**Music Composition through N-Order Adjacency Data-Structure**

Contributory work

College of Saint Benedict and Saint John’s University

By

Andrew Joseph Heroux

August 8th, 2019

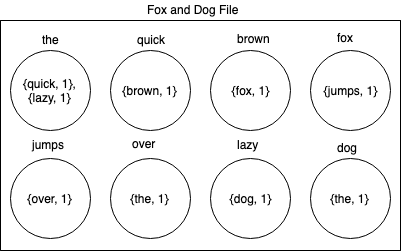
**Background 1**

Terminology: The term ‘token’ may be used in reference to measures/intervals because they are treated the same in data-processing.

This work is based on previous work from Tom Richmond, Ryan Strelow and Caitlin Harvey. Ryan and Caitlin’s work, adapted from Tom’s work, generated music by parsing \*\*kern files into measures and intervals. As the files were parsed, the frequency of each unique token was tracked; in the music generation those numbers were used to weigh the probability of any given token being selected; higher frequencies were more likely to be selected. With this system the probabilities for a given token being selected does not change as the song progresses. If there is a disproportionately high probability token within a dataset, the program could write the same token repeatedly. This would clearly not make good music. Another deficiency of this system is that the song could jump from one token to another that would never appear together in the dataset. This would allow slow melodies to suddenly become fast, or intervals to jump unexpectedly. This also made for bad sounding music.

**Order-1 Adjacency Data-Structure 2**

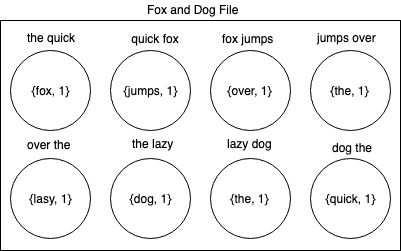
The solution to the sporadic nature of the program was having dynamic probabilities for the next token written. The initial approach was to have the program look at the last token written and select the next one from a dictionary of adjacent tokens. So, if you had a list of intervals: {1, 3, 1, 2, 4, 1, 3} and a dictionary for the token 1, the table would contain 3 and 2. It would not include 4 because it never proceeds 1. If 1 is written, then the next token would have to be either 3 or 2. Along with the table of tokens is their corresponding frequencies. In the table for 1 the frequency for 3 would be 2 and the frequency for 2 would be 1. I will illustrate the order-1 system using the sentence: “the quick brown fox jumps over the lazy dog.”



One important thing to note is the last token. For any dataset the last token (dog) will be set adjacent to the first token (the.) Without this, the token dog would have no adjacencies and when trying to write the next token, there would be no dictionary to reference.

**N-Order Adjacency Data-Structure 2.1**

By looking back one token to write the next token, it is ensured that any sequence of two exists in the dataset. An issue that came up with the order-1 system is heavy repetition, especially when dealing with intervals. In an order-1 system it is common to see tokens that have adjacencies to themselves with high relative frequencies. If an interval of 1 is highly likely to be adjacent to another interval of 1, then 1 can be written over and over again. There is no way to check if 1 has been written 2 or 3 or 4 times in a row. To solve this issue, the program needs to be able to look back at the last n tokens to determine the next token. To look back at n tokens, the program creating the data-structure concatenates a sequence of n tokens and that concatenated sequence will have a dictionary of adjacent tokens and their frequencies. To illustrate an order-2 system I will again use the sentence: “the quick brown fox jumps over the lazy dog.”



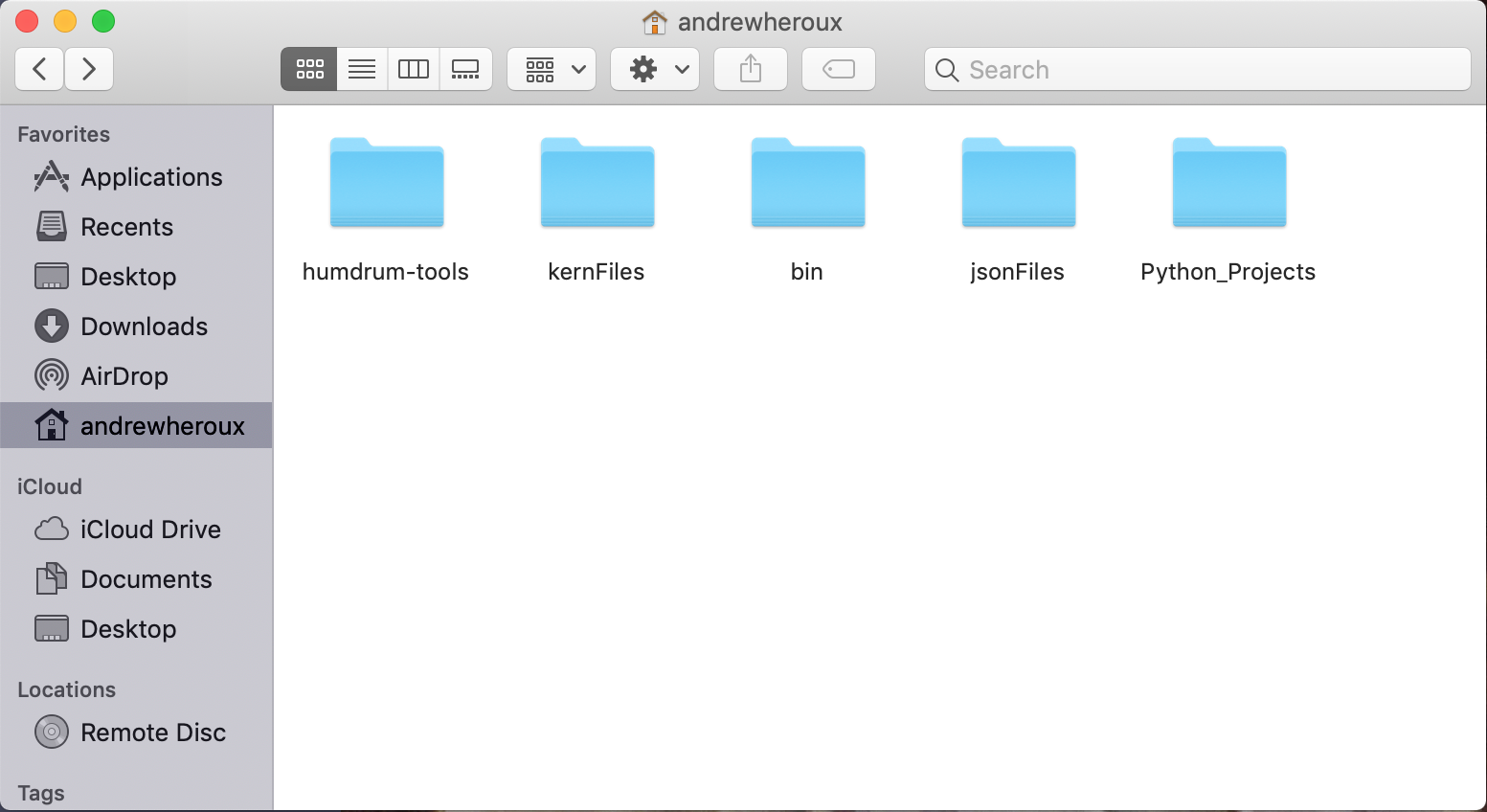
Note: In the program the tokens are actually concatenated with &&, so ‘the quick’ actually is represented as ‘the&&quick.’ It is adapted for readability.

Here you will see the last and second to last tokens having adjacencies to the beginning of the dataset. Because this is an order-2 system, the second to last token contains the last two tokens of the dataset and the last token contains the last and first tokens of the dataset. The table below more clearly shows this overlap.

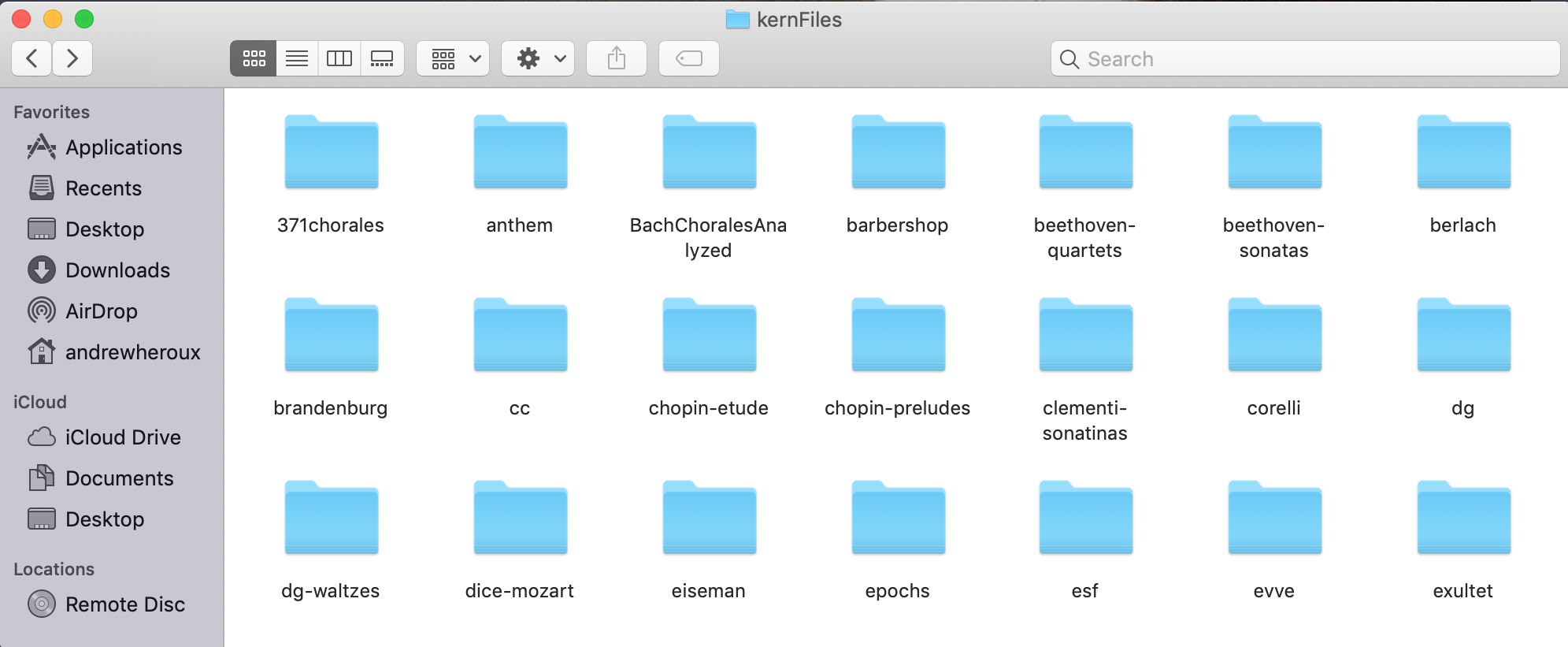
|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Concatenated Tokens↓ |  |  |  |  |  |  |  |  |  |  | Adjacent Token | Index |
| the quick | the | quick |  |  |  |  |  |  |  |  | brown | 2 |
| quick brown |  | quick | brown |  |  |  |  |  |  |  | fox | 3 |
| brown fox |  |  | brown | fox |  |  |  |  |  |  | jumps | 4 |
| fox jumps |  |  |  | fox | jumps |  |  |  |  |  | over | 5 |
| jumps over |  |  |  |  | jumps | over |  |  |  |  | the | 6 |
| over the |  |  |  |  |  | over | the |  |  |  | lazy | 7 |
| the lazy |  |  |  |  |  |  | the | lazy |  |  | dog | 8 |
| lazy dog |  |  |  |  |  |  |  | lazy | dog |  | the | 0 |
| dog the |  |  |  |  |  |  |  |  | dog | the | quick | 1 |
| Index→ | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 0 |  |  |

This table shows the overlap for the concatenated tokens and their adjacent tokens. The first concatenated token includes indexes 0 and 1. Because this, the first adjacent token is at index 3. This means there needs to be two tokens that lead to index 1. That is achieved by concatenating index 8 and 0. Expanding to order-3, the last token built on this dataset would include indexes 8, 0 and 1. The overlap grows by 1. The overlap will always be n-1 for the last element.

**Using The structConstructor program to make N-Order Data-Structures 2.3**

The structConstructor.py program is located in the scripts directory on the GitHub repository. Also included in that directory are measure-script and interval-script. Both are required to run the python program. The humdrum-toolkit is also required to run the program. Installation guide: <https://www.humdrum.org/install/github/>. You need to add kernFiles, jsonFiles, Python\_Projects and bin to your home directory. After installing humdrum and adding directories it should look like this:

Put structConstructor.py in Python\_Projects and the bash scripts in bin. The kernFiles directory should contain directories of kern files. These directories may be used to organize the music as you wish. The collection of kern files on GitHub are from <http://kern.ccarh.org/>. It is organized by genre.

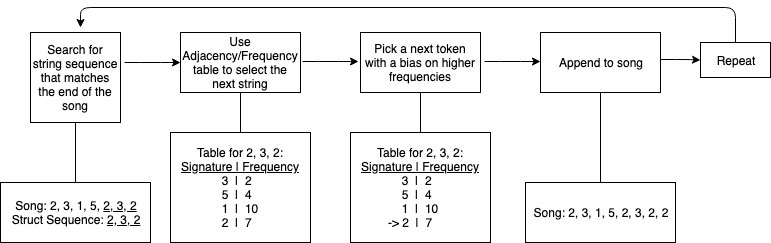


Once everything is prepared, you can run the program through the terminal. There are two ways to use the program. You can specify one collection of kern files to be processed for a single specified order by using makeStruct. From the Python\_Projects directory, type: python3 structConstructor.py <order-system> <i or m (i for interval and m for measure)> <directory>. If I want to create an order-3 measure data-structure for the folk-germany directory, I would use the following: This will produce a json file with the struct. The program can also make multiple struts in a queue. To utilize this the parameters are makeStructs <start-order> <end-order> <i or m> <directory>(you may any number of directories). To produce order 2 through 5 interval structs for folk-germany, folk-han and folk-erk, you would use the following: 

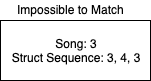
Completing a task like the one above will take a few hours.

**Understanding the Music Creation Program 2.4**

The music creation program uses the json files, made by structConstructor.py, to piece together a song and write it to a midi file. It begins by using the structParser class to load and parse the json file. The information from the json file is put into an array of MusicStructs. A MusicStruct contains information about a given sequence of strings. This information includes a sequence of signatures, a signature(the last signature in the sequence), a table of adjacent signatures and their frequencies and a string stating whether this struct appears at the beginning or end of a song. Once this data is loaded, the StructReader class uses it to create a song. The song is made by first picking a random measure. The next measure is then selected from its table of adjacent measures. This continues until the specified number of measures have been written. After having the measures written, the program determines how many notes are represented by the measures. Then the intervals are written using the same process as the measures. When the program is handling an order-2 or higher system it will compare the sequence of n strings to the last n strings written to the song. Below is an illustration of how the program adds to the song.



In the first box you can see the underlined portion of the song is the part being compared. Once the struct with the matching sequence has been found, its table is used to find the next piece of the song. This is then repeated for the number of measures specified by the user. When using a system higher then order-1, the beginning of the song is written differently. The first string is always written randomly, but if you then try to compare two or more strings there will be no match.



The way to fix this is to have the end of the struct sequence compared.



Above you can see the section of the struct sequence being compared. In the first comparison, the only part of the sequence that needs to match is the last part. Once the song has another piece, it looks at the last two elements. Finally, once the song is greater than or equal to the order of the system, the entire struct sequence is compared. After the song has been written, the midiOutputController takes the ArrayList of MusicStructs and Intervals and merges them to create a midi file.

**Music analysis 3**

The music produced has a lot of room for improvement. The primary issues are having a single note played at a time and having highly sporadic intervals. When evaluated by a Liturgical Music Student at the School of Theology at Saint John’s University, the advice given was to have intervals follow a trend of going one direction for a time and then switching. Essentially having waves of up and down intervals. At this point the program will choose randomly if the interval goes up or down and the length of the interval is determined by the struct. The other big improvement to be made is having chords implemented. Strategies for implementing this has not been looked into yet. For more information of evaluation of the music, Ryan Strelow’s work this summer has been on using neural networks to evaluate the music and using a genetic mutation algorithm to improve it.

**Kern analysis 3.1**

The research done this summer show potential for furthering this software in some regards. My primary focus has been in the new data-structure which has given me a good perspective on what we have to work with from the kern files. There are some limiting factors with kern, those primarily being reliability and dataset size. There have been some issues using the humdrum toolkit with certain kern files. Finding large sums of data has also been an issue. The issue of reliability may be alleviated, but I have struggled to find large numbers of kern files. There are 22,000 on the GitHub repository that were collected from the KernScores website. Aside from that collection, I have not found another large sum of files. Another attempt at getting more files was to locate a collection of midi files and use the mid2hum command to convert them to kern. I found a collection of 400,000 midi files on Reddit. After downloading the files, I tested the mid2hum command on a directory of Taylor Swift music. In each test the kern file produced was unusable. The numbers produced for intervals and durations were not accurate. I also attempted to do the same thing with music xml files. This involved the xml2hum command. In searching for large sums of xml files, I found many databases that had tens of thousands of files. The issue was there all required a subscription around $30.00 to access any of them. I was not sure if I would be able to scrape the websites once I had the subscription and did not want to invest money into a potentially fruitless cause, so I did not purchase any subscriptions. As of now there are only the 22,000 files to work with.

It may be good to adapt the program to work with other file types. midi has many large collections of music, but It is more difficult to parse it. In the kern format, there are delimiters for each measure of the song. At this point, the program needs entire measures, but with the flexibility of the n-order data-structure it would be possible to eliminate measures and just account for single-note durations. The other issue with parsing midi would be that it is not precise. Two notes that seemingly have the same length may very slightly. To the computer, these would not match in length. Because the n-order data-structure has to compare all of the data fed in, the midi notes would have to be given a tolerance. There has been no attempt at this yet, but I imagine this would be achievable by having a table of seen notes and when comparing the next note having a plus or minus c. There would have to be some testing done to determine what c should be. I have not looked into music xml, but that may be a good file type to consider.

**Data-Structure analysis 3.2**

The n-order data-structure is very generic. It is capable of processing any set of strings. because of this, I think it can be used in future adaptations of this project with little to no modifications. There is more testing that should be done with it. One thing that has not been determined is the threshold for overfitting when increasing the order of the system. I have determined that the size of the dataset matters. When running an order-3 script over 300 folk-germany songs, the outputs was identical to the input. Running the order-3 script over 5000 folk-germany songs yielded better results, where there were many adjacencies for most signature sequences. It is difficult to go through the files and eyeball overfittings, so it would be nice to write a script that counts the average adjacencies for each signature sequence. A good visual way to see the boundary cases would also be to use a 3d graph. Using one axis for the order of the system, one for the size of the dataset and the last for the average adjacencies. With this data, it may be possible to have a function determining the order of the system.

**Conclusion 4**

I think the contributions made to this project have improved the quality of the music. The n-order adjacency data-structure has helps with the sporadic nature of the old system. There are many things that should be revaluated. For further research, there should be a focus on implementing scales and having intelligent song progression, including the songs beginning, middle and end. As stated before, intervals need to be worked on. Having multiple notes played simultaneously is also something that needs to be implemented. I have not thought of any strategies for this yet, but I imagine implementing it will require a lot of work in the midiOutputController.