveness. The results of the poll can be found in appendix A.

Experiment 2

Aim

The aim of experiment 2 was to make the difficulty more complex. In experiment 1, the chord and note selection were constrained to a key, they were not allowed to conflict. In experiment 2, all chords with major, minor, sus4, and dom7 qualities were defined, along with 35 chromatic notes. The intention was to generate classical music.

Methods

The python program was written in a similar fashion to experiment 1.
The main difference was a significantly expanded database. Major, minor, suspended and dominant7 chords were included for the harmony. The melody was chosen to be notes contained within the scale of the chord that was currently being played. For example, if an E major chord was being played at a certain step, the melody would be selected from a predicate of notes contained within the E major scale. This pattern was repeated for all 12 root notes combined with the 4 modal scales. A predicate was initialized to ensure that any patterns such as scales and chords also considered higher/lower notes. One octave up from any note is the same note, which serves the same purpose within the pattern.

Figure 5.
Rule to expand selection to higher/lower octaves

```
chord(KeyA,  TypeA, RootA, ThirdA, FifthA) :- note(RootB), note(ThirdB), note(FifthB), chord(KeyB,
        TypeB,  RootB, ThirdB, FifthB), RootA = RootB + 13, ThirdA =  ThirdB + 13, FifthA = FifthB + 13.
```

To create a whole song, a 16 by 16 grid was defined. One chord was assigned to every step on the first axis, and 16 notes were assigned to every step on the y axis.
The hard constraints implemented for this experiment can be seen in figure 4. Optimisation statements can be found in figure 5.

The first constraint ensures that the same note isn't played twice. This was implemented due to cheating behaviour (returning the same note played repeatedly). Additional steps were added to this constraint, before utilising aggregate predicates to reduce repetition. The second constraint is the same as the first, however it prevents chords from repeating.

Figure 6.
Hard constraints for experiment 2.

```
1 Step.   :- fill(Bar, Step, Note), fill(Bar, Step+1, Note).
2.        :- assign(Bar, Key, Chord, Root, Third, Fifth), assign(Bar+1, Key, Chord, Root, Third, Fifth).
3.        :- assign(Bar, KeyB, dom7, RootB, ThirdB, FifthB), fifth(KeyX, KeyB), assign(Bar+1, KeyA,
              ChordA, RootA, ThirdA, FifthA), KeyX != KeyA.
4.        :- fill(Bar, Step, NoteA), fill(Bar, Step+1, NoteB), | NoteB - NoteA | > 10.
5.        :- grid(Bar,), assign(Bar, Key, Chord, Root, Third, Fifth), fill(Bar, Step, Note),
scale(Key, Chord, Root, Second, Third, Fourth, Fifth, Sixth, Seventh, Eighth),
          Note != Root, Note != Second, Note != Third, Note != Fourth, Note != Fifth,
Note != Sixth, Note != Seventh.
```

Figure 7.
Optimization statements for experiment 2

```
#maximize{1@1: fill(Bar, Step, Note), assign(Bar, Key, Type, Root, Third, Fifth), Note = Root;
           1@2: fill(Bar, Step, Note), assign(Bar, Key, Type, Root, Third, Fifth), Note = Fifth;
           1@3: fill(Bar, Step, Note), assign(Bar, Key, Type, Root, Third, Fifth), Note = Third;
           1: assign(Bar, KeyA, TypeA, RootA, ThirdA, FifthA), assign(Bar+1, KeyB, TypeB, RootB,
                ThirdB, FifthB), RootA - RootB == 3;
           1: assign(Bar, KeyA, TypeA, RootA, ThirdA, FifthA), assign(Bar+1, KeyB, TypeB, RootB,
                                    ThirdB, FifthB), RootA - RootB == 5;
           1: assign(Bar, KeyA, TypeA, RootA, ThirdA, FifthA), assign(Bar+1, KeyB, TypeB, RootB,
                ThirdB, FifthB), RootA - RootB == 7}.

#minimize{1@1: fill(_, Step, Note), fill(_, Step+1, Note+6);
          1@2: fill(_, Step, Note), fill(_, Step+1, Note+1);
          1: assign(Bar, Key, sus4, Root, Third, Fifth);
          1: assign(Bar, Key, dom7, Root, Third, Fifth)}.
```

The third constraint ensures that if a dominant7 chord is ever assigned to a bar, the next bar will be the root. Dominant7 to root transitions are very consonant, however the transition to any other chords can sound dissonant, unless placed in context with great care. The fourth constraint ensures notes don't jump intervals of more than ten pitches, as discussed in the literature review. The final hard constraint ensures that when a certain chord is being played, only notes contained within the scale of that chord are assigned to the melody.

The number one priority of the maximize statement is to maximize the assignment of the root note of the chord being played that bar. The 2nd priority is the third, and 3rd priority is the fifth. This is to ensure that chord-tones are emphasised. The other three have equal priority and were written to maximize chord changes that are a third/fifth/or seventh apart, respectively.

The minimize statement aims firstly to minimize use of the tritone, and the second minimizes use of the minor second. The other two priorities minimize the use of the sus4 and dom7 chord types, as these are generally only used rarely in one chord progression.

Results

The initial results of this experiment were very dissonant, which was to be expected given the complexity of the problem. With added guidance, we eventually came upon a repetitive, yet classical sounding piece. After adding constraints to avoid repetition, the piece came to sound more like Pokémon battle music. All results can be found under the experiment 2 folder of the GitHub repo.

Discussion

Initially, to be able to access the current chord being played, we initialized the grid as a 16 by 256 by 2 array. This allowed us to specify the constraint that assured the melody played within the chords scale. The first 16 was for chord assignments, the 256 for note assignments, and the 2 dimension specified different instruments. This was not a correct way to formulate the problem, as it led to the fill/3 predicate being true for 8,192 grid squares. This led to very large values being returned in the array. We fixed this by setting the grid to a 16 by 16 array. This way, for every chord assigned in axis 1, 16 notes were assigned in axis 2.

This experiment was much harder than the first, and it was difficult to constrain dissonant musical choices without making the music overly repetitive. However, it was not impossible, and we managed to create both creative and repetitive pieces.

Experiment 3

Aim

The aim of experiment 3 was to create drum rhythms using Farey sequences as done in the Anton system [7]. Previously, in experiment 1, drums were hardcoded in Clingo as shown in the following code:

Figure 8.
Hardcoded drums for experiment 1

```
:- beat(Step, bassdrum), beat(Step-1, snare).
:- beat(Step, bassdrum), beat(Step+1, snare).
:- beat(Step, DrumType), Step\4 = 0, DrumType != bassdrum.
:- beat(Step, DrumType), Step = (2; 6; 10; 14), DrumType != snare.
:- beat(Step, DrumType), Step = (1; 3; 5; 7; 9; 11; 13; 15), DrumType != hat.
```

These constraints hard code that the bass drum should be on the first step, the hi-hat every second step, and the snare on the third step, for every four steps. Although any drumbeat could be programmed this way, it is tedious, and a musician would be better served playing the beat on a drum kit. Our aim was to have an automated way of creating drumbeats from a certain BPM.

Farey sequences are a sequence of fractions [22]. The mathematical definition is the set of coprime rational numbers P/Q, with $0 < P < Q < N$, N being the maximum denominator. An example of a Farey sequence represented as a tree can be found in figure 9. Farey sequences allow mathematical segmentation of a period of time into different intervals. Rhythm in music is generally represented as a fraction. For instance, in experiment 2 we used a time signature of 1/16, such that for every chord that was played, 16 notes were played for the melody. One instrument, generally the drums, is treated as a reference frame or the denominator of the time signature, and the rhythm of other instruments can be represented as the numerator of a fraction, or the number of notes played by

this instrument for every N number played by the reference instrument. Farey sequences are commonly used to quantise music into discrete rhythms [23]. We used the filtered Farey sequence f_{27} as suggested by [23], given its optimal mapping to intervals in classical music.

Figure 9.

Farey tree of Farey sequence F_5.

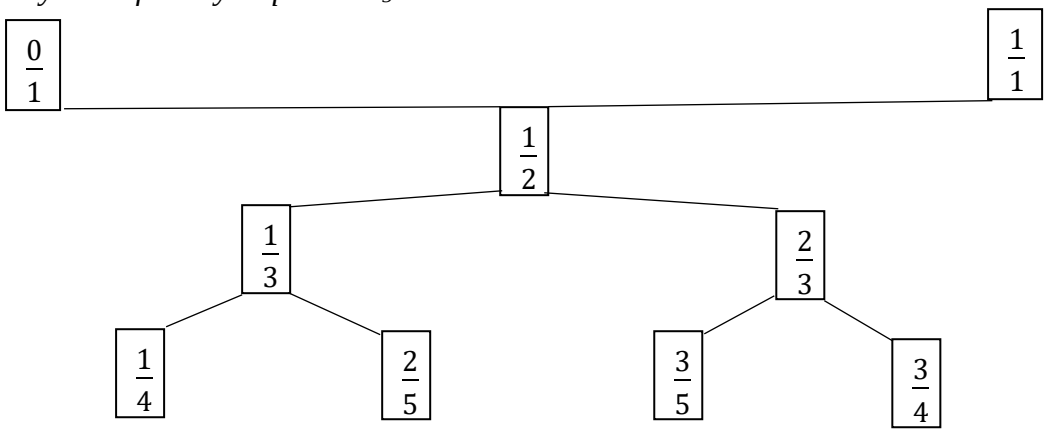


Figure 10.

Segment of the filtered Farey sequence F_{27} used to represent beat subdivisions

$$\frac{1}{24}\ \frac{1}{16}\ \frac{1}{12}\ \frac{1}{8}\ \frac{1}{4}\ \frac{2}{4}\ \frac{3}{4}\ \frac{4}{4}\ \frac{6}{4}\ \frac{8}{4}\ \frac{9}{4}$$

Regarding the bpm, these intervals represent 24th notes, 16th notes, 12th note triplets, eighth notes, quarter notes, half notes, half note triplets, whole notes, whole note triplets, double whole notes, and double whole note triplets. Intervals can be subdivided further, but at smaller intervals, they increasingly surpass the ability of the listener to detect them. In using Farey sequences, we hoped to imitate the rhythmic aspect of music, by varying the duration of silences between notes.

Methods

Representing these fractions in Clingo was not possible, as it cannot deal with decimal numbers. Instead of using the fractional ratio, we used the bpm ratio of each regarding the reference frame (quarter notes) to determine the number of notes played per minute of music. The interval between notes was calculated as 60/N, where N is the number of notes played for that time signature over a minute. This way, whenever the bpm was changed in python, Clingo would automatically update both the number of steps per minute and the time interval between them.

The sounds used were bassdrum, hi-hat, snare, tom, snareroll, ride and a crash. Given the issue of representing numbers other than integers in Clingo, it was not possible to create one sequence and then divide it by both odd and even numbers. Thus, a sequence was created for every interval type.

Initial experimentation showed there were issues with this approach. With 8 sounds filling 11 sequences of varying lengths (See Table 1), the state space of this problem is $8^{600} * 8^{480} \ldots * 8^{13}$ or $\sim 1.34e^{1732}$. Given an estimate of the number of elementary particles in the observable universe [24] , the state space of a one minute drum loop with this problem representation is roughly the same order of magnitude as the number of elementary particles in a multiverse comprised of $6.3e^{831}$ universes. Additionally, splitting what should be a single sequence into 11 difference sequences complicated the issue of writing constraints between them. Since they varied in lengths, the different types of intervals could not be mapped to each other, and thus any constraints we wrote only applied to a single interval type.

To improve representation of this problem, all triplets were removed, given they could not be represented by whole numbers. The largest array (sixteenth notes) was initialised. Every even integer ratio relative to the sixteenth note was calculated. When assigning to the sixteenth note array, each note ratio N would only assign to an index I where I mod N = 0. This reduced the state space to 8^{600} and made it far easier to form constraints which integrated different note types.

Table 1.
Interval type, number of beats N, and even interval ratios relative to 1/16 notes for a BPM of 120

Type	1/24	1/16	1/12	1/8	1/4	1/2	3/4	4/4	6/4	8/4	9/4
N	600	480	300	240	120	60	40	30	20	15	13
Ratio		1		2	4	8		16		32	
										Total N	1918

Some of the interval types were constrained to only play a single type of drumbeat. For example, all quarter intervals were assigned the bass drum, all eighth intervals were assigned the ride, and all half notes were assigned the snare. Other intervals were allowed to vary. Some intervals were constrained to not play a certain type of drum sound. 24^{th}, 16^{th} and 12^{th} note rhythms were not allowed to be assigned the crash or ride. We used a count aggregate inside a helper predicate to control the prevalence of any drum sound. Values were set arbitrarily and then tuned until they returned expected results.

Results

All results can be found in the experiment 3 results folder within the GitHub repo. Although we were able to write sequences with varying rhythms, it was very difficult to organize them into sections. As a result, most of our outputs came to resemble drum solos, whether in the genre of heavy metal (attempt 1) or jazz (attempt 5).

Discussion

The ability of Farey sequences to emulate rhythm was evident from our experimentations. Despite not involving melody nor harmony, rhythm was difficult to organize in a coherent way. The final product is very customizable. The user can specify the BPM, and control output via constraints and optimisation statements. We evaluated this system by trying to combine it with one of the recordings from experiment 2. Although the timing was not perfect, the drums did fit well with that recording.

The database initially contained more sounds; however it was difficult to include them while also specifying how often they should occur. Counting how many steps a particular drum beat was assigned too helped, and then constrained that value helped, but did not solve the problem. If we were to extend this experiment further, we would use the basic midi drum pack, which contains 47 sounds. This would make the state space unwieldy for the first problem representation.

The second representation was much more optimal, however lacked odd intervals. It would be beneficial to attempt to find a way to use these. It may be sufficient to double the sequence size, and double the interval ratio's themselves, given any natural number doubled results in an even number. This way, using the original ratios in Table 1, sixteenth notes would occur every 2 steps, twelfth notes every 3 steps, and eighth notes every 4 steps.

Experiment 2 Extension - Chord Progressions Aim

This experiment was an extension of the Experiment 2. In experiment 2 we generated a sequence of notes along with chords, while in this experiment we only focussed on creating a sequence of chords in major scales. This type of progression is usually used in mainstream pop songs with upbeat melodies.

Method

Since this experiment is an extension of experiment 2 it was based on the same database. We specifically used major chords and scales to create a meaningful progression. We used a grid of 16 rows and 3 columns, where each has 3 chords. Furthermore, we also specified that in each row no chords should be repeated. For example, at row 0 and col 0, there is A major chord, then in the next two columns it should not be repeated. Additionally, we also specified a rule where the starting chord of each row should not be repeated for 4 consecutive rows. So far, the rules that have been discussed were implemented to avoid repetition every four rows. To create a basic chord progression rule we also implemented a rule where starting chord of each row decides the scale that row is in and then the following two chords should be the IV and V chord of that scale respectively. Below in Figure 11 is the code fragment that was used to implement this rule.

Figure 11

Code fragment of chord progression experiment

```
% A grid of 16 bars as rows and 4 chords as column in each row
grid(0..15, 0..2).

% Assign a key in each bar
{fill(ROW,COL,A,B,C,D,E): chord(A,B,C,D,E),note(C),note(D),note(E)} = 1 :- grid(ROW,COL).
% Chord should not repeat in the same row
:- fill(ROW,COL,CHORD,TYPE,A,B,C), scale(_,TYPE,P,_,_,_,_,_,_), TYPE != major,  COL==0,
A!=P.
:- fill(ROW,COL0,CHORD,_,A,B,C) ,fill(ROW,COL1,CHORD1,TYPE,D,E,F),
                        scale(CHORD,TYPE,A,_,_,Q,_,_,_),     TYPE     !=     major,
COL0==0, COL1==1, D!=Q.
:- fill(ROW,COL0,CHORD,_,A,B,C) ,fill(ROW,COL2,CHORD2,TYPE,G,H,I),
                        scale(CHORD,TYPE,A,_,_,_,R,_,_), TYPE != major,
COL0==0, COL1==2, G!=R.
:- fill(ROW,COL,CHORD,TYPE,G,H,I), scale(_,TYPE,_,_,_,_,R,_,_), TYPE != major, COL==2,
G!=R.
```

Results

The result of this experiment can be found in the convertedMidi folder with a file named ‘midi_new.mid’ in our GitHub repository. Although we have only implemented one rule that specified the I-IV-V chord progression, the result was not very accurate as expected but the overall tone of the chord progression was somewhat like melodies in modern day pop songs.

Conclusion

A few barriers were faced in the production of this system. We could not customize the uniqueness of an output relative to other outputs. To have variable output, we created 50 models and randomly selected one to play. The conversion to midi added extra complexity, although it was necessary avoid the syncing issues with multiprocessing. Many of midi’s abilities were untouched, and future work should consider using aspects such as velocity and time spent between note on and note off events.

In conducting these experiments, we aimed to investigate the potential for Clingo to model music composition. The software proved to be very powerful at doing so; even succeeding at convincing 3 human listeners that its music had been composed by a human. Constraints and optimization statements allowed precise control over the output. The compositions were not perfect; however, many were enjoyable to listen to. Using logic programming allowed composition free from previous influences. Whereas statistical methods such as musenet rely on previous composition, Clingo relies only on the user’s input, often leading to novel and interesting musical phrases free from human bias not explicitly programmed.

References

[1] T. Anders, "Composing Music by Composing Rules: Design and Usage of a Generic Music Constraint System", *Strasheela.sourceforge.net*, 2017. [Online]. Available: http://strasheela.sourceforge.net/documents/TorstenAnders-PhDThesis.pdf. [Accessed: 09-Apr- 2022].

[2] Anders, T., Anagnostopoulou, C., & Alcorn, M. (n.d.). Strasheela: Design and Usage of a Music Composition Environment Based on the Oz Programming Model. In *Multiparadigm Programming in Mozart/Oz* (pp. 277–291). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-31845-3_23

[3] B. Bel, "Migrating Musical Concepts: An Overview of the Bol Processor", Hal.archivesouvertes.fr, 2022. [Online]. Available: https://hal.archives-ouvertes.fr/file/index/docid/250274/filename/744.pdf.

[4] A. Rostirolla, "AI generated pink floyd suite [instrumental]", *Youtube.com*, 2022. [Online]. Available: https://www.youtube.com/watch?v=Q55ZqfGyWU0.

[5]"MuseNet", *OpenAI*, 2022. [Online]. Available: https://openai.com/blog/musenet/.

[6] F. Everardo, G. Gil, O. Perez Alcala and G. Silva, "Using answer set programming to detect and compose music in the structure of twelve bar blues", *Ceur-ws.org*, 2022. [Online]. Available: http://ceur-ws.org/Vol-2818/paper02.pdf.

[7] Boenn, G, Brain, M, De Vos, M & ffitch, J 2008, Anton: Answer Set Programming in the Service of Music. in MPagnucco & M Thielscher (eds), Proceedings of the Twelth International Workshop on Non-Monotonic Reasoning. University of New South Wales, Sydney, pp. 85-93, Proceedings of the Twelfth InternationalWorkshop on Non-Monotonic Reasoning, 1/09/08.

[8] F. Everardo Pérez and F. Aguilera Ramírez, "Armin: Automatic Trance Music Composition using Answer Set Programming", *Fundamenta Informaticae*, vol. 113, no. 1, pp. 79-96, 2011. Available: 10.3233/fi-2011-600.

[9] Boenn, G., Brain, M., De Vos, M. and ffitch, J. (2008) Automatic composition of melodic and harmonic music by answer set programming. In: Logic Programming. Proceedings of the 24thInternational Conference, ICLP 2008.5366 ed. Lecture Notes in Computer Science . Springer, pp. 160-174. ISBN 978-3-540-89981-5

[10] Boenn, G, Brain, M, De Vos, M & ffitch, J 2012, “Computational Music Theory”. Musical Metacreation: Papers from the 2012 AIIDE Workshop AAAI Technical Report WS-12-16

[11] Boenn, G, Brain, M, De Vos, M & ffitch, J 2011, Anton — A Rule-Based Composition System. In Proceedings of ICMC 2011. ICMC, University of Huddersfield and ICMA, pp. 135-138.

[12] G. BOENN, M. BRAIN, M. DE VOS and J. FFITCH, "Automatic music composition using answer set programming", *Theory and Practice of Logic Programming*, vol. 11, no. 2-3, pp. 397-427, 2011. Available: 10.1017/s1471068410000530.

[13] R. Prieto, "GitHub - Trigork/haspie: Tool for harmonizing musical pieces based in Answer Set Programming", *GitHub*, 2022. [Online]. Available: https://github.com/Trigork/haspie.

[14] "clingo API documentation", *Potassco.org*, 2022. [Online]. Available: https://potassco.org/clingo/python-api/5.4/.

[15] P. Sobot and D. Braun, "How to call audio plugins from within Python?", *Stack Overflow*, 2022. [Online]. Available: https://stackoverflow.com/questions/37555711/how-to-call-audioplugins-from-within-python.

[16] "pedalboard", *PyPI*, 2022. [Online]. Available: https://pypi.org/project/pedalboard/.

[17] "synthesizer", *PyPI*, 2022. [Online]. Available: https://pypi.org/project/synthesizer/.

[18] - "playsound", *PyPI*, 2022. [Online]. Available: https://pypi.org/project/playsound/.

[19] "Low-fi jazz house study search results", *Youtube.com*, 2022. [Online]. Available: https://www.youtube.com/results?search_query=low+fi+jazz+house+study.

[20]"Zeitgeist Freedom Energy Exchange - Big Up", *Youtube.com*. [Online]. Available:

https://www.youtube.com/watch?v=Yzi3Mk2ULHY&list=OLAK5uy_n2ZuVdlp3N_yDl2mGZRq9JF1rQVB75p0&index=1.

[21] "course-projectunder15statespeedwalkingchampions". *Github.com.* [Online]. Available: https://github.com/RMIT-COSC2780-IDM22/course-projectunder15statespeedwalkingchampions/tree/main/results/Experiment%201

[22] J. Ainsworth, M. Dawson, J. Pianta, & J. Warwick "The Farey Sequence". [Online]. Available https://www.maths.ed.ac.uk/~v1ranick/fareyproject.pdf

[23] G. Boenn, "*The Farey Sequence as a Model for Musical Rhythm and Meter*" In Computational models of rhythm and meter. [Online]. Available https://link.springer.com/content/pdf/10.1007/978-3-319-76285-2.pdf

[24] J. Bennet, "How Many Particles Are In The Observable Universe". [Online]. Available https://www.popularmechanics.com/space/a27259/how-many-particles-are-in-the-entireuniverse/

Appendices

Appendix A

Results of a Turing test between song A (AI assisted composition) and song B (human composition)

Hi all! I'm doing a uni assignment where i'm trying to compose music with logic programming (not the DAW). As you can understand, how musical something is isn't really something you can evaluate mathematically, and I have to evaluate the results for my report.

I was hoping I could do a simple survey/turing test on here for my discussion section.
There are two electronic songs below; one was made by me, and the other was mostly composed with programs. I programmed the drums/mixed/mastered both, but for one, each section (ie verse chorus) was composed by the AI. I stuck them together and mixed it all.
Song A:
https://soundcloud.com/arlo-porter/song-a/s-JQpmQPvWqwa?utm_source=clipboard&utm_medium=text&utm_campaign=social_sharing

Song B:
https://soundcloud.com/arlo-porter/song-b/s-fgxrxf0bCTa?utm_source=clipboard&utm_medium=text&utm_campaign=social_sharing

There is a poll below with two questions, if you would like to leave your reasoning in a comment as to why you thought one seemed more computer generated than the other that would also be handy!