Course: COMP5600 Introduction to Artificial Intelligence

Group Members: Hsuan-Hau Liu, Miles Liburd-Leffler, Lakshmi Krishnaprasad, Tianhang Lan

Instructor: Bo Liu

Date: 12/11/2017

Machine-Improvised Music Final Project Report

1. Task Definition

Combine Artificial Intelligence and music, and we have machine-improved music! For the final project, our group proposed a system that could learn from sets of musical melodies and produce its own version of music. By giving the program some input files, the program would need to analyze the inputs and generate a new music file. With the help of well-designed algorithms, we hope to create a program that can produce decent musical pieces.

There are already several Music-generating programs written and shared by others on the Internet. With the help of existing open-source libraries, one can write a program that generates music easily. Python, for example, is known for having many open-source libraries out there for programmers to use. Due to its simplicity and flexibility, our group decided to us Python as our tool to create the program. However, instead of using existing libraries are built for generating music, we planned to start from scratch and use our own unique approach to write the program. Due to time constraint and difficulty of the topic, we limited ourselves to focus on improving simple songs generated by our program, rather than trying to create complex music. Since the quality of a song is subjective to listeners, our goal was to make the songs sound as consistent and melodic as possible. Our task can be broken down into several stages, mainly: processing inputs, storing data, and generating music. In the context of Hill Climbing, each note is a state, and the music-generating algorithms that we are implementing are successor functions.

When processing the inputs, the program would perform a statistical analysis on the melodies and record frequencies of each note as well as their lengths. The goal of this stage is to produce data set that contains every known note and their corresponding neighborhood. Each neighbor is assigned with a value indicating the number of occurrences followed by the current note. The neighborhoods of melodies are our evaluation metric, where each value represents the accuracy of the generated melody.

In the next stage, we need to store the data in a form that would be convenient to add elements as well as accessing. We also want to memorize that data, so that the program does not need to learn everything all over again but the new input songs. This can be easily done by exporting data to a text file. The program will generate two files: one contains the statistical data of all the notes, and one contains the names of all known songs, number of songs, at list of known note lengths, and number of notes processed. In future runs, the program would first read the existing data file to regain memory before processing new inputs.

Finally, the last step is for the program to generate its own music. The program would use the most common note in the set (note with highest frequency) as the starting note, and decide which neighbor to pick as the next note using successor functions. The note lengths are also chosen from its list, except that we would choose either arbitrarily or generate a new set to choose from. In this step, we would implement several different algorithms and compare their performances. Ideally, we hope to come up with an appropriate successor function that generates musical pieces.

2. Infrastructure

The input set consists various unique MIDI files. MIDI, which stands for musical instrument digital interface, is a type of files that contain melodic lines of music. Unlike regular audio files such as MP3 or WAV, MIDI files simply explain how the sound should be produced. The command set includes note-ons, note-offs, key velocity, pitch bend, and other methods of controlling a synthesizer. MIDI file is easy to process comparing to other types of music files. Its simple and straightforward numerical representation of music elements is a clear winner for us. To read the input data, we used an open-source library built specifically for processing MIDI files called python-midi. It is the only open-source library that we used in this project.

Originally, we hoped to write a function that can take any MIDI file as input. That way, one can use any MIDI file that he or she desires as inputs for our program. However, we quickly realized that analyzing complex music is not an easy task. Chords for example, are three or more notes played together harmoniously, which are be difficult to analyze. After several attempts on processing complex music, we decided to produce our own MIDI files instead. Each of our input contains simple melody with one note being played at a time. The output MIDI files also have the same format. In MIDI files, there exist multiple tracks, where each track is a list of MIDI events. These events contain the elements of the song, which include every note that exists in the song and the tick values of each note. Every note is represented by a number between 0 to 127, and each tick value represents the length of a note in terms of milliseconds. When processing the data, we target one specific track (since it is a single-line melody) and filter out all events except for the “note-on” event. This specific event along contains every note and their ticks. From there, we simply go through each note and record the neighbors and tick values.

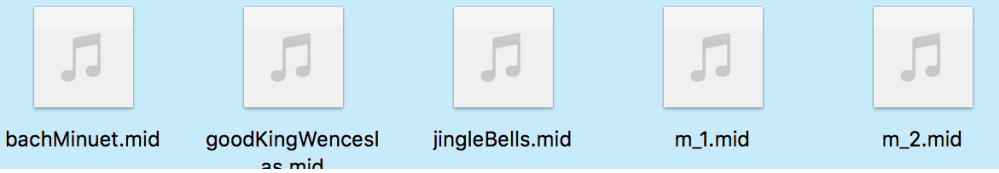

Figure 1: A Screenshot of Sample Input MIDI Files

3. Approach

Our plan was to implement some variations of Hill Climbing algorithm to accomplish the task. The basic idea is this: analyze and record the frequency of every note followed by another note, and form neighborhoods for each note. Therefore, the neighbors of a note are the set of notes appear after that note. We also want to keep track of the total number of notes that appear after a note (sum of all values of neighbors), which is roughly the number of times this note has appeared throughout the whole input set. Since note value is ranged from 0 to 127, we would have to use a number that is not within this range (-1, for example) to denote this total number. Note that the value of a neighbor divided by the total occurrences of notes followed by a note is equal to the probability of that neighbor appearing after that note. With these data, we can judge whether a neighbor is good or bad based on either the probabilities or simply the number of occurrences of a neighboring note.

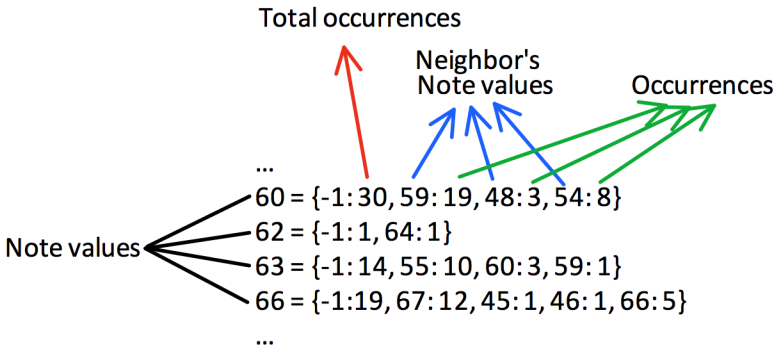

Figure 2: Demonstration of Notes and Their Corresponding Neighborhood

To give a length to the output song, we calculate the average number of notes in each input song and set it as the minimum length. If this limit is not met, the program will keep adding new notes to the song. Some of our algorithms have additional requirements that must be met in order to end a song.

3.1 Data Structure

Since we used Python as our programming language to write this program, we decided that dictionary and list would be the most suitable data structure for storing the data. There are three key information that we need to keep track of: all known notes, the neighborhoods for every known note, and the lengths or ticks used in the input songs. Although there are many elements in a song, knowing notes and ticks alone are enough to generate music. For this reason, our group decided to focus on these two elements. To remember what songs the program has already learned, we also need to store a list of song names, number of songs, and the total number of notes in these songs.

For ticks, we simply used a list to keep track of all unique ticks that were used in the input files. Since there are not many unique ticks, a list would be sufficient for this purpose. Notes, on the other hand, are a large set of data. Initially, we designed a data structure using an array, pointers, and classes (Note classes) to store and access the information. Each index of the array represents the value of the current note, and each block stores a linked-list of Note objects representing the neighborhood for that note. The advantage of this data structure is that each Note object can store a large amount of information about a note; we can store its value, frequency, tick, and possibly more information all in one object. The link list can also be sorted based on the frequency of each note, which would be useful for us when we produce the song.

The downside is that linked list does not have the ability to random access its content. This data structure also requires memory pre-allocation. Using array as the outer layer of the data structure also means that we must pre-allocate a certain amount of memory for every possible note.

Our second design, which is the one we use for our program, is a 2-D dictionary data structure. The outer dictionary is used to keep track of notes that the program has seen. The nested dictionaries are used as the neighborhood for every note, where the keys are the note values, and the contents are the occurrences of a neighbor follow by that current note. With this data structure, we can dynamically add new neighbors as we read inputs, so memory pre-allocation is not necessary. However, this does mean that we would have to sacrifice run time. Random accessing data is possible with dictionaries, and we can even define our own keys. One thing about the dictionaries is that the content does not have a particular order, though we can get around it just by simply converting the set to a list of tuples.

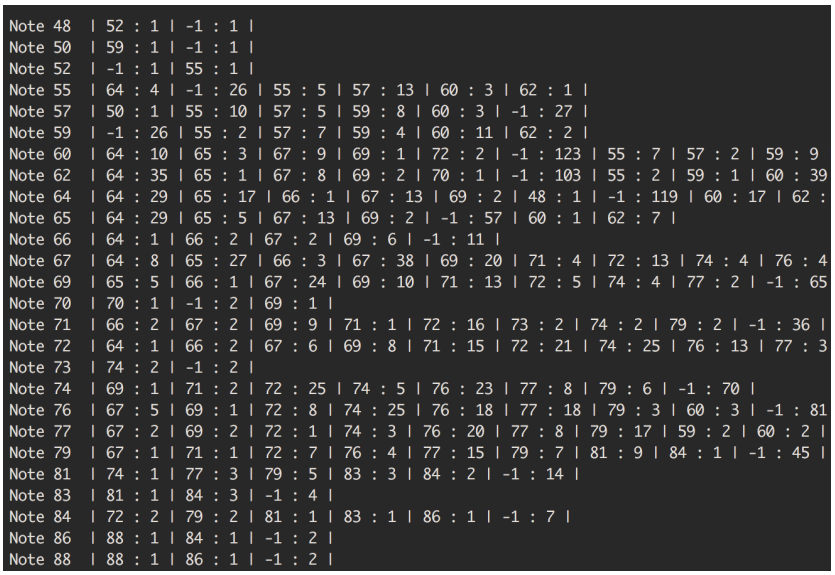

Figure 3: A Screenshot of Statistical Data of Notes

The reason why we chose the second decision is simple. It is simple to implement, flexible, and efficient in terms of storage space. Being able to dynamically add neighbors and random access data is important for us. When we build the data set, not all 128 possible notes would appear. In fact, if we were to use array, most of the blocks would have been empty. Random access allows us to quickly access the value of a neighbor. Instead of traversing through the whole set every time we choose a neighbor, we can simply use the key to identify which neighbor we want.

3.2 Algorithms

The main concern for our algorithm is how to choose the appropriate neighbors as the next notes. Our group came up with several ideas and were used in implementing the algorithms. For testing purposes, we implemented all algorithm designs and compared their performances with each other. The program source code can be found here. When picking neighbors, we choose based on their frequencies or a fixed probability. The ticks can be randomly selected or to be calculated.

3.2.1 Hill Climbing Approach

Our first algorithm uses traditional Hill Climbing algorithm with random restart. The definition of a good neighbor in this algorithm is a neighbor with higher occurrences. After choosing the neighbor, compare its total occurrence (-1 value) with that of the current note. Note that the occurrence of the neighbor is different from its total occurrence. The occurrence of a neighbor is the number of times it appears after the current note, whereas the total occurrence of a note is the number of times it has appeared throughout the whole input data set.

If the neighbor's total occurrence is higher, choose it as the next note. If not, randomly select a note among the entire note set. For ticks, we calculate the median of the tick set, and use that tick to generate another set of ticks that contains: a quarter note, an eighth note, and a half note. The tick of each note is chosen randomly among this smaller tick set. This approach prevents the program from choosing a tick that is either too small or too long, and also suitable for our output song.

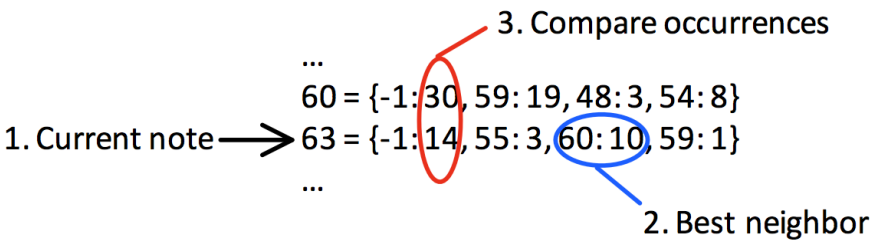

Figure 4: Hill Climbing Approach - Comparison of the Current Note and Its Best Neighbor

The result sounds reasonably decent. The randomness of the algorithm gives a small variation to the music. However, it does sound a bit too random. The tick selection method works as intended. By choosing a particular tick from the set as a quarter note and generate a set, we could even control the tempo of the song. The algorithm repeats itself if the program either does not reach the state with highest total occurrence or it has not met the minimum song length.

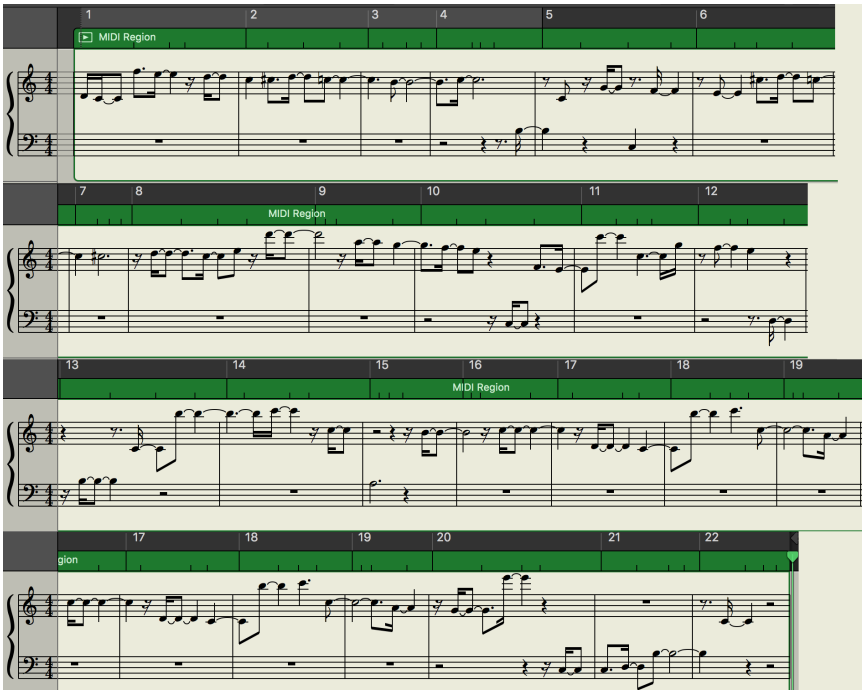

Figure 5: The Music Sheet of Hill Climbing Approach Function Output

3.2.2 Modified Hill Climbing Approach

Our second approach is a modified version of the traditional Hill Climbing algorithm. Instead of searching for a state with the best score, the algorithm simply keeps on going by choosing neighbors repeatedly until it reaches minimum length of a song. This approach focuses on the selection of the neighbors instead of finding the best note in the set. The definition of a good neighbor in this case, is simply a neighbor with high occurrences. Because we are only concentrating on the process of selecting neighboring notes, the starting and end note are set to be the note with highest total occurrence.

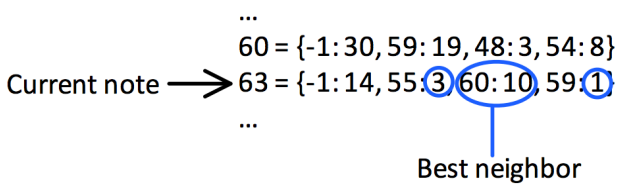

Figure 6: Neighbors Comparison

The first implementation using this approach selects the best neighbor all the time, and has a fixed length for all notes. The result turns out to be the same note playing repeatedly until it reaches minimum length. This is because the most common note (the note with highest total occurrence) itself is also in its neighborhood, which also has the highest occurrence among the neighbors. Even if we start with a different note, eventually it will hit a note that would either repeat itself, or alternate between another note until the song ends.

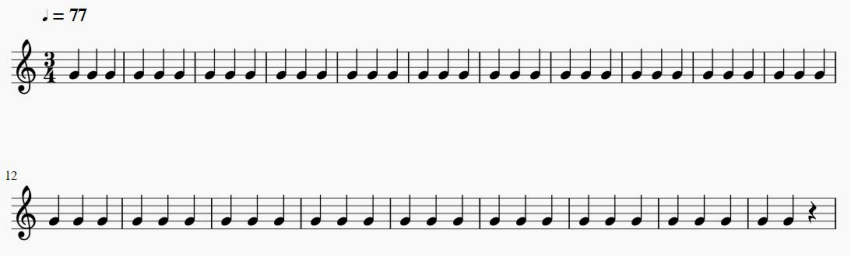

Figure 7: Modified Hill Climbing Approach with Greedy - Output

Next, we used a fixed probability to choose the best neighbor. Otherwise, choose a random neighbor. The tick selection method remains unchanged. The result turns out to be better, although there is still improvement that we could make about this algorithm. Since it is less greedy, there is a chance for the program to choose worse neighbors. However, when the probability is set to be too high, sometimes the notes will still get stuck. When it is set to be too low, the note selection becomes too random.

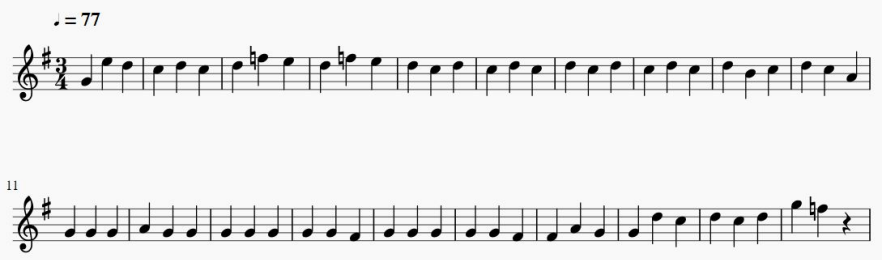

Figure 8: Modified Hill Climbing Approach with Fixed Probability - Output

We once again change only the note selection part of the algorithm and not the tick selection. This time, we use the occurrence of the best neighbor and total occurrence of the current note to calculate the best neighbor's probability. Using this probability, we will determine (by comparing with a randomly generated number between 0 to 1) whether to choose this neighbor as the next neighbor or not. If not, a random neighbor would be chosen instead. As opposed to the previous method, the probability in this case is dynamic. This time, the result sounds a bit better. It does not sound repetitive and note selection sounds reasonable; a big improvement from the previous two attempts. Now we can modify the tick selection method to make the song sound more interesting and melodic.

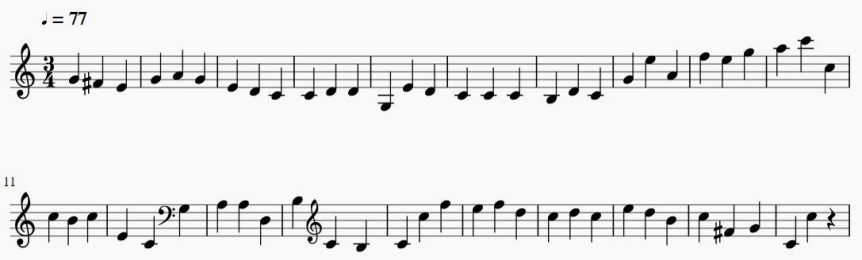

Figure 9: Modified Hill Climbing with Dynamic Probability – Output

In this attempt, only the tick selection method is changed. We will try how arbitrary ticks perform. As expected, the result was not desirable. Due to there being many long ticks in the tick set, a long tick has a higher chance of being picked. To avoid this, we tried to limit the range of the ticks being picked. Even still, the result was still not as good as we hoped to be.

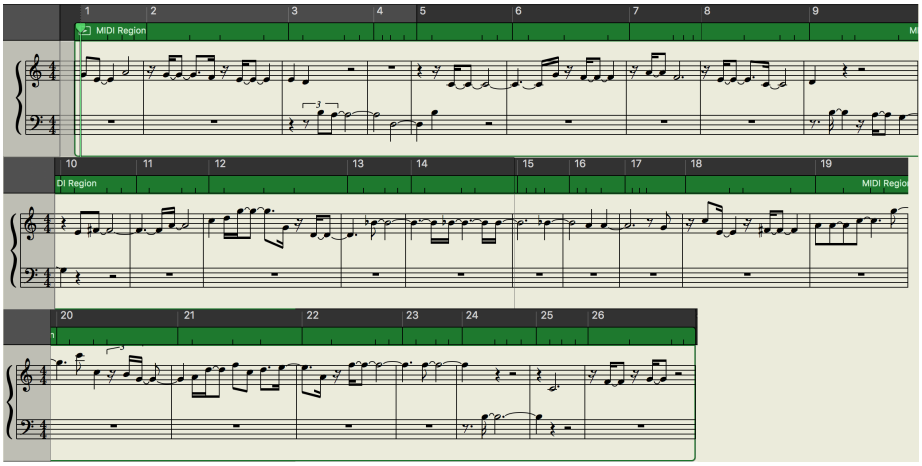

Figure 10: Modified Hill Climbing with Dynamic Probability And Random Ticks – Output

We will make yet another modification to the tick selection method. We noticed that there were a lot of long ticks in the tick set, which would make the song sound dull and boring. Therefore, we will avoid it by assigning a high probability of choosing shorter ticks in the set. The algorithm sorts the tick set in increasing order and divides the tick set into two parts: the first quarter of the set and the rest. With a probability of 90%, it will select a short tick from the first quarter of the set. Otherwise, randomly select a long tick from the rest of the set. This approach effectively lowers the chances of choosing a long tick. The result turns out to be quite decent. It has a lot of varieties of notes but not too much, good sounding melodies, and good selection of ticks. This method produces the best result so far.

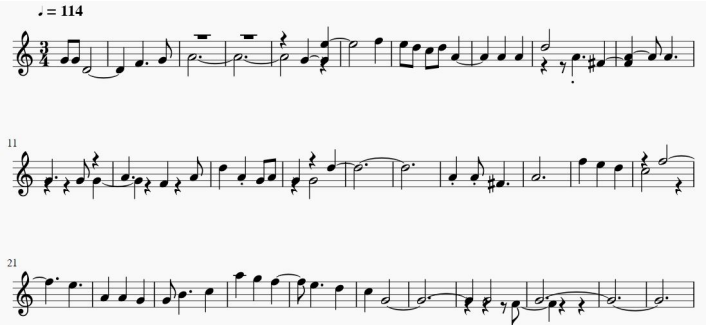

Figure 11: Modified Hill Climbing with Dynamic Probabilities On Notes and Ticks - Output

3.2.2 Modified Hill Climbing Approach

Our final modification addresses the issue of picking random ticks and notes. This approach is aimed to create set of measures (sections of the music in a music sheet) and arbitrarily select a measure to play while generating the song. The tick selection method is the same as our traditional Hill Climbing algorithm approach, where we generate a set of ticks using the median of the tick set. Instead of choosing neighbor one at a time, this algorithm first generates a certain sets of notes (measures) using dynamic probabilities. During testing, we set it to twelve measures to be generated. Each measure has eight notes in them, because every measure can have at most eight notes (eight eighth note). The ticks are randomly chosen from the generated set, and they determine how many notes there will be in each measure. The ticks are selected in a way such that it does not exceed the total length of a measure. Every time a long tick is picked and the measure length is exceeded, the algorithm will randomly pick another tick instead until an appropriate tick is chosen. This way, we are generating one measure at a time instead of a note at a time. This method is often used in real world music making – there will be repeating measures throughout the songs. This approach produces decent music pieces as well.

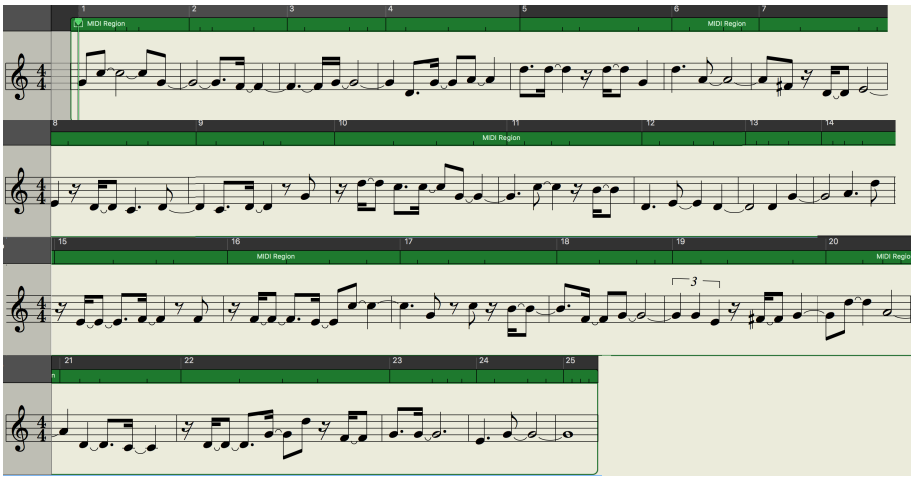

Figure 12: Measure Based Approach – Output

Challenges

Challenges that we encountered in the process include incorporating musical sound into our code, figuring out how to convert output data into digital form and export the sequence of bytes representing our musical melody into a MIDI file. As mentioned before, reading complex music was also a problem so we decided to write our own input set. Using an unfamiliar programming language and library caused us to spend quite some time on learning before actually writing the program. The most difficult of all, was to figure out how to improve our algorithms to generate better music.

Error Analysis

Most of the algorithms had expected results. The traditional Hill Climbing approach had an unexpectedly good result. In our case, our algorithm does not care about whether the best state is has reached was a global or local optima. It simply restarts whenever it reaches the top of a hill. That being said, the result was thought to be worse we hoped to achieve. The modified greedy Hill Climbing algorithm had the worst and undesirable performance. The reason is that the nature of the algorithm is greedy. It will also consider the best neighbor the best option to choose, without considering future events. This is clearly undesirable. Throughout the tests, we realized that arbitrary selection of notes would cause the output song to have too much variety, but strict selection policy would cause the song to be too plain. The solution we came up with was by choosing neighbors based on probabilities. Dynamic probability resulted better performance than fixed probabilities, since it's more flexible and changes depending on the current note. Tick selection is also the same. However, fixed probability works better for tick selection, since there does not have to be a variety of ticks.

Conclusions

In comparison, the modified Hill Climbing algorithm with dynamic probabilities on notes and ticks and measure based approach algorithm have the best performance. Both algorithms adopt the idea of limiting ticks in order to avoid high randomness, and both are choosing neighbors based on dynamic probabilities. Although the measure based approach uses generates one measure instead of one note at a time, which may be a better way to write a program to generate music, the former does produce nice sounding music as well. The difficulty of this project comes from the fact that the quality of a music is subjective to the listeners. We could only do our best to lower the random selection of notes in the song and restrict ticks. Even so, some might consider these results as failures. However, the results that we have presented are the best results that yield from our month-long experiment.

Literature Review

John Biles discussed a model in his paper, a model which can generate a musical sequence using genetic algorithms that search a space of melodic possibilities. The melodic possibilities can then be used to determine which possibilities can help generate good melodies. By mapping different chord types to corresponding scales, it can formulate sequences of notes. We may be able to utilize this technique to help the program make better note decisions.

Reference

Biles, John A. (1994) GenJam: A genetic algorithm for generating jazz solos, Proceedings of the 1994 International Computer Music Conference, 1994, San Francisco.