Composerator: Randomized Music Composition using Markov Chains

Cameron Clarke, William Cox, Garrett Parrish

1. Overview

Composerator is an algorithmic music generator that uses Markov chain based machine learning to output a piece of music that is probabilistically similar to an input piece. A Markov chain is a system that algorithmically specifies transitions between state spaces. In the case of Composerator, Markov chains are implemented as sets of probabilities that determine the probability of certain musical objects following one another.

The input comes in the form of a Musical Instrument Digital Interface (MIDI) file. This file is then parsed, and data about the pitches, note durations, and note volumes that make up the file are gathered. These data are analyzed and are compiled into a matrix representing a Markov chain of arbitrary order, whose entries correspond to the probability that a given column element follows a given row element.

Once the whole input file has been parsed and all the necessary probabilities have been computed, a random number generator is used to create lists of pitches, note durations, and note volumes that are probabilistically similar to those from the input file. These lists are then combined to form a new song, which is eventually converted back to a MIDI file and exported.

2. Planning

We planned from the outset to implement Composerator in Java using the Eclipse development environment and using GitHub for version control. Our initial plan for the program architecture was to first have a method to convert MIDI data to a list of Note objects, each of which had instance variables of types Pitch, Duration, and Volume, each of which would be subclasses of the abstract type Chainable. We then planned to use and object of type Cursor that would store a list of length n + 1 (where n is the chosen order of our Markov chain) containing a list of Chainables. The first n Chainables in the Cursor object would represent a given key, whereas the last element would represent a note that comes sequentially after the key in the input song. By letting the cursor traverse over all of the input data, we planned to iteratively update a matrix that would eventually represent the probabilities of a given chainable coming after any set of n Chainables that appears in the input file. Using this matrix, we planned to create a song that is probabilistically similar to the input song. Much more can be found in the annotated draft and final specs.

With only a few minor changes to our initially planned architecture, we were able to implement Composerator without many problems. The only major roadblock we encountered was initially interfacing with Git and finding a Java IDE that interfaced well with Git. In the end, we ended up using Intellij, which had the most seamless Git integration of any IDE we tried.

Our milestones generally worked out fine. We were able to have the majority of our project planned out and architected, in a form that is mostly faithful to our final implementation, by the first draft specification. By the final draft specification we had worked out specifically how our separate classes and methods would interface with one another, and were able to use this plan to guide the final implementation of Composerator.

3. Design and Implementation

Composerator mainly consists of two parts: MIDI I/O conversion and song generation. A call to Main.java initializes the UI, which then prompts the user to input up to three file paths to files containing MIDI data. These files are then handled in Audio_File.java, where the audio files are read into an input stream using various java audio-processing libraries. These data is then used to create Chains of Chainable objects containing the data from the input. From these data, MarkovRows, and subsequently MarkovMatrices are created

4. Reflection

The planning process we underwent before writing any of the code proved to be immensely helpful when it came time to implement everything. Our planning process was relatively straightforward: we met as a group for a few hours and focused on creating an overall roadmap for the program. We were able to hash out both an overall view of the different components of our program and a more detailed view of the classes and objects that would make up each component and how they would interact with one another.

Although we did eventually run into a few implementation challenges and had to change some parts of our design, these changes were relatively minor, and indeed much smaller than they could have been had we not had a clear, holistic picture of our project from the outset. Moreover, these challenges did not prevent us from completing our draft, final, nor our technical specifications, which all went well.

One of the most helpful choices we made early in the process of coding was to make all parts of our design as modular as possible. Originally in the planning phase, we had decided to separate some classes that could have been, in actuality, inherited from the same abstract class. This was especially the case with our Chain class. Initially, we were creating separate chains of Pitches, Durations, and Volumes, but we soon realized that it would be better to have all of these Chainables use the same Chain type.

One choice that we could have rethought was devoting so much time to creating a UI. Beyond a certain amount of tedium, creating a UI was not a huge issue in the end. However, seeing as a UI is not necessarily algorithmic in nature, our time could have possibly been better spent on the more algorithmic parts of this project. Our labor was divided well enough that the creation of the UI did not pose a large impedance to the completion of our project overall though.

There were a few unexpected roadblocks early on in the completion of the project. One major one was, surprisingly, finding a Java IDE that interfaces well with GitHub. We spent much more time than should have been necessary (about 1.5 weeks) trying to get our development environment set up, which set us back a little bit in the beginning. Also, the creation of MIDI output files turned out to me much more tricky than expected. These files need to be essentially built up byte by byte, with little margin for error without forming a corrupt file.

More positively, however, learning about both MIDI files and Java's powerful object-oriented paradigmatic focus were unexpectedly rewarding. None of the group had much previous experience with MIDI files, and figuring out how they are formed and how they can be used was interesting not only in a computational sense, but also because our group is made up of musicians so this knowledge may be directly applicable in the future. We also enjoyed working with Java and realizing the extent to which its object-oriented capabilities can be used beyond simply creating objects with somewhat trivial attributes and methods.

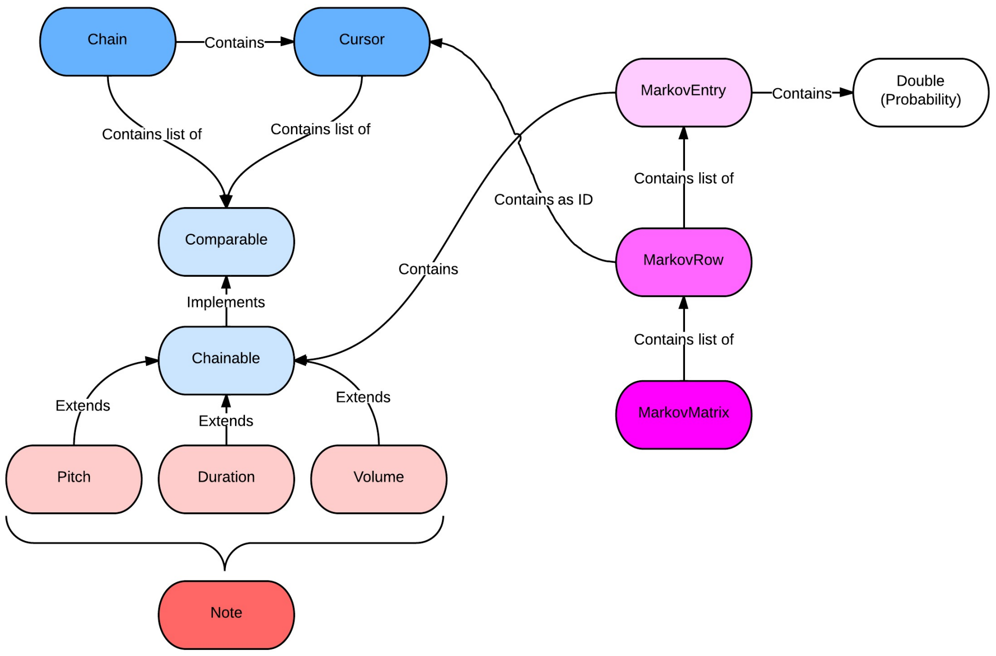
We feel that Java's object-oriented focus is especially well suited to our design. We took full advantage of Java's built-in functionality such as abstract classes private and public classes and methods, getters and setters in order to effectively and elegantly implement our original vision. This functionality made it possible to have precise yet seamless integration between all components of our project. As stated previously, program design was not any real issue, but actually crafting the classes in Java took a little effort and creativity, for example in how we chose to make Main.java rely heavily on Composeration.java. Testing the MarkovMatrix methods and their related classes was somewhat complicated simply due to their complexity, but testing MIDI in/out and UI was fairly straightforward.

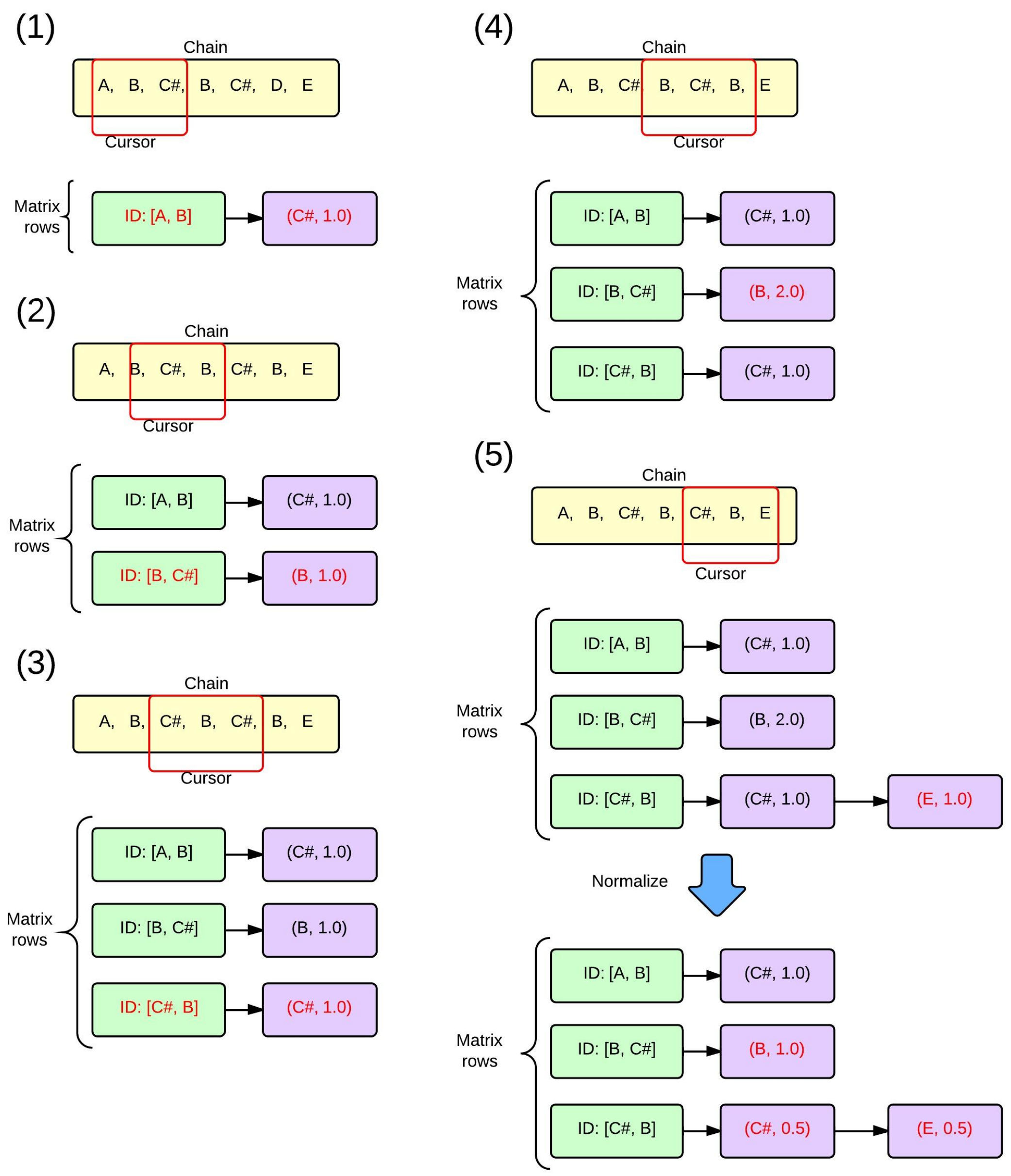
Given the opportunity to do some things differently or extend our project, we would probably like to have more explicitly planned out the actual Java implementation of all classes before beginning to implement them. Given more time, we would extend our UI to allow some sort of graphical representation of the song, in addition to having an integrated audio player.

Overall, there was much to he gained from this project. We gained valuable experience in program design and how to create something implementable beyond the command line. We gained valuable knowledge about Java, one of the most widely used programming languages today, which can only serve us well in the future. More generally, we got to experience the benefits of object-oriented programming in a real, tangible way.

Billy Cox came up with the original project proposal and is responsible for designing and implementing most of the classes and objects related to MarkovMatrix and Chainables. Garrett Parrish was instrumental in setting up Git integration and troubleshooting IDE issues, completed all the MIDI interfacing, and created the technical specification. Cameron Clarke contributed to the initial development of the algorithm at the core of the project, aided in the implementation of the Chainable classes, and compiled the final write up.

Below are the three diagrams used in the video and the revision of our final specification:




(1)
Chain
A, B, C#, B, C#, D, E
Cursor
Matrix rows
ID: [A, B]
(C#, 1.0)
(2)
Chain
A, B, C#, B, C#, B, E
Cursor
Matrix rows
ID: [A, B]
(C#, 1.0)
ID: [B, C#]
(B, 1.0)
(3)
Chain
A, B, C#, B, C#, B, E
Cursor
Matrix rows
ID: [A, B]
(C#, 1.0)
ID: [B, C#]
(B, 1.0)
ID: [C#, B]
(C#, 1.0)
(4)
Chain
A, B, C# B, C#, B, E
Cursor
Matrix rows
ID: [A, B]
(C#, 1.0)
ID: [B, C#]
(B, 2.0)
ID: [C#, B]
(C#, 1.0)
(5)
Chain
A, B, C#, B, C#, B, E
Cursor
Matrix rows
ID: [A, B]
(C#, 1.0)
ID: [B, C#]
(B, 2.0)
ID: [C#, B]
(C#, 1.0)
(E, 1.0)
Normalize
Matrix rows
ID: [A, B]
(C#, 1.0)
ID: [B, C#]
(B, 1.0)
ID: [C#, B]
(C#, 0.5)
(E, 0.5)