A Novel Approach to Automatic Music Transcription Using Electronic Synthesis and Genetic Algorithms

Gustavo Miguel Jorge dos Reis*
School of Technology and Management
Polytechnic Institute of Leiria, Portugal
gustavo.reis@estg.ipleiria.pt

Francisco Fernandez de Vega†
University of Extremadura, Spain
fcofdez@unex.es

ABSTRACT

This paper presents a novel approach to the problem of automatic music transcription using electronic synthesis with genetic algorithms. Although the problem is well known and different techniques have been applied before, evolutionary algorithms have never been considered when addressing this problem. We show that, by means of a series of steps, a polyphonic MIDI file -containing instrument's partitures- can be automatically generated from an audio recording, by extracting and separating simultaneous notes. We describe also the future steps of our research in order to improve the genetic algorithm: increasing performance and decreasing memory usage, extracting the instrument's features, accurate transcription of note's duration and, if necessary, the employment of parallel systems. The results obtained shows the feasibility of the approach.

Categories and Subject Descriptors

H.5.5 [**Sound and Music Computing**]: Signal analysis, synthesis, and processing; I.2.m [**Artificial Intelligence**]: Miscellaneous

General Terms

Algorithms

Keywords

Automatic Music Transcription, Genetic Algorithms, Multiple F0 estimation, Melody Extraction, Polyphonic Music Transcription, Polyphonic Pitch Estimation

1. INTRODUCTION

Automatic music transcription is the process in which a computer program writes the instrument's partitures of a given song or an audio signal. Hence, automatic music transcription from polyphonic audio recordings is the automatic transcription of music in which there is more than one sound occurring at the same time: multiple notes on a single instrument (like a piano) or single notes in multiple instruments.

Music transcription is a very difficult problem, not only from the computational point of view but also in a musical view since it can only by addressed by the most skilled musicians. Usually, only pitched musical instruments are considered. Therefore, recognizing drum instruments or the sounds of the singer is not discussed here.

Despite the number of attempts to solve the problem, a practical and applicable, general-purpose transcription system does not exist at the present time. The available systems clearly fall behind skilled human musicians in accuracy and flexibility [7].

1.1 Terminology

It is important in the context of music transcription and melody extraction to clarify some terms before going any further. *Pitch* is one of the fundamental aspects of music. It is the tonal height of a sound. Pitch indicates how "high" or "low" a note sounds. *Fundamental Frequency*(F0) is the lowest frequency in an harmonic series. Pitched musical instruments are usually based on a harmonic oscillator such as a string or a column of air. Both can and do oscillate at numerous frequencies simultaneously. Because of the self-filtering nature of resonance, these frequencies are mostly limited to integer multiples of the lowest possible frequency, and such multiples form the harmonic series. F0 is the corresponding physical term and is defined for periodic or nearly periodic sounds only. For these classes of sounds, F0 is defined as the inverse of the period. In ambiguous situations, the period corresponding to the perceived pitch is chosen[7]. The term *multiple-F0 estimation* refers to the estimation of the F0s of several concurrent sounds. The term *musical meter* refers to the regular pattern of strong and weak beats in a piece of music. Metrical analysis, here also called *rhythmic parsing*, refers to the process of detecting moments of musical stress in an acoustical signal and filtering them so that the underlying periodicitys are discovered. The perceived periodicitys (*pulses*) at different time scales together constitute the meter. Metrical analysis at a certain time scale is taking place for example when a person taps his foot to music[7].

All the above elements make music transcription a difficult problem, which can be considered a search problem with no exact algorithm to automatically perform the music

*PhD Student
†Advisor

GECCO'07, July 7–11, 2007, London, England, United Kingdom.

transcription. This paper presents a new approach to the problem of music transcription based on Genetic Algorithms [5].

The rest of the paper is structured in the following way: Section 2 describes our proposal while Section 3 presents our experiments and results. Section 4 describes the next steps to be performed. Finally section 5 summarizes our conclusions.

2. TECHNICAL APPROACH

The current problem with the above methods is that they attempt to conceptualize a process that is still unknown. There is no standard way to extract a musical representation from an audio or acoustic signal. The current methods make strong attempts, but in the end, they are still trying to quantify a process that is not only computationally unknown, but also only seen in the real world among skilled musicians. Klapuri likens the process to "reverse-engineering the 'source code' of a music signal"[7]. This process may never be exactly understood.

However, the forward engineering process is known: going from a musical representation to an audio representation is what MIDI cards and synthesizers perform all the time. Therefore, given a sufficient instrument model, it is easy to render any set of notes, making sense in exploring this path. Moving away from finding the correct model for the complex signals to the real problem: discover who is playing which notes.

Our method uses an electronic synthesizer which combines a genetic algorithm [2] to generate, and then, render music performances into audio signals. Our fitness evaluator compares the audio signal of each individual with our target signal (the music we want to transcribe) and returns the result of the above comparison to the genetic algorithm making the individuals evolve. In the conclusion of Klapuri's summary of transcription[7], he stresses the need for both a method for analyzing the music and a means of parameter optimization. Our system fits those precise requirements: the fitness evaluator analyzes the music and the genetic algorithm returns improved hypotheses.

Our approach is beneficial because it is not limited by any particular harmonic model. A more traditional approach might find the highest intensity frequency and automatically assume that it is the fundamental frequency. Or, it might confuse two instruments that are playing the same note for one, since their harmonics overlap so much. As a matter of fact the polyphonic music creates a complex frequency lattice that is computationally infeasible to deconstruct, even for monophonic signals (as reviewed by Gomez et al.[3]) However, with our system, this lattice will not need to be deconstructed, it will rather be reconstructed. It is our claim that by mimicking the process with which the audio was originally constructed, the transcriptions produced can get much closer to an ideal transcription.

2.1 Tools required

2.1.1 MIDI files

Unlike digital audio files (.wav, aiff, etc.) or even compact discs or cassettes, a MIDI file [8] does not need to capture and store actual sounds. The MIDI file is just a list of events which describe the specific steps that a sound card or other playback device must take to generate certain sounds. Therefore, MIDI files are much smaller than digital audio files, and the events are also editable, allowing the music to be rearranged, edited and even composed interactively.

MIDI messages, along with timing information, can be collected and stored in a computer file system, in what is commonly called a MIDI file, or more formally, a Standard MIDI File (SMF). The SMF specification was developed by, and is maintained by, the MIDI Manufacturers Association (MMA). MIDI files are typically created using desktop/laptop computer-based sequencing software (or sometimes a hardware-based MIDI instrument or workstation) that organizes MIDI messages into one or more parallel "tracks" for independent recording and editing. In most but not all sequencers, each track is assigned to a specific MIDI channel and/or a specific General MIDI instrument patch.

With the introduction of the Downloadable Sounds (DLS) format [9], it is possible to combine MIDI files with standardizes samples of musical instruments, sound effects, or even dialogue, which are used to recreate an exact copy of the sound intended by the composer and, in this case, the music or song we want to transcribe.

2.1.2 Synthesizer

The rendering of MIDI file into an acoustic audio signal is made by means of electronic synthesis. The main purpose of a synthesizer is to simulate other instruments (piano, saxophone, strings, drums, etc) or to create new sounds. Unlike the other musical instruments, which are based in acoustics principals, the synthesizers are completely electronic instruments. The synthesis methods can be divided into three different groups:

- Pure synthesis (additive synthesis, subtractive synthesis, frequency modulation);
- Sample based (wavetable, sampling);
- Physical modeling.

The pure synthesis is also referred as *analog synthesis*, and tries to create sounds based in electronic elements, such as: oscillators, filters, mixers, etc. Amongst the main techniques of this kind, there are three we want to emphasize: additive synthesis, subtractive synthesis, and frequency modulation (FM).

The additive synthesis tries to create sounds by adding diverse simple components (mainly sinusoidal waves), until we get the desired harmonics. In the subtractive synthesis, the principle is the opposite. We have a wave full of harmonics (a square wave, a triangle wave, a saw tooth wave, etc.) and, by means of filtering, some harmonics are removed or attenuated.

In frequency modulation (FM) there are no sums nor subtractions, but some kind of multiplication. What is done is to modulate a certain wave with another wave, therefore, an oscillator is controlled by another oscillator. This technique can be used to create many harmonics.

The sample based techniques are, as it sounds, based in previously recorded sound samples. For example, to simulate the sound of a flute, a synthesizer contains recorded samples of some notes of the flute. When a note is played, the synthesizer choose the nearest sample, and if necessary, it changes the tonal height by applying a pitch-shifter or something similar. In the case of the duration of the note

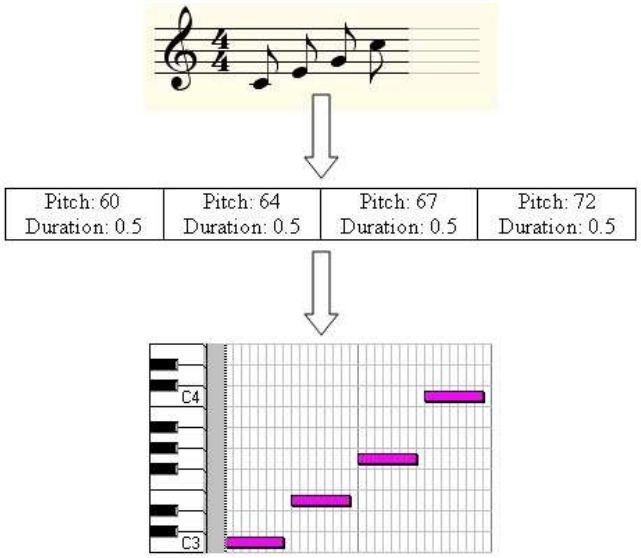

Figure 1: The top picture represents a simple music score with 4 quavers. The middle picture shows an individual encoding the score. The bottom picture represents the corresponding MIDI transcription.

being greater than the synthesizer's sample, the sample is repeated (or part of it), a couple of times, making what is commonly referred as *loops*.

Physical modeling stands on the production of the sounds of the instruments from their acoustical equations. A couple of scientists made a set of equations which define the sound produced by a certain instrument, and based on these equations, the synthesizer will produce the sound.

2.1.3 Fast Fourier Transform - FFT

A straight comparison of two acoustic signals which measures the difference between the samples would result in similar sounds being rejected because of minute differences in small factors like phase. Instead, we chose to work in the frequency domain. The power spectrum throws away phase so that differences in the phase of sine waves making up the signal are not considered. Therefore, in order to most effectively compare two sounds, a fast Fourier transform (also designated as STFT - Short Time Fourier Transform) is performed on each sound, allowing us to compare the magnitudes of all the frequencies.

In the next section, we will employ all the techniques described above, within the fitness function required by the Genetic Algorithm we employ.

3. EXPERIMENTS & RESULTS

Although genetic algorithms have been employed for signal processing [1], the nature of music transcription problem is different to a standard signal processing, as we have explained before. We describe below our approach to the problem and the results.

In order to address the problem of music transcription we decided to begin with a simpler approach: transcription of monophonic waves files and then, as a second step of our research, address the transcription of polyphonic acoustic signals.

Parameters	Values
Population Size	200
Maximum Generations	2000
Parent Selection	Tournament (tournament size = 5)
Recombination type	One Point Crossover
Prob. Mutation	0.01
Survivor Selection	Best 200 individuals (population size)

Table 1: Main parameters of the Genetic Algorithm

3.1 Transcription of Monophonic Wave files

A simple Genetic Algorithm was encoded using jMusic API[1]. This API appeared to be the ideal tool for this goal: Genetic Algorithm support, MIDI generation, it is a tool for instrument building as well as music making and it is possible to write Java applications using jMusic components. These components include a musical data structure with associated modification and translation classes as well as some graphical user interface elements.

In a simplistic overview of the Genetic Algorithm classes in jMusic, the Genetic Algorithm only works for Phrases[2], not any of the other music types like Parts[3] and Scores[4]. The central class of this framework is the PhrGeneticAlgorithm class.

Simply stated we created a standard genetic algorithm with basic operations: one point crossover, a mutation that changes the tone of a random note about a semitone and elitism. It does not have special operators regarding the concept of music transcription like: delaying a note, break a note into two consecutive notes, insert a silence between two notes, etc. Table 1 shows the the main parameters of this naive genetic algorithm.

3.1.1 Individuals encoding

Since our primary objective is to get a MIDI representation as the transcription of the audio file, our algorithm is something like evolving a set of MIDI-like individuals. Although a MIDI file consists of one header chunk[5] and one or more track chunks of bytes, we decided not to work on this level, but in an higher level, due to ease of use: it is easier to

[1]jMusic (see http://jmusic.ci.qut.edu.au/) is a library for compositional and audio processing tools. It has also native support for Genetic Algorithms and Cellular Automata.

[2]The Phrase class is representative of a single musical phrase. Phrases are held in Parts and can be played at any time based on their start times. They may be played sequentially or in parallel.

[3]The Part class is representative of a single instrumental part. A Part is made up of a number of Phrase objects, and Parts in turn are contained by Score objects which form the highest level in the jMusic data structure.

[4]The Score class is used to hold score data. Score data includes is primarily made up of a vector of Part objects. Commonly score data is algorithmically generated or read from a standard MIDI file, but can also be read and saved to file using Java's object serialization. In this way a Score's data can be saved in a more native context.

[5]A chunk is the smallest unit of content that is used independently and needs to be indexed individually. Each chunk has a type indicated by its chunk type name. Most types of chunks also include some data. The format and meaning of the data within the chunk are determined by the type name.

Parameters	Values
Number of seconds	6.75
Number of notes	23
Notes	see top part of figure 2 (tablature corresponding to the source wave file)

Table 2: Wave file for the genetic algorithm to transcribe

work with an array of notes rather than to work with many chunks of bytes.

Basically, the problem is similar to evolving an array of consecutive notes. Each note has a pitch and a duration. (see figure 1). In the case of a break (silence) the correspondent pitch is the minimum possible integer. Therefore, individuals are made up of a number of genes -notes-, each of them including two parameters, pitch and duration. Although the number of genes in the individual is variable, due to being tied to the duration of the notes, the parameter was fixed for each of the experiment performed.

The recombination operator is a basic one point crossover operator, where all data beyond the randomly selected point of cut is swapped between the two parents organisms, resulting in two new children. The mutation genetic operator, randomly shifts the pitch of a random note about {-1,0,1} semitone. The probability of both mutation and crossover are listed on Table1.

3.1.2 *Fitness function*

For monophonic transcription the fitness evaluator has to compare each individual with a pre-generated frequency time domain signal: the audio signal - WAV file - we wanted to transcribe. It was divided in time slots with size 4096 ($length = \frac{4096}{44100} \simeq 93ms$) and for each time slot a FFT was computed and the F0 extracted, thus generating a frequency time domain signal. Since each individual was a simple array of notes, the fitness evaluator calculated the difference, with a 5% tolerance, between the corresponding frequency of each note's pitch and the F0 for each corresponding time slot. The frequency of a pitch is given by the equation (1):

$$Frequency = 6.875 \times 2^{\frac{3+pitch}{12}} \quad (1)$$

To ensure there were no artifacts of unwanted frequencies, we used the Hann windowing function[4], which is stated on equation (2).

$$w(n) = 0.5\left(1 - cos\left(\frac{2\pi n}{N-1}\right)\right) \quad (2)$$

3.1.3 *Results of the monophonic transcription approach*

One experiment was performed employing a wave file with the following characteristics (see Table 2). The parameters of the genetic algorithm are stated in Table 1. The percentage of the correct transcription was about 86.9% (20 out of 23). We have noticed that, due to working with floating point numbers, there were errors in the transcription resulting from the time rounding. Sometimes, when we are working with floating point numbers, the result of the sum 0.25 + 0.25 is not 0.50 but 0.49(9) or 0.5000000000001. Hence

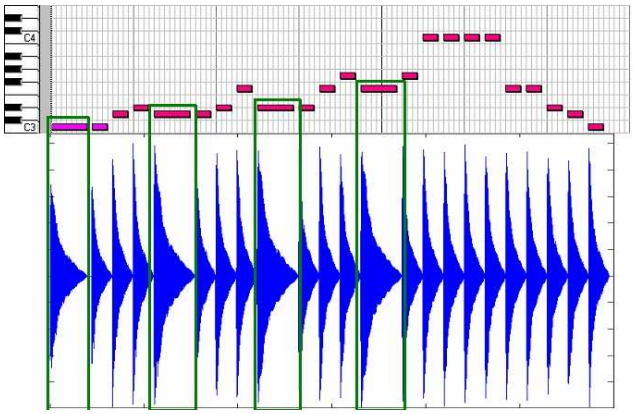

Figure 2: The audio source signal to transcribe and the corresponding tablature.

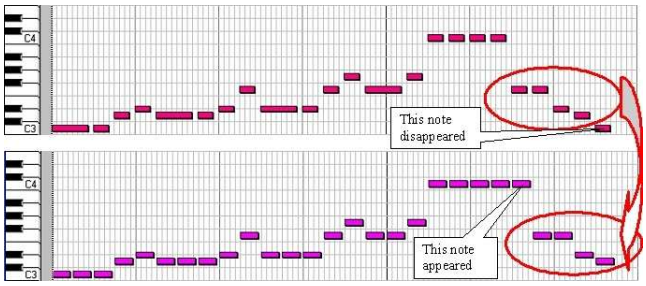

Figure 3: Transcription errors due to time rounding. The upper tablature corresponds to the music we want to transcribe and the lower tablature is the transcription generated by the naive genetic algorithm.

if we divide the result of the above sum by 0.25 the result sometimes can be greater or less than 2.00. Since there are not half time slots the result of the division must be an integer and should be always 2, but unfortunately sometimes it is 1. The figure 3 shows a transcription error due to time rounding: the upper tablature (red) corresponds to the wave file, and the bottom tablature is the transcribed version of our algorithm. This is the same for all figures. Instead of the wave file signal is shown is partiture for better understanding the process and what the algorithm must find.

The solution for this problem seamed to be very obvious: if we can't work with floating point numbers for the time, we will work only with integer numbers. The very first results of this implementation of the naive genetic algorithm with the new fitness function were impressive. The transcription of the given real audio songs was very accurate, sometimes resulting in a perfect match. And this was only the naive algorithm. Figure 4 shows that 100% of the notes were correctly transcribed. Our genetic algorithm could accurately transcribe those notes in about 29 generations and 2,7 seconds [6].

After solving the above question a new problem arises with tempo: the transcriptions are exact in pitch, and start of each notes, but not in the notes duration. We can see

[6]All tests were performed in a Pentium IV 3.0 GHz laptop computer with 512MB RAM, running FreeBSD 6.1-RELEASE operating system.

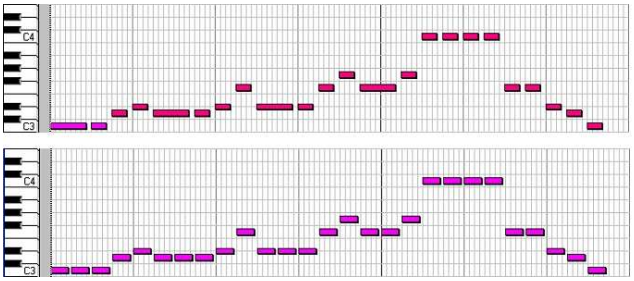

Figure 4: Working only with integer values fixed the problem of tempo

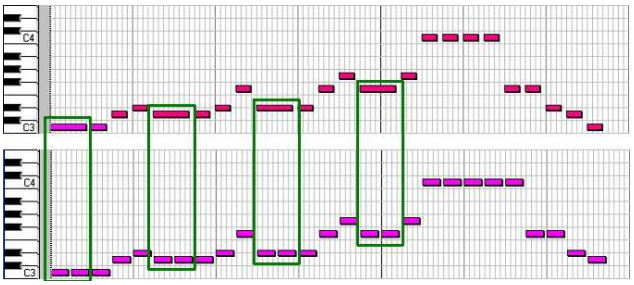

Figure 5: A note of duration 0.50 counts as two consecutive notes of duration 0.25

in the figure 5, a note with duration 0.50 results in a transcription of two consecutive notes with the same pitch as the original, each one with duration of 0.25.

Why does this happens? The main cause for this problem is the fitness function and how it is calculated. The fitness function is the sum of the difference of the frequencies for each time slot. E.g.: Lets say that our transcription has 1 note with a duration d and start a time t, hence the note is at time slots $t, t+1, t+2, (...), t+d-1$. The fitness function will look at the target audio file, and will see if the time slots $t, t+1, t+2, (...), t+d-1$ have the same frequency, corresponding to the note of the transcription. So, if there is an individual with one note with duration 2 and another individual with 2 consecutive notes with duration 1 and with same pitch as the note of the other individual, the fitness function cannot say which of them is the best because both have the right frequency at time slot 1 and at time slot 2. Dealing with notes with different durations leaded as well to some transcription errors (figure 6).

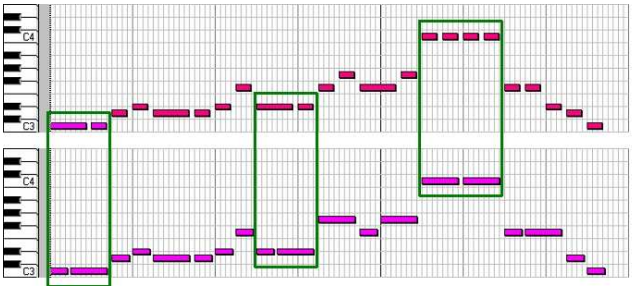

Figure 6: Transcription errors when dealing with notes with different durations.

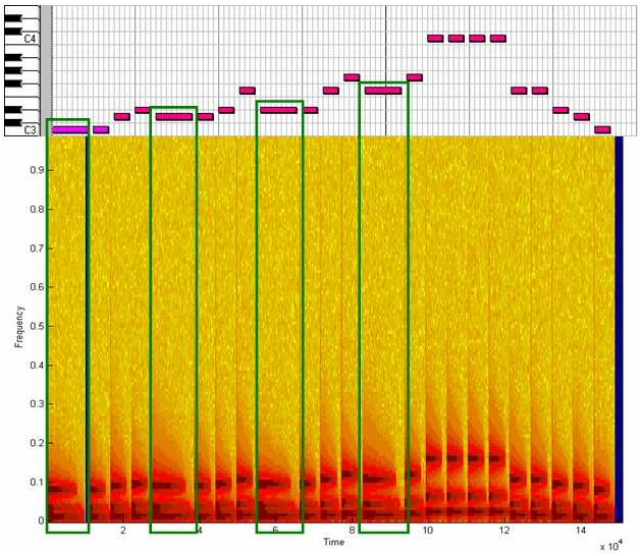

Figure 7: Using smaller window sizes it is possible to discover the duration of each note. The bottom figure is a spectrogram - frequency time domain signal - similar to the one who was processed to compute the fitness function, but with a smaller window size.

By creating a spectrogram from the audio source file, we stated that it is possible to discover whenever a note starts and where it ends, only by applying smaller time slots ($length = \frac{1024}{44100} \simeq 23.2ms$). Therefore, using a higher resolution it is possible to transcribe monophonic audio signals with a perfect match (see figure 7).

3.2 Transcription of Polyphonic Wave files

The next step to the transcription of monophonic wave files was the automatic transcription of acoustic signals. This stage is more complicated than the previous one. The individuals had to support polyphony as well as the fitness evaluator. This leaded to a complete rewriting of the genetic algorithm, since the jMusic genetic algorithm framework only works with Phrases, which do not support polyphony. We had to recode all the entire framework to work with CPhrases, which are a polyphonic version of Phrases.

3.2.1 Individuals encoding

To support polyphony the individuals had to be updated to the CPhrase structure. This structure is an array of Phrases, which is an array of consecutive notes. Each note has a pitch and duration. Figure 8 shows how individuals are encoded.

The recombination operator as also the mutation operator had to be rewritten to deal with the new individual encoding (polyphony). The recombination is still a basic one point crossover operator and the mutation operator, still randomly shifts the pitch of a random note about {-1,0,1} semitone. For future implementations we are considering to implement specific mutation operators regarding the music transcription, such as: break a note into two consecutive notes, insert a break (silence) between two notes, merge two notes, etc. to improve the robustness and speed of the genetic algorithm.

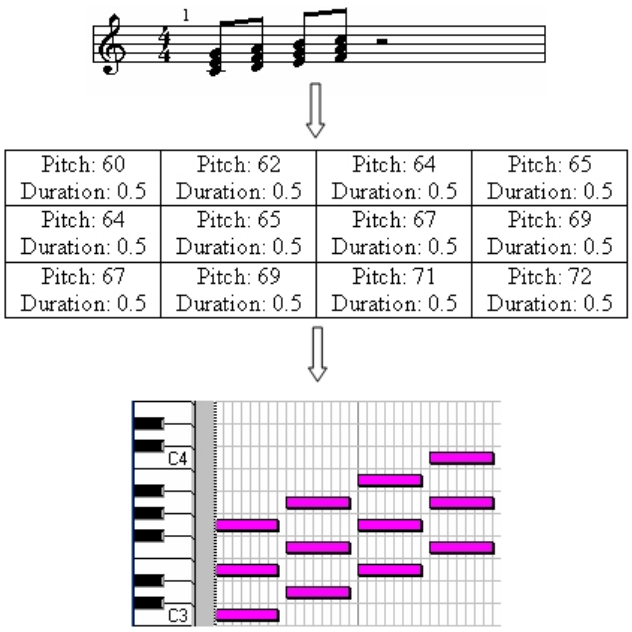

Figure 8: The top picture represents a polyphonic music score with 4 chords. The middle picture shows an individual encoding the score. The bottom picture represents the corresponding MIDI transcription.

3.2.2 Fitness Function

When we moved to transcription of polyphonic acoustic signals, the fitness evaluator had to be upgraded to support polyphony, and then compare the magnitudes of all frequencies.

In order to compare each MIDI-like individual with our target acoustic signal, it is necessary to render those individuals into audio signals as well. This is done by means of additive synthesis. We created an electronic synthesizer with the respective oscillator and envelope for this process. Therefore, our fitness evaluator renders each MIDI-like individual converting it in an audio signal and then computes it's fitness value by summing the difference between each frequency in each time slice of the song (see equation (3)).

$$Fitness = \sum_{t=0}^{tmax} \sum_{f=0}^{fmax} (O(t,f) - X(t,f))^2 \qquad (3)$$

The $O(t,f)$ is the magnitude of frequency f at time slot t in the acoustic audio signal, and $X(t,f)$ is the same for each individual. Fitness is computed from time slot 0 to $tmax$, traversing all time from the beginning to the end, and from $fmin = 0$ Hz to $fmax = 22050$ Hz, which is the nyquist frequency of 44100 Hz sample rate.

3.2.3 Results

The very first experiment employing a wave file with a C major chord (C4 + E4 + G4 + C5) and the genetic algorithm successfully transcribed those notes with a perfect match. The parameters used were the same as the previous, monophonic, version (see Table 1). These results were very enthusiastic, when compared to the three possible notes transcribed by prior implementations of automatic music transcription.

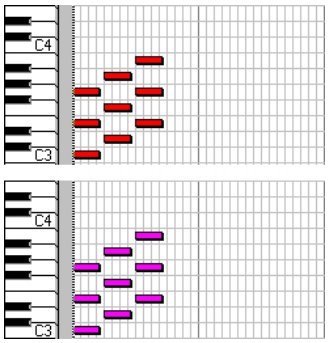

Figure 9: Transcription of 3 chords in C major scale.

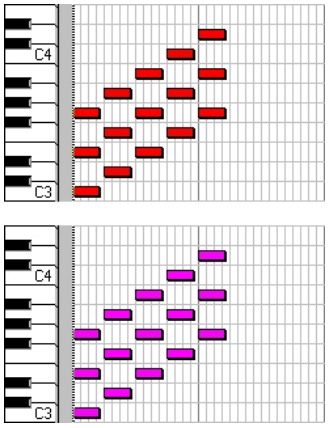

Figure 10: Transcription of 5 consecutive chords in C major scale.

Since we are working with audio data with frequency 44.1 KHz, this means that each individual, when rendered to audio data, will have 44100 samples per second. jMusic treats each music sample as a floating-point with single precision (32 bits) therefore, to deal with 3 seconds of music we will have something like 44110 $samples \times 3$ $seconds \times 32$ $bits =$ 4.233.600 for the size of each individual which are $\frac{4.233.600}{8} =$ 529.200 bytes. If we consider that our population has 200 individuals and, in each generation, 100 more individuals are generated from recombination, we have 158.760.000 bytes $\simeq$ 159Mb of memory for each generation (and we are only talking of 3 seconds of music). Since each music sample is treated like a floating-point rather than short, which as also 32 bits of precision, each operation takes several clock cycles which makes the genetic algorithm hard to compute, but still with excellent results: figure 9 shows a 100% accurate transcription of three chords in C major scale (the transcription processes was done after 6 generations and took about 6 minutes and 4 seconds).

One final test with five consecutive chords with three notes each was done. These chords were also in C major scale. To avoid the genetic algorithm to be stuck in local maxima, we had to tune the probability of the mutation to 0.1 (10%). The algorithm took 40 generations (64 minutes and 43 seconds) to transcribe those chords with 100% accuracy (figure

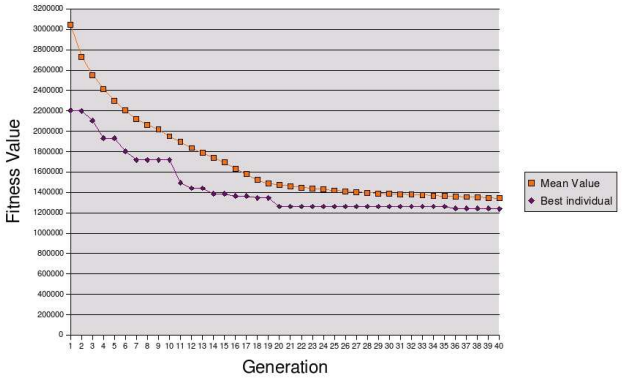

Figure 11: Evolution of the transcription of 5 consecutive chords.

10). Figure 11 shows the evolution of our genetic algorithm in order to transcribe those five chords.

4. NEXT STEPS OF OUR RESEARCH

As future work, we are looking forward to complete a series of steps: reimplementing the current algorithm focusing on better performance and less memory usage, extraction of the instrument's features (including harmonic structure and envelope), resolve the problem with the note's duration and finally, if necessary, the employment of parallel systems to increase even more the performance of our genetic algorithm.

4.1 Increasing Performance and Decreasing Memory Usage

The problem with performance and memory consuming resides in the evaluation of each individual. In order to evaluate an individual we have to convert it to an acoustic audio signal (via electronic synthesis) and process the FFT to compare with the FFT of the original audio signal. If we have 3 seconds of music, for instance, since we are working with audio data with frequency 44.1 KHz, the audio signal of each individual will have 44.100 samples per second, which are 132.300[7] samples. Since jMusic treats each music sample as a floating-point value with single precision (32 bits = 4 bytes), we will have something like 529,2 KB[8] of memory for each individual. If we consider that our population has 200 individuals and, in each generation, 100 more individuals are generated from recombination, we have 158.760.000 bytes $\simeq$ 159MB of memory for each generation (and we are only talking of 3 seconds of music).

It may seem obvious that there is no need of such memory quantity but the fact is that it is always used. When we are evaluating an individual we can generate the individual's audio signal, process the FFT to compare with the FFT of the original audio signal, store the result of the comparison as the fitness value and, finally, free the memory needed to generate the audio signal. Actually this is what our algorithm performs, but it still uses the same 159MB of memory, when dealing with 3 seconds of music. Why this happens? Because in Java it is the Garbage Collector who frees the memory, and it does not work properly: the algorithm starts getting slower and slower and the free space on disk (due to the swap) starts to get smaller and smaller. Sometimes the algorithm crashes because there are no more disk space. Even when we were forcing the Java's garbage collector with System.gc() to free the memory the problem did not disappeared.

The problem with performance also resides in the fact of each music sample being treated by jMusic like a 32 bit floating-point, even when we are dealing with audio data with only 16 bits of resolution, rather than working with integer types with 32 or 16 bits of precision. By working with floating-point types instead of integer types, each operation takes several clock cycles which makes the genetic algorithm hard to compute. Therefore, if we work with 16 bit integers we will have more performance (each operation will take less clock cycles) and the memory needed will decrease almost 50% (instead of 4 bytes for each sample, we will need only 2 bytes). Even the WAVE specification[6] sates that the audio data are integer samples and not floating point values, making sense to use the kind of values as well.

We can conclude that our algorithm has low performance and uses big quantities of memory because: memory is not freed properly, we are using 4 bytes per sample instead of using only two bytes per sample, we are working with floating-point types instead of integer types and, finally, the algorithm is implemented in Java, which is slow since it's not compiled, but interpreted. The main problem appears to reside in the Java language. If we recode our algorithm in the C++ language, we can have full control of memory management and assure that is only used the essential quantity memory, hence reducing the 159MB of memory to the size of only one individual, which is 529,2KB. The C++ code is compiled, instead of being interpreted like the Java language, therefore it makes sense to exploit this feature to increase even more the algorithm's performance using a specific compiler like the Intel compiler ICC[9]. Intel has also some performance libraries, such as: Intel Integrated Performance Primitives, Intel Math Kernel Library and Intel Threading Building Blocks. The Intel Integrated Performance Primitives[10] library has native support for signal processing, including features for: digital filtering, fourier transformation, windowing, emphasis, vector statistics, equalization and signal generation. Both fourier transform and signal generation can be used in our algorithm for FFT and electronic synthesis respectively. With IPP the content of signals can be treated as floating point, but also as integer types (increasing even more the algorithm's performance). Intel IPP libraries perform operations at a very fast rate and, with this library, FFT's can be initialized to reuse the same memory data, thus increasing the performance and reducing the quantity of memory needed.

In a few words: by rewriting our current algorithm in C++ and using IPP for sound processing, and ICC to com-

[7] 44.100 samples $\times$ 3 seconds = 132.300 samples

[8] 132.300 values $\times$ 4 Bytes = 529.200 Bytes

[9] Intel C++ Compiler

[10] Intel Integrated Performance Primitives (Intel IPP) is an extensive library of multi-core-ready, highly optimized software functions for multimedia and data processing applications, including: video decode/encode, audio decode/encode, image color conversion, computer vision, data compression, string recessing, signal processing, image processing, JPEG decode/encode, speech recognition, speech decode/encode, vector/matrix mathematics and cryptography.

pile the code, we can increase very significantly the performance of our algorithm.

4.2 Extracting Instrument's Features

In order to convert a MIDI-like individual into an acoustic audio signal we have to create a synthesizer for each instrument. The main purpose of a synthesizer is to simulate other instruments (piano, saxophone, strings, drums, etc) or to create new sounds. Those synthesizers could evolve as well to ensure the best individual will be the one who has the right note notes but also the individual who is playing the right instruments (has better synthesizers).

For the extraction of the instrument's features two approaches arise: the inclusion of the parameters of the synthesizer (harmonic structure and the envelope) in the individual genotype , hence making each individual to have his set of synthesizers or the use co-evolution to make the individuals and synthesizers evolve separately as two species: parasite and host. In the latter case, we can treat each instrument (synthesizer) as mathematical equation, using physical modeling synthesis which stands on the production of the sounds of the instruments from their acoustical equations. Those equations could evolve using genetic programming until they achieve the desired results.

4.3 Note's Duration

To solve the problem of the note's duration, we are planning to create a new mutation operator which merges two consecutive notes, if they have the same pitch. This mutation will create a new individual to ensure that the the best of both individuals (parent and child) will survive for the next generation.

4.4 Employment of Parallel Systems

In the case of our algorithm being slower than the other state-of-the-art algorithms, we are planning to employ parallel systems to increase the algorithms performance. The paralleling process may split a music into two or more different pieces and then send each peace to a different computational node to transcribe each part. We may try to find different patterns in music, split it according to those patterns (to ensure they remain intact) and then send each pattern to a different computational node. This will reduce the computation power and the reduce significantly the search space.

5. CONCLUSIONS

This paper has described the first attempt to perform Automatic Music Transcription by means of Genetic Algorithms and Electronic Synthesis.

Conducting a series of experiments, we have shown that genetic algorithms are perfect candidates for solving this problem. Genetic algorithms can cope with transcribing monophonic files and performs nicely with a series of polyphonic audio files employed for testing the new methodology.

Different encodings and fitness functions were developed for solving problems related to tempo, transcription of single notes, transcription of simultaneous notes, and more difficult: transcription of octave-related simultaneous notes.

Results demonstrated the success of the technique, and allowed confidence in trying harder problems, such as the transcription of polyphonic acoustic signals including different instruments and the possibility of extracting each instrument by evolving the synthesizer which renders each individual.

6. REFERENCES

[1] J. T. Alender. An indexed bibliography of genetic algorithms in signal and image processing. report 94-1-SIGNAL, University of Vaasa, Department of Information Technology and Production Economincs, 1995.

[2] D. E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, January 1989.

[3] E. Gómez, A. Klaupuri, and B. Meudic. Melody description and extraction in the context of music content processing. *Journal of New Music Research*, 32(1), 2003.

[4] F. J. Harris. On the use of windows for harmonic analysis with the discrete fourier transform. *Proceedings of the IEEE*, 66(1):51–83, 1978.

[5] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence*. The MIT Press, April 1992.

[6] IBM Corporation, Microsoft Corporation. *Multimedia Programming Interface and Data Specification*, August 1991.

[7] A. P. Klapuri. Automatic music transcription as we know it today. *Journal of New Music Research*, 33(3):269–282, 2004.

[8] MIDI Manufacturers Association. *The Complete MIDI 1.0 Detailed Specification*, September 1995.

[9] MIDI Manufacturers Association. *Downloadable Sounds - Level 1 Specification Version 1.1b*, September 2004.