Creating Tablature and Arranging Music for Guitar with Genetic Algorithms and Artificial Neural Networks

by

Daniel R. Tuohy

(Under the direction of Walter D. Potter)

Abstract

The methods described in this thesis address the problems of both music arranging and tablature generation for the guitar. Arranging is the process by which a piece of music is adapted so that it can be performed on an instrument for which it was not originally written. It is interpreted here as an optimization problem, the goal of which is to establish the most desirable set of notes from the original composition. In the pursuit of this goal, new methods for the automatic generation of guitar tablature were devised. Tablature is a notation system from which the majority of western guitar players read music, and contains information essential to determining the difficulty of a piece of guitar music. Genetic Algorithms are initially employed to solve both the arranging and tablature problems, and an artificial neural network is introduced at a later stage as a faster and more accurate solution to the tablature problem.

Index words: Guitar Tablature, Music arranging, Genetic Algorithms, Neural Networks

Creating Tablature and Arranging Music for Guitar with Genetic Algorithms and Artificial Neural Networks

by

Daniel R. Tuohy

B.S., The University of Georgia, 2004

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2006

© 2006

Daniel R. Tuohy

All Rights Reserved

Creating Tablature and Arranging Music for Guitar with Genetic Algorithms and Artificial Neural Networks

by

Daniel R. Tuohy

Approved:

Major Professor: Walter D. Potter

Committee: Khaled Rasheed
Ronald W. McClendon

Electronic Version Approved:

Maureen Grasso
Dean of the Graduate School
The University of Georgia
May 2006

Acknowledgments

First, to my parents. You have managed to be proud of everything I do, no matter how insignificant.

To my grandfather, who is no doubt the basis for my interest in music and is in every respect what I aspire to become.

To my friends at the AI Center (especially Brian, Julian, and Gopika), who made my time here a lot more fun than it had any right to be. You also made me a far better researcher than I would have *ever* been on my own.

To my professors. Especially Dr. Potter, who in my last year as an undergraduate unearthed a drive in me that I did not know was there.

And lastly, to Maria-Joon. Because of you, I have been far happier this year than ever before.

Table of Contents

Page

Acknowledgments iv

List of Figures viii

List of Tables ix

Chapter

1 Introduction 1

2 A Genetic Algorithm for the Automatic Generation of Playable Guitar Tablature 3

2.1 Introduction 3

2.2 Background 4

2.3 The Genetic Algorithm Approach for Tablature Generation 6

2.4 Experiment Setup and Results 9

Bibliography 13

3 Creating Guitar Tablature with LHF Notation via DGA and ANN 15

3.1 Introduction: The Guitar Fingering Problem 15

3.2 TabGA: Finding Good Tablature with Distributed Genetic Search 17

3.3 An overview of the TabGA Fitness Function for evaluating tablature difficulty 21

3.4 Improving fitness assessments with the Fitness Function Meta GA (FFMGA) 22
3.5 LHFNet: a Neural Network for Left Handed Fingering Notation 23
3.6 Results 26
3.7 Further Directions 27

Bibliography 28

4 GA-based Music Arranging for Guitar 30
4.1 Introduction 30
4.2 Background 31
4.3 The Evaluation Function for Assessing an Arrangement . 35
4.4 Methods for Obtaining Arrangements 39
4.5 Results 41
4.6 Conclusions and Further Directions 42

Bibliography 45

5 An Evolved Neural Network/HC Hybrid for Tablature Creation in GA-based Guitar Arranging 48
5.1 Introduction 48
5.2 Background 49
5.3 Neural Network Model 50
5.4 Implementing the Neural Network for Tablature Creation 52
5.5 Using generated tablatures in arrangement evaluation . 54
5.6 Results 55
5.7 Further Directions 57

Bibliography 58

6 Conclusion and Further Directions 60

Appendix

A Source Code 61

A.1 Generating a Tablature with a Neural Network 61

A.2 Arranging an original composition via GA 67

A.3 The Arrangement Evaluation Function 70

B Sample Arrangements by Genetic Algorithm 73

B.1 *Symphony No. 2* by Sergey Rachmaninoff 73

B.2 *Arioso* by J.S. Bach 74

B.3 *Jupiter* by Gustav Holst 75

List of Figures

2.1 Four different fretboard positions for the 3rd octave “E”. 4

2.2 Obtaining children from two parents. Music from “Stairway to Heaven” by Jimmy Page and Robert Plant. 7

2.3 Tablatures for an excerpt from ”As Time Goes By” by Herman Hupfeld. . . 11

3.1 Four different fretboard positions for the 3rd octave “E”. 16

3.2 An overview of our system and it’s five components. (1) TabGA takes as input music and creates tablature. TabGA’s operating parameters were set using (2) PMGA and the fitness function was optimized with (3) FFMGA. The resulting tablature is then used by (4) LHFNet to create left handed fingering notation. The architecture of LHFNet was discovered with (5) NNGA. 17

3.3 Obtaining a child from two parents in a simple GA. 19

4.1 The creation of a child tablature from two parents in TabGA. 34

4.2 An excerpt from Rachmaninoff’s Symphony No. 2 that contains examples of 5 of the 6 note categories that our system recognizes. 36

4.3 The exponential decay function that determines the reward for each note in a chord. 38

4.4 An arrangement of an excerpt from Rachmaninoff’s Symphony No. 2 by our system. We use either a greedy hill-climber or genetic algorithm (in this case, the former) to generate an arrangement from the orchestral score. The tablature for the arrangement is produced by TabGA. 43

5.1 A tablature is created by neural network. The last note is “fixed” with local search. 53

5.2 Generating an arrangement from a score. 56

List of Tables

3.1 Performance of 3 GAs on 11 musical excerpts. Number of times in 10 runs the baseline solution was found. 26

Chapter 1

Introduction

The research described here grew out of a desire for software that can reliably produce tablature for a given piece of guitar music. While the internet is a good source of tablature for popular music, there is a great deal of music for which it is still very difficult to find high quality tablature. Guitarists have the option of producing it themselves, but this can be very tricky for casual players. Commercial generators exist, but are notoriously unreliable. We believe that the methods put forth here are, if not ideal solutions, at least significant steps in the right direction.

Once we became satisified with the performance of the first method for producing tablature, a further application became feasible. We developed a genetic algorithm (GA) for generating arrangements of compositions not written for the guitar. When evaluating individual arrangements, the GA uses one of our methods for automatic tablature generation to create and evaluate the difficulty of the corresponding tablature. When combined with other heuristics for evaluating arrangements, the GA demonstrates an ability to create from a set of notes and appropriate arrangement for the guitar.

Our methods for tablature and arrangement generation are detailed over the course of four chapters corresponding to four different papers either published, soon to be published, or awaiting decision. In Chapter 1 we introduce a genetic algorithm for tablature generation. This chapter contains an overview of our tablature evaluation function, and an argument for the advantage of a GA-based approach over alternative approaches to tablature generation that have been employed in other research.

Chapter 2 provides a more detailed description of our evaluation function. A Meta-GA is used to improve the parameters of the function so that generated tablature is maximally consistent with published tablature. We also switch to a distributed genetic algorithm (DGA), which demonstrates superior search ability to the simple GA described in Chapter 1. Finally, we introduce a neural network for assigning Left-Handed Fingering (LHF) notation to tablature, which instructs the performer which specific finger to use for each note.

In Chapter 3 we turn to arranging, developing both a genetic algorithm and a greedy hill-climber for the task. Both use the DGA described in Chapter 2 to generate tablature. The DGA's evaluation function is used to assess the difficulty of arrangements based on the corresponding tablature. The system produces encouraging results, but the arranging process takes in excess of 15 minutes with the greedy hill climber and over an hour with the GA.

The bottleneck for our arranging methods is the tablature generation, which takes several seconds. In Chapter 4, we introduce a neural network for tablature generation, the output of which is enhanced with a local heuristic hill-climber (HC). This hybrid approach proves to be far faster than the DGA, and the tablature it generates is slightly more consistent with published tablature. Using this method for tablature generation, the arranging GA can generate arrangements in under a minute.

The current model we employ uses the neural network/HC hybrid for tablature generation detailed in Chapter 4. The heuristic used by the HC is the evaluation function introduced in Chapter 1 and developed in Chapter 2. For generating arrangements, our system can uses the GA described in Chapter 3.

Chapter 2

A Genetic Algorithm for the Automatic Generation of Playable Guitar Tablature

2.1 Introduction

This chapter[1] describes a method for mapping a sequence of notes to a set of guitar fretboard positions (tablature). The method uses a Genetic Algorithm (GA) to find playable tablature through the use of a fitness function that assesses the playability of a given set of fretboard positions. Tests of the algorithm on a variety of compositions demonstrate an excellent ability of the GA to discover easily playable tablature that maintains a high degree of consistency with published tablatures transcribed by humans. The algorithm was also found to generally outperform commercial software designed for the same purpose. We conclude that the GA can reliably produce good tablature for any piece of guitar music.

Stringed instruments often require a great deal of experience and decision making on the part of the performer. A given note on the guitar may have as many as five different positions on the fretboard on which it can be produced. Figure 2.1 shows a guitar fretboard, and four different fretboard positions on which the same "E" can be played. To play a piece of music, the performer must decide upon a sequence of fretboard positions that minimize the mechanical difficulty of the piece to at least the point where it is physically possible to be played. This process is time-consuming and especially difficult for novice and intermediate players and, as a result, the task of reading music from a page as a pianist would is limited only to very advanced guitar players. To address this problem, a musical notation known

[1] Tuohy, Daniel R., and W.D. Potter. 2005. In Proceedings of *International Computer Music Conference. ICMC 2005* Reprinted here with permission of publisher.

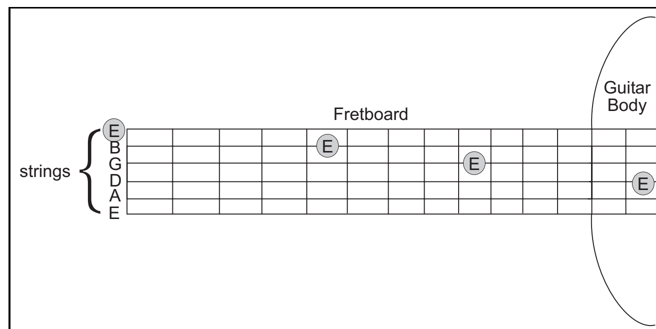

Figure 2.1: Four different fretboard positions for the 3rd octave "E".

as tablature is used. Tablature describes to the performer exactly how a piece of music is to be played by graphically representing the six guitar strings and labeling them with the corresponding frets for each note, in order.

The goal of our research is to automatically discover very good tablature for any piece of music. In the next section we will describe recent commercial and academic attempts to accomplish the same feat. We will then describe our genetic algorithm approach and offer an explanation for why it is better suited for this problem

2.2 Background

2.2.1 Commercial Software

Several programs on the market today convert music into tablature. These programs are, however, notorious for producing unplayable or unnecessarily difficult tablatures[6]. This is because they often depend too heavily upon rules of music theory relating to harmony and guitar composition[12]. For example, "open" strings are considered to be easier to play on guitar because they do not require depressing the string on a fret. However, it is often easier for a note to be played on a depressed string if that string happens to already be depressed

at that point in the music. Software that we tested seems to take the inadvisable approach of creating tablature note by note, rather than devising a strategy for the piece as a whole. This approach necessarily results in poor tablature for many pieces of music. For these reasons, current commercial software often depends upon the user to edit a tablature after it has been generated[12].

2.2.2 Academic Research

A method for discovering playable tablature was described by Samir I. Sayegh[11]. His "optimum path paradigm" describes a sequence of fretboard positions as a sequence of hand states, the goal being to find the optimum path through the states. More recently, there have been algorithms that build upon Sayegh's approach at the University of Torino[9] and the University of Victoria[10]. Both approaches have produced excellent tablatures for the pieces on which they were tested, but both have limitations. The prototype from the University of Torino cannot account for situations where more than one note needs to be played simultaneously (a chord). The program from the University of Victoria appears to be a "lazy learner" which improves its accuracy by requiring customized learning for each piece. Our algorithm aims to be able to generalize to any piece with a satisfactory degree of accuracy, though it will be difficult to assess which algorithm produces more desirable results. Another group from Doshisha University reports success in generating tablature superior to that of commercial software, but once more the program is limited to producing tablature for melodies without chords[6]. It should be noted that the former two approaches include Left Handed Fingering notation, which instructs the player on which specific fingers to use. Our approach does not currently support this feature.

2.3 The Genetic Algorithm Approach for Tablature Generation

2.3.1 The Genetic Algorithm

A Genetic Algorithm (GA)[7][8] is a stochastic search algorithm that aims to find good solutions to difficult problems by applying the principles of evolutionary biology. Evolutionary concepts such as natural selection, mutation and crossover are applied to a "population" of solutions in order to evolve a very good solution. Problems suited for a GA are those whose number of possible solutions (search space) is so large that finding good solutions by exhaustive search techniques is computationally impractical. The genetic algorithm is also useful for tackling search spaces with many scattered maxima and minima. This property of GAs is essential in the search for tablature, as the search space can be quite large, generally on the order of 3^n where n is the number of notes.

2.3.2 Implementation

The population for our GA is a collection of tablatures that are valid, though not necessarily desirable, for a given piece of music. The initial population is generated randomly. A tablature "chromosome" is defined as a sequence of chords. A chord is a "gene" and consists of fretboard positions for all the notes in that chord. A chord, in this sense, is any combination of notes that are played simultaneously. Pieces are generally not evolved all at once, but rather should be divided into logical excerpts and evolved separately. Evolving an entire piece is generally undesirable as it would likely create a search space too large for a GA to search effectively in a reasonable amount of time.

The parameters for the GA are those which have empirically led to convergence on the most fit individuals and were tuned manually. Our GA has a population size of 300 and uses binary tournament selection with two-point crossover. The crossover rate is set at 60% and the mutation rate (which mutates a single random chord) is set at 7%. To increase the overall performance of the program, we found it useful to run the GA several times for

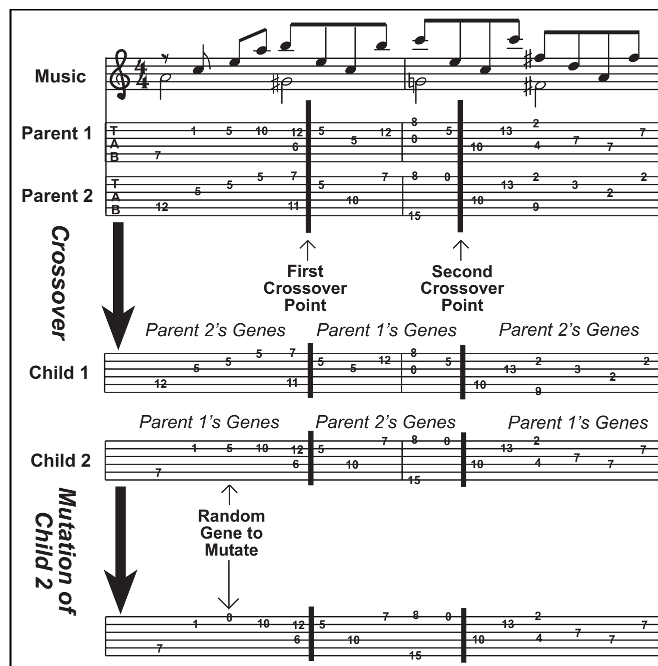

Figure 2.2: Obtaining children from two parents. Music from "Stairway to Heaven" by Jimmy Page and Robert Plant.

100 generations and save all of the most fit individuals found. We then run the GA once more with a population composed predominately of these individuals. Very often this final run will produce an individual more fit than any other found so far. This is often the best individual ever found as defined by our fitness function. The chance that this individual is optimal varies with the length of the excerpt and the complexity of the corresponding search space. This "seeding" scheme is often necessary because of the deceptive nature of the problem. Deception arises because each piece of music has several globally competitive but structurally divergent optima.

2.3.3 Fitness Function

Our function analyzes a tablature in many different ways to assess its playability. This assessment is partially informed by the analysis of finger-positioning complexity of Heijink and Meulenbroek[5]. Unfortunately, it has proven very difficult to accurately capture difficulty by simply keeping track of the movement of individual fingers and our approach is therefore significantly different from that of other academic tablature generators. We acknowledge that an approach that assesses difficulty as a function of each finger's movements is more elegant and potentially more accurate, but we found implementation of such a scheme to be impractical. We found it satisfactory, and far simpler, to devise a set of heuristics that estimate difficulty by defining general properties to which good tablature tends to adhere and measuring tablature against these. The function can be thought of as calculating two separate classes of tablature complexity, difficulty of hand/finger movement and difficulty of hand/finger manipulation.

Hand Movement

These calculations essentially estimate the total amount of lateral hand and finger movement across the fretboard that is necessary to perform a tablature by executing simple calculations on the fret numbers. The more movement that is required, the more the function penalizes the tablature's fitness. Fitness is penalized according to three factors in this category: The total number of times that strings must be depressed, the number of frets that separate adjacent notes, and the number of frets that separate each note from the average fret of the six surrounding notes in the piece of music. Fitness is rewarded proportionally to the number of open notes, as they do not require any lateral movement by the left hand to be executed.

A number is accumulated for each one of these factors and each has an associated scaling coefficient that has been tuned so that generated tablature more accurately coincides with published tablature.

Hand Manipulation

This portion of the fitness function attempts to analyze what the left hand itself is doing and assess the difficulty of the requisite hand positions. There are two predominant methods by which a guitarist can finger multiple notes. The open chord method requires depressing strings individually for each note. Alternately, one may place a finger across a fret over several strings and use the remaining fingers for any other notes on higher frets. This is known as a barre chord.

After much manual analysis of the possible open and barre chords, we settled upon several heuristics that define valid chords of both types. At every note in a tablature, we count how many notes can be played sequentially without violating one of these heuristics. This number is then penalized according to the difficulty of the chord. The difficulty of the chord is assessed by measuring how many fingers are involved, how far they must be spread, and so on. The heuristics that define valid chords and that determine the difficulty of those chords are too numerous to enumerate here. It is also by no means true that our heuristics capture the target concept defined by the mobility of the human hand, but rather that they approximate the concept well enough for our purposes.

2.4 Experiment Setup and Results

2.4.1 Experiment Setup

A common metric for establishing the success or failure of a tablature generator is the percentage of fretboard positions in the generated tablature that are consistent with a published tablature. Over a very large body of work this should provide a rough indication of the generator's reliability, but the metric is not perfect. While playability is the main concern for a human when creating tablature, it is not the only concern. In cases where notes can be placed on the fretboard in multiple positions without significant differences in playability, the position chosen by a professional could seem essentially arbitrary. Because each guitar

string has a slight but noticeable difference in timbre, tone quality becomes a factor for notes with more than one viable position. If differences in playability are very slight, the chosen position could even be little more than the personal preference of the tablature creator.

Consequently, our results do not lend themselves to numerical representation. Our metric for success is simply that our generator never create a tablature significantly more difficult than necessary. This is hard to accurately express statistically, since there is no entirely reliable objective measure of difficulty. If there were, it would be our fitness function. We will instead demonstrate a weakness of commercial tablature generators to which our approach is resistant. We tested several different generators including a few pieces of free software, and the commercial applications TablEdit and GuitarPro 4. The TablEdit software generates tablature one measure at a time, and hence tends to make mistakes in phrases that cross bar lines. The GuitarPro software uses an unknown algorithm that seemed to produce the most professional tablatures and was used as the primary benchmark against which we subjectively evaluated our algorithm.

2.4.2 Results and Conclusion

We ran our GA on a wide variety of both popular and classical musical literature. In all, selections from 34 pieces of music were tested. The majority of tablature coincided with published tablatures and the algorithm never produced a tablature that was significantly more difficult than the published tablature. We consider the GA a success because it reliably produces playable tablature even when departing from the published versions. Figure 2.3 is an example of the type of departure our generator never made. It is an excerpt from jazz standard “As Time Goes By” by Herman Hupfeld.

In this case the tablature generated by the GA is the same as the published tablature. The commercial generated tablature begins differently and as a result is probably unplayable. The first six notes are placed low on the fretboard and this results in a difficult jump from the second fret to the tenth fret between the second and third beats. The human professional

Figure 2.3: Tablatures for an excerpt from "As Time Goes By" by Herman Hupfeld.

and the GA, however, account for the fact that the hand will have to be high on the fretboard and place early notes in the tablature in that area as well.

The reason our approach does not make the same mistakes as the commercial generator is the implicit parallelism with which the GA processes the tablature. While the GA evaluates the first six positions of the commercially generated tablature as fit when they are by themselves, when those positions are viewed in context the tablature's fitness suffers and hence tablatures containing those positions are selected less often and eventually purged from the population.

There is a general property we noticed of generated tablatures which diverged from the published versions. We found that these tablatures were not necessarily more mechanically difficult, but were often more difficult to read from the page because they do not conform to

a conventional practice of human tablature creators. This practice dictates that higher notes should be played on higher strings, and lower notes should be played on lower strings. Our algorithm, and any algorithm that consults only finger-positioning in assessing difficulty, is unaware of this and in trying to minimize mechanical difficulty will occasionally violate the practice. It would be reasonably simple to implement a fitness penalty for infractions of this nature, but because our goal was to minimize mechanical difficulty rather than precisely match published tablatures we did not.

Mistakes made by the commercial software usually only result in tablatures that are unnecessarily difficult but, as is shown in the example, sometimes create tablatures that are unplayable. The advantage of our Genetic Algorithm over other approaches becomes apparent when creating tablature for sections of music that require knowledge of later sections to be transcribed correctly. Generated tablatures did not always maintain consistency with the published tablatures, but deviations proved to be mostly inconsequential to overall playability and never disastrous. We offer as evidence of the validity of our approach that we never encountered a piece of music for which the GA produced a tablature that was significantly more difficult than the published counterpart.

Bibliography

[1] Colgan, B., Stang, A. *Led Zeppelin: Acoustic Classics Vol. 1.* Warner Brothers, Miami, 1995.

[2] Colgan, B., Stang, A. *The Solo Guitar Big Book*. Warner Brothers, Miami, 2001.

[3] Davis, L. *The Handbook of Genetic Algorithms*. Van Nostrand Reinhold, New York, 1991.

[4] Guitar Pro. *Guitar Pro 4 Demo*. 2004

[5] Heijink, H., Meulenbroek, R.G.J. "On the complexity of classical guitar playing: functional adaptations to task constraints", *Journal of Motor Behavior*. 34(4), 339-351, 2002.

[6] Miura, M., Hirota, I., Hama, N., Yanigida, M. "Constructing a System for Finger-Position Determination and Tablature Generation for Playing Melodies on Guitars", *Systems and Computers in Japan*. 35(6), 755-763, 2004.

[7] Holland, J. "Genetic Algorithms", *Scientific American*. 267, 44-50, July, 1992.

[8] Liepens, G.E., Potter, W.D. "A Genetic Algorithm Approach to Multiple-Fault Diagnosis", *Handbook of Genetic Algorithms*. 237-250, Van Nostrand Reinhold, New York, 1991.

[9] Radicioni, D., Anselma, L., Lombardo, V. "A segmentation-based prototype to compute string instruments fingering", *Proceedings of the Conference on Interdisciplinary Musicology*. Graz, Austria, 2004.

[10] Radisavljevic, A., Driessen, P. "Path Difference Learning for Guitar Fingering Problem", *Proceedings of the International Computer Music Conference*. Miami, USA, 2004.

[11] Sayegh, S. "Fingering for String Instruments with the Optimum Path Paradigm", *Computer Music Journal*. 13(6), 76-84, 1989.

[12] Wang, J., Tsai-Yen, L. "Generating Guitar Scores from a MIDI Source", *International Symposium on Multimedia Information Processing*. Taipei, Taiwan, 1997.

Chapter 3

Creating Guitar Tablature with LHF Notation via DGA and ANN

3.1 Introduction: The Guitar Fingering Problem

This chapter[1] describes a system for converting music to guitar tablature. At run time, the system employs a distributed genetic algorithm (DGA) to create tablature and an artificial neural network to assign fingers to each note. Three additional genetic algorithms are used to optimize the fitness function of the DGA, the operating parameters of the DGA, and the learning environment of the Neural Network. These steps are taken in the hope of maximizing the consistency of our algorithm with human experts. The results have been encouraging.

Stringed instruments suffer from a one-to-many relationship between notes and the positions from which they can be played. Figure 3.1 shows four different positions from which the same "E" can be produced. These are called fretboard positions and are described by two variables, a string and a fret. It is the burden of the performer to choose a fretboard position for every note in a piece of music. Tablature is a musical notation system that represents music as fretboard positions rather than as notes, freeing the performer from tedious decision making. Tablature is ordinarily created by humans. There have been commercial attempts to generate tablature automatically, but these are notorious for creating unnecessarily difficult or unplayable tablature [3] [11].

It is the goal of our system to be able to consistently generate tablature competitive with that created by humans. This problem has been given considerable attention in the last few years. Examples of three recent academic attempts to achieve a similar goal were conducted at

[1]Tuohy, Daniel R., and W.D. Potter. 2006. To appear in Proceedings of *Industrial and Engineering Applications of Artificial Intelligence and Expert Systems. IEA/AIE'06.* Lecture Notes in Artificial Intelligence, Springer-Verlag, Berlin. Reprinted here with permission of publisher.

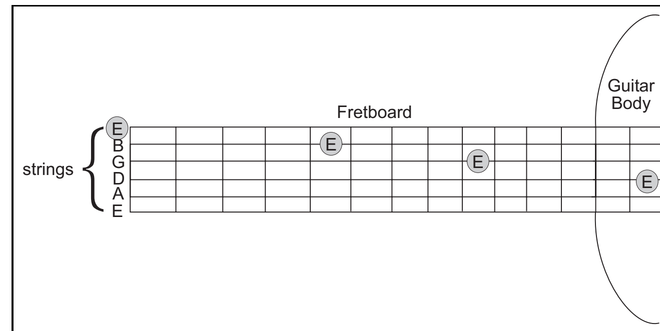

Figure 3.1: Four different fretboard positions for the 3rd octave "E".

the Universities of Doshisha, Victoria, and Torino, each with apparent weaknesses. The model from the University of Doshisha has been designed only for use on monophonic melodies, and cannot create tablature for pieces including chords [3]. The model from the University of Victoria is a learning algorithm that consults published tablature to learn different musical styles, and an assessment of performance on an independent test set is not provided [6]. We will only compare our results to those from the University of Torino, as this approach has the same aspirations as our own [5][4].

In this chapter we will describe four genetic algorithms and one neural network. In section two we will describe TabGA, the distributed genetic algorithm designed to create tablature from guitar music. Section 2 also provides an analysis of PMGA, a Meta GA for optimizing the operating parameters of TabGA. A Meta GA, as used in this chapter, is defined as a GA that executes another GA in its fitness evaluations. Section three is an analysis of the fitness function of TabGA, upon which the ability to create tablature competitive with human experts is entirely dependent. In section four we describe FFMGA, a Meta GA used to tune the fitness function of TabGA. In section five we discuss LHFNet, the neural network we have implemented to determine appropriate left handed fingering for guitar tablature.

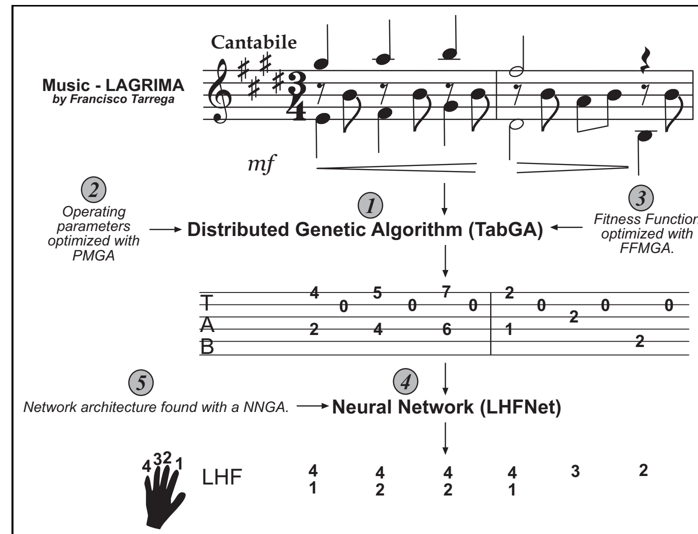

Figure 3.2: An overview of our system and it's five components. (1) TabGA takes as input music and creates tablature. TabGA's operating parameters were set using (2) PMGA and the fitness function was optimized with (3) FFMGA. The resulting tablature is then used by (4) LHFNet to create left handed fingering notation. The architecture of LHFNet was discovered with (5) NNGA.

Section 5 also will address NNGA, a GA used to determine the architecture for LHFNet. An overview of the entire system is illustrated in Figure 3.2.

3.2 TabGA: Finding Good Tablature with Distributed Genetic Search

3.2.1 A GA for the Tablature Problem.

In previous work we detailed how genetic algorithms can be used to find good tablature [9]. At the start of a run, a seed population of hundreds of random, but musically valid, tablatures is created. The workings of genetic algorithms are well-known, and figure 3.3 illustrates how crossover and mutation function in this domain.

3.2.2 Why we use Genetic Search.

Suppose that, on average, a note can be played in three different fretboard positions (this is lower than the actual average). Then a tablature with n notes has 3^n possible tablatures. Exhaustive search techniques are quickly rendered impractical as the number of notes is increased. One might assert that this does not present a problem, since there does not seem to be a reason to create tablature for more than a few notes at a time. Why should we be concerned with notes to be played several seconds in the future when deciding on a fretboard position for a given note? Consider the 1st and 10th notes in a piece of music. The fretboard position of the 1st note is only directly dependent on those of the few notes that immediately follow it, say only as far as the 4th note. However, the 4th note may depend upon the 7th, and that on the 10th and so on until there is a break in the music that allows the performer time to move his hand freely. This establishes a dependency chain within phrases that makes it inadvisable to assign positions to notes sequentially, as is the approach (and weakness) of many commercial tablature generators. For this reason, we choose to break music into logical phrases manually, then process each phrase with the GA.

3.2.3 Genitor II Background

The DGA that we have implemented in TabGA is based on the Genitor II system devised by Whitley and Starkweather [14]. Genitor II is an implementation of a DGA, also called Island GAs. It was chosen because of its reliability for converging quickly on very good solutions over a wide variety of NP-hard problems [1]. Adherence to the steady-state and island paradigms is characteristic of Genitor II. A Steady State GA is one in which individuals are introduced into the population individually, rather than as a whole new generation. An Island GA is one in which the population is divided into separate sub-populations, each of which conducts genetic search independently. It is this aspect which provides the distributed nature of the DGA. Each island consists of a small population of individuals and uses a Steady State GA to evolve that population. Because of sampling bias inherent in the initial population of each

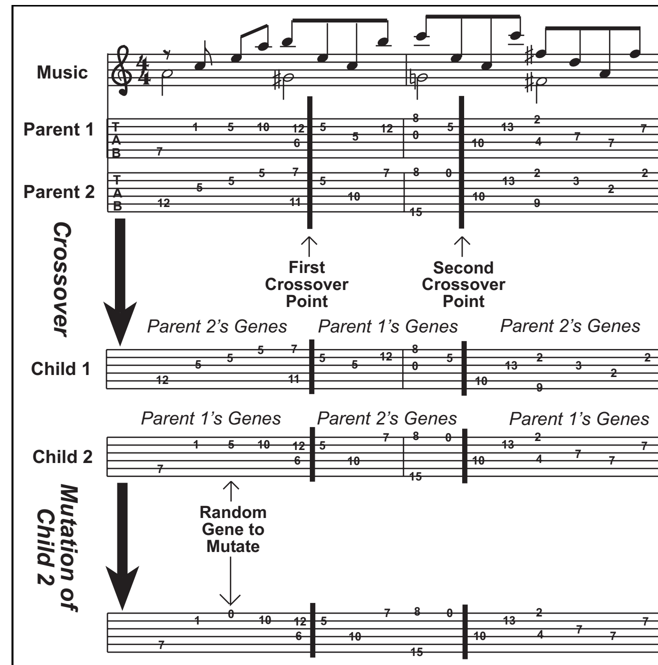

Figure 3.3: Obtaining a child from two parents in a simple GA.

island, the islands will naturally focus on different parts of the search space. The islands periodically swap their best individuals between one other, which encourages the sharing of beneficial characteristics that might proliferate in distant parts of the search space.

Whitley specifically mentioned that problems with "multiple structurally incompatible solutions" are well-suited for exploitation by the Genitor II scheme. The application he has in mind is the optimization of weights in a neural network. Such problems may have many globally competitive solutions that, when recombined with each other, yield very poor results. This is exactly the problem that we face with tablature generation. Consider a piece of music that is optimally played low on the fretboard, but could be as easily played higher on the fretboard. Attempting to recombine a tablature of each type would result in an unplayable tablature. The results presented by Whitley and Starkweather on neural network optimization indicate that Genitor II is better suited to such problems than other genetic

search techniques. The Genitor II scheme presents us with more operating parameters than would a simpler evolutionary search technique, so we again turn to genetic search to find good values for these parameters.

3.2.4 Optimization with the Parameter Meta GA (PMGA)

A set of DGA operating parameters can be represented as a chromosome composed of eight variables. These are the population size, the number of islands, the interval at which islands swap individuals, the number of individuals that are swapped, the linear bias of the ranked-based selection, two variables governing adaptive mutation, and a binary variable indicating whether to use one or two point crossover. In PMGA we have chosen to use a meta GA based loosely on Whitley's Genitor I (non-distributed) approach to search for the best set of values for TabGA [13].

To assess fitness, a DGA is run on eleven different musical excerpts. For each excerpt, we evolved a population of tablatures until no improvement was found in the best individual for 5000 replacements. We previously ran a hand tuned DGA 100 times on all eleven excerpts to establish a baseline tablature for each excerpt that should be close to an optimal solution as defined by our fitness function. We compare the best individual at the end of the evolution process with the baseline.

PMGA is a steady-state scheme with ranked based selection and a population size of 150. Uniform crossover and average (arithmetic) crossover are used interchangeably since both are applicable. Mutation is random and occurs with a probability of 10% at each gene. We ran PMGA for 20,000 replacements, at which point very little improvement in average population fitness was being made. We then ran each of the 150 DGA parameter sets in the final population ten times and averaged their performance, taking the DGA with the best average performance as our final solution.

3.3 An overview of the TabGA Fitness Function for evaluating tablature difficulty

The fitness function for our GA is required to assess the difficulty, or "playability", of tablature. We have elected not to use a system based on minimizing the distance traveled for each finger. This method was first detailed by Sayegh and has been expounded upon in other research [7] [6] [5]. Systems that adopt this paradigm analyze of the movement of each finger, and then use a cost function that penalizes each finger's movement. However, net finger movement alone does not constitute difficulty. Many other factors, some very difficult to articulate, are also important. We have implemented a fitness function based on the author's musical expertise. We have endeavoured to include in our fitness function as many relevant difficulty factors as possible, though none are dependent upon accurately accounting for each finger.

We will now provide an overview of our fitness function, which is quite lengthy. We will provide as much information as possible, but will summarize specific details for the sake of brevity. We have included as many metrics of difficulty as possible, and will allow a Meta GA (FFMGA) to decide the importance that each metric should be given.

3.3.1 Biomechanical Factors

There are twelve factors considered to be biomechanical in nature because they assess the physical difficulty of hand movement.

How the Hand Moves.

The first eight biomechanical factors are concerned with the total amount of movement the hand must undertake. These are the number of times frets must be depressed, a reward for the number of times open strings are played, the total fretwise distance between sequential chords, a penalty for larger fretwise distances, total fretwise distance between each chord and the average of the six surrounding notes, a penalty if this distance is particularly large,

total maximum fretwise distance within chords, and total largest fretwise distance between any two notes of sequential chords.

How the Hand is Held.

The remaining four factors reward tablatures that allow the hand to remain in the same position as often as possible, allowing each finger to be lifted or pressed exactly once. We look for hand positions that cover the tablature to the maximum extent. For each position, there is a variable that holds the number of notes for which that position can be held and a variable that carries a penalty associated with the difficulty with holding the position.

3.3.2 Cognitive Factors

There are two other factors that are cognitive in nature, and assess the difficulty that may arise from the violation of certain accepted conventions of guitar performance unrelated to physical difficulty. Guitar players will be startled by tablatures which, even if physically less difficult, are not intuitive. The first of these conventions is that higher notes should be played on higher strings. In line with this convention, a tablature incurs a penalty if a note that is higher than the preceding note is played on a lower string, or a lower note on a higher string. A second convention gives a slight penalty to notes placed higher on the fretboard. This establishes a preference for positions lower on the fretboard in cases where two candidate fretboard positions are of similar difficulty.

3.4 Improving fitness assessments with the Fitness Function Meta GA (FFMGA)

We have a total of fourteen difficulty factors, and each has an associated weight that determines its contribution to tablature fitness. In previous work we tuned these weights manually, hoping to find a set of weights that would best capture the actual difficulty of tablature. We seek to improve upon our efforts with a Meta GA. The chromosome for a DGA in FFMGA

is fourteen continuous valued variables corresponding to the weights of the difficulty factors. Each variable is given a range of possible values that we consider to be reasonable, either 0-30 or 0-3 depending on which order of magnitude we deem most appropriate for that variable. For assessing fitness we have taken from published tablature books and the web a selection of excerpts from 30 different pieces of music comprising a total of 751 notes and their corresponding fretboard positions. We selected the excerpts in a random fashion, though preference was given to excerpts that moved around the fretboard, thereby requiring a more intelligent approach to tablature creation. The fitness of a fitness function chromosome is the percentage of generated fretboard positions consistent with those assigned by human experts. The implementation of FFMGA is nearly identical to that of the PMGA, and was run for 20000 replacements.

3.5 LHFNet: a Neural Network for Left Handed Fingering Notation

Tablature sometimes includes information on which specific fingers to use at each fretboard position. This is known as Left Handed Fingering (LHF) notation, and further relieves the performer from having to decide on appropriate performance technique. We have employed a Neural Network to handle this task. We chose to use a Neural Network because of the vast number of inputs that are potentially viable. There is, for each learning pattern, a set of 59 variables that we consider and there is no obvious method for determining which are relevant and how they should be weighted. By using these variables as inputs to a neural network we free ourselves from having to make such determinations. The model we have adopted consults a neural network at every fretboard position in a tablature to determine the appropriate finger to be used. The outputs are values between .1 and .9 for each of the four fingers. We then assign whichever finger has the highest value as the finger for that fretboard position.

3.5.1 Training Data Acquisition

We trained the network with tablature from the web. Tablature was taken from the popular classical guitar tablature repository at www.classtab.org [16]. We chose thirty pieces of music that included LHF notation. The pieces are all classical, including works from Bach, Villa-Lobos, Tarrega, Sor, and others. In total we generated 6800 training patterns, one for each note, from the data.

3.5.2 Network Architecure and Inputs

For exploratory purposes, we used the neural network package NeuroShell to get an idea of what network architecture was most suitable for this problem [15]. We tested multilayer, Ward, and simple back-propagation architectures with several combinations of activation functions and layer sizes. The standard three-layer back-propagation architecture with sigmoid activation functions was found to fit the data the best. We will focus our search on networks of this type and use genetic search to determine hidden layer size, learning rate, momentum, and inputs. The neural networks are built with source code from the GAIL package developed by Brian A. Smith [8].

3.5.3 Optimizing LHFNet with the Neural Network GA (NNGA)

Network Inputs.

For each note we have extracted a set of 59 possible network inputs. The first two are the fret and the string of the current note. The next twenty are comprised of attributes of the most recent note on which each finger was used. These are the fret, string, notes since, chords since, and a binary variable indicating if the note is in the same chord as the current note. There are then four binary variables indicating whether or not each finger is "free", which is dependent on how many notes have elapsed since the finger was last used, and if the string on which the finger was last used has been depressed by a different finger in the intervening time. The next 15 variables are the fret, string and finger of the previous five notes. The

next 10 are the fret and string of the next five notes. The remaining eight variables detail the chord that the current note occupies. These are number of notes, number of notes on open strings, the highest string that is open, the number of notes higher than the lowest fret in the chord, the highest played string, the lowest played string, the highest played fret and the lowest played fret.

We extracted all of these features from the data because all are potentially valuable, but we do not know a priori which ones are important. We use NNGA to pare down the inputs for us.

Chromosome and Fitness.

The chromosome for one of our networks is a bit string of length 59, two continuous valued variables for learning rate and momentum, and an integer for the number of hidden nodes. Each bit in the bit string determines whether or not an input is included. We set the range for learning rate at .05-.25, momentum at 0-.2 and hidden layer size at 8-17. These ranges were chosen based on what we have observed in exploratory testing with NeuroShell.

Network fitness is determined by accuracy on a production set, which comprises a randomly selected 15% of the 6800 patterns. This is also the size of the test set, and the remaining 70% are used for training. Training has concluded when no improvement in mean squared error has been found in 250000 events.

The GA.

NNGA is similar to the Genitor I style used in PMGA and FFMGA. We set population size at 100, mutation rate at 7%, and use rank-based selection with uniform crossover. The GA was run for 30000 replacements.

Table 3.1: Performance of 3 GAs on 11 musical excerpts. Number of times in 10 runs the baseline solution was found.

	1	2	3	4	5	6	7	8	9	10	11	*Total*
Optimized DGA (TabGA)	10	10	10	10	10	10	7	6	1	0	8	82
Hand Tuned DGA	10	9	10	10	10	10	7	2	1	1	7	77
Simple GA	10	7	7	10	4	3	2	0	0	0	1	44

3.6 Results

3.6.1 Optimized DGA vs. DGA vs. Simple GA

We are interested in comparing the effectiveness of the simple GA, the hand-tuned DGA, and the DGA found with PMGA. We ran the three GAs on each piece of music ten times for 20,000 replacements, which is sufficient time for convergence, and recorded how many times each algorithm found the baseline solution. Each GA was run with the parameter settings found to work best for that GA, either empirically or through testing, and all had the same population size.

The DGA parameter set found by PMGA seems to afford us only a marginal benefit, if any. The optimized DGA found the baseline solution in 74.7% of its runs, compared to 70% for the hand tuned DGA. However, there is a noticeable benefit to the DGA over the simple GA, which found the baseline solution only 40% of the time. When we consider only the "difficult" pieces, namely excerpts 5-11, the difference is even more remarkable. On these excerpts the optimized and hand tuned DGAs found the baseline solution in 60% and 56% of cases respectively, while the Simple GA found the solution in only 14.3% of the cases. As may have been expectd, no improvement upon the baseline solutions was made.

3.6.2 Consistency of TabGA and LHFNet with human experts

The accuracy achieved by TabGA on our test set is 91.1%. That is, 91.1% of the fretboard positions in the generated tablature were consistent with the human-created tablature. This is a marked improvement over the DGA using our previous fitness function, which has an accuracy of 73% on the same set of data. In addition, deviations from published tablature did not result in significantly more difficult tablature. For LHF notation, the accuracy achieved by LHFNet on the independent test set was 80.6%.

The University of Torino reports accuracies of 98% and 90% respectively, but these numbers were achieved on a very different set of data. The best tablatures that we were able to find on the web for the same pieces of music on which their system was judged had the characteristic that nearly every note was placed at the lowest possible position. Tablatures of this nature are not particularly difficult to create, and our system achieved an accuracy of 98.9% on data from the same pieces of music. We could not test our neural network on this data, however, since left handed fingering notation for these pieces was not available to us. For most of the thirty excerpts on which our algorithm was run, tablature creation is much more difficult. For this reason, we contend that the versatility and validity of our approach can be justly asserted.

3.7 Further Directions

This system has been used in the development of a framework for creating arrangements of pieces of music never before written for guitar, with compelling results. It acts as part of an evaluation function for a guided heuristic search that distinguishes arrangements on the basis of both playability, as defined by TabGA, and musical merit[10].

Bibliography

[1] Gordon, V.S., Whitley, D.: Serial and Parallel Genetic Algorithms as Function Optimizers. 5th International Conference on Genetic Algorithms (1993) 177–183

[2] Heijink, H., Meulenbroek, R.G.J.: On the complexity of classical guitar playing: functional adaptations to task constraints. Journal of Motor Behaviour **34(4)** (2002) 339–351

[3] Miura, M., Hirota, I., Hama, N., Yanigida, M.: Constructing a System for Finger-Position Determination and Tablature Generation for Playing Melodies on Guitars. Systems and Computers in Japan **35(6)** (2004) 755–763

[4] Radicioni, D., Lombardo, V.: Guitar Fingering for Music Performance. Proceedings of the Internation Computer Music Conference. Barcelona, Spain. (2005)

[5] Radicioni, D., Anselma, L., Lombardo, V.: A segmentation-based prototype to compute string instruments fingering. Proceedings of the Conference on Interdisciplinary Musicology. Graz, Austria. (2004)

[6] Radisavljevic, A., Driessen, P.: Path Difference Learning for Guitar Fingering Problem. Proceedings of the International Computer Music Conference. Miami, USA. (2004)

[7] Sayegh, S.: Fingering for String Instruments with the Optimum Path Paradigm. Computer Music Journal **13(6)** (1989) 76–84

[8] Smith, B.: GAIL: Georgia Artificial Intelligence Library Neural Network Package. University of Georgia. (2004)

[9] Tuohy, D., Potter, W.D.: A Genetic Algorithm for the Automatic Generation of Playable Guitar Tablature. Proceedings of the International Computer Music Conference. Barcelona, Spain. (2005)

[10] Tuohy, D., Potter, W.D.: GA-based Music Arranging for Guitar. Submitted to IEEE Congress on Evolutionary Computation (2006).

[11] Wang, J., Tsai-Yen, L.: Generating Guitar Scores from a MIDI Source. International Symposium on Multimedia Information Processing. Taipei, Taiwan. (1997)

[12] Whitley, D.: Genetic Algorithms and Neural Networks. Genetic Algorithms in Engineering and Computer Science (1995) 1–15

[13] Whitley, D., Kauth, J.: GENITOR: A Different Genetic Algorithm. Proceedings of the Rocky Mountain Conference on Artificial Intelligence.

[14] Whitley, D., Starkweather, T.: GENITOR II: a distributed genetic algorithm. J. Expt. Theor. Artif. Intel. **2** (1990) 189–213

[15] NeuroShell 2 Neural Network Kit by Ward Systems Group. http://www.wardsystems.com.

[16] Classical Guitar Tablature. http://www.classtab.org

[17] A-Z Guitar Tabs. http://www.guitaretab.com

Chapter 4

GA-based Music Arranging for Guitar

4.1 Introduction

In this chapter[1] we describe a system for converting a piece of music into an appropriate arrangement for guitar. The system is supported by previous research into the generation of guitar performance information (tablature) with genetic algorithms. This technology is a significant part of an evaluation function that is designed to differentiate between candidate arrangements on the basis of both faithfulness to the original piece and on playability. We describe both a genetic algorithm and a greedy hill-climber that use this function to find good arrangements. Both techniques have demonstrated an ability to create viable arrangements of the pieces tested.

Arranging is the process by which a piece of music is adapted so that it can be performed on an instrument or set of instruments other than those for which it was originally written [3]. When arranging a piece of music, the piece may require significant alteration in order to be playable on a new instrument. Such alterations are necessary when a piece contains notes too high or too low to be played on an instrument, or when there are simply too many notes for a performer to play. A piano player has only ten fingers, a guitar has only six strings, and a trumpet has only one tube. These limits impose restrictions on what can be played on the instrument. When adapting a piece of music, one must be cognizant of such limitations and carefully consider how to work around them. This often requires that notes be moved

[1]Tuohy, Daniel R., and W.D. Potter. 2006. To appear in Proceedings of *IEEE Congress on Evolutionary Computation. CEC 2006.* Reprinted here with permission of publisher.

up or down an octave, shortened, lengthened, or eliminated altogether. A good arranger will make these adjustments in such a way that the original intent of the composer is preserved.

The arrangements we are concerned with here are the subset known as "reductions". Reductions are arrangements in which as few creative liberties as possible are taken and the goal is only to reproduce the original composition while ensuring that it is possible to be performed. When creating a reduction an arranger does not introduce new harmonies, counter-melodies, or bass lines, but merely takes as much as possible of that which the composer has provided. There are, however, two types of alteration that we incorporate into our arrangements for guitar. Alterations of note-length are implicit in the process, as notes may be incidentally shortened or lengthened depending on the appropriate string usage of the generated arrangement. Explicitly, notes are automatically moved up or down an octave to fit onto the instrument.

Our strategy for creating arrangements is to maximize a function which assesses the competency of arrangements. In the next section we discuss research on the topic of using evolutionary computation for music generation. We shall also provide a brief introduction to guitar tablature and discuss TabGA, a genetic algorithm for creating guitar tablature which is vital to the evaluation function of our arranging system. In section three we define the arrangement evaluation function, and the fourth section describes a hill-climber and a genetic algorithm that attempt to optimize this function. In the final section we discuss the results of our experimentation with the automatic generation of arrangements.

4.2 Background

4.2.1 Academic Research

There has been an enormous amount of research into the automatic creation of music, particularly with techniques from evolutionary computation [4][25]. Successful unions of EC and music tend to be limited in scope, and much research is ongoing. One of the first papers on the subject, by Horner and Goldberg, used genetic algorithms for thematic bridging [7].

Two successful and often cited projects are those by Biles and Jacob. Biles created an Interactive Genetic Algorithm that could produce pleasing jazz solos with the aid of a human fitness function [2]. Jacob's system used a Genetic Algorithm to compose music using musical motives provided by a composer. The goal of all of these systems is to create music which should seem altogether new to the listener [8]. Our system, on the other hand, is designed to adapt existing music for performance on the guitar.

Work on creating arrangements of compositions automatically is rare. Jacob's system includes an "arranging" module, but the term is used in a different sense, that of arranging phrases into larger statements. Another system by Nagashima and Kawashima uses chaotic neural networks to "arrange" music, but by this they mean creating variations on melodies [12]. Neither system actually creates arrangements as the word is strictly defined in music. Perhaps this is not surprising because arranging is not a task one would normally be inclined to automate. The process of arranging for an ensemble or piano, for example, is predominantly creative because so few limitations are enforced by the instruments. It is because solo guitar is such a limiting instrument that the problem of automating arranging is so interesting. Ensuring that an arrangement is within the realm of playability is a complicated, yet relatively uncreative, task to which it is useful to apply computation [11][15][16][17].

We know of no other research pertaining to automatic arranging for the guitar or any instrument, and we believe this system to be the first of its kind.

4.2.2 The Guitar Fingering Problem and TabGA

Stringed instruments are unique in that nearly every note can be played in multiple positions on the instrument. Choosing between these positions can be a laborious process, and guitar players often use tablature to record exactly how a piece is to be played. Instead of notes, tablature encodes music as a sequence of positions on the fretboard. It does this by graphically representing each of the six guitar strings as opposed to the five lines of a music staff. For each note there is a number indicating the fret on which the note should be played,

and this note is placed on the appropriate string in the tablature. The tablature is then read from left to right by the performer.

Tablature information is essential to determining how difficult a piece of music is on guitar, or if it can even be played at all. For this reason, tablature is of paramount importance when arranging for guitar. A potentially beautiful arrangement of a piece of music is worthless if it cannot be played.

Previously, we developed a system called TabGA that reliably produces competent tablature for any piece of guitar music [18]. TabGA is a distributed genetic algorithm that evolves tablature for a given sequence of notes. The fitness function provides selection pressure on the basis of tablature playability. Both this function and the operating parameters of TabGA were optimized with meta-genetic algorithms in order to maximize the consistency of generated tablatures with those created by human experts. The result was a high level of consistency, above 90%, as well as good results even when the generated tablature diverged from published tablature. The system also employs a neural network to assign left-handed fingers to each note, though this feature is supplementary and not particularly relevant to this chapter. Both the GA and the fitness function are detailed in [19]. Tablature lends itself to very natural processing by search algorithms and can be encoded as a chromosome for a GA very easily. Each note, or chord if there are multiple notes played simultaneously, constitutes a "gene" in the tablature chromosome. The reproductive process used in TabGA is illustrated in Figure 4.1.

4.2.3 The Goal of our System

We are attempting to *preserve* beauty, not *create* it. To automate arranging we attempt to minimize the creative input of the system and maximize the creative input of the composer. The task of endowing a machine with creativity is a daunting one, and one which we do not wish to address in this research. The arrangements produced by our approach will not interweave new musical material with the original composition, but attempt to reproduce

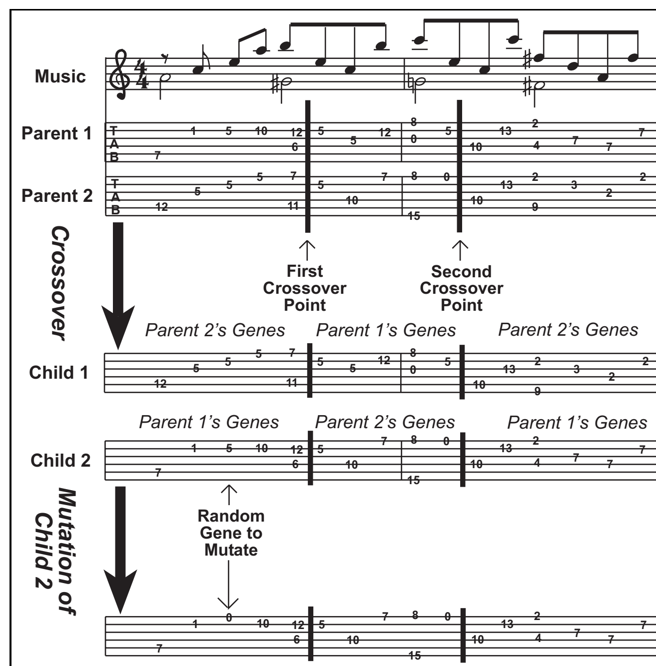

Figure 4.1: The creation of a child tablature from two parents in TabGA.

the original composition as accurately as possible while ensuring that the arrangement is below a reasonable difficulty level.

4.2.4 On the Subjectiveness of the Domain

The reader should be aware that we do not intend for our results to be precisely reproducible. We will include all the information which we consider to be insightful into the domain, and will omit some parameters which were chosen somewhat arbitrarily.

Further, we are not specifically devoted to the fitness function or search strategies that we shall describe. The components of our evaluation function are those which we considered to be important based on our current understanding of the arranging process. We do not expect that they are sufficient for a complete description of an arrangement's virtue, but believe that they are satisfactory for our purposes. Consequently, they will not at this time be the

subject of rigorous experimentation. Should our system demonstrate an ability to generate pleasing arrangements of compositions, we are disposed to believe that any system employing a similar evaluation function and search strategy would demonstrate similar competency.

4.3 The Evaluation Function for Assessing an Arrangement

As noted above, we do not expect our approach to have a fine-grained ability to differentiate between the competence of arrangements, which is the target concept of this domain. We have a set of heuristics by which we judge an arrangement, and expect only that these heuristics are sufficient to be effective. An effective heuristic has the ability to distinguish the good arrangements from the poor. We consider this to be a worthwhile goal, because even two humans who can tell the difference between good and bad arrangements could easily disagree on which of two good arrangements is superior.

The evaluation function described in this section is a first attempt, and will no doubt be subject of improvement in the future. It is integer based, but there is no technical limit on the scale of fitness values and we do not normalize them. The function is designed to assess arrangements, and to so we attempt to satisfy two objectives. These are to determine the playability of the tablature and the degree to which an arrangement preserves the composer's original intent. There are four components satisfying these two objectives, and these are weighted and then added together to assess the overall competence of an arrangement.

4.3.1 Maintaining Playability

In producing guitar arrangements, one is limited by the abilities of the human hand. A good arrangement must not exceed the dexterity or reach of the fingers. To assess difficulty, TabGA is given a candidate arrangement and returns a viable tablature for the arrangement. TabGA also returns the fitness value of the tablature, which corresponds to the difficulty of the tablature. This value is the first component of our fitness function and is referred to as the Playability Component (PC). A higher PC value indicates an easier tablature.

Figure 4.2: An excerpt from Rachmaninoff's Symphony No. 2 that contains examples of 5 of the 6 note categories that our system recognizes.

4.3.2 Preserving the Original Composition

The second objective is to maximize the extent to which the arrangement sounds like the original piece of music. In order to do this effectively we acknowledge that not all notes are created equal. Melodic notes are regarded as essential whereas others are supplementary and are important to varying degrees. We currently allow for six categories of note, five of which are illustrated in figure 4.2. We distinguish notes belonging to the melody, harmony, counter-melody, and bass. We have two additional categories for notes in the harmony and bass that are constituents of moving lines. Each category of note has an associated weight that represents how important it is to the piece of music. For our experimentation we specify note types by hand, but only because we have not written an algorithm to read music. Techniques exist for automating this task, especially for melody and bass lines [21][10]. In designing a function to judge arrangements, we have consulted a book by Corozine on general arranging practices and a book by Michaels on arranging specifically for the guitar [3][9]

The weight for each note is the extent to which its inclusion benefits the fitness of the arrangement and is referred to as the Note Weight Component (NWC).

$$NWC = \sum_{i=0}^{N} incl(i) \times weight(i)$$

Where N is the number of notes in the original piece and *incl(i)* evaluates as one if the note is included in the arrangement and zero otherwise. The weight associated with note i is returned by *weight(i)*.

We also recognize the importance of contiguous notes in the moving lines, so there is a fitness bonus when consecutive notes in a moving line are included. This is referred to as the Moving Line Continuity Component (MLCC).

$$MLCC = \sum_{i=0}^{L} \sum_{j=0}^{N_i} incl(j) \times incl(j-1)$$

Where L is the number of moving lines within the phrase, N_i is the number of notes in line i and *incl(j)* evaluates as one if note j is included in the arrangement and zero otherwise.

A fitness reward is also assigned relative to the number of notes in each chord. This factor benefits an arrangement based on the number of notes from each chord included. However, the incremental increase in the reward for each additional note decays exponentially. The first note included in a chord increases the bonus for that chord far more than the fourth or fifth. In this way we hope to maintain an even distribution of notes throughout the arrangement. This will help discourage the arrangement from becoming intermittently thick. We use the term "thick" to describe arrangements which have so many notes that important musical content is obscured, and "thin" those arrangements which have too few notes drawn from the original composition. We hope to prevent both as far as possible with the Notes in Chord Component (NCC).

$$NCC = \sum_{i=0}^{C} 1 - e^{-.5 \times \sum_{j=0}^{O_i} incl(j)}$$

Where C is the number of chords in the original piece of music. The decay function is shown in Figure 4.3, and approaches a bonus of 1.0 for six notes (the maximum possible given the six strings of the guitar).

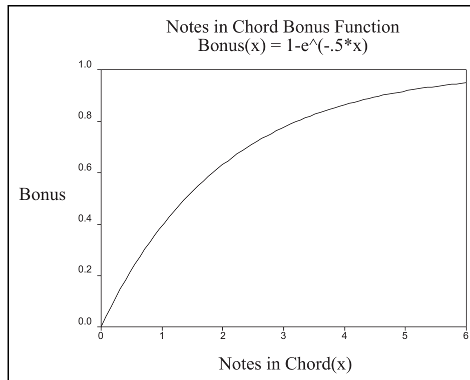

Figure 4.3: The exponential decay function that determines the reward for each note in a chord.

The fitness, or competence, of an arrangement is then simply:

$$Fitness = PC + NWC + MLCC + NCC$$

It should be noted that each of the four components are weighted, and these weights are fully intended to be adjustable at the user's discretion. The default weights are 1.0 for PC (which tends to be in the range of -1000 to 500, though there is no enforced limit), 16.0 for NWC, 1.0 for MLCC and 5.0 for NCC. The default weights for the note types are 2.5 for bass line notes, 1.0 for counter-melody notes (which often are also boosted by MLCC) and 1.0 for notes in the harmony. Melodic notes are unweighted because they are necessarily included in the arrangement. These weights have been subject to only brief experimentation, and are intended to be adjustable to insure that components do not overpower each other in fitness assessments. We would not, for example, wish to place too high an importance on playability. This would result in unnecessarily thin arrangements because it is easier to play fewer notes. Likewise, placing too much importance on faithfulness to the original piece may result in an unplayable arrangement with far too many notes. Adjusting these weights also allows the system to produce arrangements for users of varied ability levels, most effectively

by increasing or decreasing the weights associated with the Playability Component and the Note Weight Component. These two components are primarily responsible for controlling the balance between ease of performance and the number of notes included.

4.4 Methods for Obtaining Arrangements

There follows a description of two methods for selecting notes to maximize the function described above. The first is a greedy hill-climber, which attempts to mimic (to a modest extent) the human strategy of arranging. The hill-climber considers the most important notes for inclusion before any others, and every note found to increase the fitness becomes a permanent addition. This biases the search even further in favor of those notes which were already considered to be more important. We also have implemented a genetic algorithm, simply because GAs are often better suited to function optimization in general. The representation for both the genetic algorithm and the hill-climber is a binary string where each bit represents a note in the original piece of music. A one indicates that the note is included in the arrangement while a zero indicates its absence.

4.4.1 Arranging Music with a Greedy Hill-Climber

In this model we attempt to arrange the piece starting with the most important notes and ending with the least important. This is accomplished by setting a minimum allowable weight, and decreasing this weight whenever all the notes at or above that weight have been given a chance for inclusion without success. Whenever a note is found to improve the fitness of the arrangement, it becomes a permanent addition to the arrangement. Because of this, we consider the algorithm to be greedy. Whenever a note is added, all the other notes considered to be of equal or greater importance (as determined by weight) are given another chance to be included, effectively restarting the search at the current minimum allowable weight level. The algorithm is described by the following instructions.

1. Set a high minimum allowable note weight.

2. Attempt to add a note that is more heavily weighted than the current minimum allowable weight.

3. If the note increases the fitness of the arrangement, it becomes a permanent addition. Every note above the current minimum allowable weight is now eligible to be tried again, go to **2**.

4. Else, try the subsequent note above the current minimum allowable weight.

5. If every note has been tried without any improvement, decrease the minimum allowable weight and go to step **2**.

6. If the minimum allowable weight is 0, and every note has been tried, we have reached a local optimum.

Each time a note is accepted into the arrangement, we restart the hill-climber at a random point in the bit string and regard the bit string as circular. This is to avoid biasing the arrangement towards notes at the beginning, which would receive more opportunities to be included if the algorithm began at the beginning every time a note was accepted.

4.4.2 Arranging Music with a Genetic Algorithm

Because experimentation is exceedingly time-consuming in this domain (results must be played, or at last looked at, by a guitar player to be judged correctly), the parameters of our GA have been chosen somewhat arbitrarily. Our GA is steady-state, so only one individual is replaced in each iteration [23]. New individuals are created by two-point crossover and mutation occurs at every bit with a probability of 3%. Parents are chosen with binary tournament selection. Each piece of music is broken into logical phrases so that the search space remains manageable. Points in the music which allow the performer a great deal of time to move his/her left hand are ideal for segmentation. Methods exist for performing this segmentation automatically for the guitar by Radicioni et al. [14], though we currently

perform the segmentation ourselves. The GA is then run on each phrase individually, and the results of each are appended together for the final solution. Once the first phrase has been arranged, each subsequent phrase includes the fretboard positions of the last three chords of the previous phrase. In this way, the GA has information about where on the fretboard the performer's hand is at the end of the previous phrase and can create tablature for the subsequent phrase accordingly. It should be noted that all melodic notes are always included (i.e., set to one in every chromosome) and as such do not contribute to the dimensionality of the search space. We run the GA for 1000 iterations (replacements).

Because each fitness evaluation requires running TabGA to create tablature, the GA requires on the order of an hour to create an arrangement of a lengthy phrase. The greedy hill-climber required far less execution time, about 15 minutes to reach a local optimum. It should be noted that execution time for both varies with phrase length.

4.5 Results

Neither system for arranging, nor the fitness function, have as yet been subjected to rigorous experimentation. Arrangements require several minutes to create with the hill-climber, and more than an hour with the genetic algorithm. Consequently, we have had time only to run the algorithms on a limited set of music and appraise the initial output. We selected three pieces of music on which to test our approach. These are the moderato from the second movement of *Symphony No. 2* in E Minor by Sergey Rachmaninoff , a selection from *Jupiter* by Gustav Holst, and the beginning of *Arioso* from Cantata No. 156 by J.S. Bach [13][6][1]. The former two are pieces for full orchestra, the third is adapted for four trombones. We began with the Rachmaninoff, and in our first run created an arrangement that was too difficult for an intermediate guitarist. To bias the search more towards playability we decreased the weight given to the Note Weight Component by 25%. The second run produced an arrangement which we found to be quite acceptable. Part of this arrangement is shown in figure 4.4. This same configuration produced an acceptable arrangement of the Holst excerpt

on the first try. The first run for the Bach gave us an arrangement that seemed too thin and was very easy, so we reset the Note Weight Component to its original value and ran it again. The second run gave us a pleasing arrangement with a reasonable tablature.

The arrangement in figure 4.4 was obtained with the greedy hill-climber. This results in local optima which are, on average, of lower fitness than those found with the GA. After obtaining our arrangements for both pieces we ran the genetic algorithm with the same weights. The arrangements obtained with the GA were, in our estimation, of roughly the same quality as those obtained with the hill-climber even though the fitnesses were better. We attribute this to the coarseness with which our fitness function can differentiate arrangements. We believe that although it can reliably discern the good arrangements from the bad, it can not reliably distinguish the good from the excellent or the bad from the awful. Any improvements made to the fitness function in the future will likely enhance the legitimacy of the GA.

We are pleased with the results of our experimentation. We were able to obtain arrangements as good as those we would have been able to create ourselves yet in a fraction of the time. For further results, see [20]. This website provides examples of arrangements produced by our system, both as tablatures and as mp3 recordings.

4.6 Conclusions and Further Directions

Our system would benefit from further automation. Currently, the notes of the original composition must be entered into the computer together with their appropriate categories, which takes about 15 minutes for a 30 second phrase. There is reason to believe that both of these tasks, particularly the former, could be handled automatically, potentially eliminating the need for user input. Indeed, methods have already been described in other research [22][21].

There are several contexts in which we believe our system could be useful. For the user uninterested in arranging music themselves, the internet is a superb resource for free and

Figure 4.4: An arrangement of an excerpt from Rachmaninoff's Symphony No. 2 by our system. We use either a greedy hill-climber or genetic algorithm (in this case, the former) to generate an arrangement from the orchestral score. The tablature for the arrangement is produced by TabGA.

easily accessible MIDI files for almost any well-known piece of music. MIDI files are electronic versions of music from which can be read all of the information used to create arrangements with our system. Combined with software for extracting this information, one could quickly obtain arrangements for any piece of music for which MIDI or sheet music can be found.

Those who wish to have more control over the arranging process can also benefit from our system. A user can arrange the piece themselves, then set all the notes in their own arrangement as being "essential" (or melodic, in our system) and run the search. This will likely add notes to the existing arrangement, potentially enriching the sound. Users can also search for the best guitar tuning and key for the piece, using simple algorithmic methods for adapting the input and running several searches. The system can be used as a source of musical ideas. A user can easily scavenge the output of the system for useful material that might not have occurred to them in the course of arranging a piece themselves.

We have described an evaluation function that is accurate enough to produce arrangements which sound good. However, there is much room for improvement. The four components we have detailed here are certainly not a complete account of tablature competency and would benefit from further investigation.

Bibliography

[1] J.S. Bach, *Arioso from Cantata No. 156, arr. Patrick McCarty for four trombones*, Rochester, NY: Ensemble Publications, 1961.

[2] J.A. Biles, "GenJam: A genetic algorithm for generating jazz solos," *Proc. International Computer Music Conference*, Aarhus, Denmark, 1994, pp.131–137.

[3] V. Corozine, *Arranging music for the real world: classical and commercial aspects*, Pacific, MO: Mel Bay, 2002.

[4] A. Gartland-Jones and P. Copley, "The Suitability of Genetic Algorithms for Musical Composition," *Contemporary Music Review*, vol. 22(3), pp. 43–55, 2003.

[5] H. Heijink and R.G.J. Meulenbroek, "On the complexity of classical guitar playing: functional adaptations to task constraints," *Journal of Motor Behaviour*, vol. 34(4) pp. 339–351, 2002.

[6] G. Holst, *The Planets*, London, England: Boosey and Hawkes, 1921

[7] A. Horner and D.E. Goldberg, "Genetic algorithms and computer-assisted music composition," *Proc. Fourth International Conference on Genetic Algorithms*, San Francisco, USA, 1991, pp. 479–482.

[8] B.L. Jacob, "Composing with Genetic Algorithms," *Proc. International Computer Music Conference*, Banff Alberta, Sept. 1995, pp. 452–455.

[9] D. Michael, *Arranging for Open Guitar Tunings*, Anaheim Hills, CA: Centerstream Publishing, 2003.

[10] C. Meek and W.P. Birmingham, "Automatic Thematic Extractor," *Journal of Intelligent Information Systems: Special Issue on Music Information Retrieval*, vol. 21(1), pp. 9–33, July 2003.

[11] M. Miura, I. Hirota, N. Hama, and M. Yanigida, "Constructing a System for Finger-Position Determination and Tablature Generation for Playing Melodies on Guitars," *Systems and Computers in Japan* vol. 35(6), pp. 755–764, 2004.

[12] T. Nagashima and J. Kawashima, "Experimental Study on Arranging Music by Chaotic Neural Network", *International Journal of Intelligent Systems*, vol. 12, pp. 323–339, 1997.

[13] S. Rachmaninoff, *Symphony No. 2 in E Minor, Op. 27, in Full Score*, Mineola, NY: Dover Publications, 1999.

[14] D. Radicioni, L. Anselma and V. Lombardo, "A segmentation-based prototype to compute string instruments fingering," *Proc. Conference on Interdisciplinary Musicology*, Graz, Austria, 2004.

[15] D. Radicioni and V. Lombardo, "Guitar Fingering for Music Performance," *Proc. International Computer Music Conference*, Barcelona, Spain, Sept. 2005, pp. 527–530.

[16] A. Radisavljevic and P. Driessen, "Path Difference Learning for Guitar Fingering Problem," *Proc. International Computer Music Conference*, Miami, USA, Nov. 2004.

[17] S. Sayegh, "Fingering for String Instruments with the Optimum Path Paradigm", *Computer Music Journal*, vol. 13(6), pp.76–84, 1989.

[18] D.R. Tuohy and W.D. Potter, "A Genetic Algorithm for the Automatic Generation of Playable Guitar Tablature," *Proc. International Computer Music Conference*, Barcelona, Spain, Sept. 2005, pp. 499-502.

[19] D.R. Tuohy and W.D. Potter, "Creating Guitar Tablature with Neural Networks and Distributed Genetic Search,", to appear in *Proc. International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems* Annecy, France, 2006.

[20] D.R. Tuohy, *Tablature and mp3s of arrangements generated by genetic algorithm*, http://www.ai.uga.edu/tuohy/excerpts.html.

[21] H.-H. Shih, S.S. Narayanan, and C.-C.J. Kuo, "Automatic main melody extraction from MIDI files with a modified Lempel-Ziv algorithm", *Proc. International Symposium on Intelligent Multimedia, Video and Speech Processing*, May 2001.

[22] J. Wand and L. Tsai-Yen, "Generating Guitar Scores from a MIDI Source", *Proc. International Symposium on Multimedia Information Processing*, Taipei, Taiwan, 1997.

[23] D. Whitley and J. Kauth, "GENITOR: A Different Genetic Algorithm," *Proc. Rocky Mountain Conference on Artificial Intelligence*, Denver, CO, 1988, pp. 118–130.

[24] D. Whitley and T. Starkweather, "GENITOR II: a distributed genetic algorithm", *Journal Expt. Theor. Artif. Intel.*, vol. 2, pp. 189–213, 1990.

[25] G. Wiggins, G. Papadopolous, S. Phon-Amnuaisuk, and A. Tuson. "Evolutionary Methods for Musical Composition", *International Journal of Computing and Anticipatory Systems*, vol. 4, 1999

Chapter 5

An Evolved Neural Network/HC Hybrid for Tablature Creation in GA-based Guitar Arranging

5.1 Introduction

In this chapter[1] we describe a technique for creating guitar tablature using a neural network. Training data was parsed from an online repository of human-created tablatures. The contents of both the input layer and the set of training data have been optimized through genetic search in order to maximize the accuracy of the network. The output of the network is improved upon with a local heuristic hill-climber (HC). We implement this model in an existing system for generating guitar arrangements via genetic algorithm (GA). When compared to the original system for generating tablature, we note modest improvement in tablature quality and drastic improvements in execution time.

Tablature is a notational system that describes to a guitarist how a piece of music should be played. Because notes can be played at several different positions on the instrument, playing directly from sheet music requires tedious decision making about where best to play each note. Tablature frees the performer from this responsibility by displaying fretboard finger positions in lieu of actual notes. We employ an evolved neural network model with local heuristic search for the purpose of generating guitar tablature for a piece of guitar music.

We have implemented this model for tablature generation into a system for automatically arranging pieces of music not originally written for the guitar. The process of arranging

[1]Tuohy, Daniel R., and W.D. Potter. 2006. Submitted to 2006 International Computer Music Conference.

frequently requires eliminating notes in the original composition so that the arrangement is not too difficult to perform. We use a genetic algorithm to find the most relevant selection of notes from the original piece that does not burden the performer excessively. In doing so, the GA requires our model for tablature generation in order to create tablatures corresponding to candidate arrangements. These tablatures are necessary in order to assess the difficulty of the arrangements.

In section two we describe previous work on both the task of tablature creation and that of arrangement generation. Section three details the evolved neural network model for generating tablature and a hill-climbing search for improving those tablatures. In section four we discuss the implementation of the tablature generation model as part of a genetic algorithm for automatically arranging music. Finally, we summarize the improvements the new model has made in terms of both speed and reliability.

5.2 Background

Generating tablature for the guitar that is competitive with tablature created by human experts is a difficult task. It has often been treated as an NP-hard graph-search problem, based on the research of Samir Sayegh [7], [5], [6]. Another method used genetic algorithms to evolve playable tablature [9]. These models for tablature creation are dependent upon an evaluation function. Defining such a function for this domain is exceedingly difficult because of the innate complexity of the human hand [3]. We conjecture that learning how to create tablature directly from human-defined tablature will yield a more reliable, human-like method.

We do, however, require an evaluation function to assess the difficulty of the tablatures generated by this model. Previously, we developed a GA for generating arrangements, specifically reductions, of music to allow the music to be played on guitar [11]. This GA evaluates arrangements partially on the basis of the difficulty of the corresponding tablature. For the current project, we evaluate tablature difficulty with the function originally used for fitness

evaluations in a GA for generating tablature. This function will also guide a brief local search on the tablatures generated by our neural network method.

5.3 Neural Network Model

The approach to tablature generation presented here constructs tablature sequentially with a neural network (i.e. one note at a time). A danger inherent in this approach of processing tablature sequentially is that when placing a note on the fretboard, it is often beneficial to be aware of where subsequent notes will be played. With this in mind, we are careful to construct a set of network inputs that encompasses as much relevant information about the context of a note as is feasible. This includes data describing the notes both preceding and, crucially, those succeeding a note. We have chosen to implement a standard three layer feed-forward network with sigmoid activation function trained with error back-propogation.

The network has 20 outputs with a continuous range between 0 and 1. These correspond to 19 frets on the guitar and the open strings. The fretboard selected for a note by the network is the viable position nearest to the output with the highest value. Networks were constructed using a Neural Network package from the University of Georgia [8].

5.3.1 Training Data

Our data are taken from a popular online library of tablature for classical guitar [1]. We have selected excerpts from 75 different pieces of music, constituting a wide variety of styles and difficulty levels. We chose to use tablature as data because it is readily available in ASCII format, which allows for the generation of a large training base of patterns. It does however, deny us access to timing information, which is present in traditional sheet music but not in tablature notation. From the 75 excerpts we generated 1853 patterns, one for each note and corresponding fretboard position.

5.3.2 Network Inputs

A training pattern corresponds to information about the context of a single note and consists of 64 inputs. 41 of these pertain to notes played previous to or simultaneously with the note being processed. For each pattern, we generate the number of notes in the current chord, a number between one and four corresponding to the "octave" of the current note, a number between one and 12 corresponding to the notes C through D#, the octave and note of the highest and lowest notes in the chord, the average fret of the notes in the previous three chords, the fret and the string used the last time the note occured in the piece, the fret and string of the last four notes, the number of chords played since the last four notes were played, the number of notes in the previous two chords, a binary value for each string indicating which strings are already in use in the current chord, and all of the possible frets and strings of the current note.

23 additional inputs comprise data pertaining to notes that have yet to be assigned fretboard positions. These are the number of notes in the next two chords, the octave and note of the next six notes, the number of chords between the current note and the next six notes, and the fret and string furthest from the guitar body of the nearest subsequent limiting note. A limiting note is one which can only be played in one position, or can only be played on the sixth fret or higher. In this way, the network has information about future notes which require the hand to be at or above a certain place on the fretboard. The final input indicates how many chords remain until the next limiting note must be played. These inputs are all those which we thought could possibly be informative.

5.3.3 Evolving the Network Inputs and Training Data

By "evolve", we do not mean that the weights of our network are optimized with genetic search [13]. Weights are set using a standard back-propogation algorithm. Rather, genetic search is used to select from among all possible inputs those which are truly informative, and to select from 75 possible excerpts which 65 result in the most accurate trained network

when used as training data. We are, in effect, evolving the input layer of the network as well as the data set on which the network is trained. Irrelevant inputs and noisy training data can potentially confuse a neural network, and we have no reason to believe that our data are devoid of either. We use GAs to optimize bit strings denoting which inputs and which training excerpts are used. We have done so using GAs based roughly on Whitley's GENITOR model [14]. We have opted to use GAs rather than less computationally intensive methods because a network can be fully trained in less than three minutes.

The fitness of a bit string is the accuracy of the corresponding trained network. Accuracy is the percentage of fretboard positions selected by the network that coincide with those in published tablatures. To determine which fretboard position is selected by the network, we first determine which output node has the highest value. The fret corresponding to this node is then compared with all viable positions for that note, and the position nearest to this fret is considered to be the output of the network.

Each network is trained until no decrease in the mean squared error on an independent test set is found in 400000 learning events, calculating error at intervals of 200 events. One GA was used to optimize a bit string of length 64, corresponding to the 64 possible inputs. Another optimized a bit string of length 75, with the requirement that the bit string denote the inclusion of at least 65 excerpts as training data.

5.4 Implementing the Neural Network for Tablature Creation

In order to generate a tablature from a sequence of notes, we generate all of the data comprising an input sequence for the first note in the piece. These data are run through the feed-forward network and the node with the value closest to one is taken to be the output. The fretboard position appended to the tablature is the one nearest to the output of the network. This process is repeated for every note in the piece of music, in sequence.

Chords are processed recursively. Because the network could make a mistake in placing the first notes it encounters, we recursively generate every possible chord configuration that

Figure 5.1: A tablature is created by neural network. The last note is "fixed" with local search.

can be reasonably played. The chord inserted into the tablature is the one which is most consistent with the network's output. We judge consistency to be the cumulative difference between the assigned fretboard positions and the output of the network.

5.4.1 Improving network-generated tablatures with hill climbing

After evolving both the input layer and the training set, our network has an accuracy of 94.0% on the 65 excerpts selected via genetic search. This indicates that for 94.0% of the notes were assigned the same position on the fretboard that was present in the published tablature. This degree of consistency is such that the network reliably produces tablature similar or identical with that created by humans. However, the network is susceptible to rare cases of bad judgement, and tablatures occasionally include an obviously misplaced note or chord.

To correct this, we run a brief local search that passes over each note in the generated tablature a single time. For each note or chord, every possible fingering configuration is tested to see if it improves the tablature. To test improvement, we use an evaluation function previously used as the fitness function in a GA for tablature creation. A complete description of this evaluation function, which was optimized with a meta-GA to generate tablature consistent with published tablature, can be found in [10]. We only pass over the tablature one time so that the neural network's influence is not over-powered by that of the evaluation function. We have found that using the evaluation function for brief hill-climbing fixes obvious mistakes, but using it for a full hill-climbing optimization decreases consistency with published tablatures. Of particular interest is that even a single pass of the hill-climber decreased overall consistency on our data, although only slightly. This suggests that the network, despite it's occasional mistakes, is more adept at creating tablature than an algorithm for optimizing the evaluation function. However, we must acknowledge that hill-climbing is indeed useful because it corrects obvious mistakes. Inconsistency with published tablature that is introduced by the local optimization tends to be more harmless than the mistakes that are corrected. Figure 5.1 shows an example of a tablature generated for a piece using the neural network, with one of the fretboard positions "fixed" using local search.

5.5 Using generated tablatures in arrangement evaluation

In [11] we describe a genetic algorithm that automatically generates arrangements for a piece of music. Fitness is a function of both playability (as defined by the aforementioned tablature evaluation function) and on "faithfulness" to the original composition. Briefly, "faithfulness" is determined on the basis of the number and importance of notes that are included. A note's importance is related to the part of the music it occupies. For example, notes in the melody are considered to be more important than those in the harmony. Fitness evaluation is also biased to promote continuity in moving lines, and to promote an appropriate distribution of

notes throughout the arrangement [2], [4]. A complete mathematical account of this function can be found in [11].

The playability of an arrangement is defined as the fitness of the corresponding tablature according to the same evaluation function we use for hill-climbing. Using orchestral scores as the source of potential notes, we then generated arrangements of excerpts from *Symphony No. 2* by Rachmaninoff, *Arioso* by Bach, and *Jupiter* by Gustav Holst. Figure 5.2 illustrates the process for arranging part of an excerpt from the Rachmaninoff, with notes omitted from the arrangement in gray.

5.6 Results

We compare our neural network model to a GA model for tablature creation because it is the only one for which we have significant performance data. All techniques were tested on a set of 65 excerpts constituting 1717 notes. Tablatures generated with a GA model that optimizes our evaluation function are 86.9% consistent with published tablature. This increases to 90.8% using the neural network with full hill-climbing. Using only brief hill-climbing, consistency increases to 91.1%. Using the network alone, consistency is even higher at 91.4% but has occasional obvious mistakes. This number is lower than the 94% achieved by the network on notes individually. This is because when a generated tablature diverges from a published tablature, the appropriate positions of notes subsequent to the point of divergence can be different than they would have been had there been no divergence.

The advantage in speed is more obvious. The average time necessary to create tablature for the Rachmaninoff piece in Figure 5.2 by GA was 5.7 seconds. To create tablature for the same piece using the neural network with local search was 55.6 milliseconds. This difference is much more noticeable when *arranging* the excerpt. An excerpt or phrase can be arranged by GA using a neural network for tablature creation in under a minute, while it would take an hour or more if tablature creation were handled with a GA.

Figure 5.2: Generating an arrangement from a score.

Tablature and mp3 recordings of the arrangements created, as well as the tablature data set which we used, are available at http://www.ai.uga.edu/tuohy/excerpts.html.

5.7 Further Directions

We are pleased with the performance of our network, but there is room for improvement. Because we have trained our network on tablature, our network does not have access to timing information when making decisions. A network would likely perform better if it had tempo and note length information to work with, perhaps by extracting data from MIDI files and specifying correct tablature by hand [12].

Bibliography

[1] Classtab.org (2006). Classical Guitar Tablature. http://www.classtab.org

[2] Corozine, V. (2002). *Arranging music for the real world: classical and commercial aspects*. Pacific, MO: Mel Bay.

[3] Heijink, H. and R. Meulenbroek (2002). On the complexity of classical guitar playing: functional adaptations to task constraints. *Journal of Motor Behaviour 34*(4), 339–351.

[4] Michael, D. (2003). *Arranging for Open Guitar Tunings*. Anaheim Hills, CA: Center Stream Publishing.

[5] Radicioni, D. and V. Lombardo (2005). Guitar fingering for music performance. In *Proceedings of the International Computer Music Conference*, pp. 527–530. International Computer Music Association.

[6] Radisavljevic, A. and P. Driessen (2004). Path difference learning for guitar fingering problem. In *Proceedings of the International Computer Music Conference*. International Computer Music Association.

[7] Sayegh, S. (1989). Fingering for string instruments with the optimum path paradigm. *Computer Music Journal 13*(6), 76–84.

[8] Smith, B. (2004). Gail: Georgia artificial intelligence library neural network package. *University of Georgia*.

[9] Tuohy, D. and W. Potter (2005). A genetic algorithm for the automatic generation of playable guitar tablature. In *Proceedings of the International Computer Music Conference*, pp. 499–502. International Computer Music Association.

[10] Tuohy, D. and W. Potter (2006a). Generating Guitar Tablature with LHF Notation via DGA and ANN. In *Proceedings of the IEA/AIE*. International Society of Applied Intelligence.

[11] Tuohy, D. and W. Potter (2006b). Ga-based music arranging for the guitar. In *Proceedings of the 2006 Congress on Evolutionary Computation (to appear)*. http://www.ai.uga.edu/tuohy/arranging.pdf

[12] Wand, J. and L. Tsai-Yen (1997). Generating guitar scores from a midi source. In *Proceedings of the International Symposium on Multimedia Information Processing*.

[13] Whitley, D. (1995). Genetic algorithms and neural networks. *Genetic Algorithms in Engineering and Computer Science*, 1–15.

[14] Whitley, D. and J. Kauth (1988). Genitor: A different genetic algorithm. In *Proceedings of the Rocky Mountain Conference on Artificial Intelligence*, pp. 118–130.

Chapter 6

Conclusion and Further Directions

The Neural Network/Hill-climber hybrid described in chapter 4 is our current method for tablature generation, and we believe it to be quick and reliable. An obvious enhancement that we have not implemented is to include information about note-length and tempo as input to the network. We have omitted this information because our training data was in tablature form, which often lacks such information.

We consider the two methods for arrangement evaluation explored in chapter 3 to be of roughly equal quality. Both were able to generate pleasing arrangements on the 3 pieces of music on which they were tested (tablature for all three pieces can be found in Appendix B). The arranging methods would benefit most noticeably from further research into the arrangement evaluation function. In its current form, the function offers only rough insight into the quality of an arrangement.

Appendix A

Source Code

The following code was written in Java (version 1.5).

A.1 Generating a Tablature with a Neural Network

For this tablature creation scheme, we store a piece of music as a list (class NoteList) of notes (class NoteNode). Each NoteNode has the following variables associated with it:

- **int** string: An integer between 1 and 6 designating the string on which the note is played, this will be filled in by the network.
- **int** fret: An integer between 0 and 20 designating the fret on which the note is played, this will also be filled in by the network.
- **int** index: An integer indicating the chord that the note belongs to, this allows us to identify which notes are played simultaneously.

```
//A procedure that traverses the NoteList chord by chord. Calls
//the tabANote method if the chordhas only one note, and the
//tabAChord method otherwise.
public void createTab(){
   boolean done = false;
   NoteNode cur = list.first;
   int i = 0;
   while(!done){
     //find beginning and end of chord, store list index of the last node in e
     NoteNode trav = cur;
     int e = i;
     //find end of chord
     if(trav!=list.last){
```

```
      while (trav.index == trav.next.index) {
        trav = trav.next;
        e++;
        if (trav == list.last) {
          break;
        }
      }
    }

    //tab all notes from cur through trav (the first and last notes of the current chord)
    if(i-e==0){
      tabANote(cur,i);
    }
    else{
      tabAChord(cur, trav, i, e);
    }

    //if last note of chord is the last note of the piece, done = true
    if(trav==list.last){done = true;}

    //go to next chord
    cur = trav.next;
    i=e+1;
  }
}


/*This method initializes some parameters for use with a recursive tabbing method.
 *@param b is the first NoteNode of the chord
 *@param e is the last NoteNode of the chord
 *@param f is the NoteList index of the first note of the chord
 *@param l is the NoteList index of the last note of the chord
 */
public void tabAChord(NoteNode b, NoteNode e, int f, int l){
  //run a recursive function that ALWAYS fails, and, in the base case
  //assigns new fretPositions to the chord notes if they are more desirable
  //in this way, it tries every possible chord configuration, not biasing
  //towards the lower notes


  //for use in recurseChord, keeps track of which strings are in use
  int[] stringsUsed = new int[6];

  //for use in recurseChord, keeps track of the current best fretboard positions
  int[] bestFBPs = new int[(l-f+1)*2];

  //keeps track of bestFBPs' fitness, attempting to minimize
```

```
  bestChordFit = 10000;

  recurseChord(b, e, f, l, 0, b, f,stringsUsed);
  NoteNode trav = b;
  //assign the best fretboard positions found to the chord
  for(int j = 0;j<=(l-f)*2;j=j+2){
    trav.fret = bestFBPs[j];
    trav.string = bestFBPs[j+1];
    trav = trav.next;
  }
}


/*A recursive method that tries every viable combination of fretboard
 *positions, returning the set that is most consistent with the output
 *of the network (minimum value of cumDiff).
 *This is to avoid biasing the assignment of fretboard
 *positions in favor of the first notes in the NoteList which
 *could occupy strings better suited for subsequent notes
 *@param begin is the first NoteNode of the chord
 *@param end is the last NoteNode of the chord
 *@param first is the NoteList index of the first note of the chord
 *@param last is the NoteList index of the last note of the chord
 *@param curNode is the NoteNode currently being tabbed
 *@param curInd is the NoteList index of the NoteNode being tabbed
 *@param stringsUsed keeps track of strings already have been assigned notes
 *in the current chord
 */
public void recurseChord(NoteNode begin, NoteNode end, int first, int last,
                         int cumDiff, NoteNode curNode,
                         int curInd, int[] stringsUsed){

  //get the input pattern
  fretPosition[] curFPs = t.getFPs(notes[curInd]);
  String s = getPattern(curNode,curInd);

  //if there is only one option, install it and proceed
  int netOutput = 0;
  if(s==""){
    curNode.fret = curFPs[0].fret;
    curNode.string = curFPs[0].string;
    netOutput = curFPs[0].fret;
  }
  else{

    //convert string into an input pattern for the network
    Pattern p = interpretString(s);
```

```
  //find the output of the network (Fire_TabNN), which
  //is the output node with the highest value.
  double[] outputs = Fire_TabNN(p.inputs);
  for (int o = 0; o < outputs.length; o++) {
    if (outputs[o] > outputs[netOutput]) {
      netOutput = o;
    }
  }
}

//if we are at the last note in the chord, we are at the bottom of the recursion
if(curInd==last){

  //try every fretboard position iteratively and evaluate fitness
  for (int i = 0; i < curFPs.length; i++) {

    //if the string of this FPB is not already in use in the chord
    if (stringsUsed[curFPs[i].string - 1] == 0) {

      //add the difference between the current fret and the output of
      //the network to cumDiff
      int fit = cumDiff + Math.abs(netOutput - curFPs[i].fret);
      curNode.fret = curFPs[i].fret;
      curNode.string = curFPs[i].string;

      //add a component for chord playability, this punishes
      //an FPB set for requiring the hand to stretch over too
      //many frets
      NoteNode playTrav = begin;
      int minFret = 25;
      int maxFret = playTrav.fret;
      for(int p = 0;p<=(last-first);p++){
        if(playTrav.fret>maxFret){maxFret = playTrav.fret;}
        if(playTrav.fret<minFret&&playTrav.fret!=0){minFret = playTrav.fret;}
        playTrav = playTrav.next;
      }
      if(minFret!=25){
        fit = fit + (maxFret - minFret);
        if (maxFret - minFret > 3) {
          fit = fit + 5;
        }
        if (maxFret - minFret > 4) {
          fit = fit + 20;
        }
      }
```

```
        //if fit is the best so far, store FBPs in bestFBPs
        if(fit<bestChordFit){
          bestChordFit = fit;
          NoteNode trav = begin;
          for(int j = 0;j<=(last-first)*2;j=j+2){
            bestFBPs[j] = trav.fret;
            bestFBPs[j+1] = trav.string;
            trav = trav.next;
          }
        }
      }
    }
  }

  //we are not at the last note, recurse for every viable fretboard position
  else{
    for (int i = 0; i < curFPs.length; i++) {
      if (stringsUsed[curFPs[i].string - 1] == 0) {
        int newCumDiff = cumDiff + Math.abs(netOutput - curFPs[i].fret);
        int[] sUsed = (int[]) stringsUsed.clone();
        sUsed[curFPs[i].string - 1] = 1;
        curNode.fret = curFPs[i].fret;
        curNode.string = curFPs[i].string;
        NoteNode newCur = curNode.next;
        int newInd = curInd+1;
        recurseChord(begin, end, first, last, newCumDiff, newCur,
                     newInd, sUsed);
        sUsed[curFPs[i].string - 1] = 0;
      }
    }
  }
}

/*Assign a fretboard position for the note
 *@param f is the NoteNode being assigned a FBP
 *@param b is the NoteList index of the NoteNode being assigned an FBP
 */
public void tabANote(NoteNode b,int f){
  NoteNode trav = b;
    int fret = -1;
    int string = -1;
    //get pattern
    fretPosition[] fps = t.getFPs(notes[f]);
    String s = getPattern(trav,f);

    //if only one fret position possible, assign that position
    if(s.equals("")){
```

```
      fret = fps[0].fret;
      string = fps[0].string;
    }

    //else, run the network and assign the position dictated
    //by the network.
    else{
      Pattern p = interpretString(s);
      double[] outputs = Fire_TabNN(p.inputs);
      int netOutput = 0;
      for(int o = 0;o<outputs.length;o++){
        if(outputs[o]>outputs[netOutput]){
          netOutput = o;
        }
      }

      //find the FPB that is closest to the output of the network
      //thet is taken to be the network's output
      int minDif = 20;
      int actualOutput = fps[0].fret;
      for(int o = 0;o<fps.length;o++){
        int dif = Math.abs(netOutput-fps[o].fret);
        if(dif<minDif){
          minDif = dif;
          actualOutput = fps[o].fret;
          fret = actualOutput;
          string = fps[o].string;
        }
      }
    }
    trav.fret = fret;
    trav.string = string;
    trav = trav.next;
}
```

A.2 Arranging an original composition via GA

```
//A steady state Genetic Algorithm for evolving a bit string
public void evolve(){

    for(int i = 0;i<replacements;i++){

      //select two parents with binary tournament selection
      int p1 = tourneySelect();
      int p2 = tourneySelect();

      //perform two-point crossover
      String cross = twoPointCross(p1, p2);

      //perform uniform bit-flip mutation
      String childChrom = mutate(cross);


      //create child arrangement
      ArrangementChromosome child =
                  new ArrangementChromosome(childChrom,chords,t,noteCoeffs);

      //if child is valid (every chord can be placed on the fretboard without
      //collisions), replace worst member of the population
      if(child.valid){
        int indexOfWorst = 0;
        int indexOfBest = 0;
        double worstFit = population[0].fitness;
        double bestFit = worstFit;
        for(int p = 0;p<popSize;p++){
          if(population[p].fitness<worstFit){
            worstFit = population[p].fitness;
            indexOfWorst = p;
          }
          if(population[p].fitness>bestFit){
            bestFit = population[p].fitness;
            indexOfBest = p;
          }
        }
        bestIndex = indexOfBest;

        //keep track of best fitness for output at end
        if(bestFitYet<bestFit){
          bestFitYet = bestFit;
        }
```

```
      //replace worst individual in the population if the fitness
      //if the child's fitness is superior
      if(child.fitness>worstFit){
        population[indexOfWorst] = child;
      }
    }

    //else, the arrangement is invalid, and the replacement does not count
    else{
      i--;
    }
  }
}

//Simple binary tournament selection
public int tourneySelect(){
  int x = generator.nextInt(popSize);
  int y = generator.nextInt(popSize);
  if(population[x].fitness>population[y].fitness){
    return x;
  }
  return y;
}

//Uniform Bit-Flip Mutation
public String mutate(String c){
  String chrom = new String("");
  int i = 0;
  for(int h = 0;h<chords.length;h++){
      for(int g = 0;g<chords[h].notes.length;g++){
        if(chords[h].notes[g].type.equals("M")){
          chrom = chrom + "1";
        }
        else{
          if(generator.nextInt(1000)<mutProb){
            if(c.charAt(i)=='1'){chrom = chrom + '0';}
            else{chrom = chrom + '1';}
          }
          else{
            chrom = chrom + c.charAt(i);
          }
        }
        i++;
      }
    }
    return chrom;
}
```

```
//A method for simple two point crossover
public String twoPointCross(int p1, int p2){
  String chrom = new String("");
  int c1 = generator.nextInt(population[p1].chrom.length());
  int c2 = generator.nextInt(population[p1].chrom.length());
  int lowPoint = Math.min(c1,c2);
  int highPoint = Math.max(c1,c2);
  for(int i = 0;i<population[p1].chrom.length();i++){
    if(i<lowPoint||i>highPoint){
      chrom = chrom + population[p1].chrom.charAt(i);
    }
    else{
      chrom = chrom + population[p2].chrom.charAt(i);
    }
  }
  return chrom;
}


//This method creates an arrangement by generating a bit string
//whose length corresponds to the number of notes in
//the original composition.
public ArrangementChromosome createArrangement(){
  String chrom = new String("");
    for(int h = 0;h<chords.length;h++){
      for(int g = 0;g<chords[h].notes.length;g++){

        //All melody notes are automatically included
        if(chords[h].notes[g].type.equals("M")){
          chrom = chrom + "1";
        }
        else{
          if(generator.nextInt(1000)>500){
            chrom = chrom + "0";
          }
          else{
            chrom = chrom + "1";
          }
        }
      }
    }
    return new ArrangementChromosome(chrom,chords,t,noteCoeffs);
}
```

A.3 The Arrangement Evaluation Function

The following code is the fitness function used in the evaluation of arrangements in our arranging GA.

```
public void setFitness(){

    //create a new list that will store the arrangement in TabNN
    NoteList list = createList();


    //This new instance of TabNN creates and evaluates
    //a tablature for the arrangement
    TabNN t = new TabNN(arrangement,list);

    arr = t.c;

    //Fitness of the arrangement, at first, is equivalent
    //to the playability of the tablature
    fitness = t.c.fitness;

    //bonus for including notes
    double noteBonus = 0;

    //true if the last note in the counter-melody, moving bass, or moving
    //harmony was included in the arrangement
    boolean lastCMNIncluded = true;
    boolean lastMBNIncluded = true;
    boolean lastMHNIncluded = true;

    int i = 0;

    //traverse each chord
    for(int j = 0;j<chords.length;j++){

      //traverse each note
      for (int h = 0; h < chords[j].notes.length; h++) {
        if (chrom.charAt(i) == '1') {

          //if note is in counter-melody
          if(chords[j].notes[h].type.equals("C")){
            noteBonus = noteBonus + counterCoeff;

            //bonus for contiguous notes in the counter-melody
            if(lastCMNIncluded){
```

```
      noteBonus = noteBonus + counterBonus;
    }
  }

  //if note is in moving bass line
  if(chords[j].notes[h].type.equals("S")){
    noteBonus = noteBonus + movBassCoeff;

    //bonus for contiguous notes in a moving bass line
    if(lastMBNIncluded){
      noteBonus = noteBonus + movBassBonus;
    }
  }

  //if note is in moving harmony
  if(chords[j].notes[h].type.equals("D")){
    noteBonus = noteBonus + movHarCoeff;

    //bonus for contiguous notes in a moving harmony line
    if(lastMHNIncluded){
      noteBonus = noteBonus + movHarBonus;
    }
  }
  if(chords[j].notes[h].type.equals("H")){
    noteBonus = noteBonus + harmonyCoeff;
  }
  if(chords[j].notes[h].type.equals("B")){
    noteBonus = noteBonus + bassCoeff;
  }
}

//set boolean to true if the note is in the counter-melody
//and in the arrangment
if(chords[j].notes[h].type.equals("C")){
  if(chrom.charAt(i) == '0'){
    lastCMNIncluded = false;
  }
  else if(chrom.charAt(i) == '1'){
    lastCMNIncluded = true;
  }
}

//set boolean to true if the note is in a moving bass line
//and in the arrangment
if(chords[j].notes[h].type.equals("S")){
  if(chrom.charAt(i) == '0'){
    lastMBNIncluded = false;
```

```
        }
        else if(chrom.charAt(i) == '1'){
          lastMBNIncluded = true;
        }
      }

      //set boolean to true if the note is in a moving harmony line
      //and in the arrangment
      if(chords[j].notes[h].type.equals("D")){
        if(chrom.charAt(i) == '0'){
          lastMHNIncluded = false;
        }
        else if(chrom.charAt(i) == '1'){
          lastMHNIncluded = true;
        }
      }



      i++;
    }
  }
  fitness = fitness + noteFactor*noteBonus;

  i=0;
  for(int j = 0;j<chords.length;j++){
    double nic = 0;
    for (int h = 0; h < chords[j].notes.length; h++) {
      if (chrom.charAt(i) == '1') {
        nic++;
      }
      i++;
    }

    percentage = 1 - Math.exp(-.5*nic);

    fitness = fitness + percentage*nicBonus;
  }

}
```

Appendix B

Sample Arrangements by Genetic Algorithm

Contained here are examples of three arrangements generated by the methods described in this thesis. Mp3s of these arrangements, as well as the original scores, can be found at the following website:

http://www.ai.uga.edu/tuohy/excerpts.html.

Also on this site are the tablatures we used for training and evaluation of our hybrid method for tablature generation, as well as the tablatures generated by that method.

B.1 *Symphony No. 2* by Sergey Rachmaninoff

This excerpt is from the first moderato in the second movement of Rachmaninoff's *Symphony No. 2*. It was arranged by GA, using the Neural Network/HC hybrid for tablature generation:

```
‘  ‘  ‘  ‘   ‘  ‘  ‘  ‘   ‘  ‘  ‘  ‘   ‘  ‘  ‘  ‘   ‘  ‘  ‘  ‘   ‘  ‘

8-----7--5-|-3--------0-|----5--7----|----5-------|-0--------1-|-0---
-----------|-0-----0----|------------|-------8--6-|-0----------|-----
-----------|-0-----0----|-5--------5-|-5----------|------------|-3---
-----------|----2-------|------------|----7--5--3-|-2-----2--3-|-----
---7--7----|-------2----|------------|-7--------0-|----3-----0-|-----
8----------|-0--3-------|-3----------|-5----------|-3-----3----|-3---

‘  ‘   ‘  ‘  ‘  ‘   ‘  ‘  ‘   ‘

---0-|----------0-|---------|-0--
3----|-3----------|-3--1--3-|-1--
-----|----------0-|-1-------|-0--
0----|-3--0--3----|-0-------|----
-----|-0----------|----3----|-3--
-----|------------|---------|----
```

B.2 *Arioso* by J.S. Bach

This excerpt is from Bach's *Arioso*, adapted for four trombones by Patrick McCarty. It was arranged by Greedy Hill-climber, using a GA for tablature generation:

```
‘  .  ‘  .  ‘  .  ‘  .   ‘  .  ‘  .  ‘  .  ‘  .   ‘  .  ‘  .  ‘  .
-----------------------|------------------------|-------1---------
------------------1--3-|----------------------1-|----------3------
2--------------3-------|-0--------------2--3----|-------------3---
3-----------0-----2----|----------0--2-----0--2-|-3-----------0---
3----------------------|-3-----3----------------|-1---------------
1-----------------1----|-0-----------------0----|-----------------

‘  .   ‘  .  ‘  .  ‘  .  ‘  .   ‘  .  ‘  .  ‘  .  ‘  .
-----|----3--0----------------|-----------------------|
-----|----------1-------------|-----------------------|
-----|-------------3-----2--3-|-2-----0--------0--2---|
0--0-|-2-----------------3--2-|-3--------3------------|
3----|-3-----------------3----|-----------------------|
-----|------------------------|-1-----------------0---|

‘  .  ‘  .  ‘  .  ‘  .   ‘  .  ‘  .  ‘  .  ‘  .   ‘  .  ‘  .  ‘  .  ‘  .
1-----------------0--1-|----------------------1-|-0----------------------|
---------------3-------|----------------0--3----|-------1--------1-------|
------0-----------0--2-|-0-----------------2--0-|-0-----------------2--0-|
0-----------3----------|-------3--2--3----------|------------------------|
------------------3----|-2-----------------2----|-------------------2----|
-----------------------|------------------------|------------------------|

‘  .  ‘  .  ‘  .  ‘  .    ‘  .  ‘  .  ‘  .  ‘  .
---------------------0--|------------------------|
------------------1-----|-3--------------0-------|
---------------2--0-----|-0-----------------0----|
3--------------------3--|-0-----3-----3--------3-|
0-----------------0-----|-2-----------------2----|
------------------------|------------------------|

 ‘  .  ‘  .  ‘  .  ‘  .   ‘  .  ‘  .  ‘  .  ‘  .   ‘
------------------------|-------1--5-------------|----
-------0--1--1--------0-|----3--0-----0--------1-|-1--
----0--------------0----|-2-----------0----------|-0--
-2--------------2-------|-3-----------0-----3----|----
-3----------------------|------------------------|-3--
-------------------0----|-1-----------3-----3----|----
```

B.3 *Jupiter* by Gustav Holst

This following excerpt is an arrangement from a full score for *Jupiter* from Holst's *The Planets*. It was arranged by GA, using the Neural Network/HC hybrid for tablature generation:

```
‘  .   ‘  .  ‘  .  ‘  .   ‘  .  ‘  .  ‘  .   ‘  .  ‘  .  ‘  .   ‘  .
-----|------------------|------------------|------------------|-----
-----|-1-----1--4--3----|-4--6--4-----3----|-1--3--1----------|-----
0--3-|-----------------3|-------------3----|-------------3----|-0---
-----|-1----------------|-------------1----|-1----------------|-1---
1----|-------------1----|------------------|------------------|-----
-----|------------------|-3-----------3----|-------------1----|-----

‘  .   ‘  .  ‘  .  ‘  .   ‘  .  ‘  .  ‘  .   ‘  .  ‘  .  ‘  .   ‘  .  ‘
-----|------------------|----1--3-----3----|-3--1--------1----|---------
-----|-1-----1--4--3----|-4-----------4----|-------4----------|-4-----4-
0--3-|-----------------3|-------------3----|------------------|---------
-----|-1----------------|------------------|------------------|-------1-
1----|-------------1----|------------------|-------------3----|---------
-----|------------------|-3-----------3----|------------------|---------

‘  .   ‘  .  ‘  .  ‘  .   ‘  .  ‘  .  ‘  .   ‘  .  ‘  .  ‘  .   ‘
6--3-|-1-----1--------3-|-1-----------6--3-|----------------6-|-8--
-----|-3-----------4----|------------------|-6-----6-----8----|----
3----|-3----------------|-3-----3-----3----|-7----------------|-5--
-----|-0----------------|-0----------------|-0----------------|----
-----|-------------3----|------------------|-------------6----|-6--
3----|------------------|-------------3----|------------------|----
```