Registration number 6144993

2015

Guitar tablature transcription for MIDI

Supervised by Dr Gavin Cawley

University of East Anglia
Faculty of Science
School of Computing Sciences

Abstract

In fretted string instruments such as the guitar there are multiple ways to play the same note, this creates a problem when playing conventional (staff) notation, as only the note and duration are shown, not how to play the note on the guitar. Tablature is a solution for this problem, the guitar neck can be represented as lines for the strings of the guitar and numbers on those lines representing the fret to be pressed. This method shows the player how to play each note specifically rather than what note to play. Constructing guitar tablature from known musical notes presents a difficult problem due to the restrictions on a player's hand and the instrument; the maximum number of voices that can be played on a guitar is the number of strings on that guitar (usually 6) and there are multiple different ways of playing the same note as the range of tones of guitar strings overlap. There is also a constraint on the player's hand as the player must use 4 fingers to play all the available notes, and their hand takes time to transition between different frets and strings of a guitar. This report introduces the problem of constructing a tablature output from MIDI files, and how to optimise this output using a genetic algorithm, then details the implementation of a program to do so.

Acknowledgements

I would like to thank Gavin Cawley for his continued support and enthusiasm on this project.

Contents

List of Figures

List of Tables

1. Introduction

The aim of this project is to create a piece of software to read in existing MIDI files and output a formatted sheet of guitar music as tablature. As there is a large amount of music Stored in MIDI files which have not been transcribed into tablature this will be a useful piece of software. There is room for expansion beyond basic transcription as the tablature can be optimised to be played as easily as possible. The project can also be expanded to allow for other fretted instruments such as the lute.

1.1. Tablature

Tablature is a method of writing music which indicates the position in which a note is played, rather than the pitch of that note. This is commonly used over conventional staff notation for fretted string instruments as it provides a more direct representation of what to play. For guitar tablature there are 6 horizontal lines, each of these lines represents a string, with the highest line representing the highest pitch (first) string and the lowest line representing the lowest pitch (sixth) string. Numbers on these lines represent what fret to play on that string, “0” represents playing the string without pressing at a fret (open string), “1” being pressing the string at the first fret, and so on. The note order runs from right to left with notes in the same horizontal position being played at the same time to form a chord. Notes can be separated by vertical lines (bar lines) which are used to segment music into groups based on timing, generally the time interval between two bar lines is constant.

The example tablature in Figure 1 is played as: the 10^{th} fret on the 4^{th} string, the 3^{rd} fret on the 4^{th} string, the 4^{th} fret on the 3^{rd} string, an open 1^{st} string, the 5^{th} and 7^{th} fret on the 5^{th} and 6^{th} string respectively, and finally the 7^{th} and 9^{th} fret on the 5^{th} and 6^{th} string respectively.

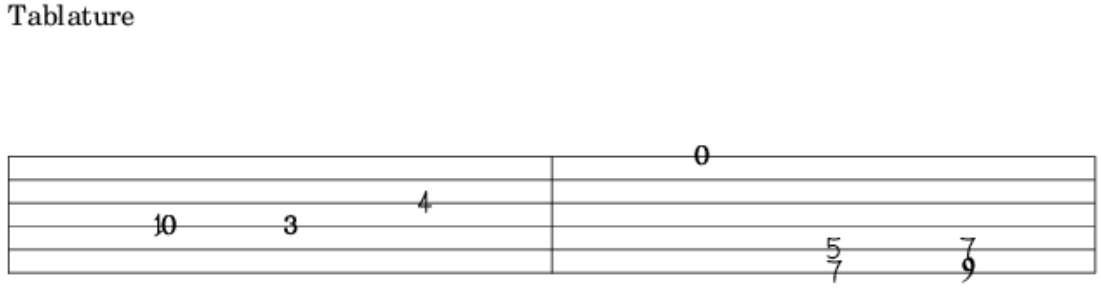


Figure 1: A simple tablature example

1.2. Main sections of the program

The main sections of this project are:

- Reading in a MIDI file and obtaining the relevant data
- Taking the note values and constructing playable tablature
- Formatting and outputting the constructed tablature

The intention of the program is to read in a MIDI file using the "javax.sound.midi" package which can then use *Sequence* and *MidiEvent* objects to get the data from the MIDI file.

The most complex section of the project will be in creating tablature from gathered MIDI information. There are multiple ways to play the same note on fretted instruments, so the initial aim is to construct tablature that is playable; for instance there cannot be two notes played on the same string at the same time and the player's hand cannot move instantly (or reach) from fret one to fret twenty, so the fret numbers must be fairly close to each other if the time between the notes is short or the notes are played at the same time. Not all guitars have the same amount of frets, and strings can be tuned differently which may cause problems for the transcriber.

Once the basic program functions, the tablature can be optimized so the player needs to move their fret hand position as little as possible and can therefore play the music as easily as possible. The problem of optimising the tablature will be a combinatorial

optimisation problem, as there are a set number of discrete ways to play any given note on a fretboard. The most basic way to output the created tablature is using the ASCII tab format, however this could be improved to be output in a more professional format by using other methods.

To test the software, a program that converts tablature into MIDI files will be used on existing tablature to create a MIDI file that can be converted into tablature again using the program. The accuracy of the new tablature when compared to the initial version used to create the MIDI file should give an approximation of how good the tablature is.

2. Related work and context

2.1. MIDI

MIDI stands for Musical Instrument Digital Interface and was developed in 1983 (MIDI Manufacturers Association et al., 1996). The MIDI messages sent describe events in a particular piece of music, they can specify a note being played or stopped, or details about the music such at tempo and title. All MIDI messages consist of up to three bytes, one status byte and one or two bytes containing the information of the message. These messages are split into two groups, ones that affect a particular route of communication (channel wide) or ones that affect the whole system (system wide).

Note on messages are a form of channel message and are the focus of this project. The status byte of a *note on* message is 9, the first data byte is an integer number corresponding to the note value and the second data byte is the velocity of the note. The value 60 in the first byte is normally the C4 note (middle C on a conventional keyboard) and any increment or decrement in the data value represent an increase or decrease of a semitone respectively; a semitone is the pitch difference between two adjacent notes. The velocity value represents how hard the note is played, the higher this value is the harder the note is played, and a value of zero is comparable to a *note off* message. As the system is only concerned with *note on* messages any note with a velocity of 0 can be discarded. These *note on* messages also have a timing value or *tick*, which represent the relative time of events, however do not have a fixed amount of time attached to them,

rather the amount of time that a tick corresponds to is relative to the tempo of the music. The music in MIDI files is split into different tracks, these are used to partition the data for better organisation when creating a MIDI file, for example tracks could be used to separate data for different instruments.

The "javax.sound.midi" package can be used to extract the information from a MIDI file in Java, it provides the tools to take in a MIDI sequence from a file, and obtain the tracks, events and individual messages from that file, identify the values they contain. This will allow the relevant information to be obtained with comparatively little computation.

2.2. Methods of determining tablature

When considering translating from a known input to guitar tablature there are multiple methods that can be used, the problem can be described as an optimisation problem, where the choice of what position on a fretboard is to be taken (or rather the transition between two positions) is assigned a cost and the total cost over the entire piece evaluates to a number which can be reduced or increased with different solutions to the problem.

2.2.1. A greedy solution

A greedy algorithm is a heuristic method to solve a problem, at each stage in calculating the solution the best possible next step is taken, "[a greedy algorithm] makes a locally optimal choice in the hope that this choice will lead to a globally optimum solution" (Cormen et al., 2009). As a solution to guitar tab optimisation a greedy algorithm would leave out the choice of the first played note, which may be chosen randomly, or a number of starting positions can be chosen and the optimum of those may be taken as the final result. This kind of algorithm may be comparatively efficient, but less effective in general than a more in depth solution as when a local maxima is found no other choices are considered, even if they would lead to a global optimum. For this reason, a greedy solution may be used in a early prototype, but would not be fit for the final solution.

2.2.2. Hidden Markov model

A hidden Markov model (HMM) is a statistical model for time-series data in which the input states are unknown (hidden) and the output is known. It is a tool for representing probability distribution over as set of observations. For any observation at a given time, O_t a probability distribution must be able to be calculated over it.

The defining properties of a HMM are:

- O_t is created by a process with a hidden state, S_t.
- Given the value S_{t-1} S_t is independent of all other states, meaning that the state at any given time is all the information needed to accurately predict the next state in the sequence.

$$P(S_{t+1}|S_0, S_1 ... S_t) = P(S_{t+1}|S_t)$$

- Given S_t, O_t is independent of all other states.

In the case of this project the unknown input states will be the string and fret played, the known output will be the note obtained from the MIDI file and the sequence of states will bet the tablature sequence. The method used to traverse the hidden Markov model would most likely be the Viterbi algorithm.

Viterbi algorithm

The Viterbi algorithm, proposed by Viterbi (1967), is a method for estimating the optimum path through a series of finite discreet states. It was initially designed for sequential decoding and can be used on hidden Markov models. The Viterbi algorithm looks at the current state and the previous one to decide what the most likely value for the current state is. Sayegh (1989) proposed using the Viterbi algorithm to solve the problem of fingering for string instruments. By weighting each transition between finger positions on the guitar the problem of computing tablature can be thought of a path finding problem on a directional graph, the optimal path through this graph can be the minimum or maximum path depending on the implementation of the weight.

2.2.3. Genetic algorithms

Developed in 1975 by John H. Holland (Holland, 1975) genetic algorithms are a heuristic optimisation method based upon the theory of evolution by natural selection by Darwin (1859).

Natural selection in biology

All the hereditary information of an individual is encoded in its genes, the genes are sections of DNA that code for functional molecules in the body. The genetic makeup of an individual is referred to as its genotype, and the observable characteristics that result from the genotype are called the organism's phenotype.

Darwin made five observations which were summarised by Mayr (1982):

1. A population would grow exponentially if all the population would reproduce successfully.
2. The population of a species remains relatively constant.
3. No two individuals are identical.
4. There are a limited number of resources that a population can access.
5. A large amount of variation in a species is passed on to children.

From these facts three inferences were made:

1. Because more children are produced than survive there must be a fight for survival.
2. The individuals that die or survive are not determined by random chance the more suited individuals survive (natural selection).
3. Over time a population will adapt to better suit its environment and eventually lead to speciation.

The probability of any individual surviving is based on its fitness and adaptation to the environment it is in. The more fit an individual is then the more chance of it reproducing and passing on its genetic information (genes) to the next generation. Through this method of natural selection each successive generation becomes fitter than the last as genes that are beneficial to the survival of the individual remain in the gene pool.

Creating an algorithm based on natural selection

A particular solution in the solution-space can be thought of as an individual, and the components of that solution are the individuals genes. A population of solutions is created and is evolved by merging and mutating solutions. The first step is to create an initial population, which is generally done by randomly generating solutions to the problem so the entire search-space can be covered.

Once an initial population has been created it is evolved using a selection process,two parent individuals are chosen based on their fitness as a solution. The genes from these solutions are taken and combined via crossover to create a new individual that will be in the next generation. This process is comparable to biological reproduction, each gene of a child is taken from one of the two parents at random. After the crossover is complete the solution can then be put through a mutation process where genes have a very low chance to be changed to a random different gene, this maintains diversity in the population while at a low rate, however if the probability of mutation is set too high the search could become too random. Once a child solution has been formed, another two individuals are selected to create a child. Depending on the implementation this can continue to create a new population of equal size which replaces the old population, or old individuals can be replaced as new ones are made. As long as the selection process is weighted towards more fit individuals the solutions in the population will tend towards becoming more optimal.

In a tournament selection a sub population is created containing a random selection of individuals from the main population (for simplicity two will be taken). The fittest of these two individuals are then selected to reproduce; and a second parent is chosen in the same way. On average each individual will be selected twice, the most fit individual will win both times, the median will win once and the least fit will never win. This will

result in a roughly linear weighting of chance to be selected to reproduce.

Generational vs steady state

As described above, there are two main methods of evolving the population of a genetic algorithm - generational and steady state (Noever and Baskaran, 1992). In a generational version of the algorithm the population evolves by selecting two parents, randomly combining them, and adding them to a temporary population. This process is repeated, adding the offspring to the new population until it is the desired size, then the new population replaces the old as the next generation.

In a steady state genetic algorithm two individuals are selected and combined to make two offspring (another method exists where only one offspring is created, however the function is the same), these two offspring then replace their parents. When the same amount of children have been made the population can be thought of as having moved to the next generation as in the steady state version.

In a generational algorithm, the previous generation is removed from the solution-space at one time, this means that no parent survives to the next generation, whereas in a steady state algorithm an individual may survive for many generations(even the entire course of the genetic algorithm), this can be due to its fitness or simple luck.

2.2.4. Hill-climbing search

As mentioned previously tablature transcription can be thought of as a graph traversal problem, which can be solved using a hill-climbing (or gradient descent) search. This is an iterative improvement algorithm where a solution is initially generated with little or no optimisation (a greedy or random solution could be taken, for example) and then each node can be swapped for another value at that node, and if this change is an improvement it is kept. This process is repeated until no more possible changes result in improvements.

There are two main issues with this method, the first is that a local maximum may be found, this is a solution where no other surrounding possibilities result in a positive effect on the solution, however the solution as a whole is not optimal. The second issue

is that a plateau may be found, this is where neighbouring solutions are of equal fitness and the hill climb would randomly move between solutions.

In both of these situations no improvement to the solution is being made, so the algorithm would halt prematurely One solution to this problem is to search from multiple starting points and take the optimum solution from the result of the collection of hill-climbs. This method would not completely solve the issue, however would increase the solution's effectiveness. Another method is to allow the hill climb to move to worse solutions in the hopes of escaping local maxima or plateau. The chance of a choice being selected decreases with the amount that choice would reduce the effectiveness of the solution, and the longer the algorithm has been running the less chance of moving to a worse solution there is. This process is called simulated annealing (Russell et al., 1995). Tabu search may also be implemented to prevent plateaus and local maxima, tabu search also allows less optimal solutions to be chosen (as in simulated annealing) however previously visited solutions are less likely to be revisited, as they are declared "tabu".

2.3. Output

2.3.1. The TAB program

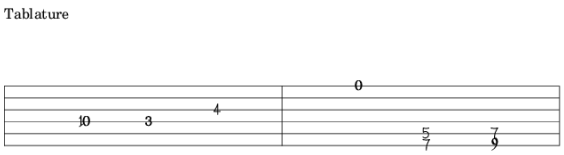
Wayne Cripps developed a command line program to output a postscript file in tablature format, it reads in a text file with the ".tab" extension in order to output the final tablature in multiple formats (Williams, 2006). The input file has a specific format, the file header specifies the title of the piece, the name of the output file and information that affects the entire file, such as whether notes should be in between lines or on the lines; or whether the file should include page numbers. The format of the lines to print out notes is that a line begins with a number corresponding to the length of a note or an "x" if no length is needed. The characters after this specify the values to go on each line, starting from the top and moving down; if multiple characters are needed on the same line (such as the number 10) the number must be immediately preceded by 'N'. By outputting a file in this format the TAB program can be used to generate a well formatted output postscript file to display the tablature.

An example TAB input file is given in Figure 2a and output in 2b. The header consists of the lines "{Tablature}" and "$line=o", these denote the title of the file and that the tablature is to be written on the string lines respectively. The "b" represents the initial bar line. The next three lines all represent separate notes, the "x" means that no time value is to be given for the note, the spaces (marked as "-") indicate the amount of lines without any writing, the "N" on the first of these states that the next two characters should be outputted on the next line, and the numbers are those to be written on the specified line. The new line represents a new event, and move the output across horizontally. The next line draws the bar line in the middle of the file, then another single note line with no time is given. For the following two lines, however, there are multiple notes on the same line, and as there is no "N" character they are written on separate lines of the tablature.

```
{Tablature}
$line=o

b
x---N10
x---3
x--4
b
x0
x----57
x----79
b
e
```

(a) Input file for the TAB program, spaces are replaced with "-" for clarity



(b) Tablature output from running the input file through the TAB program

Figure 2: An example input file and the generated output from the TAB program

2.3.2. Ultimate-guitar format

Websites such as ultimate guitar[1] use simple ASCII characters to display tablature, a mono-spaced font (one where every character has the same width) is used, with 6 lines of text representing the strings of the guitar. In each line dashes are used to represent space where a note is not played and the fret numbers are written as usual. It is possible

[1] www.ultimateguitar.com

to allow for multiple playing techniques such as bends and hammer-ons in this format, however for this program advanced techniques are not considered and only basic notes are displayed.

2.4. Segmentation

In many pieces of music the song is split into different parts (such as verse and chorus). These sections are comparatively simple to detect for a human, however present a complex problem for automatic recognition(Chai, 2006). It has been proven that the addition of segmentation to the determination of tablature can improve accuracy when generated tab is compared to a human expert transcriber (Radicioni et al., 2004). This is due to the fact that the transcription of each phrase is separate from that of the previous one, as having one large transition between two phrases can be made up for by ease of playing the phrases themselves.

2.5. Literature review conclusion

The MIDI standard has been overviewed, and the package "javax.sound.midi" has been briefly discussed and will be used in the final implementation to read in music to be transcribed. For the implementation of the algorithm to determine the tablature a greedy solution will initially be used to create a minimum viable solution, however a more advanced algorithm would be highly desirable. For this a genetic algorithm has been chosen due to the fact that fitness of a solution can be defined well, which allows for a genetic algorithm to be used. A heuristic method is required as the search space of the problem becomes very large with an increase in song length (or rather, number of notes). The genetic algorithm is also much less mathematically complex, making an implementation more understandable.

3. Genetic algorithms - an illustrative example

For this example an array of five bits will be considered, with the correct solution to the problem being the array $[1,1,1,1,1]$. For this example a cost, rather than fitness, will

be used. Fitness shows how well a given array fits the solution and the cost represents how "unfit" it is, the larger the cost, the worse the solution. Any solution's cost can be defined as the number of incorrect bits - for the correct solution this cost is 0, and for the array $[0,0,0,0,0]$ the cost will be 5.

A population size of 5 is taken, and an initial population is randomly generated:

1. $[0,0,1,1,0]$ - cost 3
2. $[0,0,1,0,0]$ - cost 4
3. $[0,1,1,0,1]$ - cost 2
4. $[0,0,0,1,1]$ - cost 3
5. $[1,0,0,0,1]$ - cost 3

For the tournament selection two individuals can be taken at random, say 2 and 3, then take the least costly, 3, and use that as a parent. Another two random individuals can then be taken, 1 and 4, and as they have the same cost either will do, for this example solution 1 is taken. Now two parents have been selected, genes can be randomly chosen from those parent individuals to create an offspring using:

$$c = [g1_3, g2_4, g3_3, g4_3, g5_4] \tag{1}$$

Where c is the child solution, g1...g5 are the genes and the subscript number is the number of the parent individual the gene will be taken from. Thusly

$$c = [0,0,1,0,1] \tag{2}$$

Each gene then has a chance of mutating, for this example the second gene will mutate and the bit will flip from "0" to "1" so that

$$c = [0,1,1,0,1] \tag{3}$$

Which is then added to a temporary population while four more solutions are found in this way. This new population will replace the old one, potentially looking something like this:

1. $[0,1,1,0,1]$ - cost 3
2. $[0,1,0,1,1]$ - cost 2
3. $[1,0,0,1,1]$ - cost 2
4. $[0,0,0,1,0]$ - cost 4
5. $[0,1,1,1,1]$ - cost 1

This new population is the second generation, new generations are continually made in the same fashion until the solution (an individual with a cost of 0) is found.

4. Design and scope

4.1. Overview

For the implementation of the program a rapid prototyping approach was taken, as the problem has had comparatively little research done into it the requirements found at the beginning of the project might change as new information was discovered. The idea being that the minimum testable solution could be programmed and then checked, once this was created it would be built upon and improved in order to achieve the next testable program. This cycle continues until a suitable product has been made.

4.2. Requirements

The requirements of the program were defined using the MoSCoW requirements prioritisation method (Clegg and Barker, 1994). This method separates the requirements of software into four categories, those that are required for the system to be complete are defined as “must”, items that are a priority but have alternate (less effective) solutions are defined as “should”, items that are perks that would benefit the system but are not critical are “could”, and items that could be included in the future but are not needed are labelled as “will not”

Must

- Read a MIDI file.
- Output readable tab.
- Output valid tab.

Should

- Implement a genetic algorithm to compute tablature.
- Write output in Wayne Cripps' format.

Could

- Implement a distributed genetic algorithm.
- Implement secondary optimisation (as opposed to shortest distance between frets).
- Implement a transposition method (alter pitch of the notes in the MIDI file).
- Implement timing and bar lines.
- Implement segmentation of the input music.

Will not

- Correct unplayable input.
- Implement a graphical user interface.
- Consider advanced techniques such as bends, palm muting and hammer on/off.
- Only select specific tracks - it is assumed all MIDI tracks are to be played on the guitar.

4.3. Structure

The Program is split into four packages, *geneticAlgorithm* which contains all of the components to calculate the tablature; *guitarItems* which contains the classes to deal with playable events and the fretboard model; main which contains the class with the main method of the program; and *notelist* which contains the classes to deal with reading, writing and storing the lists of notes for a file. The structure of the program and clases can be seen in UML diagram in Figure 3.

4.4. Playable Events

An interface, *Playable*, is defined to represent both chords and single notes from the MIDI file, this interface is used so that when creating a list of all the notes played, a list of *Playable* objects can be created, rather than having a list or array where multiple notes are at the same time, a note or chord object can be used instead. This allows for much easier iteration over note lists as each item in the list represents all of the notes played at that time.

4.4.1. Notes

The *Note* class inherits from *Playable* and accommodates the single notes read in (integer MIDI values) and/or notes for the output (integer string and fret values), this allows a single *NoteList* class to be used when the notes are read in from the MIDI file and for when the notes are assigned playing positions on the fretboard.

4.4.2. Chord

The Chord class contains an *ArrayList* of *Note* objects and contains methods for obtaining a list of all playable chords for a given input, and methods for validating a given chord. A chord must be checked to ensure it is valid before it can be used in the algorithm, it must be ensured that:

- There are not more notes to be played than strings on the instrument.

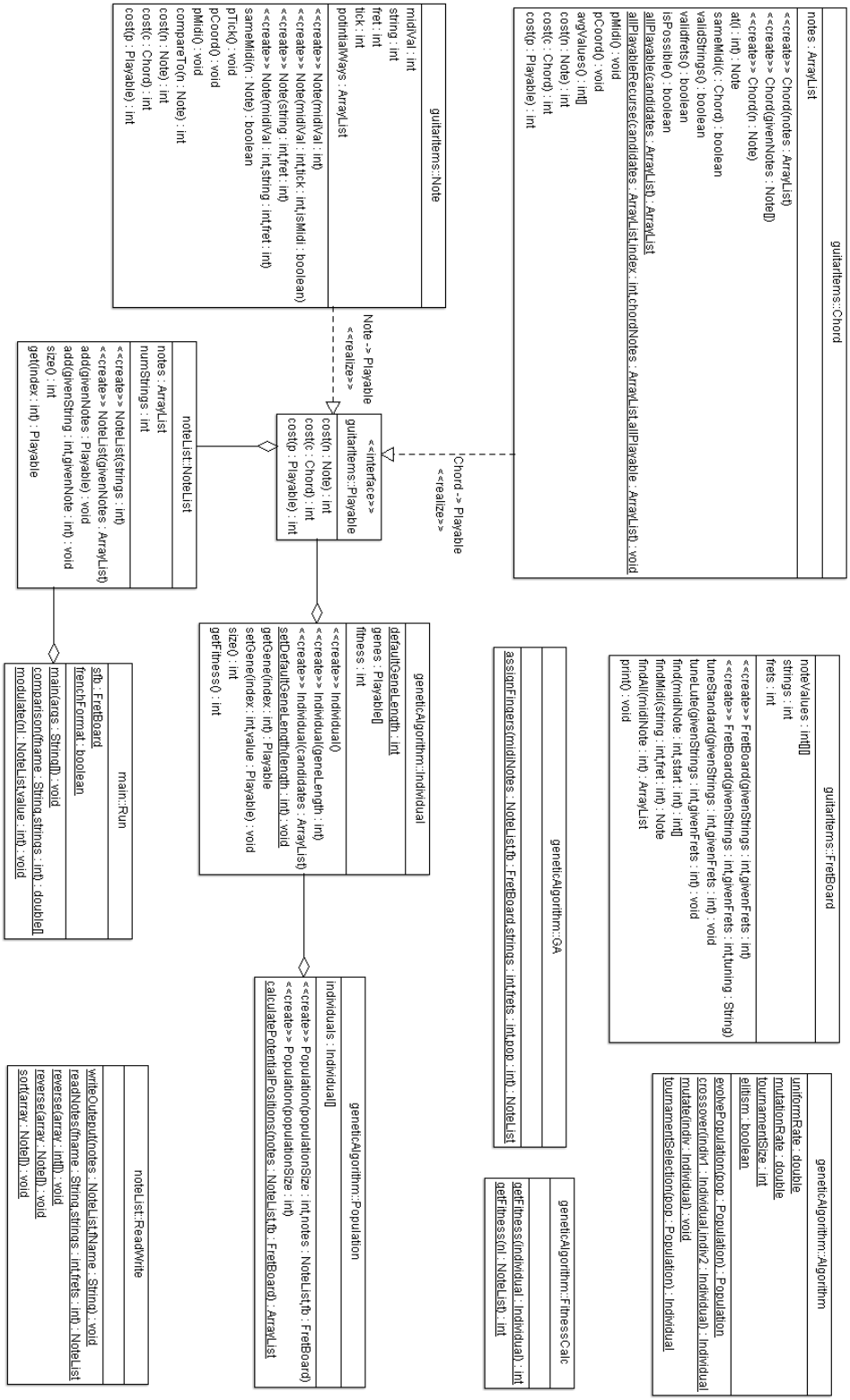


Figure 3: Class diagram for the implementation of the project

- No two notes are played on the same string.
- The hand must be able to reach from the lowest fret in the chord to the highest (a span of 5 frets has been assumed).

4.5. The genetic algorithm classes

The first class in the *geneticAlgorithm* package is *GA*, it contains the method to take in a *NoteList* object with MIDI note information and return one with finger position information.

The *Individual* class holds the genes for each individual in the population, these are stored as an array of *Playable* objects. In addition to this it provides methods to get and set gene values, find the number of genes and find the fitness of the individuals. *Population* holds an array of individuals and a method to determine the complete graph (represented as an *ArrayList* of *ArrayList*s of *Playable* objects) with all possible ways of playing each playable event.

The *Algorithm* and *FitnessCalc* classes contain static methods for running the genetic algorithm, these classes are never instantiated, they provide the algorithms to be used by the other classes.

4.6. Representing the guitar neck

A *FretBoard* object is created with the guitar fretboard represented as a 2D array, for example a 6-string 22-fret guitar would require a matrix of size 6 by 23 (22 frets and the open string), the x index of this array representing the fret and the y index representing the string. The value at these indexes is the integer value of the note when played at that position on the guitar. This method allows tuning to be changed easily as incrementing the fret by one increases the MIDI value by 1, so all that is needed for an entire string is its starting MIDI value, the rest can be filled in accordingly.

4.7. The representation of a piece of music

The class *NoteList* is defined to represent all of the notes in a piece of music in a *ArrayList* of *Playable* objects. The static methods from the *ReadWrite* class are used to take in the MIDI input, convert it to a *NoteList* and then output the found tablature after the genetic algorithm has been executed.

4.8. The *run* class

This class holds the main method for the system, it also contains the method *comparison* for testing the output against a known file used in Section 5. The method *modulate* is used to move all the midi values of a *NoteList* object by a constant value, this method could be used to alter the tuning of a piece to match different pitches, and change certain pieces to be played on instruments with different tunings.

4.9. Algorithms

4.9.1. Reading in notes

The method of reading in the notes uses the "javax.sound.midi" library to represent the data. The file is represented as a *MidiSystem*, and the *getSequence* method can be used to extract the sequence of MIDI tracks, these contain the note on events with the corresponding MIDI values that represent the notes being played.

A *note* object containing a MIDI value is created for every MIDI note on command in every track, then the notes are ordered in terms of the time as when reading in the notes initially they will be ordered first by track, then by time. Once the notes are ordered they can be looped through and if there are multiple notes at the same time a chord can be constructed from the notes and the playable object is added to a note list. As it is assumed that all the tracks are for the same instrument, no filter is required to avoid unnecessary tracks. See Algorithm 1 for pseudo code of this process.

Algorithm 1 pseudo code for reading in MIDI notes

```
NoteList readNotes = new NOTELIST()
NoteList playble = new NOTELIST()
for each track in the MIDI file do
    for each MIDI event in the track do
        if the event is a note on then
            unordered.ADD(new NOTE(midiValue))
        end if
    end for
end for
SORT(unordered)
NoteList output = new NOTELIST()
int prevTime = ordered.GET(0).tick, currentTime, numAtCurrentTime =0
Note[] notesAtCurrentTime = new Note[strings]
for each note in ordered do
    currentTime = note.tick
    if currentTime == prevTime then          ▷ add current note to the list of notes at this time
        notesAtCurrentTime[numAtCurrentTime] = note
        numAtCurrentTime++
    else                                     ▷ add the note or chord to the NoteList
        if numAtCurrentTime == 1 then
            orderdNoteList.ADD(notesAtCurrentTime[0])
        else
            orderdNoteList.ADD(new CHORD(notesAtCurrentTime))
        end if
        notesAtCurrentTime = new Note[strings]
        notesAtCurrentTime[0] = note     ▷ the current note is now the only note at this time
    end if
    prevTime = currentTime
end for                                      ▷ the last playable must be added
if numAtCurrentTime == 1 then
    orderdNoteList.ADD(notesAtCurrentTime[0])
else
    orderdNoteList.ADD(new CHORD(notesAtCurrentTime))
end if
```

4.9.2. Initial generation

For the initial generation of the genetic algorithm it is sufficient to allow the algorithm to select any finger position on the fretboard for each note, this position can then be checked to ensure that playing the assigned string and fret produced the correct note. This, however, would increase run time, and would require a very high cost to be given to incorrect notes, however they still may be selected (although in an unlikely case). To prevent this, the search space can first be restricted by calculating a list of all the potential ways of playing a given note, by only using these as options in the genetic algorithm the chance of playing wrong notes is completely removed. By using this method input can also be checked to ensure that notes have at least one way of being played on the fretboard, and run time can be reduced. Methods of restricting the search space are shown in Section 4.9.3.

4.9.3. Determining playable positions for given notes

Find all the potential ways of playing a given note

By using the matrix representation of the fretboard defined in Section 4.6 the potential ways of playing a note can be found by checking the MIDI value of the note against the values in the array. The indices of the matrix locations with the same value as the MIDI note will give the finger positions which will produce the correct pitch.

A recursive method to find all possible ways of playing a chord

Initially a chord is comprised of a list of *Note* objects with MIDI values, for each of these notes the matrix method defined above can be used to find a list of the valid ways of playing each note in the chord. This will result in a list of lists where the items in the outer list represent each note, and the items in the inner lists represent the ways to play that note. Once this nested list has been acquired all of the combinations that will produce a valid, playable chord must be obtained (see section 4.4.2). A recursive method with the following inputs has been defined for this purpose.

- *candidates* - the nested list.

Algorithm 2 pseudo code for finding every possible way to play a given note

```
potentialWays = new ARRAYLIST<PLAYABLE>()
for i= 1:numStrings do
    for j= 1:numFrets do
        if fretBoard[i][j] = noteMidi then
            potentialWays.ADD(new NOTE(i,j))
        end if
    end for
end for
if No Notes Found then
    display warning
end if
return potentialWays
```

- *index* - the index of the note it is looking at.
- *chordNotes* - the list of notes currently being used to form a chord.
- *allPlayable* - the list of playable combinations.

Initially the index is 0 (looking at the first note) and the lists of notes and playable combinations are empty. The nested list remains constant throughout the algorithm. For each way of playing the note at the current index the algorithm will add that note to the *chordNotes* list, and if there are more notes the method will recurse with the index incremented by one. If there are no more notes to add to the list a chord is created from the *chordNotes* list and its validity is checked. if the chord is valid it is added to the *allPlayable* list. After recursing or creating a chord the last added note is removed so it can be replaced with a new note, allowing for a new combination to be tested. See Algorithm 3 for the pseudo code for this algorithm.

4.9.4. Cost

The cost function is possibly the most problem-specific element of a genetic algorithm, it is the method used to determine how effective a solution is. In this algorithm the cost

Algorithm 3 pseudo code for a recursive method to determine all valid ways of playing a chord

```
function ALLPLAYABLERECURSE(ArrayList<ArrayList<Playable» candidates, int index,
ArrayList<Note> chordNotes, ArrayList<Playable> allPlayable)
    current = candidates.get(index)
    for i = 0:current.SIZE( )do
        chordNotes.ADD(current.GET(i))
        if index < candidates.size()-1 then
            ALLPLAYABLERECURSE(candidates, index+1, chordNotes, allPlayable)
        else
            Chord c = new CHORD(chordNotes)
            if c.ISPLAYABLE( )then
                allPlayable.add(c)
            end if
        end if
        CHORDNOTES.REMOVE(chordNotes.size()-1)
    end for
end function
```

of each note is calculated by the size of the transition between the current note and the previous note. A high transition cost means that the hand must move a lot to reach the next position, so the higher the cost the less desirable the solution.

Cost between two notes

The cost (fitness) function implemented between two notes calculates the sum of the difference in each coordinate. This method works for transitions between two notes, but for transitioning between a note and a chord (or two chords) other methods must be used.

Cost between chords

The first implementation of a cost function uses the lowest note in the chord (or finds both lowest notes if the transition is between two chords), once the note(s) have been found the transition can be treated as if it were the cost of transitioning between two notes. This also can be applied to the highest note. The pseudo code for this can be found in Algorithm 4. A second implemented method of determining the cost between chords is to take the average of each of the coordinates, then round each value to the nearest integer. This method gives a better representation of the position of the entire hand, rather than just one finger.

4.9.5. Crossover and mutate

The crossover algorithm takes in two parent individuals, it loops from 0 to the number of genes in the individual and randomly selects one parent to take a gene from to insert into the child solution. Pseudo code for this method can be seen in Algorithm 5.

To mutate an individual solution a random number is generated for each of its genes. If it is lower than a set threshold then a random valid replacement is generated using the methods defined in Section 4.9.3.

Algorithm 4 pseudo code for the cost function using the lowest note in a chord

```
function COST(NoteList notes)
    cost = 0
    for 1:notes.SIZE( )do                          ▷ no cost for the first note
        Playable p = notes.GET(i)
        current = new NOTE()
        prev = new NOTE()
        if p instanceof Chord then c = (Chord)p
            current = c.GETLOWEST()
        else
            current = (Note) p
        end if
        Playable pr = notes.GET(i-1)
        if pr instanceof Chord then c = (Chord)pr
            prev = c.GETLOWEST()
        else
            prev = (Note) pr
        end if
        cost += (ABSOLUTE(current.stringValue-prev.stringValue))
        cost += (ABSOLUTE(current.fretValue-prev.fretValue))
    end for
    return cost
end function
```

Algorithm 5 pseudo code for crossing over two parent individuals to make a child

```
function CROSSOVER(Individual indiv1, Individual indiv2)
    child = new INDIVIDUAL(indiv1.geneLength)
    for i = 0:child.geneLength do
        rndm = RANDOM()
        if rndm > 0.5 then
            child.SETGENE(i, indiv1.GETGENE(i))
        else
            child.SETGENE(i, indiv2.GETGENE(i))
        end if
    end for
    return child
end function
```

Algorithm 6 pseudo code for mutating genes

```
function MUTATE(Individual indiv)
    rndm = RANDOM()
    for i = 0:indiv.GENELENGTH() do
        if rndm < mutationRate then
            indiv.genes[i].GENERATENEWGENE()
        end if
    end for
end function
```

4.9.6. Tournament selection

For selecting a fit individual whose information will be passed down to the next generation a tournament selection method was implemented, a smaller population was created with a random selection of the individuals in the main population, the fittest of these can then be selected. A larger sample than two individuals is taken to find parents, this results in a non linear selection process - *i.e.* there is more chance to select fitter individuals (Harvey, 2011).

Algorithm 7 pseudo code for tournament selection of parents

```
function TOURNAMETSELECTION(Population pop)
    tournament = new POPULATION(tournamentSize)
    for i = 0 : tournamentSize do
        randomId = (int) ( RANDOM() * pop.SIZE())
        tournament.SAVEINDIVIDUAL(i, pop.GETINDIVIDUAL(randomId))
    end for
    return tournament.GETFITTEST()
end function
```

4.9.7. Evolving the population

After randomly generating an initial population it must be evolved using the previously defined methods. To do this a new, empty population of the same size as the old one is created, the most fit individual is saved as an identical copy into the new generation, this prevents the algorithm regressing to a worse state. A tournament selection process (see Section 4.9.6) is held twice to find two fit individuals to use as parents, these are combined using the crossover method (Section 4.9.5) and mutated using the mutate method (Section 4.9.5) before being added to the new population. Once the new population has been filled, the old generation is replaced, and the process is looped until an optimal solution is found. A single optimal solution cannot be guaranteed so instead the process is continued until improvements have not been made for some successive generations. This creates a heuristic solution where the final solution is good, whilst not necessarily being the global optimum.

Algorithm 8 pseudo code for evolving a population of individuals

```
function EVOLVEPOPULATION(Population pop)
    while population is becoming more fit do
        newPop = new POPULATION(pop.SIZE())
        newPopulation.SAVEINDIVIDUAL(0, pop.GETFITTEST())
        for i = 1 to populationSize do
            parent1 = tournametselection(pop)
            parent2 = tournametselection(pop)
            newIndividual = crossover(parent1, parent2)
            mutate(newIndividual)
            newPopulation.add(newIndividual)
        end for
        Replace the old population with the new one
    end while
end function
```

4.9.8. Output

To write the output to a .tab file a method was created that took in a *NoteList* object and a *String* for the file name, the method used the *java.io* library to write the desired output to a file. First the header is written, this states the output should be written on the lines of the output and not in between, it also states the title to be written. The starting bar line of the piece is written to the file then every playable instance of the *NoteList* is looped through and an 'x' is written - this specifies no timing is to be given to the playable instance. If there is only one note at the given time, spaces are outputted to move the note onto the correct line and the fret number is outputted (with a leading 'N' if the fret number is greater than 9). If the playable instance is a chord then a similar process is used, however it is looped for each note and the amount of spaces needed is calculated using the difference between the note and the previous note in the chord, or the top of the fretboard if it is the first note in the chord. If a sufficient amount of notes have been printed a bar line is output and a new line begins with the starting bar line. Once all the notes have been written to the file a closing bar line is printed and the end symbol outputted and the method is complete.

Algorithm 9 pseudo code for writing the output in Wayne Cripps format

```
write header
for each playable instance do
    output "x"
    if playable is a chord then
        if chord is not valid then
            output a message and skip the chord
        end if
        create an array from the notes in the chord
        for each note in the array do
            calculate the string difference between the two notes (or the first string)
            for i = 0: string difference do
                output " "
            end for
            if the note's fret > 9 then
                output "N"
            end if
            output the note's fret
        end for
    else
        for i = 0: the note's string do
            output " "
            if the note's fret > 9 then
                output "N"
            end if
            output the note's fret
        end for
    end if
    output "\n"
    if the line has enough notes then
        output "b\n\nb\n"
    end if
end for
output "b\ne\n"
```

5. Analysis and results

A series of experiments were carried out, for these experiments, three files from Wayne Cripps' library[2] were used: *La Villanella Balletto* by Vincenzo Capirola, *Pavana Alla Venetiana* by Johan Ambrosio Dalza and *Volte de Provence* by Adrian Le Roy. A ground truth test using these three pieces was employed, this test employs a certain standard as a benchmark for results to be tested against. For this test the tablature for the pieces listed above are the benchmark that the program output will be tested against.

5.1. Comparison of the generated output to that of known tablature

For these experiments tab files transcribed by a human are converted into a MIDI file using Wayne Cripps' typesetting program, the generated file is then run through the genetic algorithm to generate a new tab file. These two files can be compared and the difference measured, this provides a comparison as to how close the algorithm is to a human transcriber. The population size used in the algorithm was made to be proportional to the number of playable instances in each file, this was done because a fixed population size could cover a large amount of the search space for one file and a small proportion of another. A constant tournament size, mutation rate and crossover probability was also taken.

5.1.1. Comparison of average chord and lowest note

For comparing the methods of calculating the cost of transitioning between chords a population size of 1000% of the number of playable instances was taken and no other methods or variables were changed. The average was taken over 100 runs to increase the reliability.

Table 1 shows that the average number of notes found the same as the human transcriber is worst when the cost is determined from the lowest pitch note, and highest for the method using the average position of the chord. Using the lowest note in the chord would have the tendency to push chords to the high end of the fretboard in order

[2] http://www.cs.dartmouth.edu/ wbc/tab-serv/tab-serv.cgi

to make the lowest note close to the neighbouring notes. Conversely using the highest note to represent the chord the chords tend to take the lowest possible position. Using the average of the note positions in the chord to calculate cost all notes in the chord tend toward the position of the last note, this can result in the chord being played mainly on adjacent strings centred around the last note played. Using the lowest note to represent the chord resulting in the least accuracy may be due to the fact that chords in these pieces are more likely to be played lower on the fretboard, this would also be supported by the fact representing the chord as its highest note resulted in improved accuracy. The average note position of a chord better represents the overall position of the hand when playing a chord, rather than the position of one finger, this may be the reason it was the most effective implemented method of determining cost between chords.

L.V.B			P.A.V.			V.D.P		
Lowest note	Highest Note	Average	Lowest note	Highest Note	Average	Lowest note	Highest Note	Average
29.48	44.03	58.29	8.44	42.75	68.00	13.29	36.98	75.01

Table 1: Percentage of playable instances accurately transcribed in all three pieces with the lowest note and average cost functions

5.1.2. Comparison of performance with population size

For this experiment, values were recorded over differing population sizes as a percentage of the number of notes in the piece. The algorithm was run 100 times with the average chord cost being used and the averages of the following values were recorded:

- Outputted notes or chords the same as the original tablature.
- The percentage correct.
- The cost of the most fit solution.
- Time taken to run the genetic algorithm.

As shown in Figure 4 the percentage of notes identified as the same as the original tab greatly increases until a population of approximately 200% - 300% of the number of notes (or chords) played, then plateaus for all pieces. This may be due to a few factors:

1. There is no chord optimisation, the difficulty of playing a given chord is not accounted for in the cost function.

2. Tablature transcribed by the expert does not always attempt to make the least possible movement along the neck, in some pieces the lower end of the neck is preferred.

3. As the cost reaches an optimum little to no improvement can be made.

The cost (Figure 5) follows a similar pattern of decreasing (becoming better) until a population of approximately 200% - 300% of the playable events, the convergence of the cost to a similar number indicates an optimal solution has been found. This would also help explain why the percentage correct plateaus, as the different methods of tablature transcription implemented by humans have not been included in this algorithm.

The time taken to run the genetic algorithm can be seen if Figure 6 and is roughly linearly proportional to the population size, it is also proportional to the number of notes (genes) in the piece, with *Pavana Alla Venetiana* having the most notes and *Volte de Provence* having the least. This is due to a runtime complexity of $O(P)$, where P is the population size.

6. Conclusion

The reading in and outputting of notes defined in this solution both work well for their function, the optimisation of reading in notes is acceptable as $O(N)$ run time is the minimum as every note needs to be covered once. The tablature format is readable so it fits the purpose, however with out timing or segmentation, note spacing can become an issue.

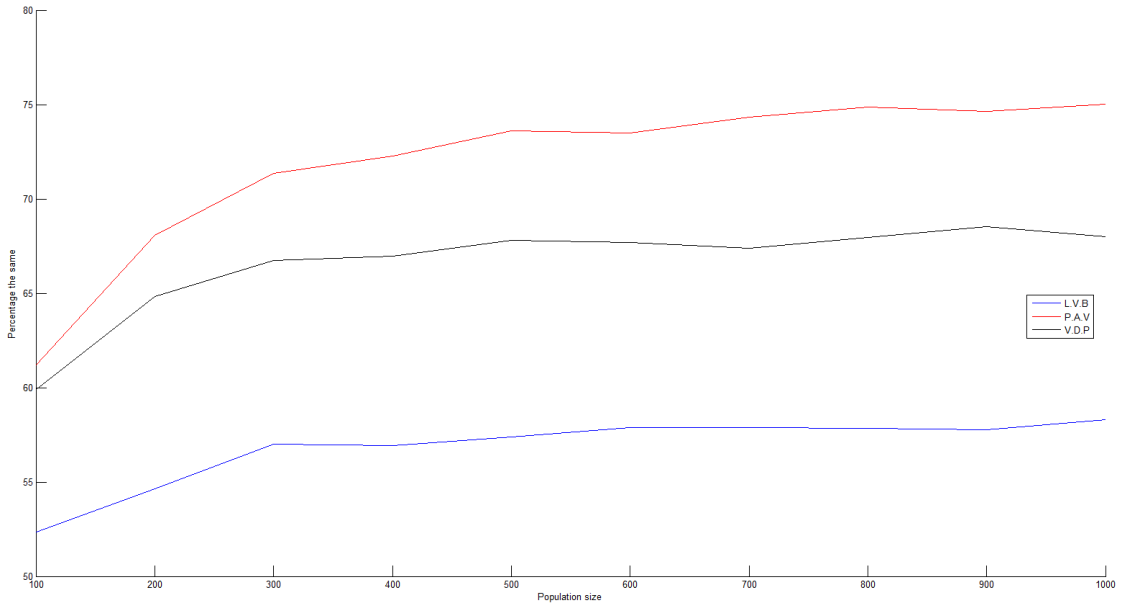


Figure 4: Number of notes identified the same as a human expert against population size

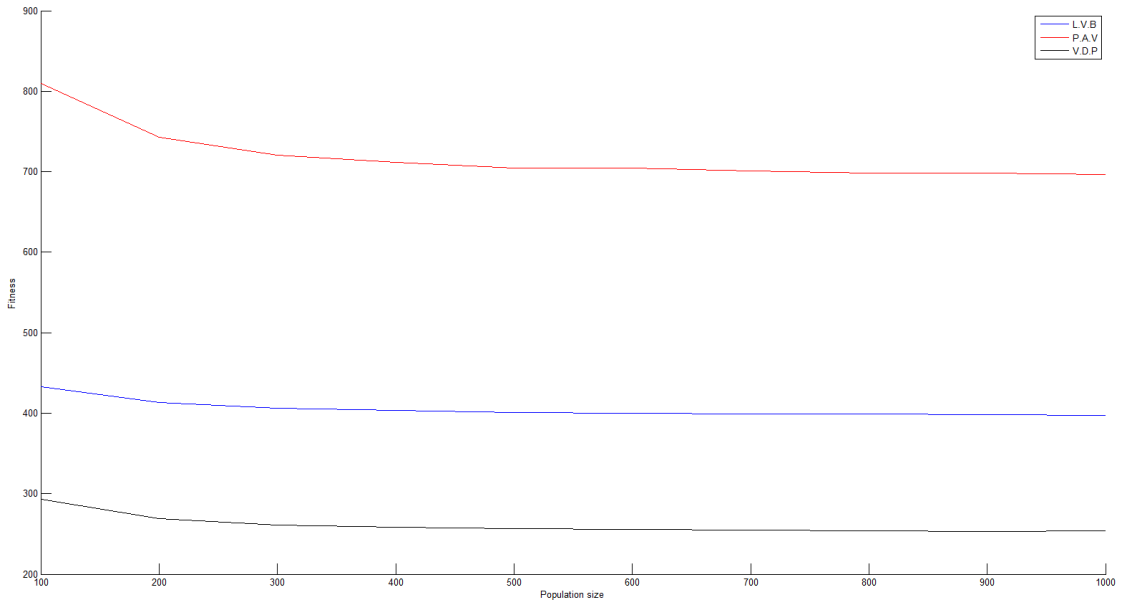


Figure 5: Cost against population size

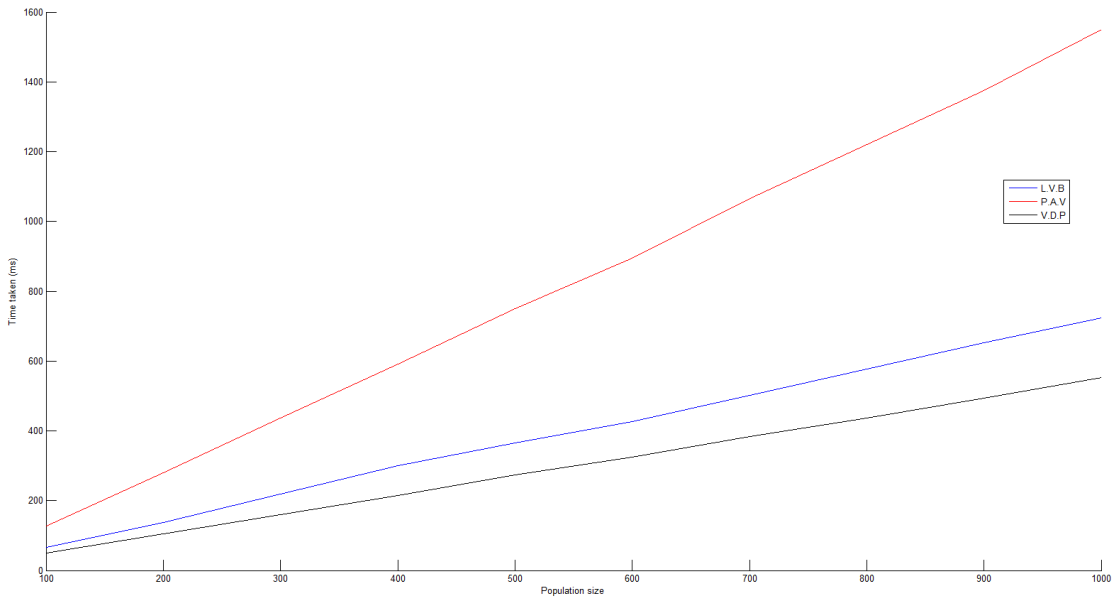


Figure 6: Time taken to run the genetic algorithm against population size

The algorithm for constructing tablature defined in this project has had an acceptable performance in the ground truth tests, however further tests on the algorithms performance with longer pieces of music, and a wider variety of styles of music would be desirable. There would be additional factors to be considered when these tests are being performed as different guitar styles have tendencies to use different parts of the fretboard, for example modern rock chords tend to be played on the lower strings of the guitar, whereas lute music is more likely to be on the low end of the fretboard.

6.1. Further work

For the reading in of notes the only major improvement to this method would be the calculation of note duration when notes are read in. For the output of tablature, other formats could be considered, however these are not strictly needed because the output is readable enough. The output could be improved with the addition of timing to the notes to allow for bar lines, and with the incorporation of segmentation to better break up the tablature.

The genetic algorithm outlined could be improved by implementing a distributed ge-

netic algorithm where multiple separate populations are evolved simultaneously, this allows for greater genetic diversity between solutions, and a solution closer to a global optimum becomes more likely. Implementing this solution would create a better genetic algorithm, however, it may not significantly improve results when comparing the output to human created tablature because the fitness tended to plateau after a certain population size. The solution could be improved with the implementation of other fitness criteria, potential fitness criteria that may improve the algorithm are:

- Cost for chords based on a chord's difficulty to play.
- Using a system where note and chord positions are weighted so that the low end of the fretboard may be prioritised.
- Notes that are played on an open string require no hand movement as the fingers only need to be lifted off the strings rather than moved to the fret location.
- The difficulty of transitioning between two positions on the fretboard can be reduced if there is enough time to move between fret location.
- Some different methods to determine note to note cost such as taking the position of the left hand and increasing cost based on whether the note is reachable from that position.

Segmentation could also be implemented as this has been shown to improve performance in other work Radicioni et al. (2004). This was deemed too time consuming to fit into the current design, but would increase the performance of the algorithm, especially in long pieces of music as music is commonly divided this way.

References

Chai, W. (2006). Semantic segmentation and summarization of music: methods based on tonality and recurrent structure. *Signal Processing Magazine, IEEE*, 23(2):124–132.

Clegg, D. and Barker, R. (1994). *Case method fast-track: a RAD approach*. Addison-Wesley Longman Publishing Co., Inc.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT press.

Darwin, C. (1859). *On the origin of species by means of natural selection or the preservation of favoured races in the struggle for life*. John Murray.

Harvey, I. (2011). The microbial genetic algorithm. In *Advances in artificial life. Darwin Meets von Neumann*, pages 126–133. Springer.

Holland, J. H. (1975). *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* U Michigan Press.

Mayr, E. (1982). The growth of biological thought. *Diversity, evolution, and inheritance*.

MIDI Manufacturers Association et al. (1996). *The complete MIDI 1.0 detailed specification: incorporating all recommended practices*. MIDI Manufacturers Association.

Noever, D. and Baskaran, S. (1992). Steady state vs. generational genetic algorithms: A comparison of time complexity and convergence properties. *Preprint series*, pages 92–07.

Radicioni, D., Anselma, L., and Lombardo, V. (2004). A segmentation-based prototype to compute string instruments fingering. In *Proceedings of the Conference on Interdisciplinary Musicology*, volume 17, page 97. Citeseer.

Russell, S., Norvig, P., and Intelligence, A. (1995). A modern approach. *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs*, 25.

Sayegh, S. I. (1989). Fingering for string instruments with the optimum path paradigm. *Computer Music Journal*, pages 76–84.

Viterbi, A. J. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *Information Theory, IEEE Transactions on*, 13(2):260–269.

Williams, L. (2006). Tab a guide to wayne cripps' tablature typesetting program.

Appendix A Example output

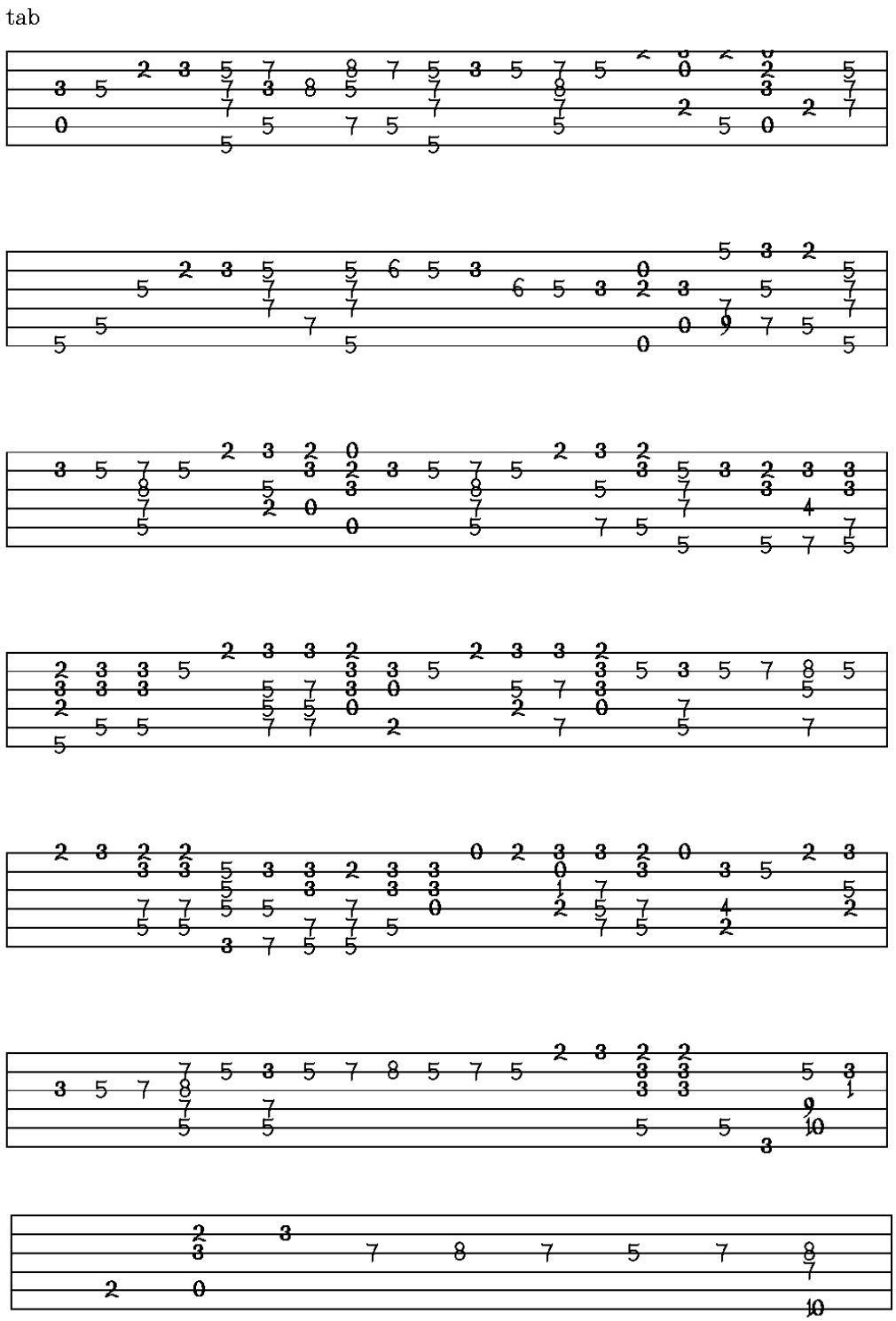


Figure 7: *Volte de Provence* formatted output at a population of 1000% the number of notes in the piece

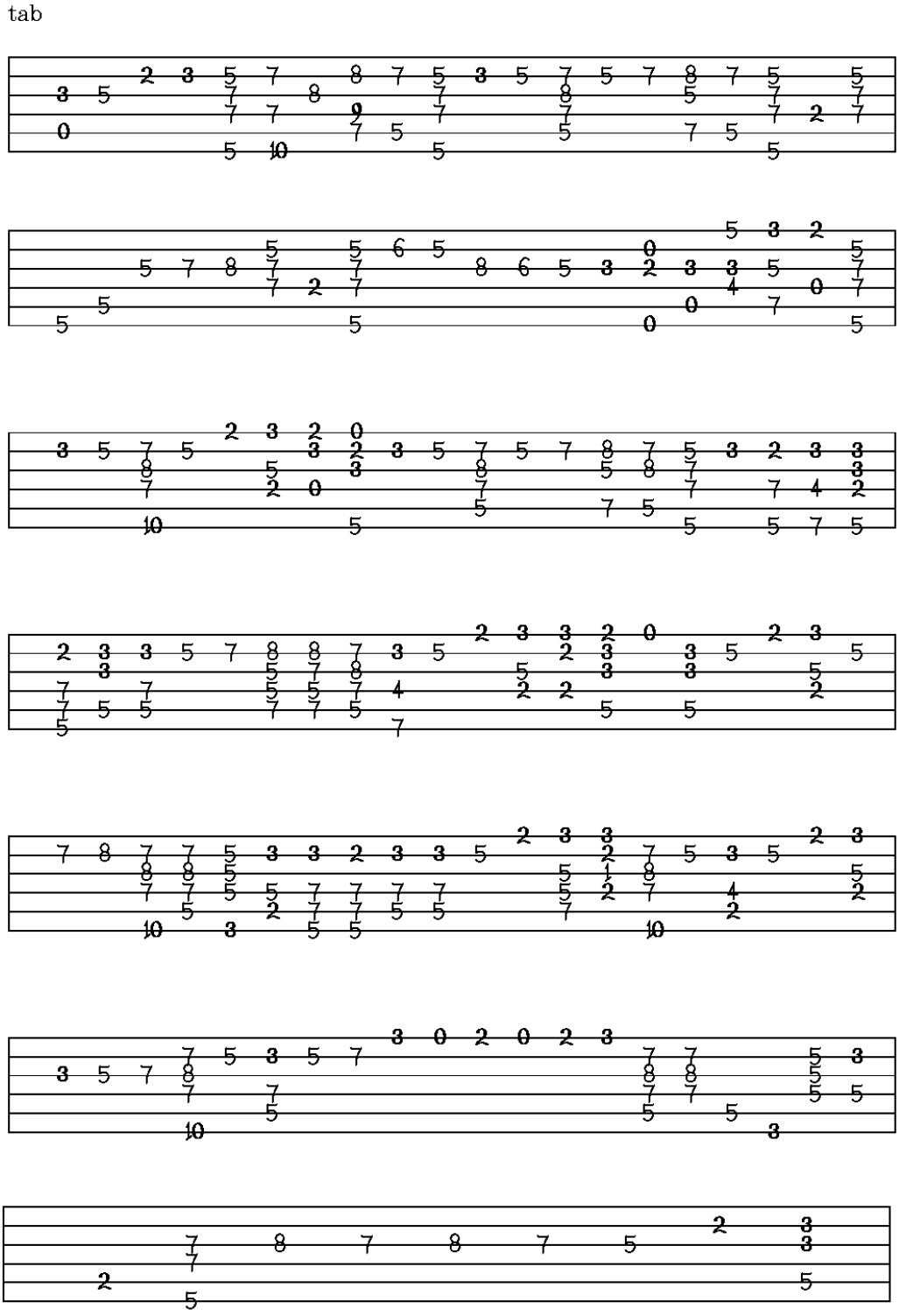


Figure 8: *Volte de Provence* formatted output at a population of 100% the number of notes in the piece