LazyChords

M1 Project

Final Report

ENS DE LYON

Contents

Introduction

Chords have been omnipresent in Western music since the Renaissance. From classical music to pop music, going through blues, jazz and rock'n'roll, they have been used in many different ways. Their role is predominant in determining the style of a tune: its "mood" isn't contained in the melody line but mostly in the chord progression.

Two parameters are to be taken into account when finding chords for a melody: style and harmony rules. These rules are based on a mechanical analysis of the melody, and fitting a style of chord progressions is obtained by mimicking what has been done in this style of music. These considerations led us toward the possibility of automatising, given a melody line and a style, the writing of the chord progression: the project LazyChords was born.

In the field of automatic generating chord progressions, LazyChords is not a pioneer; SongSmith by Microsoft and Band-in-a-Box by PG-Music were created before with the same objective. SongSmith was proven to be a failure due to its poor performance, while Band-in-a-Box has been always considered as a right-hand man for musicians. However, Band-in-a-box is very complicated and aims only at experts, who are accustomed with musical concepts.

Our new software was designed to be immuned from these flaws. LazyChords not only possesses sharp algorithms, which ensure effective work, but also has a user-friendly interface. The main purposes of LazyChords are giving insights of the power of harmony to beginners, and providing a tool for experimented musicians to get quickly and painlessly a chord accompaniment of quality for any melody.

Our algorithm relies on two orthogonal approaches: an empirical one, using Machine learning, and a theoretical one, using Music rules. The former allows to imitate existing chord accompaniments, while the latter guarantees good-sounding results.

In this report, we will first present our software; then explain the algorithm we use, through its three parts: Machine learning, Music rules, and integration of these; then describe all the side-work we produced; and finally evoke the improvements we plan to do.

1 The software

LazyChords takes a score as input, written in an ABC file, and outputs a chord progression that fits the melody. A sound player is provided in order to let the user listen to the melody he provides together with the resulting chord progression, in order to know immediately if it sounds well. To create this accompaniment, we chose to put one chord per measure, and to optimize on the chord associated to each measure

This optimization is done through music rules, that can be defined by the user. Trying to deal with the style properly is done by a machine learning part, which is also considered as a rule. Figure 1. gives a better view of the overall software architecture.

The core of the software is the Music part, which optimizes according to the rules. The chord progression output by the software is mainly decided by this part. The machine learning part is based on Hidden Markov Models. The database to learn from has been conceived, and filled by our team.

The software is written in C++, and multi-platform. We chose to use the following libraries, that are available on the three main Operating Systems (Unix, Mac, Windows):

- Boost: `http://www.boost.org/`
- Qt 5 for the graphic user interface: `http://qt-project.org/`
- abcMIDI to convert abc file into midi file to play them: `http://abc.sourceforge.net/abcMIDI/`
- abcjsto have a score rendering: `https://github.com/paulrosen/abcjs`
- the noun project icon set: `http://thenounproject.com/`

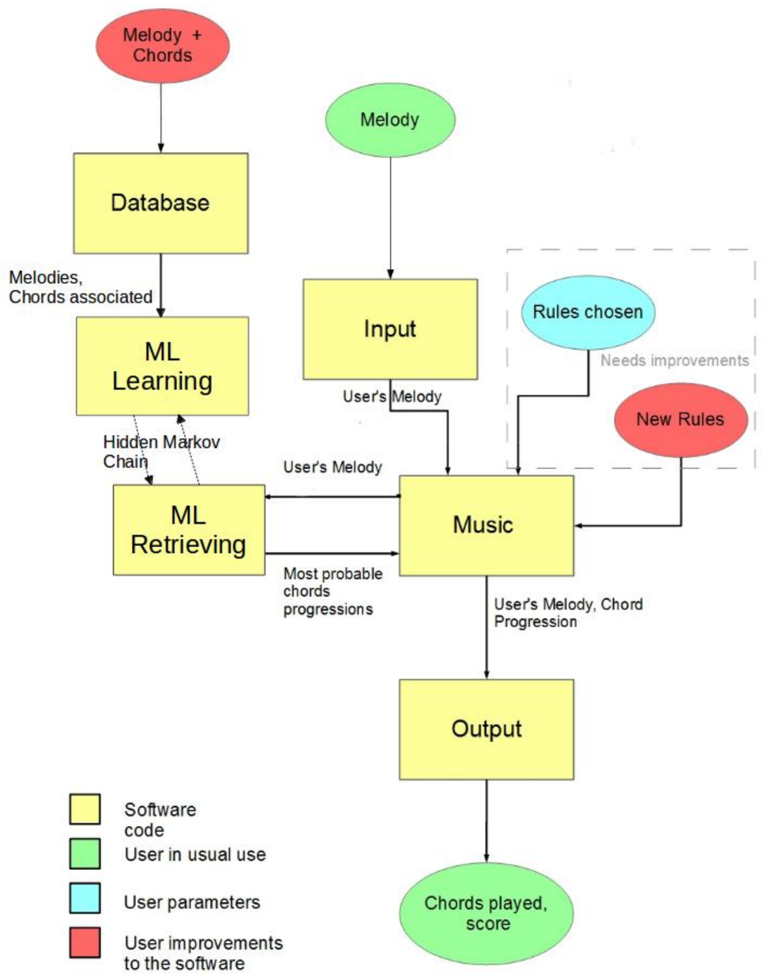

Figure 1: Architecture of LazyChords

2 Musical Background

In this section, we will introduce some basic notions of music essential to understand the whole project. We largely restrict the global musical definitions of such notions to the simple fragment considered throughout this project.

A *note* is a continuous sound played on an instrument or sung by someone. It has a length (its duration) which is a rational number, and a height (the frequency of the sound wave). A *melody* is a finite sequence of notes or silences. We consider indeed only monophonic melodies (at most one note at a time). The corresponding sound thereafter is the sequence of notes/silences played one after each other with respect to their height, duration and the sequence order. A melody is divided into regular measures which are all of size 4, counted in number of beats.

A *chord* is a finite ordered set of notes with the same duration, but those ones are played at the same time. We only consider chords of size 3 in this project (the most popular), though our data structure allows chords of size 4 also. We consider a fragment of this large set of "chords" containing only "classical ones", that sound well (triads and seventh chords). A *chord progression* is a finite sequence of chords. We only consider regular chord progressions: all the chords have the same length, which is length of a measure, so 4 beats.

(Most) musics are composed of (at least) two parts: a melody and a chord progression. The melody is usually what people remembers, but it's the chord progression that gives the colour of the music.

The goal of our software, formalized with those definitions, is the following: Given a melody, what is the chord progression that sounds "the best" with it ? The whole point of this project is to understand how quantify this.

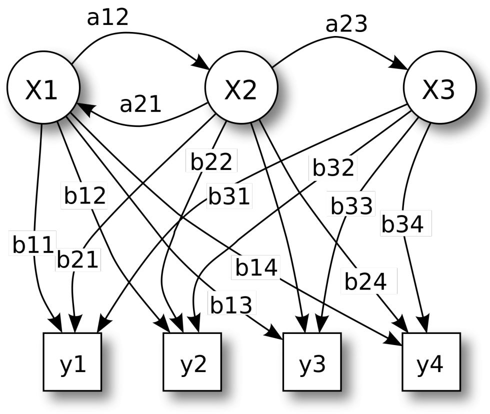

Figure 2: A Hidden Markod Model

3 Machine Learning

First, let us recall that *Machine learning* is "a scientific discipline that explores the construction and study of algorithms that can learn from data"[1]. Here, our purpose is to use Machine learning paradigm so as to be able to propose, given an input melody, a sequence of chords that statistically fits a Database.

We will first give a bibliographical overview of the the use of Machine learning for similar purposes, then the model we chose (Hidden Markov Model), and how we implemented it, alongside with the complexity of our algorithms.

3.1 Hidden Markov Models

Hidden Markov Models are a statistical tool widely used in machine learning problems that involve sequential data, such as text or music for example. The figure 2 shows an example of a HMM. It is composed of *hidden states* (in the example, x_1, x_2, and x_3), that form a Markov Chain, which means that they are linked by edges weighted by a probability. The other part of the HMM are the *observations* (in the example, y_1, y_2, y_3 and y_4). For each state, we have a probability distribution that indicates the probability to output each of the observations.

At the beginning of the process, the computation starts in one of the hidden states, according to some probability distribution π, called the *initial probability distribution.* Then, at each time step, one state is

[1]http://en.wikipedia.org/wiki/Machine_learning

chosen among the neighbours of the current state, according to the distribution induced by the weights of the edges, and this state becomes the new active state. After that, an observation is chosen randomly according to the distribution of the state, and is emitted.

From the user point of view, the current active hidden state is not known, the only information he has is the sequence of observations that are emitted by the model.

3.2 State of the art

In this part, we present bibliographical results on problems similar to ours.

3.2.1 Allan & William

The objective of the work by Allan and William [1] is to construct Hidden Markov Model that automatically compose a sequence of chords (a chord progression) for a rough melody.

For the melody as well as the chord progression, they use a discretization step size of one beat, with the assumption that the notes and chords do not change during a beat (there are often 3 or 4 beats per measure).

To train the HMM they used the Baum-Welch method which maximizes the probability that the model will emit a given set of sequences of observations.

Their Hidden Markov Models do not only produce a chain of chords, but also the harmonic functions that the chord is serving (i.e. the alto, tenor and bass).

3.2.2 Songsmith by Microsoft Research Ltd.

The objective of Songsmith (the original name is MySong) [2] is producing immediately the chord sequence coincident with the input melody. The technique of Songsmith is based on the harmonization-HMM by Allan & William.

The developers collected about 300 music sheets that contained both music notes and chords. From the database, they created two matrices to train the HMM. First, they ignored the melodies of the music sheets and kept only statistics on the transition of chords and obtained two **chord transition matrices** (one for major mode, and one for minor mode), describing the probability that one chord appears right after another. Then they created the **melody observation matrix** which represents the number of times a certain note is associated to a certain chord.

For a matter of normalization, they also fixed the length of the measures. As a parameter of the software, the user is able to set some factors : the “happy factor” and the “jazzy factor” that influence a bit the output (basically by determining the mode and the kind of chords used). To generate the chords (which is called **decoding**) that coincide with the melody, the developers used the Viterbi algorithm [3] (a popular algorithm to find hidden states of HMM) to produce the most likely chord sequence.

One of Songsmith’s strength is its computing speed : it creates a chord progression almost immediately. However, although they fixed some parameters to enhance the precision of the generated chords, because of the small set of musical notes and chords, and the too simple HMM they use, the generated chord sequences are very far from the expected chords according to feedbacks from users.

3.2.3 Orio et al.

The work by Orio et. al. [4, 5] is on using HMM to automatically generate a **score following**, which is an accompaniment (audio synthesis or sound processing) to a human performer. Different from Songsmith, which orients to popular users, this method aims at helping musical experts, and automatically produces the accompaniment for their song.

The problem is, given the observation (the melody), how to find the optimal sequence of **hidden states**. The procedure of finding this sequence is called **decoding**. There are two levels of states: states at the **lower level**, which model the incoming signals like notes or rests (musical notation to denote the absence of note played), and states at the **higher level**, which model all events written in the music sheet as notes, trills (rapid sucession of notes), chords, rests... Each state at the higher level is made by a set of states at the lower level.

Normally, training of HMM is performed using the Baum-Welch method, as in [1], that maximizes the probability that the model will emit a given set of sequences of observations. However, Orio showed that the procedure is not suitable to train HMM and introduced a new technique, which maximizes a more complex formula. Decoding with Viterbi algorithm is widely used, but it is not suitable for real-time score following. The reason is that for a given sequence of observation, the algorithm outputs the most probable path in the hidden states. In fact, for score following, the main goal is to find the state locally corresponding to the emitted note, so Orio et. al. used the algorithm by Durbin et al. [6] where the probability of each individual state is locally maximized.

3.3 Implementation choices

As shown by the bibliographic review, *Hidden Markov Models* have been widely used in similar contexts, that's why we decided to use them for our problem.

For LazyChords, each hidden state of the model describes a Chord, and each observation describes a measure of a melody (further explanations below). Then, a song (melody and chords) corresponds to a path through the chain (the sequence of chords), each state emitting the associated measure of the melody.

The main problem of Markov chain is that they don't have any memory. To overcome this problem, we restrained ourselves to chord progressions with a period of 4 chords (very common in nowadays musics). Then, for each chord C, we created 4 states $S_{C,1}$, $S_{C,2}$, $S_{C,3}$ and $S_{C,4}$ in the model, one for each possible position in the progression. To ensure the validity of the progression, we enforced the following property :

$$\forall C \neq C',\ \forall i, j \in \{1,2,3,4\},\ (j \neq i+1 \mod 4 \Rightarrow p(S_{C,i}, S_{C',j}) = 0)$$

Overall, there are 288 possible chords, so we had 1152 states in the model

As for the observations, we had to tackle an issue: of course, we could not decide the set of observations to be the set of every possible measures. It is far too huge, and would be pointless: the HMM would have provided results only if the input melody was already in the database. Thus, we had to implement an efficient description of a melody, which is a kind of high level description. We discussed several choices, and chose a vector of the accumulated duration of notes in the measure, which represents in a way a footprint of the melody. This allows a realistic but not too general description. Moreover, there is then a very natural way of defining the distance between two observations (distance induced by the infinite norm); so that we can get the closest known observation (from the database) of the input melody. This representation has however the drawback of loosing all the temporal information of a measure: as an example, a measure "C D E F" will be encoded exactly the same way as "F E D C".

Using the HMM is done in two parts. The first one is the *learning* part, with a database. In this part, the path through the chain and the resulting observations are completely known for every song in the database. So the purpose is to compute the corresponding probability distributions of the transition function and the observation function. This is done by computing the empirical mean of each transition in the matrix. One obvious shortcoming of this method is that if a chord doesn't belong to any song of the database, we have no information regarding the transition from the corresponding state to any other state, and thus the means are not defined. We addressed this issue by initializing the matrix so that these "orphan" states had equal probability to go to any other valid state, and to output any valid observation. In a future prospect, it could be worth considering using unsupervised learning for example through the Baum-Welch algorithm (which we have implemented). We could thus use entries without annotations, which are usually much easier to find on the Internet. However, it is unlikely that using exclusively this kind of learning would lead to coherent enough results, so this algorithm should be applied only after an initial supervised learning, to refine the probabilities.

The second part is the inference process, or retrieving. Given the probability distributions of the HMM and a melody, it allows to infer what path in the chain can have emitted such a melody (thus, to find a fitting chord progression). For that we also implemented a classical algorithm, the Viterbi algorithm, which is a dynamic programming algorithm. It computes the most probable sequence of states given a sequence of observation, and the corresponding conditional probability. We had to modify the algorithm because the Music workpackage expected us to provide not only one but the n most probable sequences of chords, so as to be able to compare them with other rules. Also, two different versions were implemented. One is fast but needs a large database to be efficient, because for each measure, it considers only the closest observation in the set of known observations. Thus, if this observation was not very frequent in the examples of the database, the algorithm outputs poor results. The other method is a lot slower (several minutes for a single execution) but compares the input melody with every known observation, and add a corrective factor to the probability to penalize the most distant observation (with regard to the metric we defined, the infinite norm). Thus, it allows to overcome partly the problem of rare observations and works rather well even with a small database.

3.4 Complexity

If we denote by $Data$ the size of the database (total number of measures), s the total number of hidden states in the model, o the number of different observations, n the number of expected chord progressions, and m the length of the input melody, we have the following complexities:

- learning: $\Theta(Data(s^2 + so)$ (for each entry of the database, we update the probability matrices);
- fast retrieving: $\Theta(s^2 \cdot mn)$ (classic Viterby algorithm, but with the n best progressions);
- slow retrieving: $\Theta(c^2 \cdot mn \cdot o)$ (the additionnal factor comes from the fact that we consider all observations instead of only the closest one).

4 Music Rules

A parallel approach was studied in the conception of the software: the notion of music rules. This relies on the idea that, if Machine Learning is meant to detect intricate musical notions, such as style, we can implement many musical notions manually. In fact, chord progressions follow logical rules, which are given by the musical theory of harmony.

The first idea that arises is to implement these rules is to give to each chord progression generated by the Machine learning a value, depending on these harmony considerations. Then, the objective is to find the best chord progression, balancing between what the Machine Learning part outputs and what music rules output. Also, we wanted to be able to modify those rules dynamically, for at least two reasons. First, this allows us to improve easily the quality of those rules (empirically), which is essential in the process of writing convincing rules. Secondly, this would allow us to extend the writing of music rules both to the user (through a simplified interface), and to contributors with a deeper knowledge of harmony than ours, but without necessarily any coding knowledge.

However, writing a complete language for music rules, which first seemed interesting, arises some inherent issues. First, an exhaustive language would necessarily be complex to use, which can be a problem if non-programmers musicians want to contribute in rule writing. Also, without any knowledge on the shape of the rules, we wouldn't have been able to write an efficient algorithm to find the optimal chord progression.

Thus, we have chosen a modular approach: we wrote several types of rules, that can be divided into the two following categories:

- Local coherence between melody and chord;
- Chords transition.

The first category of rules is applied on one chord at a time, ensuring that the chord fits well with the notes of the current measure. The second category of rules only applies on chords, independently of the melody, ensuring that two neighbouring chords follow a coherent progression. The rules falling into that category are particularly well explained by harmony.

For the first category of rules, we want to give a value, a score of how a chord fits on a measure, whereas for the second category, we will give a score to pair of chords. We can actually achieve that by writing the rules as *extended arithmetic expressions* over variables describing some aspects of the melody and/or chords, depending on the rule type, either by boolean or statistical values. This is enough to capture everything we want to say about the melody and the chords (basically, we can encode "if then else" kind of statements), and arithmetic expressions are a convenient way to give a score, either to one chord and a measure, or for pairs of chords.

The five rules we have implemented are listed in the appendix.

5 Algorithm

Once we have the music rules and the machine learning results, we still need to output the best chord progression. This part presents how we unify them, and the obtained results.

To unify the machine learning results with the music rules, we add a bonus to chord progressions that are returned by the machine learning.

The obvious naive algorithm would be to generate all possible chord progressions and calculate the value given by the music rules on each one of them. However, the complexity of that algorithm is $\Omega(c^m)$, c being the number of possible chords and m the number of measures.

We propose another algorithm using dynamic programming which has a complexity of $O((c^2 + ML)m)$ where ML is the number of chord progressions returned by the machine learning algorithm. Here is the algorithm:

Let $Best(i,t)$ be the value of the best chord progression (only counting music rules) on the t first measures of the melody and that finishes by chord i. Let $val(i,t)$ be the sum of the values of how chord i fits on measure number t (each rule of the first category giving a (most likely) different value). Let $link(i,j)$ be the value of the rule that describes the link between chord i and chord j. Let $overall(cp)$ be the value over the whole melody of the chord progression cp according to music rules only. Let $Best$ be the value of the best chord progression (now adding machine learning). We have the following equations from which we can deduce the algorithm:

1. $Best(i,t+1) = \max_j Best(j,t) + val(i,t+1) + link(j,i)$
2. $Best = \max\{Best(i,n), i\} \cup \{overall(cp) + MLBonus(cp), cp \in ML\}$

5.1 Results

We could not test formally the results of our HMM, but here we give some empirical clues. As we already stated, our database is too small to deduce efficiently the expected chord progressions. Yet, the musicians of the group found the results of the slow version very promising; yet its running time is at the moment too prohibitive to be integrated in the released version. On the other side, the faster version works only on melodies that are relatively close to these in the database (else it may only give arbitrary chord progressions of probability zero).

Concerning the whole algorithm, including all the rules, we have had convincing results on pop, rock'n'roll, traditional, and even classical music, in a short execution time. In the released version, chords very rarely felt strange to our ears. Sometimes selected chords are questionable (for example minor chords on a major melody, or vice versa); but the overall result is convincing. Nevertheless, it cannot handle more intricated music styles in terms of harmony, like jazz.

6 Production

6.1 Website

A website with all the needed information, and a link to download the source code, has been created. It can be found at the following address: `http://www.lazychords.nicolascarion.com/index.php`.

6.2 Databases

If one wants the Machine Learning algorithms to actually work, and to output proper chords given an input melody, proper chords must be defined for these algorithms. So a database defining this criterion must be created.

To that extent, we decided to select melodies and songs that already exist through the pop culture, and write down the melody, and the corresponding chords the artists play. But there are major difficulties here. First, we decided to restrict ourselves to specific melody formats. More specifically, we restricted ourselves to one chord per measure. Thus, many songs do not fit our requirements, as the chords may be played along a non trivial rhythm in existing songs. Then, it seems that no reliable, already existing database, exists. Indeed, score and chords can be found through the internet, but are most of the times inaccurate, or even incorrect. That is amplified by the fact that main melodies are often vocal, thus not totally representable through scores. Also, no sufficiently simple, reliable, automated solution seems to exist to this problem. Indeed, no reliable audio to any classical format converter has been found: mistaking a note by a semitone can easily create unpleasant harmonies. But a database must be created, in order to, at least, test the algorithms. Considering the size of the project, we created a database by hand, either by listening to audio files directly, or by comparing and correcting existing scores with audio files. This, at least, led to a small but hopefully reliable data for the algorithms.

In order to implement this database we chose to use only text files to store the data. The main reasons for this choice are that it is easy and fast to implement, and we only needed two functions in the database, add a file and load all the songs of a certain style. Thus, we have one file for each style, containing all the songs of this style.

6.3 User's interface

The user must provide a melody for LazyChords to generate chords. We chose the abc format as it is readable, and has all the information we need about the melody. For now, the parser does not parse all of the specifications of the ABC format as we did not need all features.

We chose to use Qt 5 to implement the Graphic User Interface (GUI) because it is C++, easy to use, portable, and provides nice interfaces. The GUI provides all needed buttons: an import file button, to put the ABC file of the melody, a button to generate the chords, and all player related buttons, like play, stop, two different volumes (one for the chords, one for the melody), but also the tempo. The GUI provides an emphasis of the chords as they are played, to help the user follow the song. It has also a score rendering using the library abcjs, and the possibility to look at the ABC file to see if it is the right one.

To play the music, we use ABCToMidi to create a midi file and then the library FMOD to play it, which added some licensing issues, as FMOD is free for independent or non-commercial uses, but it is not open-source, so we can not put the sources of FMOD, thus the software requires the user to install FMOD on its computer to be able to use the software with the sound player.

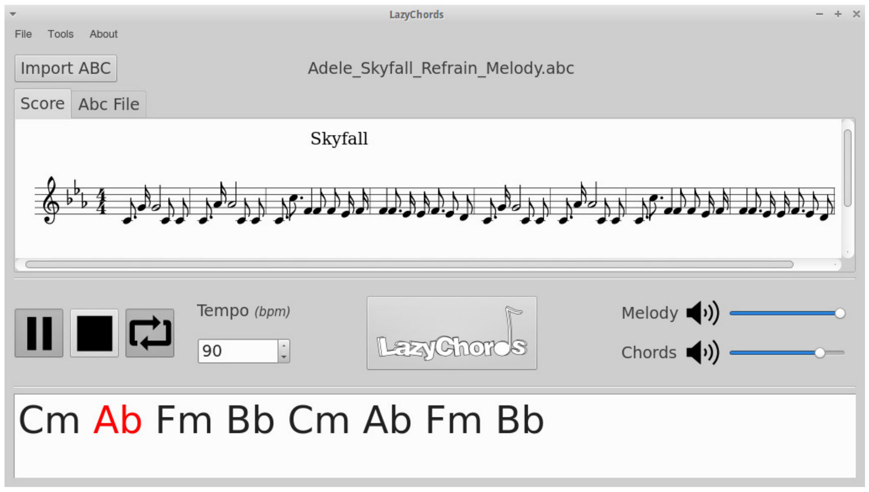

Figure 3: Graphic User Interface

7 Conclusion and Perspectives

The software we provide is clearly not a full software, it needs some improvements. However, results are already quite good, and we really would like to improve them, so the software can really be used by musicians and gets to the point where its name is really relevant, this means the user would not even have to think about chords and would get a very nice chord progression to play. We reached the objectives we chose, with a little restriction about style, and we created the basis of a much more evolved software. This M1 project is only the beginning of a larger project, which will hopefully last for years.

The project will continue, because at least half of the members are willing to continue working on it. This legitimates this part of improvements that are planned to be in the next version of LazyChords which would start to be implemented in January. We would also be very happy if some people would like to join us on this project. We designed LazyChords, not only to work after three months of work, but to be able to improve its quality. For instance we defined music rules so that the user can describe his own rules, giving him much power. We knew that it would be very hard for us to release in December a software enabling the user to make his own choices. Anyway, we built it this way, so we can continue in this perspective, without having to restart this part from scratch. This is essential to get more contributors. We may not have produced the best chord finder we could have done with the same amount of work, because defining the rules description well took time, but we probably created the best basis for the project to live afterwards.

The main improvement we would like to do is the management of music rules. We want to simplify rules writing, make it more intuitive, and provide a way to write them in the GUI. We also would like the user to be able to choose how to balance the rules (including the Machine Learning part), in the GUI of course. The idea is that a musician, non computer scientist, could improve the rules, and add new ones, without any knowledge in C++. We also would like to be able to choose the rules from a database, in order to get usual rules and choose exactly the ones the user wants, maybe with predefined sets of rules to help users who are not aware of music concerns enough, or to specialize on a certain style of music.

We would like to improve the database for the Machine Learning algorithm in order to get better, so that the software could get more taste of style, since it could be hard to get the right chords for each style only with rules. Thus we need a larger database, with much more styles of music.

The Machine Learning part in itself can be improved too. There is an algorithmic perspective of decreasing the running time, because the retrieving algorithm (as well as the main unifying algorithm), that uses dynamic programming, could easily be parallelized. Moreover, it would be interesting to find new ways of representing the observations (a boolean presence vector, some notes…), and compare, formally, the corresponding results.

We also wanted to implement a SQL database for music rules. The idea is to provide a database on the website with concurrent access to get a bigger database for music rules, so users can contribute and share their rules with everyone. We already worked on that, but the main issue is that the management of music rules in the software is not easy to do yet.

We also want to improve the GUI to enable editing the score and the chords. Now, the user can only play the chords he got from the software, but he can not change one of them that sounds odd to see if it could sound better with a new one. Also if he found a note in the score that is not the right one, he can not change it easily. He has to edit the abc file by hand (using his own text editor), and to reload it in LazyChords. Adding a cursor that follow the music on the score would be interesting then to find easily what note to change if one is not the right one.

Developing a web and an android application is also one possible improvement, maybe with less functionalities first. An android application could be very useful for people who do not want to bring their computer everywhere and get chords as soon as they want to play some music.

Improvements can also be done on input. First, we do not parse every ABC file, lots of things allowed by the ABC format are not taken into account. Improving the parser to parse all ABC files could be very interesting for a real use of the software, as it is already not so easy to find the ABC file of the music you like; and if you are not able to write it yourself, finding one that we parse could be complicated. A huge improvement could also be to implement a microphone input, but it is very complicated to get the right notes, even limiting ourselves to only one instrument, or to voice only. This is an improvement we all would like to add, but we know it is hard to make it work and it costs a lot of work even to know if we can achieve this well enough. It is more of a wish than a real objective for now, but it can be later. We know projects which tried to implement this kind of microphone input, and we may rely on them for this objective.

Acknowledgements

We would like to thank Benjamin Girault, who has helped us a lot, with useful advice, restraining some ideas when they were too risky or too far from what we could actually do. We would like also to thank the expert that pointed the flaws of our midterm report, so we hope we corrected them properly for this report.

References

[1] Allan, M., Williams, C.K.I. Harmonising Chorales by Probabilistic Inference. NIPS 2005.

[2] Simon, I. Morris, D. Basu, S. MySong: Automatic Accompaniment Generation for Vocal Melodies. CHI 2008.

[3] Forney, D. The Viterbi algorithm. Proc IEEE 61(3), 1973

[4] Orio, N., Déchelle, F. Score Following Using Spectral Analysis and Hidden Markov Models. ICMC 2001.

[5] Schwarz, D., Orio, N., Schnell, N. Robust Polyphonic MIDI Score Following with Hidden Markov Models. ICMC 2004.

[6] Durbin, R., S. Eddy, A. Krogh, and G. Mitchison (1998). Biological sequence analysis. Cambridge, UK: Cambridge University Press.

Appendix

Utilisation

Installing dependencies

On Archlinux All the needed packages are available in the official repositories. To install them, the following command should be enough :

```
$ pacman -S fmodex boost boost-libs cmake qt5-base qt5-webkit
```

On Ubuntu Due to licence issues, the library FMODex on which our software relies is not available in the repositories. There are two options to solve this issue : the first one is to download and install by hand the library (`http://www.fmod.org/download-previous-products/`). The other solution is to rely on the cmake script provided in the archive, which downloads a local version of the library if not found on the computer. Please note that it won't install FMODex properly on your computer.

The other packages can be installed as follow :

```
$ sudo apt-get install libboost-dev libboost-filesystem-dev cmake \
qt5-default libqt5webkit5
```

On other Unix-like systems This project has not be tested on other systems, including Mac OS, but all the library used are cross-platform so in principle it should work correctly. Please make sure you install Qt 5 (`http://www.qt.io/download-open-source/`), Boost (`http://www.boost.org/users/download/`) and FMOD (`http://www.fmod.org/download-previous-products/`).

Get the software

On windows You can just download an installer from the download page of the site (`http://www.lazychords.nicolascarion.com/download.php`)

On other systems Download the source archive from the website (`http://www.lazychords.nicolascarion.com/download.php`). Then you can extract the source and start the build process :

```
$ tar xvzf lazychords0.1b.tar.gz
$ cd lazychords
$ mkdir build
$ cd build
$ cmake ..
$ make -j
$ cd ..
$ ./LazyChords
```

Usage

Use the import ABC button and select the file of your melody (this should be an ABC file). You can look at the score to see if it is the right melody, you can also play it with the play button, and choose the tempo if the tempo is not right. You can toggle the repeat button if you want your music to repeat. You can then generate chords with the LazyChords button in the middle of the button bar. You can mute, change the volume separately of chords and melody by clicking on the volume icons (to mute/un-mute) or using the sliders. When you play the music with the chords, the chord that is played is displayed in red, and the box moves to always show you the chord played. So you can easily follow the song, but also mute the chords and play your own instrument and know when to play the chords, only watching this part of the screen.

Contribution

Everyone can contribute to the project, and we are likely to have more people working on it, or at least some feedback from people trying it. To get contributors we would have to spread the software among musicians. Also having some feedback from beginners to know exactly what they need to improve their knowledge about chords using LazyChords. New active contributors could be very interesting, they can bring new ideas, new ways of thinking the music rules to get better results. An other way to contribute will be brought by the online music rules database we would like to have on the website. This would need enough people to create music rules to be relevant, but with the huge power of the rules we defined, users are really free to create good rules, sharing rules with beginners could then be a good way to share their knowledge of music. Creating a forum is also an idea to give the possibility to discuss pertinence of rules, or to help beginners with them. If you want to contribute do not hesitate to contact us at the following e-mail address: benjamin.gras@ens-lyon.fr, or to go on the website. We use git to work, and there is a full documentation of every function implemented in the software, using Doxygen to create it, so a new contributor can easily join us. This documentation can be found on the website.

Our music rules

We implemented five music rules. As we said, they are encoded using arithmetic expressions, which are described by the following grammar, with the usual priority conventions:

$$\begin{aligned} A, B \quad ::= \quad & k \in \mathbb{R}_+ \mid v \text{ or } v(i) \text{ or } v(i,j) \mid A + B \mid A - B \mid A * B \mid A/B \mid \\ & A^B \mid [A = B]^2 \mid abs(A) \mid \dots \end{aligned}$$

Let's then define those rules. We only have to describe how to interpret the variables (v or $v(i)$ or $v(i,j)$), a rule being an arithmetic expression on those variables.

Local coherence of melody and chord

Rule 1 The goal of this rule is to check if the notes on the beats (1, 2, 3 or 4, since we consider only measures of 4 beats) are present in the chord:

Variables $v(i,j), i = 1, 2, 3, 4, j = 1, 2, 3$

Interpretation $v(i,j) = 1$ if the note on the ith beat is equal to the jth note of the chord, $v(i,j) = 0$ otherwise

Rule 2 The goal of this rule is to check if the notes of the chord are in the melody in a long enough time:

Variables $v(i)$

Interpretation $v(i) = p$ where p is the percentage of the time the ith note of the chord appears in the melody

Rule 3 The goal of this rule is to check if the longest notes of the melody are present in the chord:

Variables v

[2] $[A = B]$ is the boolean value of the proposition "$A = B$", given an evaluation of the variables

Parameter v the minimum length of notes we are going to look at

Interpretation v is the percentage of notes which aren't in the chord and that last more than x (default is 1 beat)

Rule 4 The goal of this rule is to check if the notes of the chord are in the key: if we do not actually look at the notes of the measure of the melody for this rule, it is still dependent of the melody, since the key is one of its parameters. It is the reason why we have put this rule in the same category as the previous ones:

Variables $v(i), i = 1, 2, 3$

Interpretation $v(i) = 1$ if the ith note of the chord is in the key, $v(i) = 0$ otherwise

Chords transition

Rule 5 The goal of this rule is to check if pairs of adjacent chords are harmonious, logical, ... It applies on each pair of measures:

Variables $v(i, j), i = 0, 1$ (0 is the precedent chord, 1 the current one), $j = 1, 2, 3$ (notes of the chord)

Interpretation To each of the 12 possible notes is associated an integer between 0 and 11 such that the difference of two notes (mod 12) is the number of halftones between them. $v(i, j)$ is then the corresponding integer of the jth note of the ith chord. This allows us to test intervals between those chords, which is enough to implement most harmony considerations.

Organisation feedback

The organisation of the project went well from a general point of view. We achieved our objectives in time. We had nice relations between the members of the project even if some disagreements have been a bit hard to deal with. Moreover, a least half of the team wants to continue the project, which shows that it was well enough to keep people willing to work together. The organisation was a bit strange due to the fact that the coordinator did not know music or was even good to implement things, so the organisation was built on more people, one directing the project from a musician point of view, and two others directing the code writing (some coding rules were chosen, some code has been cleaned to be more understandable). This structure was a good point with no one having too much organisation to do, but enough organisation overall.

Yet, everything was not perfect. First, the organisation failed to make every member involved enough, mainly non musicians members. We think this issue arose from the lack of leadership of the coordinator, in the sense that he did not motivate people to work on the project.

Then, communication was an issue, some people thought some things implicit that were not for others. This slowed down a lot the merge between the different parts. We chose to give us two weeks to merge, in order to let us some time to improve some particular points of the software. This was a huge fail, we took more than four weeks to merge, deleting all the time for improvements. What makes this less problematic is that we did manage to have something already quite good, and we continued to work on the parts even if the merge was not finished, trying to merge as much as possible between parts relying on others. The reason of this much longer time to merge is the lack of communications, that made some parts overwritten many times. The lack of strength in the asks to respect the deadlines combined with the amount of work we had to do these weeks played also a role in that lateness.

An other failure in the project was that some people wrote a lot in parts they should not have with respect to the workpackages. This happened because of the two issues before, some people less involved put less energy trying to make everything work well, or thought they provided all the needed functions even if some were missing to use the code properly. This also shows some failure in the workpackages, we did not manage to put enough amount of motivation in each group, leading some to work slower, or to need other people to work on them.

The last issue of organisation was not to deal well enough with the lack of confidence of some people into others, leading to the willing to re-implement some parts to make them work easier, the database format, for instance, was a huge issue of this kind. We think this is part of the previous point, we should have more separated the workpackages, enhancing communications between workpackages that were linked and reducing feedback from people who were not involved. Even if it is needed to have some feedback from other points of view, it should only be feedback and not authoritative arguments.

To conclude this feedback on organisation, we can say that the workpackages were not well distributed, that communications were not good enough, but overall it went well due to the motivation and the work of most of the members.