Music Theory Research

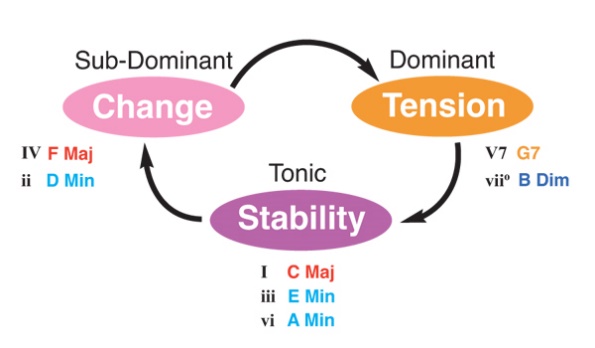
Diatonic chords:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| I | 1 | 3 | 5 | 7 | 9 |
|  |  | Major | Maj7 | Maj9 |
| ii | 1 | b3 | 5 | b7 | 9 |
|  |  | Minor | Min7 | Min9 |
| iii | 1 | b3 | 5 | b7 | b9 |
|  |  | Minor | Min7 | Min7b9 |
| IV | 1 | 3 | 5 | 7 | 9 |
|  |  | Major | Maj7 | Maj9 |
| V | 1 | 3 | 5 | b7 | 9 |
|  |  | Major | Dom7 | Dom9 |
| vi | 1 | b3 | 5 | b7 | 9 |
|  |  | Minor | Min7 | Min9 |
| viii | 1 | b3 | b5 | b7 | b9 |
|  |  | Diminished | Min7b5 | Min7b5b9 |

Could make program build chords using modes of major (or potentially harmonic minor for jazz) scale diatonically. I would create a class *chord* with properties *name, intervals, isDiatonic*. Diatonic chords could inherit initial intervals from their triad counterparts and then be built up, whilst chromatic chords would have to each be made as separate instances of a *chord*. Different chord voicings could be made by randomising the inclusion of non-essential notes (i.e. the 5th in a non-diminished chord or the 7th in a 9th chord).

In the function to identify which chord should come next, the current chord should be identified along with the time in the phrase at which it is played. This will help create chord resolutions. For instance, if the current chord is a V chord, the next should probably be a tonic chord, creating a perfect cadence (this is related to the musical idea of functional harmony). I should create a list of cadences and program those into the script to create better chord progressions. However, the randomness of the program should be kept. A Vdom7 chord should only increase the chances of the next chord being a tonic, not make it a certainty. Therefore the cadences should increase the probabilities related to choosing the next chord.

Another important aspect of functional harmony is chord substitutions (http://www.jazzguitarlessons.net/chord-substitutions.html), this will help my program create more interesting chord progressions. An easy example is replacing a chord with a different chord that shares some notes with the original. I can incorporate this into my program by first getting the program to choose a chord function, and then choose a chord which fits that function (which could be diatonic or could be a substitution of a diatonic chord).

Therefore each *chord* object will need the property *chordFunction* (which could either be “tonic”, “sub-dominant” or “dominant”, the three types of functions).

Programming

My program requires a system for randomly choosing items based on some sort of probabilities for each item. For example, the program might need to choose between playing a D with a probability of 80% and playing an E with a probability of 20%. What will make this harder is the fact that the probabilities are not static, they must change based on what was played before. This problem is not just relevant for individual note choices, it will also be vital for choosing chords and rhythm. For this reason, I plan to have one function ***choose(****items****)*** which will take a 2D array of whatever needs to be chosen in an array with its probability weight (for the previous example: [[D, 0.8], [E, 0.2]] ). After finding <http://eli.thegreenplace.net/2010/01/22/weighted-random-generation-in-python/> I have decided to use the following algorithm since “[This] subtraction method is the fastest when you need one-off selections with different weights”, which is exactly what I need.

**def** weighted\_choice\_sub(weights):

rnd = random.random() \* sum(weights)

**for** i, w in enumerate(weights):

rnd -= w

**if** rnd < 0:

**return** i

I have altered this to fit my program better (since *items* will be a 2D array)

**def** choose(items):

sum=0

for i in items:

sum += i[1]

rnd = random.random() \* sum

**for** i, w in weights:

rnd -= w

**if** rnd < 0:

**return** i

What’s even better about this algorithm is that I don’t need to make sure that the weights all add up to 1, since *rnd* is multiplied by the sum of the weights.

Unfortunately, however, this is only half of the problem. The other (trickier half) is assigning these probabilities/weights to each item. This will have to be done slightly differently for each type of item, but the core code will be the same.

The chord progression generating procedure looks like this:

1. Create a sequence of note lengths (1-4 beats each)
2. Look at the scale tone last used, see if any known cadence is possible/useful (depending on how far into the progression this chord is)
3. Generate a chord function based on this
4. Choose a scale tone whose diatonic chords have that function
5. Choose chord type (dom7, maj9, b13 etc.)
6. Decide whether to use a diatonic chord or a substitution (by choosing a chord sharing some notes with the diatonic chord)
7. Choose whether to use a different chord voicing (by eliminating certain non-essential notes such as thirds and perfect fifths)
8. Generate the chord required (only generate the notes that will be needed in the chord voicing)\*
9. Create a simple rhythm for the chord to be played with
10. Add the chord to the MIDI file with its respective duration
11. Repeat for more chords (go to step 2)

\*For efficiency, this will be a separate function that can be used whether a diatonic chord is chosen or a substitution is chosen.

*(A variable for ‘jazziness’ could be used to decide how often a diatonic chord will be chosen versus a substitution. A similar variable could then be used if a diatonic chord is chosen in order to decide how extended the chord is – generally, the more notes in a diatonic chord the ‘jazzier’ it may sound.)*

The melody generator is much simpler:

1. Look at the first chord, the notes in it and how long it is played for
2. Create a sequence of note lengths (that can last slightly longer than the length of the chord)
3. Add actual notes to these values (favouring chord tones)
4. Repeat for each chord in the progression for as many loops as is desired

I will have one external function (along with *choose()* ) that will be used in both of these procedures to eliminate redundant code:

*createRhythm(durations, length, overlap)*:

* *durations*: a 2D array of note durations and their base probabilities (can change whilst the function is running)
* *length*: an integer value of the overall time that is needed
* *overlap*: a Boolean (true/false) value for whether the note values have to add up to *length* or can be slightly over.
* This will run through the *length* given assigning random note durations (based on the dynamic probabilities of each note duration in *durations*)
* The probabilities will be changed each time from the original set – durations similar in length to the last duration chosen will have a value added to their respective probabilities.
* Will return a list of durations using MIDI duration standards

*(The exact weights of the probabilities and the value by which they are increased will probably need to be altered whilst I am writing the code in order to create generally pleasant rhythms. My current thinking is to make sure ‘durations’ has got each note duration in the right order (longest to shortest). Initially the probabilities should probably be some sort of bell curve distribution, with the added weights depending on the distance within the list between each duration and the previously chosen duration. I could add in a variable for ‘rhythmic variance’ that would decide the magnitude of the added weights, with a lower value increasing the added weights of each duration. Due to the weighted choice algorithm on the previous page, I do not need to make sure that the sum of the weights is always the same. This should make the probability-based programming easier.)*

As part of the chord progression generator, I need an algorithm that can discover possible chord substitutions based on what notes are in the original diatonic chord. I will have to try to make this algorithm as efficient as possible as it could potentially slow down the whole program. (This may not be coded as a separate function, but it is still an important algorithm in my code).

*Substitute(chord):*

1. Look in *chord*’s *intervals* list and identify the notes making up that chord.
2. Choose two notes from the chord and find their interval in terms of how many thirds they are away from each other
3. Choose a random number out of (1,3,5,7) and put the first note chosen in that numbers spot of the new chord. Then add in the other note in whatever place it takes (keeping the interval between the two notes the same).
4. Choose whether to use a different chord voicing (by eliminating certain non-essential notes such as thirds and perfect fifths)
5. ‘Fill out’ the new chord by randomly choosing major/minor thirds to get each ‘next’ note in the chord (not including eliminated notes in the particular voicing).
6. Create and return a new *chord* object with the notes that have just been chosen

*(This particular algorithm will most likely require a lot of tweaking in order to make sure the substitute chords are not too dissonant. I will probably add in a variable for how ‘abstract/jazzy’ the chords should be, with a higher value leading to more chords being substituted out.)*