Analyzing and Mutating Structural Patterns To Create New Music in the Style of Other Music

Drew Jex and Dan Ventura

Abstract

A common element in music is the concept of repeating patterns. It is the careful organization and combination of notes and rhythms that forms these patterns, and indeed the very essence of music itself. Because music is definitive and pattern-based, we as humans recognize it intuitively as music, distinct and discernible from other random sounds. Music can therefore be roughly defined as ”a combination of patterns, plus some variety.” If we can describe music in terms of such patterns, then it becomes at once easier to define, manipulate, and imitate. We propose a method for imitating and creating music from existing music through the analysis, manipulation, and reconstruction of repeated patterns found in music.

Introduction

When it comes to music, the range of possibilities is endless. The broad spectrum of various musical styles and flavors has slowly grown over the ages as various cultures and people have added to history's vast collection of unique songs. Each song composed is different, yet most are easily distinguishable as music and not just some strange, random compilation of sounds. We propose that an integral element of music is the existence of temporal and melodic patterns (Rolland 1999). Our brains can more easily grasp such patterns because they are distinct and discernible, separate from other sounds we hear. When these patterns are repeated over time, music is born.

Music is comprised of both temporal patterns and melodic patterns. Without time, music cannot exist. Furthermore, music contains definitive, melodic notes. The combination of notes with a repeated, temporal beat is the essence of music. From the underlying beat, other rhythms form through the careful placement of notes. When these melodies are repeated, even at small levels, they become more memorable to us. In our research, therefore, we have adopted our own definition of music to be ”a combination of repeated patterns, plus some variety.” Our methods for creating music are rooted in this simple, core definition.

If music can be broken down into a combination of patterns, then it becomes easier to manage, manipulate, and imitate. Patterns by definition can be copied. They can also be easily changed because they are easy to define. In this paper we demonstrate how we can break a song into a collection of patterns, manipulate those patterns through a variety of means, and then reconstruct those patterns in order to create a new song imitating the style of the original song through simple pattern-imitation (Pachet 2003).

Such a computer algorithm has the potential to make song-writing much more efficient, and even completely automated. A computer algorithm which can create music based on songs we already like makes it easier to tailor likes and dislikes to the individual. Songs that we hear in computer games and movies can sound different for each person based on their personal taste. This project opens the door for increased productivity among composers and a better listening experience tailored for each client. The ability to create original music from previous compositions minimizes the need for a human composer to fulfill every demand in the fast-paced song industry. Custom ringtones and jingles could be automated based on user preferences. Game designers could generate background music for their games in real time. Composers could use generated music as inspiration for their own compositions, potentially overcoming writers block. Clients could create a personalized playlist of songs and then generate new music written in the style of songs they enjoy the most. This project has the potential to impact composers and consumers alike.

Approach

Our approach to developing a computer algorithm which can create original, interesting music is rooted in the idea that music is a collection of patterns that be deconstructed, manipulated, and finally put back together. By defining music as having patterns and not just being some abstract entity that causes emotional response, we have defined it in terms that a computer can understand, and is therefore much more manageable.

Patterns, as we have mentioned, can be imitated and changed very easily. By breaking a song down into a manageable set of patterns, a computer can then easily copy, rearrange, or mutate particular patterns however we want. The concept makes creating new music relatively simple because the data model is so flexible.

Our approach, however, does not merely create these patterns from scratch. By providing the computer with a song or set of songs, the algorithm can essentially implement patterns and structures that we as the listener might already find enjoyable. The algorithm, therefore, has a blueprint from which to work, and from there can further rearrange, mutate, or imitate individual patterns.

Our approach essentially contains three processes in order to create a new song:

1. Deconstructing an existing song into a manageable set of patterns.
2. Manipulating, rearranging or copying the patterns.
3. Putting the patterns back together.

Furthermore, we will refer to the computer algorithm that completes these tasks as ”MusicMaker”.

1. Deconstruction

The first step MusicMaker performs is the deconstruction of an existing song into a collection of patterns. This song provided must be in MIDI format because such a format can easily be parsed and converted into other formats. This is precisely what MusicMaker accomplishes during the deconstruction phase.

To perform deconstruction, MusicMaker primarily uses two helpful components: an open source PHP MIDI2Txt converter and the MusicParser.

Midi2Txt (mf2)

When confronted with a MIDI file, the PHP Midi2Txt library converts the MIDI file into a more human-readable format, referred to as ”MF2T” format. The resulting text file contains simple instructions for which notes to play and how long to play them:

```
MFile 1 3 480
MTrk
0 TimeSig 4/4 24 8
0 Tempo 600000
0 PrCh ch=1 p=1\n
0 On ch=1 n=70 v=94
0 On ch=1 n=70 v=94
120 On ch=1 n=67 v=82
120 On ch=1 n=74 v=82
120 On ch=1 n=67 v=82
120 On ch=1 n=74 v=82
240 On ch=1 n=79 v=93
240 On ch=1 n=79 v=93
480 On ch=1 n=70 v=76
480 On ch=1 n=70 v=76
600 On ch=1 n=67 v=74
600 On ch=1 n=67 v=74
720 On ch=1 n=79 v=60
720 On ch=1 n=67 v=60
720 On ch=1 n=79 v=60
720 On ch=1 n=67 v=60
960 On ch=1 n=77 v=74
960 On ch=1 n=77 v=74
960 On ch=1 n=77 v=64
960 On ch=1 n=77 v=64
960 On ch=1 n=77 v=55
960 On ch=1 n=77 v=55
```

This file contains its own grammar and rules for how to display MIDI data. The rules for reading a MF2T file can be found at http://valentin.dasdeck.com/php/midi/documentation.htmmf2t.

MusicParser Now that we have a human-readable text file of the original song, we need to convert it to the data model used in MusicMaker. To accomplish this, we have written a MusicParser that, given a MF2T file, converts the file to an internal data structure in MusicMaker using the specific grammer for a MF2T file.

The data model used in MusicMaker stems from the idea that a song is composed of repeating patterns. A detailed explanation of the model is provided below.

Song Object A song is defined as set of parts and a multi-dimensional array of indexes which correspond to particular parts. The first dimension of the array stores each track of the song. The second dimension corresponds with the index of the part that is played at that point in the song. Although parts in the set do not have to be equal in length, parts played at the same time across tracks are expected to be the same length. (see below)

```
class Song {
   public string title;
   public string author;
   public parts = [Part]; //array of part
       objects
   public structure = [Track][Part_Index];
       //defines structure of song
}
```

For example, a song could contain the following valid data:

```
Song song = new Song("Drew Jex", "New
    Song");
song.parts = [new Part(), new Part(), new
    Part()];
song.structure = [[0, 0, 1],[1, 2, 1]];
```

This song has two tracks, one of which plays the part located at index 0 twice followed by the part located at index 1 and the other which plays the part at index 1 followed by index 2 then finishes with the same part at index 1 again. At the beginning of the song, the parts at index 0 and index 1 would be playing at the same time.

This model makes it simple to rearrange parts or repeat sections without having to copy and paste all the corresponding notes and rhythms. When a song is played, the program simply converts each track in MusicMakers data model to a track in MIDI format along with each corresponding array of parts. Keep in mind that the song structure emphasizes specific notes and rhythms, not dynamics and other musical elements. These may be added later when we analyze patterns at a bigger level than just notes/rhythms.

Part Object A part is defined as having a set of measure objects, a tempo, dynamic, time-signature, and an instrument identifier (which corresponds directly with MIDIs instrument database). The model is outlined below:

```
class Part {
   public int instrument;
   public TIME_SIG time_signature;
   public int time_increment;
   public int tempo;
   public int volume;
   public measures = [Measure];
}
```

Each measure is played in the order they are in the array.

Measure Object A measure is defined similarly to what you would expect in traditional notational music. We have implemented them in the MusicMaker data model even though they are technically used in MIDI format. The object contains an array of note objects and an array of chord objects, shown below:

```
class Measure {
   public notes = [Note];
   public chords = [Chord];
}
```

Note Object A note is defined as having a time (relative to the measure), any number of specific tones being played at that time, any special effects, and then the length of the note:

```
class Note {
   public time;
   public note = []
   public effect;
   public length;
}
```

Pattern Object Pattern objects are used to define a particular repeating subsection or pattern found in the song. When MusicMaker runs its algorithm, it first defines each deconstructed piece in terms of pattern objects. Pattern objects are not part of the Song model. They are used to track repeating elements in the music so that MusicMaker can create something new. Later, we will discuss how MusicMaker finds and creates these pattern objects.

Notice how pattern objects are not explicit copies of a particular part, but rather templates for creating similar parts. For example, they do not contain any indication of rhythm or timing. Rather, they consist of a length, an array of notes, and a number representing the number of notes that were played in the original song. The reason for this loose definition is to introduce variety when creating the generated song. These are flexible data objects, however, and further observation of the original piece can be recorded here as we continue work on the program. (Johnson 2016)

Also notice that there is a difference between a ”pattern of music” and a ”pattern object”. For most of this paper, we refer to repeating patterns as literal notes and rhythms in the music itself that are copied over and over throughout a song. When MusicMaker breaks down the song into pieces, it looks at the actual pieces of music to discover where patterns are located in the song. After making that discovery, it creates ”pattern objects” to represent the patterns found. These pattern objects are simply redefining those pieces so that variety and other elements can be introduced into the final piece.

This pattern object is not used as part of the overarching Song model, but it is used when analyzing the original song and identifying repeated elements. A pattern object consists of the following:

```
class Pattern {
   public int length;
   public chords [Chord];
   public int num_notes;
   public note_sack [int];
}
```

Once the MusicParser parses a MF2T file to the MusicMaker data model, MusicMaker can now begin searching the model for repeated patterns. Understand that the model contains a separate part object for each track. Each analysis, manipulation, and reconstruction of the parts (at this point) is done independent of other tracks. Future work would include more integration between tracks during analysis.

Another component of MusicMaker, MusicAnalyzer, performs the essential task of finding patterns. The pseudo code for this function is provided below:

```
  patterns = [], structure = [];
      //initialize arrays
  if (partObj.timeIncrement ==
      patternLength) {
    while (!empty(partObj.measures)) {
      next =
          array_shift(partObj.measures);
          //get next measure
      currentKey = key(next);
      currentMeasure = next[current_key];
      patterns.push(new
          Pattern(patternLength,
          currentMeasure.chords,
          currentMeasure.notes)); //create
          new pattern obj
      id = patterns.length - 1;
      structure[currentKey] = id
      foreach (partObj.measures as key =>
          value) {
        if (getDiff(currentMeasure, value)
            < similarityValue) {
          structure[key] = id;
          unset(partObj.measures[key]);
              //delete any measures that
              meet similarity threshold
        }
      }
    }
  }
  return new Object(patterns,
      ksort(structure))
}
```

In addition to the original song, this function exposes two other important parameters: patternLength and similarityValue.

Pattern Length The getPatterns() function essentially breaks the song into pieces and stores them as pattern objects. This parameter indicates how long each pattern will be. Each number represents a 16th note for a 4/4 song, which at this point is the shortest a note can be defined in my data model. Therefore, choosing a high value (such as 16) means MusicMaker will break the song into pieces of 16 16th notes each, or one measure. This allows for greater variety in the final result. Choosing a smaller value will make your final song much more similar to the original music.

The setup of MusicMaker is intended to analyze songs written in 4/4, but this can easily expand to other time signatures in the future with little modifications. Interstingly, MusicMaker can still generate music regardless of time signature; the data is just stored in a data model that was created specifically for 4/4 songs.

Similarity Value Before explaining what this value means, it's important to understand how the function getDiff() works. getDiff() takes two measure objects as parameters and returns a number representing how similar they are. A higher number means there is less similarity, while a 0 means they are identical. Each number represents a single change in either note value or note time.

Similarity Value, therefore, represents how many differences can exist between two patterns before they should be considered the same. This is used to correlate two patterns that may sound very similar but have just a small number of differences. Typically, a higher structural-level sounds best with a higher similarity value. Picking a small structural level and high similarity would result in the song repeating the same pattern over and over again.

The final song MusicMaker produces is dependent heavily on these two values. Choosing a high pattern length and a low similarity value will result in few repeating patterns because there are few patterns that exist which are long and nearly identical. Choosing a low pattern value with a low similarity value, on the other hand, will produce much more repetition. Choosing 1 as the pattern length and 0 as a similarity value returns the same song that MusicMaker was given because it has no way to create patterns that are anything else other than the original. Depending on how different you want the final song, these values can be adjusted accordingly.

getPatterns() returns an array of pattern objects and a two-dimensional array representing the structure of the part object analyzed. The index of the first element in the structural array represents the measure number, while the second element contains the indexes of the patterns in the pattern array that were found in that particular measure.

For example, the following could be examples of a returned array of patterns and its associated structural array when analyzed at the quarter-note level:

```
//Array of Pattern Objects

(
  [0] => PatternObject()
  [1] => PatternObject()
  [2] => PatternObject()
  [3] => PatternObject()
  [4] => PatternObject()
  [5] => PatternObject()
  [6] => PatternObject()
  [7] => PatternObject()
  [8] => PatternObject()
  [9] => PatternObject()
  [10] => PatternObject()
)

//Structural Array
(
  [0] => N|N|N|0
  [1] => 1|2|2|0
  [2] => 1|2|3|4
  [3] => 5|6|7|8
  [4] => 2|2|9|N
  [5] => 10|2|2|N
  [6] => 10|2|3|4
  [7] => 5|6|7|8
  [8] => 2|2|9|0
)
```

Each number in the structural array is simply an index to a pattern object. In the first measure, therefore, the pattern at index 1 is played during the first quarter of the measure, the pattern at index 2 is played for both the second and third quarters, and the pattern at index 0 is played for the last quarter of the measure. This structure is useful because it lays out the basic pattern structure for an entire song. These numbers can be just as easily modified, copied, or deleted as the patterns they represent.

2. Manipulation

Now that we have a definition for the structure of patterns found in a song, MusicMaker can make changes to the patterns or structure to create something new. This phase of music-generation is most important in determining what the final result will sound like. There are many approaches that can be taken as to how, when, and to what extent MusicMaker can make changes. Because of this, MusicMaker is set up to modular during manipulation so that we can experiment with several different methods.

Note that it can be easy to misunderstand what manipulation actually means. The definition of a Pattern object itself was designed to introduce variation into the final result. In this sense, manipulation has already occurred in the deconstruction phase. A pattern object takes away key definitions of a piece of music, namely the rhythms, so that MusicMaker is forced to do produce something different. Simply redefining pieces of a song into something a little more general means that it is more difficult to transform it back into what it came from, and variety is therefore introduced. This was done on purpose. Manipulation, therefore, is simply restoring pattern objects back to subpieces in the song model in a way that is pleasurable, yet still unique from its initial state in the original song. This is accomplished by redefining key musical elements that have been taken away on purpose, but using methods that can create something of similar quality.

In future work, the pattern object may be set up to be modular as well so that we have more control over what information about a piece of a song we want to store and what we don't care about. This way, we can collect whatever data we want and manipulate the pattern object however we want. In this version of MusicMaker, however, we simply keep track of the notes being played and rely more on the overall structure of the song to maintain a repetitive element in the final song.

In our first module, we simply combined notes and rhythms randomly for each of our Pattern objects. What's useful about keeping track of patterns, however, is that no matter what combination of notes and rhythms we choose to create for each pattern, even if it is random, a strong sense of repetition has already been defined by the structural array that was created during the deconstruction phase. This, all on its own, at least maintains the fact that our new song will be just as repetitive as its original counterpart. /*Indeed, part of our experiment was to see whether simple repetitions in music, regardless of the quality of those repetitions, can still create something interesting and enjoyable.*/

For further modules, we created several tools that would develop pattern objects into a more repetitive and a more pleasant final result than randomly combining notes. These simple strategies are outlined below:

getScore() The MusicAnalyzer component contains a simple function that returns a number representing how many patterns it could find in a given part at different levels, starting from small patterns and growing to find larger patterns. This function take into account both rhythms and notes. More points are awarded for repeated patterns that are longer in length. For a pattern to be considered a match, the notes and rhythms must line up exactly at whatever level is being analyzed. A higher score means the part contains more patterns, and therefore can be classified as "better" under our own definition of music. The pseudo code for this function is outlined below:

```
public function getScore(part) {
  score = 0, incrementLength = 1;
      //initial score set to zero, starts
      at smallest count = 1/16 note
  while (incrementLength <=
      part.timeIncrement) {
    rhythmAnalysis = [], noteAnalysis =
        [], measureNumber = 0, initialNote
        = null;
    foreach (part.measures as measure) {
      foreach (measure.notes as note) {
        noteAnalysis[listNumber][posNumber]
            = abs(note - lastNote);
        rhythmAnalysis[listNumber][posNumber]
            = 1;
      }
      measureNumber++;
    }
    while (!empty(rhythmAnalysis)) {
      nextRhythm =
          array_shift(rhythmAnalysis);
      nextNote = array_shift(noteAnalysis);
      foreach ($rhythmAnalysis as key
          => value) { // => $value
        if ((value === nextRhythm) &&
            exists(1, rhythmAnalysis)) {
          score += pow(incrementLength,
              2); //so it rewards bigger
              matches of patterns
          if (noteAnalysis[key] ===
              nextNote) {
            score += pow(incrementLength,
                2);
          }
                unset(rhythmAnalysis[key])
                    &&
                    unset(noteAnalysis[key]);
        }
      }
    }
    incrementLength *= 2;
  }
  return score;
}
```

We used this function to create better results from Pattern objects. Instead of just taking whatever random selection the computer made, we created a large number of different possibilities, then chose the possibility that returned the highest score. This was done for each pattern in the entire song.

This could also be used in a genetic algorithm ...

Separating the Rhythms For many songs, there are sections of music that use the same rhythms as a previous section, but this time with different notes. In getDiff(), keep in mind that both notes and rhythms are used to determine how similar two sections are. Two sections with the same rhythm but completely different notes, therefore, can still be considered very different. Separating rhythms from notes when comparing sections helped solve this interesting issue.

When looking at differences between measures, we keep track of the result taking notes into account and the result as if all the notes were the same. This gives us an idea of which measures are similar rhythmically. When creating the notes for the final song, MusicMaker ensures that sections that contain similar rhythms are given the same rhythm, even if the notes assigned are completely different.

Choosing notes that are close together We noticed that the intervals between notes in songs are generally fairly close together, rarely going beyond an octave. We applied this same principle in MusicMaker when selecting which notes to choose in the final piece. As opposed to randomly selecting notes from the set of notes in the Pattern object, we choose the note that is closest to the previous note but that has not yet been chosen.

When the combination of these strategies are implemented, we end up with a subpiece of music associated with each pattern object. At this point, we are ready to combine these subpieces into the final song.

3. Reconstruction

After applying creating subpieces for each pattern during manipulation, it is relatively simple to recombine the pieces into a song. With a single song, this is especially easy because we literally just need to convert our subpieces, which already contain necessary rhythm and note data, back into Part objects. Finally, we combine the Part objects into a Song object, and the data model is complete with a new song from the original.

There is the option, however, to rearrange or combine subpieces in a different way than how the structural array is organized. This is especially possible if you are trying to combine elements and patterns from multiple songs. Once again, that is a goal for future work.

Once a complete Song object has been created, it must be converted back to MF2T and then finally to MIDI format. Using the MusicParser component, this is a simple process. After creating a MF2T file from the Song object, the PHP Midi2Txt provides an interface to convert MF2T back to MIDI format. When the MIDI file is created, it can then be played back for the user to hear. For testing, it was most convenient to use a simple web-client that included a simple JavaScript library that played MIDI files in-browser called ”Midi.js” (mid).

Survey

In order to test the success of creating new music using the previously described methods, we created a web-based survey seeking input on how well the music imitates the original music (Martn 2015). We are asking this under the assumption that if our music can create something that sounds similar to the original music, then it's also somewhat similar in the level of enjoyment experienced by the listener.

Obviously, we don't intend for MusicMaker to copy songs. Rather, we want it to implement similar patterns and structural elements in new music. For this reason, we did not set the similarity value or pattern length value to be too small when running our algorithm or else there would be potentially too little variation from the original music. Rather, we picked a value that seemed to consistently produce new music with enough variety to sound unique but still contain the key structures and repeated elements in the original song. Because we separated the analysis of rhythmic patterns from melodic and rhythmic patterns, they are each defined below:

- *Rhythmic Similarity Value*: 4
- *Rhythmic Pattern Length*: 1
- *Melodic and Rhythmic Similarity Value*: 4
- *Melodic and Rhythmic Pattern Length*: 2

To publicize the survey, we sent out invitations via social media and word-of-mouth. Anyone who had the ability to hear could take the survey.

In our survey, we first asked recipients to listen to an already exiting song and then compare it with two generated songs that were created by MusicMaker. The recipient had to decide which computer-generated song was created from the existing song. There were two such questions asked. Then, we asked recipients to do the opposite; they first listened to the computer-generated song and then selected from

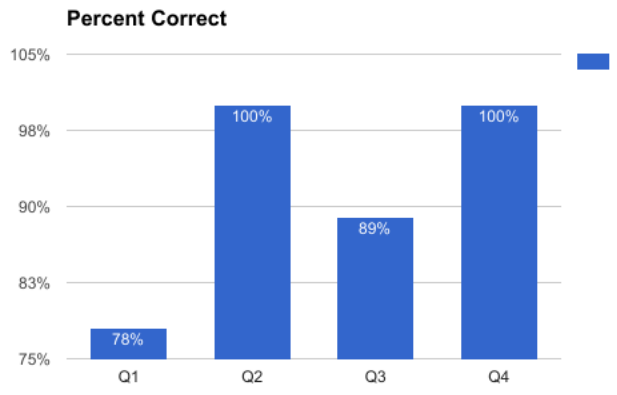

Figure 1: A visual representation of the percentage of correct responses for each of the first four questions asked in the survey where people were asked to match the original song with its computer-generated counterpart.

two options which song was the original song used in the algorithm. Once again, there were two questions of this type that were asked.

In order to prevent bias, we ensured that all songs played in a single set were in the same key and contained the same primary instrument, the piano.

Then, we provided instructions for the person taking the survey to visit the open-source library of sheet-music provided by MuseScore (mus), download the MIDI version of a song of their choice, and then upload it for MusicMaker to analyze. After that, MusicMaker generated a new song from the song they chose, and they were asked to give the new song a ranking based on how well it imitated the original. The time required for MusicMaker to generate a new song was typically 2 or 3 seconds. Finally, we asked the person for how many years of musical training they've had.

Results

The results from our survey are shown in Figures 1 and 2. These results are taken from a database that recorded responses from individuals taking the survey. A total number of 10 people took the survey.

The results suggest that a high percentage of people across various musical backgrounds can correctly identify the computer-generated song that was created from its already-existing counterpart, and vice versa. There were two questions that nobody got incorrect.

Only 10 people took our web-survey, but of those 10 people, the musical training ranged from 0 to 16 years and there were only 3 incorrect responses total from only 2 individuals.

The most common rating for similarity between the uploaded song and generated song was 5, with the lowest being only 3. Even those recipients that reported having many

(1).png (1).png

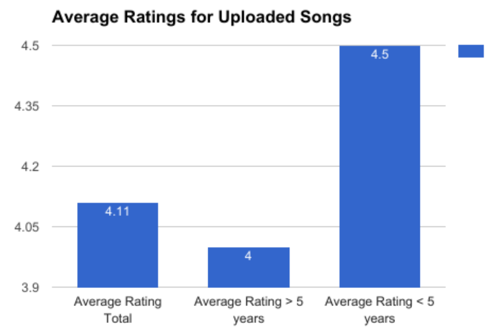

Figure 2: A visual representation of the average ratings for the similarity between the user-uploaded song and its computer-generated counterpart, a 1 being not similar at all and a 5 being very similar. Number of years refers to number of years of musical training.

years of musical training (at least 5) gave an average score of 4.

Discussion

The survey suggests that most people, perhaps regardless of musical training, could answer at least 75 percent of the survey questions correctly. Such a high percentage shows that MusicMaker can, to some degree, create new music that sounds similar in style and structure to existing songs. The end goal of MusicMaker, however, is to eventually be able to more intelligently mutate the song structure and pattern structure to create something that is perhaps hardly traceable to the original song, yet still maintains the key component of repetitive patterns and perhaps chord-progressions, etc. The main goal for this research was to evaluate how well MusicMaker can mimic music using the basic idea of imitating the structure of repeated patterns. Based on results, it appears that MusicMaker can successfully do this.

If I were to do this survey again, I would ensure that many more people participated before analyzing results. I would also ask some questions more in line with the person's opinion of the overall quality of music that MusicMaker created. It would be valuable to know if they thought the generated song was perhaps too similar to the original for it to be considered unique.

Conclusion

Based on the results from our survey, we have determined that MusicMaker can successfully generate new music from existing music with a high level of similarity, especially when observing the structure of repeated patterns. It is not clear at this point how original the generated songs are; this can be part of future work.

We believe it was important to establish a solid framework for MusicMaker so that it has plenty of room to grow and improve the efficiency and quality that it can generate new music. This involved redefining music in our own terms so it clarified the direction we should take, which in this case was the focus on repeating patterns. By giving MusicMaker the ability to break apart a song into patterns, we've provided a relatively simple method for generating new songs with a sense of repetition and structure which we propose are core, fundamental elements of music.

There is plenty of room for future work in this project, and several of those ideas have already been mentioned in this paper. Indeed, MusicMaker is only at its beginnings; it has a solid framework for breaking apart, analyzing, and reconstructing existing songs. There are many things we would like to improve for the future.

First of all, it would be interesting to make the process of finding patterns and manipulation more robust. Additional data, such as note volume and rhythms, could be analyzed and looked at when recreating each subpiece from pattern objects (Ventura 2012). On a more global level, an analysis of overall chord structures and common notes could be taken so that note and rhythm selection was based more on statistics than pure chance. These statistics could even be stored in a database that MusicMaker could add to as it analyzed and generated more and more music.

Secondly, the potential of MusicMaker could be increased with the ability to analyze several songs and combine elements from each one into the final result. This would perhaps better mask the original song from which the generated song was created and provide additional opportunity for variety and a more tailored experience for the listener. Instead of having to chose a single song to imitate, you could select a set of songs and then listen to a generated song that combined pleasurable elements from each. With this idea in mind, a genetic algorithm using the getScore() function could be used to find the combination of parts from multiple pieces that resulted in the most pattern-based song. Combining random sections from different songs could prove to be very difficult, but if there was an efficient algorithm that would ensure that each song was in the same key and if we had a way of evaluating the combination of parts, then a genetic algorithm would be possible. This is perhaps the most exciting project for future development.

Thirdly, getScore() could be more modular to accommodate other interesting song elements. Fundamentally, we believe that repeated patterns make up the root of any song, but final scores could consider other interesting elements. This would potentially lead to higher-quality subpieces created from patterns. It would also be interesting if we ran the function not merely on each subpiece, but a combination of subpieces to ensure transitions between subpieces was adequate.

Lastly, more integration between tracks and patterns could be analyzed. It would be interesting to identify which notes are commonly played with other notes across tracks, as well as which notes commonly follow other notes. With this knowledge, you could more easily align patterns that sound good together or ensure transitions between patterns sound natural. At this point, integration between different subpieces is non-existent.

References

Johnson, Daniel; Ventura, D. 2016. Musical motif discovery from non-musical inspiration sources. *ACM Computers in Entertainment—Special Issue on Musical Metacreation*.

Martn, Daniel; Pachet, F. 2015. Improving music composition through peer feedback: experiment and preliminary results. *24th International Joint Conference on Artificial Intelligence* 1407:27–31.

Mf2t/t2mf. http://valentin.dasdeck.com/php/midi/downloads/README.TXT.

Midi js. http://www.midijs.net/.

Musescore. https://musescore.com/sheetmusic?instruments=0.

Pachet, F. 2003. Representing musical genre: A state of the art. *Journal of New Music Research* 32(1):83–93.

Rolland, P.-Y. 1999. Discovering patterns in musical sequences. *Journal of New Music Research* 28(4):334–350.

Ventura, Dan; Murray, S. 2012. Algorithmically flexible style composition through multi-objective fitness functions. *Proceedings of the 1st International Workshop on Musical Metacreation* 55–62.