5-2013

An Expert System for Guitar Sheet Music to Guitar Tablature

Chuanjun He
che3@utk.edu

To the Graduate Council:

I am submitting herewith a dissertation written by Chuanjun He entitled "An Expert System for Guitar Sheet Music to Guitar Tablature." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Computer Science.

Michael Vose, Major Professor

We have read this dissertation and recommend its acceptance:

James Plank, Itamar Arel, Marianne Woodside

Accepted for the Council:

Carolyn R. Hodges

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

An Expert System for Guitar Sheet Music to Guitar Tablature

A Dissertation

Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Chuanjun He

May 2013

This dissertation is dedicated to my parents.

Acknowledgements

I would like to thank my advisor, Dr. Michael Vose who has shown me the requirements for being a qualified researcher over the past 4 years. I also deeply appreciate his encouragement which has helped me build confidence. It is my honor to express my highest respect and most heartfelt thanks to him.

Thanks to the Department of Electrical Engineering and Computer Science for my Teaching Assistant funding.

Thank you to my committee: Dr. James Plank, Dr. Itamar Arel, Dr. Marianne Woodside.

Abstract

This project applies analysis, design and implementation of the Optical Music Recognition (OMR) to an expert system for transforming guitar sheet music to guitar tablature. The first part includes image processing and music semantic interpretation to interpret and transform sheet music or printed scores into editable and playable electronic form. Then after importing the electronic form of music into internal data structures, our application uses effective pruning to explore the entire search space to find the best guitar tablature. Also considered are alternate guitar tunings and transposition of the music to improve the resulting tablature.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This document describes an expert system which can convert guitar sheet music to guitar tablature. There are two main parts in the system: optical music recognition (OMR) with image processing, and, guitar tablature generation with artificial intelligence.

OMR software interprets the sheet music into editable and playable digital form. Normally, the result is saved as a midi file for play back or MusixTeX for music engraving [wik11].

Unlike Optical Character Recognition (OCR) which recognizes the text and parses the words sequentially, OMR should handle more complicated situations such as multiple voices on the same staff which should be played simultaneously. Therefore, the analysis of the spatial and semantical relationship between music symbols is a crucial part of the music interpretation.

Once OMR is complete, guitar tablature generation can begin (see figure 1.1). Unlike the piano, many notes can be played at several different positions on the guitar, which causes a large search space. Also many requirements should be taken into consideration, eg. the skill level of the performer, hand position and movement, note and chord fingering, etc.

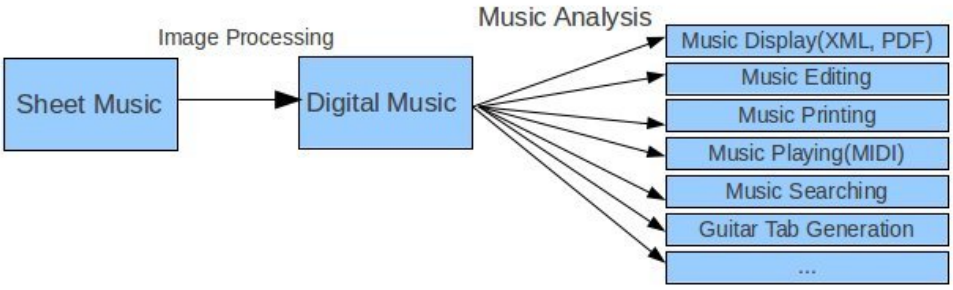

Figure 1.1: Sheet Music Processing Stages

Whereas this project is about generating guitar tablature, it doesn't concern editing the music from which the tablature is generated.

1.1 Optical Music Recognition (OMR)

1.1.1 OMR Background

"Music is an art form whose medium is sound and silence. Music notation represents aurally perceived music through the use of written symbols. " [wik13d] Until recently, most sheet music was in print format.

With the rapid development of computer hardware and the Internet, the electronic version of sheet music is becoming more popular as it can be conveniently shared. There are currently several commercial and open source software programs which can perform sheet music recognition. However, they are not highly reliable nor very effective.

1.1.2 Distinguishing Features of OMR

OMR is a type of document imaging and optical character recognition, however, it is more difficult than the normal document image recognition due in part to the reasons below:

1. Architecture: Unlike the normal passage or paragraph, guitar sheet music has a more complex architecture. For example, multiple track polyphonic music involves several parallel voices in a single stave, which complicate semantic interpretation.

2. Polymorphism: Each music symbol can represent its own meaning, while at the same time it can represent another meaning if modified by other music symbols. For instance, the meaning of a note is modified by a sharp. Also, the same music event can be presented by different music symbols, as a $\frac{3}{8}$ duration note could be represented by a quarter note with a augmentation dot or a quarter note and a eighth note.

3. Three dimensional property: Unlike simple document image recognition, in addition to the x and y coordinates of musical symbols, the time dimension of music should also be taken into consideration.

4. Interconnection: For the normal optical character recognition, the characters are separated. But in OMR, most music symbols are connected by staff lines. Hence, in order to separate music symbols, an effective method should be developed to handle the interconnection problems.

1.1.3 Related Research

Early OMR research dates back four decades to MIT and other universities. Dennis Pruslin and David Prerau made the first attempt to automate the recognition of sheet music, [Pru66, Pre70]. Since then, other universities and institutes around the world have set up their own specific research centers. However, no previous work deals effectively with semantic interpretation involving several parallel voices in a single stave.

Waseda University, Japan

The researchers from the Waseda University presented a robotic organist at the International Exposition in 1985. By reading sheet music with a CCD camera, the robot could play music on the electronic organ in nearly real-time, [MHS+85, KOS+87, Mat88, Mat89].

This robot used hardware to achieve sheet music recognition. Firstly, by using a line filter as the basis of staff line detection circuitry, hardware was designed to locate the horizontal lines in order to compensate for skew angle.

Secondly, the note heads were located by using a template matching method while a slicing technique was applied to recognize the remaining objects. A slice was taken through an object at a predetermined position and orientation. Then the number of transitions from foreground black pixels to background white pixels was counted. Carefully chosen slices provided a differing number of transition counts to recognize the symbols.

The Digital Knowledge Center, John Hopkins University

The Digital Knowledge Center of John Hopkins University finished an NSF Digital Workflow Management project in 1998. This project was called "Lester S. Levy Collection of Sheet Music" which digitized a collection of more than 29,000 pieces of American popular sheet music spanning the years 1780 to 1960 (`http://levysheetmusic.mse.jhu.edu`), creating "sound renditions and enhanced search capabilities for the collection". Audio files and full-text lyrics were obtained using optical music recognition software written by staffs from the Peabody Conservatory at Hopkins.

Ichiro Fujinaga was one of the members at the center. Using a projection method, he worked together with Alphonce and Pennycook to solve the OMR problem [Fuj88, FAPB89, FAPH91, PBF07, Fuj96, FP97, FMS98, CDD+00, MDF02, Fuj04, DDPF07].

By treating the projection average value as a threshold, the horizontal projection of the whole input image was used for staff line detection. He didn't remove staff lines but only identified their positions.

Then, the system used the k-Nearest-Neighbor (kNN) scheme for classification. The properties of a music symbol included its width, maximum height, area, and a measure of rectangularity.

Lastly, music notations were formalized by means of a context-free and LL(k) grammar. His software also used syntactical rules and included an interactive manual correction tool.

Centre For Scientific Research in Music, the University of Leeds, UK

Dr. Kia C. Ng from the University of Leeds developed a software called Automated Music Score Recognizer has a X Window GUI and works on Unix platforms, [Ng11, NB96a, NBC95b, Ng95, NJ03, NB92, NB96b, NBC95a, Ng02, Ng01].

This software divides the composite music symbols into lower-level graphical primitives which are then classified by using a k-Nearest-Neighbor (kNN) classifier. After recognition, "sub-segmented primitives are reconstructed and contextual information is used to resolve ambiguities".

The centre is now developing computer software to recognize handwritten music manuscripts. Their sub-segmentation module adopts a mathematical morphology approach, using skeletonization and junction points to guide the decomposition of composite features. The segments are then disassembled into lower-level graphical primitives, such as vertical and horizontal lines, curves and ellipses.

The University of Waikato, New Zealand

Dr. David Bainbridge from the University of Waikato developed a web version of his PhD software called CANTOR in conjunction with the New Zealand Digital Library Project, [BB01, BNMW+99, Bai97, BB03, BC97, Bai94, Bai96].

Dr. Bainbridge presents a combination of projection techniques incorporating flood-fill algorithms to recognize different music symbols. Both horizontal and vertical projections are used to create a signature of each object, and match that with projections of previously identified objects. He also introduces a Primitive Expression Language which defines build-in rules of how to combine different primitive symbols together for the music semantic purpose.

The software can also accept different types of music symbol sets such as the square-note notation which was used prior to the invention of five-line staff notation.

Surrey University, UK

Nicholas Paul Carter got his PH.D degree from Surrey University, [Car89, Car92, Car94]. His dissertation investigates the image segmentation process based on a method that uses the Line Adjacency Graph (LAG).

Initially, by producing the run length encoded version of the image, the runs of pixels (Segments) are generated vertically. Then, "by proceeding from left to right across the image and considering pairs of columns of run length encoded data, the segments were grouped together to form Sections, which are the nodes of transformed LAG".

The so built graph is analyzed to detect the staff lines and symbols lying on it. By checking section properties, the symbols can be recognized. For example, the Staff Line section should meet requirements concerning aspect ratio, forward and backward connectivity, and curvature. Another example is that a quarter note is composed by two Sections: one had a high vertical aspect ratio (the stem) and the other has average thickness slightly less than the staff line spacing (note head).

His dissertation is one of the most cited references to date.

Department of Systems and Informatics, University of Florence, Italy

Ivan Bruno and Paolo Nesi from University of Florence published many OMR papers, [BBN01, BBN03, BBN04, BBN07a, BBN08]. They designed an Object Oriented Optical Music Recognition System (O^3MR), which is an off-line system. "The off-line system does not have strong temporal bounds in terms of time to produce the output, but only in the requirement of quality in the recognition with a low error percentage".

This project sets a feed forward neural network to perform the identification of the music symbols. Then after the recognition, basic music symbols are composed on the basis of a set of build-in Music Notation Rules.

Swiss Federal Institute of Technology Zurich, Switzerland

Roth presents an OMR system with rule-based classification, [Rot93, Rot94]. Staff line thickness and staff space is estimated by the vertical-run-length of foreground and background pixels over the input image. The staff line thickness is set to be the average of the foreground black vertical run while the average value of the background white vertical run is the staff space.

Staff lines are then located by locating groups of five peaks which form a horizontal projection of the image. Vertical lines such as stems and bar lines are also detected and removed.

Finally, he uses a Mathematical Morphology Method to classify the music symbols.

Other Research Scholars

Besides the researchers mentioned above, many other research scholars also contribute to the OMR area.

Reed uses "template matching, the Hough transform, line adjacency graphs, character profiles, and graph grammars" in his research. "The initial experiments

indicate recognition rates in excess of 95% are obtainable for good quality music of moderate complexity", [Ree95, RP96].

Bertrand Coüasnon developed "a new method called DMOS (Description and MOdification of Segmentation)". "It consists of a grammatical formalism of position (to define knowledge) and a parser allowing a dynamic modification of the parsed structure. This modification allows the introduction of context (symbolic level) in segmentation (numeric level) in order to improve recognition. With knowledge represented by grammar, the DMOS method offers a separation between knowledge and program, and an automatic parser generation (through a compilation phase)", [CC94, CBSB95, CRUE95, CC95, ACD00, MACdR05, Cou06].

1.1.4 OMR Software

There are currently several OMR software packages available, both commercial software and open source software. This section introduces the software available and summarizes performance.

1. Commercial Software:

    - Capella-scan: Windows platform, $249.95;
    - PhotoScore (Neuratron Corporation): Windows and Mac Platform, $369;
    - SharpEye (Visiv Corporation): Windows Platform, $169;
    - Vivaldi-Scan: Windows Platform, £119.00;
    - Nightingale: Mac Platform, $310;
    - SmartScore (Musitek Corporation): Windows and Mac Platform, $399;
    - Finale (MakeMusic Corporation): Its music-scanning module is SmartScore Lite, $600.

2. Open Source Software:

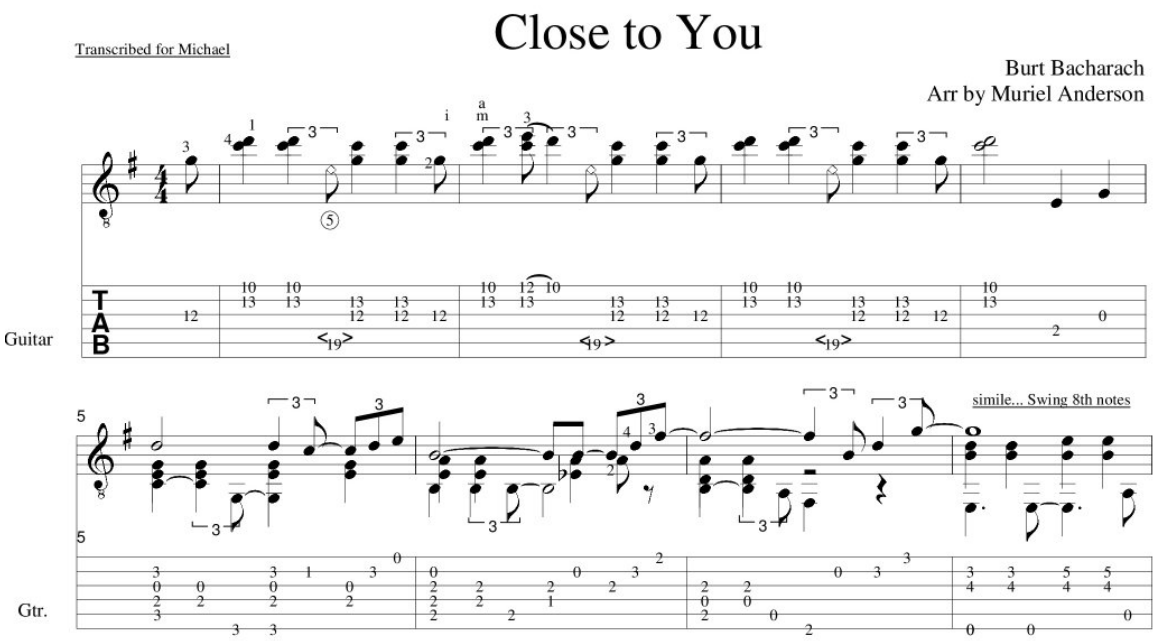

Figure 1.2: Original Image

- Audiveris: written in Java.
- OpenOMR: written in Java.

The evaluation/trial versions of the commercial software packages, and the open source software packages are tested by using figure 1.2 (partially shown). Conclusions are then given based on the test results. The following sheet music is selected for the reasons below:

1. This music sheet contains two stave sets, one is the normal music staff while the other is the guitar tablature.

2. The music sheet contains many music symbol types: clef, key signature, most of the rest types, not only the normal note head, but also harmonics and triplets.

3. Some music symbols in this image are overlapped. For example, a flag and a triple, also some note heads are combined together.

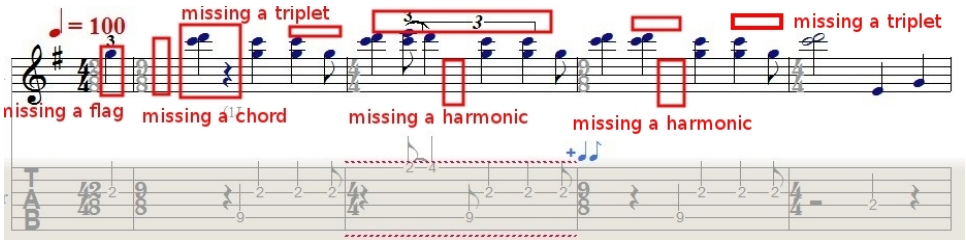

Figure 1.3: PhotoScore Result 1

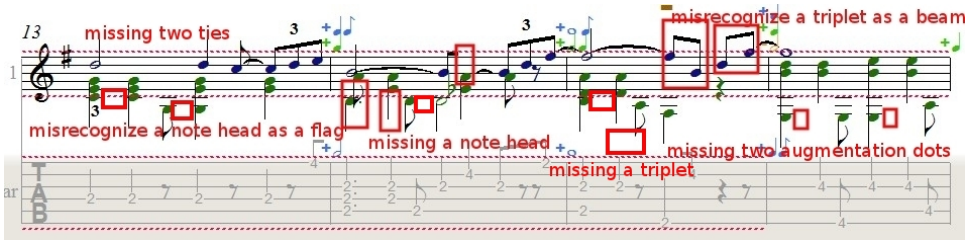

Figure 1.4: PhotoScore Result 2

4. There are many ties and slurs on the image. Also in some measures two or more tracks should be played simultaneously.

PhotoScore

This demo version has symbol recognition errors, and fails to read the triplets and other tuplets. In addition, it doesn't ignore the 6-string guitar tablature. See the result in the figure 1.3 and 1.4.

SharpEye

In addition to symbol recognition errors, this software can't identify triplets or harmonics. But unlike PhotoScore, it does ignore the guitar tab in the music symbol recognition procedure (see figure 1.5 and 1.4).

SmartScore/Finale

SmartScore, formerly named MIDISCAN, was the first commercial OMR software published in 1991. As a scanning and scoring application, this software is available

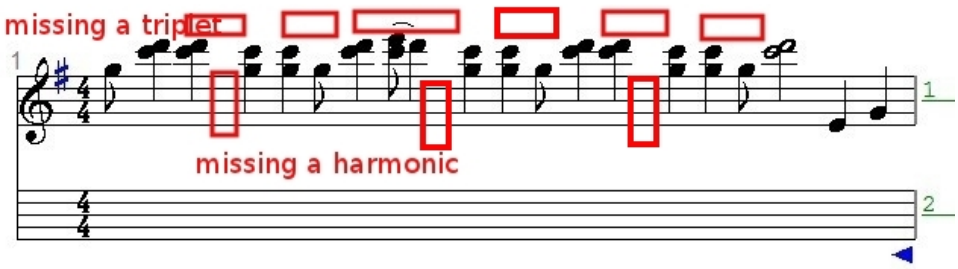

Figure 1.5: SharpEye Result 1

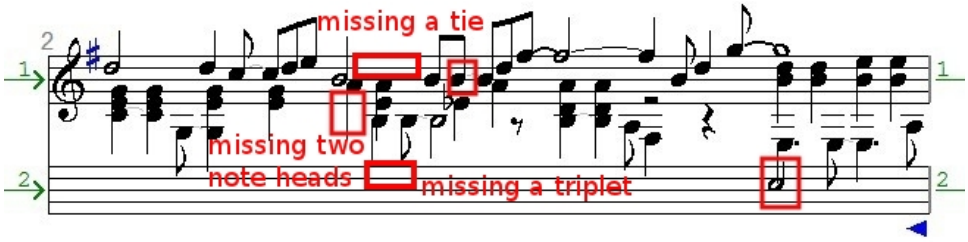

Figure 1.6: SharpEye Result 2

for both Windows and Mac operating systems. Like the others, it also has symbol recognition errors. A serious problem with this product is that it cannot handle a tilted image, see the result in the figure 1.7, 1.8 and 1.9.

Open Source Software - Audiveris

Audiveris uses "Linear Adjacency Graphs to record the pixel information as a directed graph structure, which is composed of pixel contiguous runs organized in sections". Staff lines, ledgers and legato signs are considered as horizontal sections, while stems and barlines are vertical sections. The structure of a music sheet is then be identified by these sections. At last using accumulated training based on user input, the author uses a neural network to identify the music symbols.

This software was written in Java using 420 classes and 130,000 lines of commented Java code [Bit11]. This software has some disadvantages:

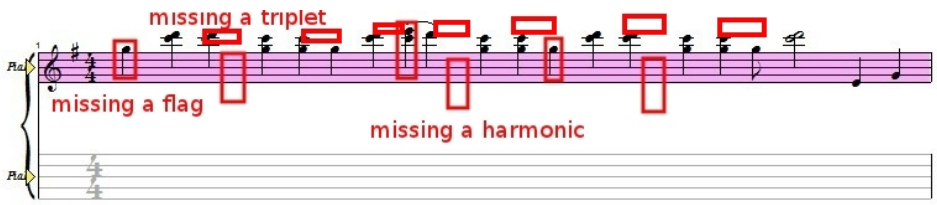

Figure 1.7: SmartScore Result 1

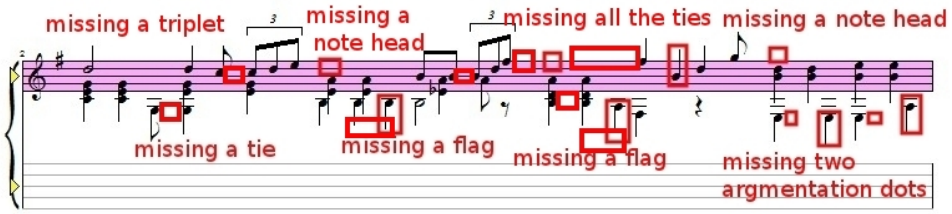

Figure 1.8: SmartScore Result 2

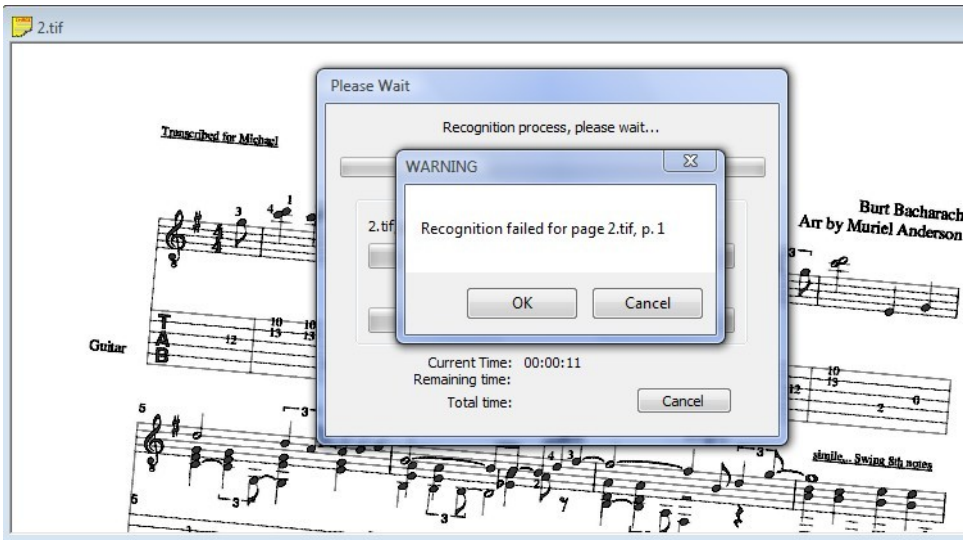

Figure 1.9: SmartScore Result 3

Figure 1.10: Audiveris mis-recognition

1. Many music symbols are mis-recognized, ie, a chord with four note heads is mistakenly identified as a sixty-four rest; a beam with four note heads is recognized as five beams (see figure 1.10).

2. This software can't play some music staff sheets properly if it contains both staves (5 staff lines) and guitar tablature (6 staff lines).

Open Source Software - OpenOMR

Arnaud F. Desaedeleer developed OpenOMR for his MSc degree and published it as an open source software [Des06]. In his thesis, a fast fourier transform of the image is used to deal with the skewed picture. After straightening, the music symbols can then be detected by a neural network. Finally, a midi file is generated based on the music symbols.

This software has the disadvantages of Audiveris above, and also:

1. Polyphonic scores are not supported. From figure 1.11, the note heads can't be recognized if the note heads are connected or they are solid.

2. Doesn't differentiate between Bass and Treble Clefs when interpreting sheet music;

3. Can't detect minims or semibreves (hollow note heads).

Conclusion

Generally speaking, the performance of all commercial software is better than their open source counterparts. In chapter 7, the OMR software is discussed and evaluated

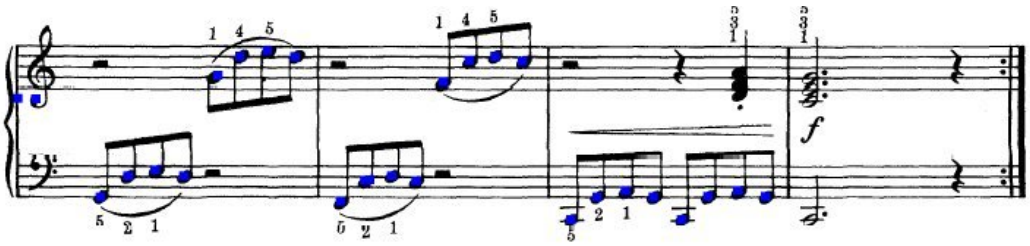

Figure 1.11: OpenOMR mis-recognition

further. Also, the performance of our software is compared with the evaluation version of commercial software and free software mentioned above.

All the commercial software can find most of the most important music symbols (note head, beam and flag). However, some packages have trouble in identifying these symbols with interference (for example, two connected note heads, a flag overlapped with a triplet, etc). For other symbols, the software fails to recognize harmonics, triplets and ties.

Some software have trouble in processing a tilted/distorted image, or can only deal with the normal music staff sheet with 5 staff lines. If there is a 6 staff line guitar tablature, the system won't ignore this stave.

Besides recognition problems, some restrictions also exist for the input images. All the commercial softwares can handle only the white/black image, and refuse to accept the gray scale image. Also the input image format should be tiff or bmp, so common formats like pdf, jpg or gif can't be accepted.

All software can only process one image at a time; there is no parallel computing. Some software can't combine the midi result of each page together if it is a multi-page music sheet.

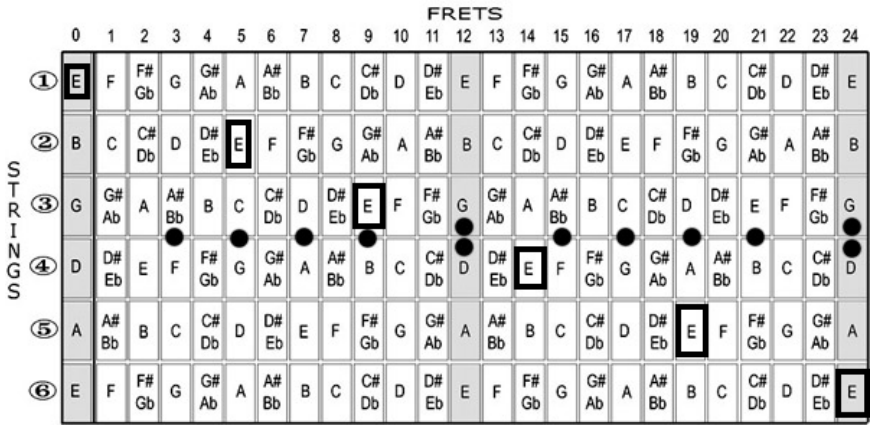

Figure 1.12: Guitar Finger Board

1.2 Tablature Generation

On the guitar there could be as many as six different positions for a note. Take the E note at the first fret on the first string as an example; there could be 6 positions on the fret board (see figure 1.12)

In consequence, for many common musical chords, there could be several ways of playing that chord. Based on some constraints and optimization criteria, search algorithms could be used to find out the best solution in the search space.

1.2.1 Related Research for Guitar Tablature Generation

Several papers have been published to discuss how to generate guitar tablature. [RD04a, MHHY04, TP05, TP06a, TPC06, TP06b, TP06c, TP06b, Rut09].

Some of the papers (those published after 2004) use a genetic algorithm to seek an optimal result. The Genetic Algorithm, however, suffers from an inability to reliably obtain an optimal result. Normally, each chord has several possible playing positions on the fret board. By selecting a playing position randomly for each chord, a queue with chord playing positions is obtained. The population pool contains n such queues.

The fitness function measures the ease of hand/finger movement and ease of hand/finger placement. Parents (population members) are chosen for crossover, with probability proportional to fitness, to form children.

For crossover, the algorithm picks up the k_{th} node from two parents queues, then exchanges the nodes after the k_{th} node of two queues. In this way, each child queue obtains the first part from the first parent and the latter part from the second parent. For mutation, because several playing positions are available for each chord, another playing position is randomly chosen. The children replace the population pool to form the next generation. The Genetic Algorithm conducts a form of stochastic hill-climbing that does not exhaustively consider the search space and cannot guarantee optimality.

University Of Georgia

Daniel R. Tuohy from University Of Georgia explores several heuristic methods, [TP05, TP06a, TPC06, TP06b, TP06c].

One is to use a genetic algorithm. Generally speaking, the principle of survival of the fittest allows better solutions (individuals) to reproduce more than those less fit, which leads to gradual improvement over time. However, the genetic algorithm is time consuming, since hundreds of generations should be produced before obtaining satisfactory results.

Base on the genetic algorithm, this author also explores Evolved Neural Network for tablature generation. "Training data was parsed from an online repository of human-created tablatures". The input layer of the neural net was optimized through genetic search in order to improve the accuracy of the network. Then, a local heuristic hill climber is used to improve the output of the network. They report that they have made modest improvement in tablature quality and significant improvements in execution time when compared to their original system for generating tablature.

University of Torino, Italy

Another previous paper [RAL04] used a graph-based representation for the possible chord position sequences. Each vertex represents a playing position for a note, and edges connect time-adjacent positions. Each edge is labeled with a weight representing the cost of the transition between incident playing positions. Using a shortest path algorithm, the best ways to play the music could be found. However, this paper only considers one note played at a time rather than multi-notes within a chord played simultaneously.

Universidade Federal de Pernambuco, Brazil

This paper [TDSR04] used Viterbi's algorithm (a type of dynamic programming algorithm) to assign the right hand fingers.

The program maintains a set of 20 different hand positions. For each chord, the program select an optimal hand position from the hand position set with the minimum transformation cost and application cost.

Transformation Cost is a function that establishes a cost to change from the current hand position to the next hand position, while Application Cost is a function that computes the cost of applying the hand position.

Other Research Scholars

Besides the papers introduced above, several other researchers also discussed the guitar tab and tuning problems. For example, [TA12] presents "a formal language for assigning pitches to strings. Final optimization relies on heuristics idiomatic to the tuning, the particular musical style, and the performer's proficiency".

The paper by [MHHY04] does not consider multitrack polyphonic music.

Paper [fWL97] uses weighted rules based on the harmonics to select appropriate chords from database and match constituent notes in that chord. Then they choose the finger chart from the database according to the chords. The users could modify

the chord or finger chart from the graphic user interface if they are not satisfied with it. However, this paper doesn't discuss the fitness function and fails to show the best optimal result.

The paper [BTSB12] generates the tablature by using only the audio waveform. They analyzed "the inharmonicity relations between the fundamentals and the partials of the notes played to estimate both the notes played and the string/fret combination used to produce that sound". "A procedure to analyze chords is described which makes use of the inharmonicity analysis to find the simultaneous string/fret combinations used to play each chord".

In the paper [RD04b], dynamic programming is used for the guitar tablature generation based on a cost function and a gradient descent search is employed to improve the coefficients of the cost function. This paper introduces "path difference learning" whose goal is to adjust the cost function weights until the desired path becomes optimal within the dynamic programming search. Also, it constructs tablature which has been optimized for playing the song in reverse (going backwards in time). By using an existing published guitar tablature as the training set, the method obtains optimized weights for the cost function. This paper omits consideration of how a playing position previous in time to the current position can influence what choice of playing position will be optimal at a future time (after the current playing position).

1.2.2 Guitar Tablature Generation Software

There are currently several Guitar Tablature Generation software packages available, both commercial software and open source software, [wik13a]. This section introduces available software and summarizes how well each application performs. Software such as Guitar Pro and TuxGuitar can produce guitar tablature from a midi file, but some can only do the generation from user input. However, the algorithms used by these software are unknown or undocumented, [Gui13], [Tux13], [Tab13], [Pow13].

In this section, three midi files are used to test software (all songs use standard tuning `EADGBE`):

1. what_a_wonderful_world.mid;
2. alone_again.mid;
3. close_to_you.mid.

TuxGuitar

TuxGuitar supports editing the music score and guitar tablature, also can import MIDI files to generate guitar tablature, [Tux13]. The guitar tablature generation results are show in figures 1.13, 1.14 and 1.15. If there are more than three notes in the chord, tuxGuitar may generate unplayable playing positions since finger positions are too far apart. Consider for example, the seventh chords in the first test case and the eleventh chord in the second test case.

TablEdit Tablature Editor

TablEdit Tablature Editor has the same function as TuxGuitar which can be used to create and edit the sheet music/guitar tablature, [Tab13]. It can also convert MIDI files into guitar tabs.

Figure 1.16, 1.17 and 1.18 are the results of testing. Much the same as TuxGuitar, this software doesn't do a good job. As the second measure of the first example, the hand movement is large, moving from the fourth fret to the ninth; also some chords are not playable since the notes are too far apart (see the 9th chord in figure 1.17).

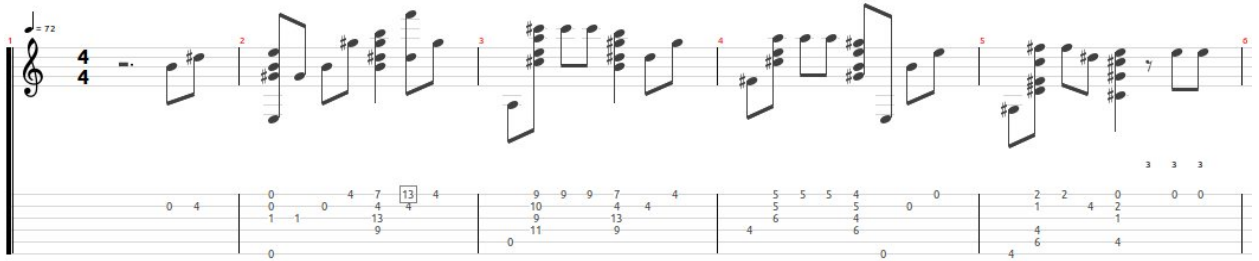

Figure 1.13: TuxGuitar Results: what_a_wonderful_world.mid

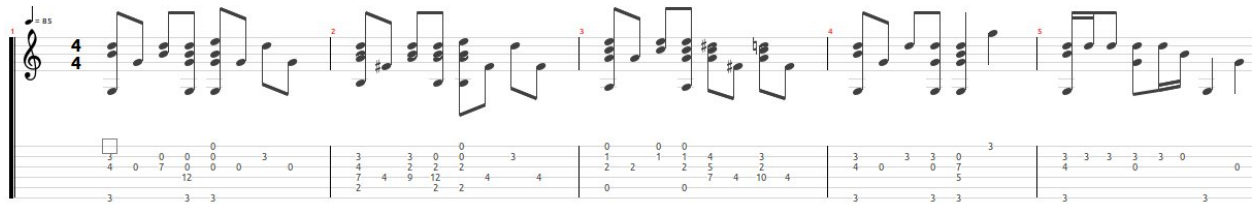

Figure 1.14: TuxGuitar Results: alone_again.mid

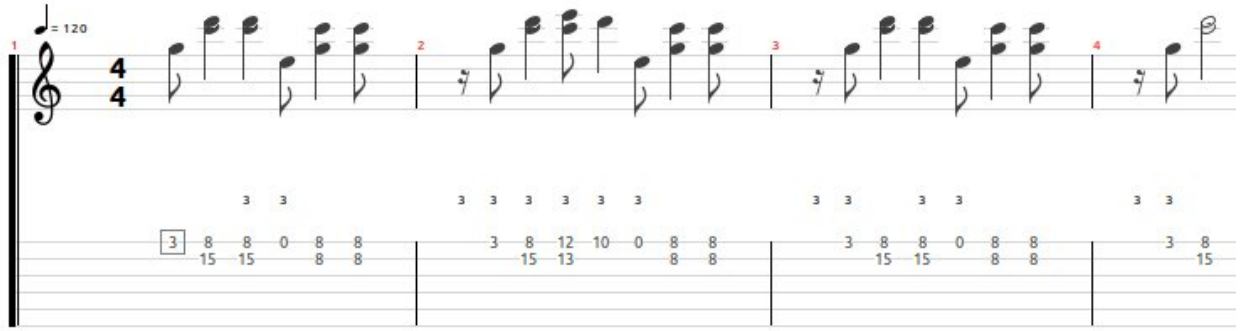

Figure 1.15: TuxGuitar Results: close_to_you.mid

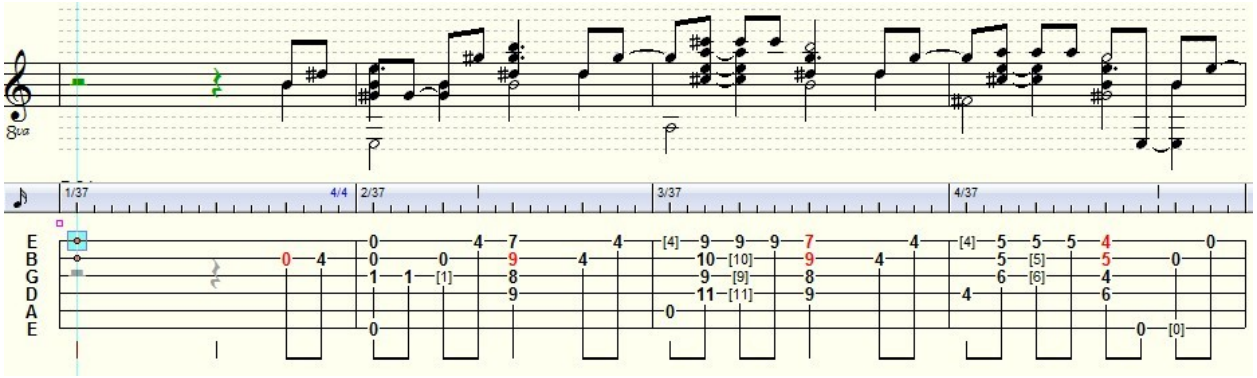

Figure 1.16: TablEdit Results: what_a_wonderful_world.mid

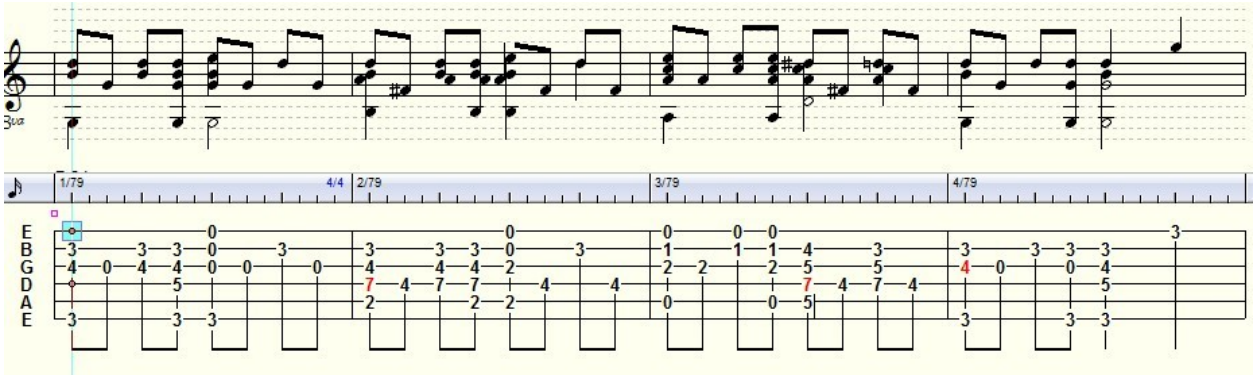

Figure 1.17: TablEdit Results: alone_again.mid

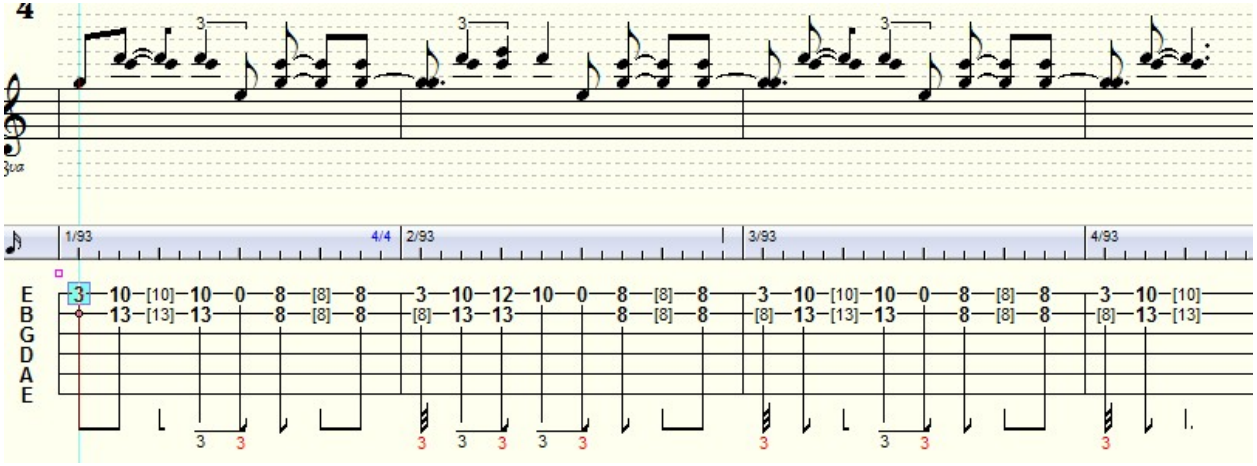

Figure 1.18: TablEdit Results: close_to_you.mid

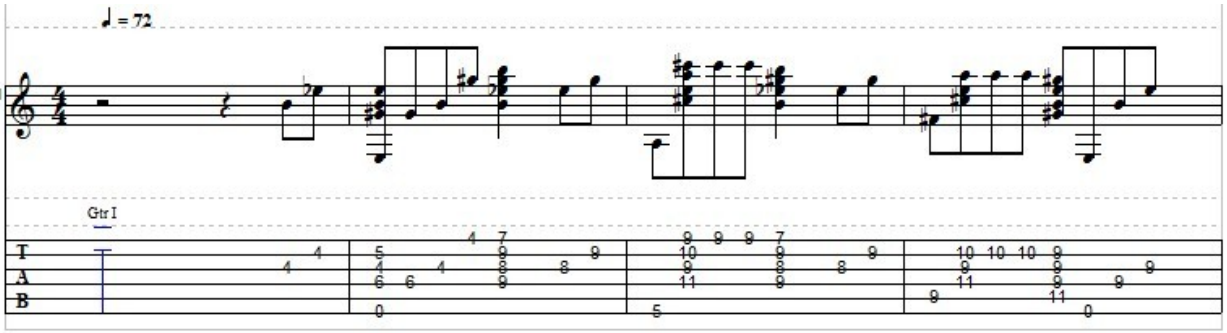

Figure 1.19: Power Tab Editor Results: what_a_wonderful_world.mid

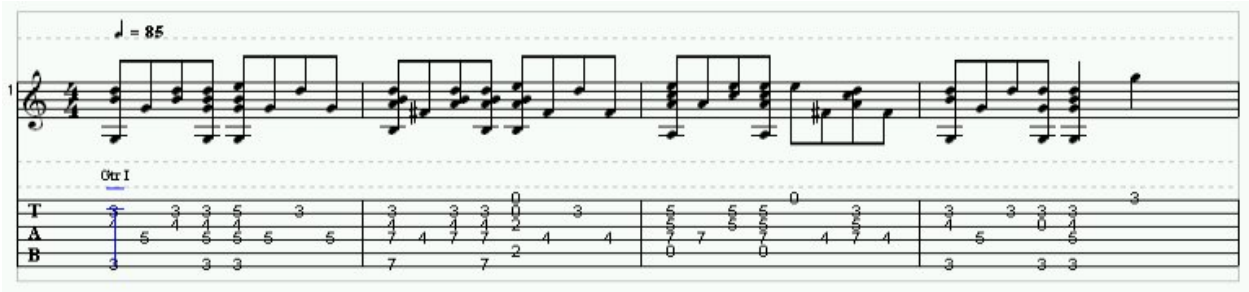

Figure 1.20: Power Tab Editor Results: alone_again.mid

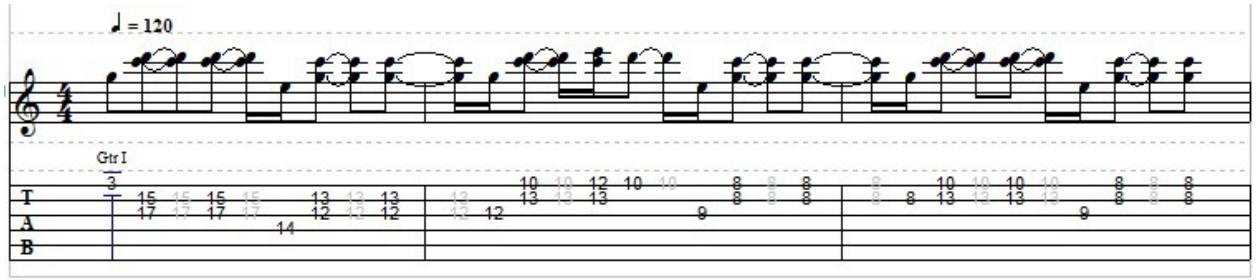

Figure 1.21: Power Tab Editor Results: close_to_you.mid

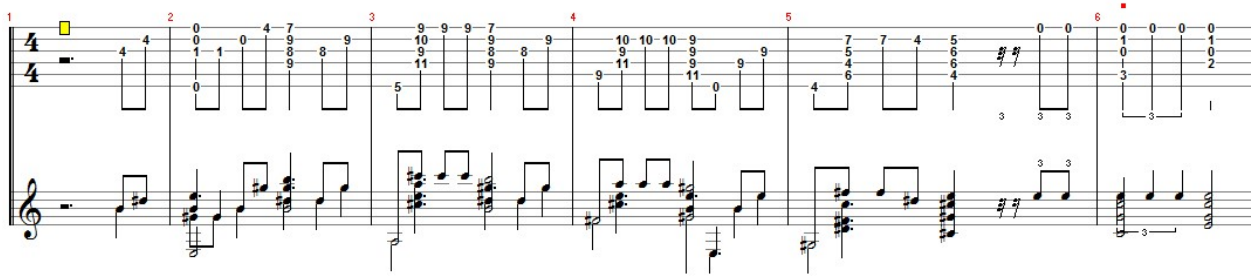

Figure 1.22: Guitar Pro Results: what_a_wonderful_world.mid

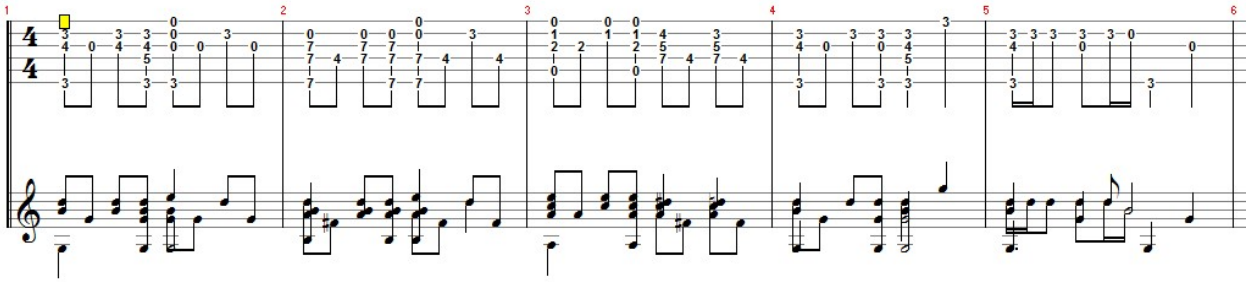

Figure 1.23: Guitar Pro Results: alone_again.mid

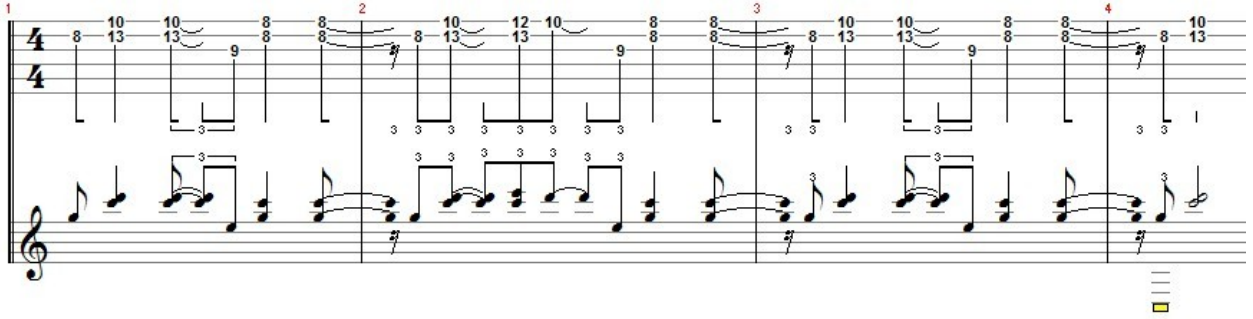

Figure 1.24: Guitar Pro Results: close_to_you.mid

Power Tab Editor

In the reference [Pow13], some limitations of this software are listed. For example, "when using high and low melodies, redundant rests that are left out (as in regular notation) are counted as errors".

The tablature results are listed in figure 1.19, 1.20 and 1.21, which show the same problem of large or unnecessary hand movement (see chords 9 through 11 in figure 1.19, and the movement to and from the eight fret in firgure 1.21).

Guitar Pro

Guitar Pro can be used to generate and edit fretted instrument tablature from a midi file, [Gui13]. Generally speaking, Guitar Pro does a better job than the other three software. However, not all of the tablature results are satisfiable. For example, some users may prefer the smallest hand movement. But as shown in the second measure of the figure 1.22, the player's hand should move from the fourth fret to the ninth. Also, in the third example 1.24, there is unnecessary movement from the eighth fret up and back down again.

Conclusion

TuxGuitar has the worst performance, using a naive algorithm preferring the lowest string for the note playing position, without considering whether the guitar tab is playable.

All four software have the weaknesses below:

1. No suitable guitar tuning is suggested if the midi file is unplayable under the standard tuning.

2. Although guitar tablature generation algorithm of these software are unknown, from the results we may see the minimum hand movement is not a priority.

3. Users may have their own preferences to the guitar tablature. For example, open strings, lower fret numbers, or minimum hand movement. But these software provide no way for the user to configure the generation process.

4. Not all the software let the user move notes to a different string in the graphic user interface.

5. Only one result is generated. It is better to generate several results based on the users' preferences and then let them to choose the guitar tablature.

1.3 Structure of this Dissertation

Previous systems have dealt with individual parts of the complete task which this dissertation considers. We implement a complete proof-of-concept system that includes all of the following:

1. Image binarization and anti-distortion;

2. Musical Symbols Location and Identification;

3. Musical Symbols Semantic Interpretation;

4. Midi Generation;

5. Guitar tablature generation.

Moreover, our system is capable of processing multiple track polyphonic music involving several parallel voices.

1.4 Conclusion

There are two main parts in this document: optical music recognition (OMR) and guitar tablature generation. This chapter has presented the background, related

research paper and software of both parts, and has noted disadvantages of existing software. Finally, the structure of this dissertation is outlined.

Chapter 2

Image Preprocessing

In this chapter, image preprocessing is discussed. The major steps are as follows. After image format conversion and binarization, the image is loaded into the memory. Then, the image is rotated if it is tilted or distorted due to poor scan quality.

2.1 Image Loading Stage

2.1.1 Image Format Conversion and Binarization

Netpbm Formats

Netpbm defines a set of graphic formats, including the portable pixmap (PPM), greymap (PGM) and bitmap (PBM). PBM is Netpbm bi-level monochrome image format.

Netpbm graphics have two modes: ASCII mode and binary mode. Netpbm pbm ASCII format images have "a raster of *Height* rows, in order from top to bottom. Each row consists of *Width* integer gray values separated by a space, in order from left to right" [net13].

For the binary format, a single byte contains pixel data for 8 pixels. Within each byte every bit represents one pixel and there is no space between bytes. The bit value 1 represents black while 0 represents white.

Image Conversion

The image we get from the scanner software could be a pdf or jpg format file. It is better to convert such compressed file formats into a bit map format such as Netpbm before the image processing.

In this project, the *convert* program from the ImageMagick software suite is used to convert the input image into the Netpbm bi-level PBM binary format.

Some options of the "convert" program are used:

1. -density geometry: horizontal and vertical density of the image;
2. -monochrome: transform image to black and white;
3. -black-threshold value: force all pixels below the threshold into black.

Image Binarization

Since the input image is converted into bi-level PBM format, the image binarization is automatically done.

An alternative simple method can be used to complete the binarization. After scanning the image from gray scale format file into the memory, a threshold value is chosen. From the histogram of pixel gray values, pick the peak as the threshold and make every pixel with an intensity below the threshold white and every pixel with an intensity at or above the threshold black.

2.1.2 Image Loading

Memory Map Method and Binary PBM File

"A memory-mapped file is a segment of virtual memory which has been assigned a direct byte-for-byte correlation with some portion of a file or filelike resource. Once present, this correlation between the file and the memory space permits applications to treat the mapped portion as if it were primary memory" [wik13c]. Memory mapping

files can improve performance. In this project, POSIX mmap() function is used to map the image from disk into memory. For PBM binary format, 8 pixels are stored in a char type variable in the memory mapped char array.

2.1.3 Image Data Structure

After memory mapping, the image is stored in the PIC data structure below:

```
typedef struct PIC
{
    char *data;    //pointer to memory mapped file
    char *checked; //whether the pixel is checked or not
    int width;     //image width
    int height;    //image height
}PIC;
```

There are some assumptions concerning the image:

1. Coordinate origin (0, 0) is located at image's upper left;
2. X axis is vertical, increasing direction is down;
3. Y axis is horizontal, increasing direction is to the right;
4. The height and width of the image is H and W.

2.2 Image Anti-distortion

Image distortion is caused in the image scanning step. Most of the time it is a small angular tilt or partial slightly curved. In order to better recognize the image in the proceeding steps, the shifted pixels should be moved back to their ideal places.

2.2.1 Hough Transformation

Hough transformation can be used for line detection. After line detection, the image's tilted degree is returned and the image can be rotated based on this degree from coordinate origin.

A line in (x, y) plane can be presented in normal parameterization form as: $xcos\theta + ysin\theta = r$. A point in the (x, y) plane corresponds to a sinusoid in the parameter (θ, r) plane, while a point of intersection of sinusoids in the parameter plane corresponds to a line in the (x, y) plane.

For each point (x_i, y_i), calculate $r(\theta') = x_i cos\theta' + y_i sin\theta'$ for each $\theta' \in [0, \pi)$, then check the histogram of all resulting points $(r(\theta'), \theta')$, and use the most prevalent θ' to find out the tilt degree θ.

However, one disadvantage of Hough transformation is its high time complexity. Suppose 10° is calculated above and below the horizontal line in 1° steps, time complexity could be *O(20*height*Width)*; Also, if the straight staff lines become curved during the scanning process, the Hough transformation may not work well.

2.2.2 Image Distortion Correction Algorithm

Basic Ideas

Image distortion is caused by the offset between the pixel's current coordinate and its ideal coordinate. The pixels can be considered to be shifted vertically if the image is slightly tilted or partially curved. Image distortion correction can be done if this vertical offset is calculated.

Suppose in the $W \times H$ image data(i, j), there are five staff lines as in Figure 2.1. Define the evaluation function as:

$$E(a, \lambda) = \sum_{j=0}^{H-1} data(a, j) \times data(a + d, j + \lambda)$$

in which d is a constant, and summation is restricted to pixels within the image.

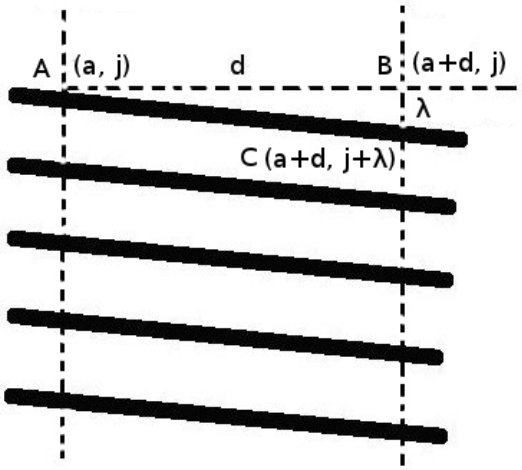

Figure 2.1: Image Anti-distortion Example

From figure 2.1 we notice that the function above can reach its maximum value when λ is the offset between the pixel's current position and its ideal position. Consequently, we can check when $E(a, j)$ reaches its peak.

Image Distortion Correction Algorithm

1. λ should be in the range *[-staffspace, staffspace]*, because the location of the function's peak is periodic; the peak is attained when λ is *offset+n*staffspace* $(1 \leq n \leq 5)$.

2. Divide the image into N vertical strips. In each part (vertical strip) the algorithm is used to calculate and apply the offset.

2.3 Conclusion

This chapter introduces the image preprocessing process. After the image format conversion and binarization, memory mapping method is used for loading the pgm format image into an image data structure. By using an effective distortion correction algorithm, an input suitable for OMR is obtained.

Chapter 3

Image Processing and Optical Music Recognition

In this chapter, how the music symbols are located and identified is described. Firstly in order to eliminate the interference, all staff lines and stems are located and removed. Some simple music symbols are then recognized based on built-in rules. After that, remaining symbols are identified by a comparing method.

3.1 Stave Recognition

3.1.1 Staff Line Width and Staff Space Calculation

Run Length Encoding Algorithm

"Run Length Encoding Algorithm is a form of data compression in which runs of data (sequences in which the same data value occurs in consecutive data elements) are stored as a single data value and count, rather than as the original run" [wik13e].

For example, suppose a hypothetical single scan line is: *110110110*. The encoding result could be *2B1W2B1W2B1W* or *B212121*, with B representing a black pixel and W representing white.

Staff Line Width and Staff Space Calculation

In order to get the staff line thickness *staffthickness* and the distance between two staff lines *staffspace*, Run Length Encoding method is used to represent the image vertically. The histogram of black runs and white runs are both checked. The peak of black runs is *staffthickness* while the peak of white runs is *staffspace*. This is because the music sheet is full of staff lines extending from the left side to the right side.

3.1.2 Staff Line Location

Staff lines are located and removed because most of the musical symbols are connected by these horizontal lines. In order to find out in which rows the staff lines are located, all the pixels are projected to the left:

$$\sum_{j} pic.data[i][j], (i \in [0, H)).$$

If there is a staff line in a row, the summation value is greater than other rows without a staff line. A threshold is used here to do the determination. There are many ways to set this threshold: 0.7 times the image width, 0.9 times the histogram peak, or ask for user input.

3.1.3 Stave Recognition

By using the method mentioned above, staff lines can be detected. But a single staff line's information is insufficient to analyze the whole image. In consequence, we have to merge the single staff line's information together with such information as location, thickness, distance between two stave lines, etc.

Normally, there are 5 staff lines in a stave. The distance between two staff lines in the same stave is much smaller than the distance between two different staves. In this way staves can be differentiated by checking the distance. This method also works for 6 staff line guitar tablature.

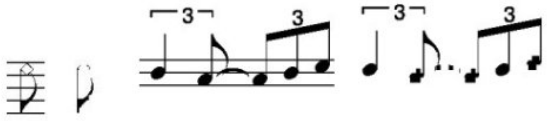

Figure 3.1: Run Length Encoding Staff Line Removal Result

After the stave detection, the staff lines are stored in the stave structure.

```
typedef struct stave
{
    int middle[SIZE];   //row number for staff lines in the stave
    int count;          //how many stave lines in this stave set
    int staffthickness; //stave line thickness
    int staffspace;     //distance between two staff lines
    int left;           //smallest horizontal coordinate
    int right;          //largest horzontal coordinate
}stave;
```

3.1.4 Staff Line Removal

In this section, some staff line removal methods are discussed.

Run Length Encoding Method

Run Length Encoding Method can be used to remove staff lines. Only the runs located near the staff lines are checked. The run is deleted if its height is smaller than a threshold. However, this method removes more pixels than actually needed, especially on the musical symbols' border (see figure 3.1).

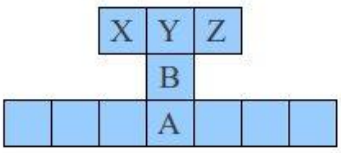

Figure 3.2: Adjacent Pixel Analysis Method

Figure 3.3: Adjacent Pixel Analysis Method Result

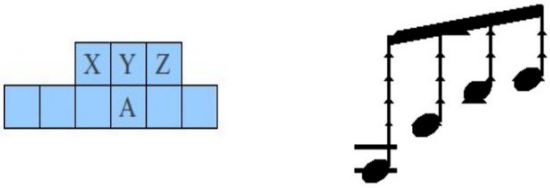

Figure 3.4: Adjacent Pixel Analysis Method Result 2

Adjacent Pixel Analysis Method

Pixels adjacent to the pixels on the staff lines are analyzed to see whether there is another symbol overlapped with a staff line. If there is such a symbol, the pixels on the staff line are kept.

Every pixel on a staff line is checked from left to right. In the figure 3.2, suppose A is a black pixel on the staff line. If the pixel B above A is black and one of the 3 pixels X, Y, Z is black, A is removable. This method is simple and intuitive. But the result is not always acceptable because this method sometimes removes more or less pixels than wanted (see figure 3.3).

We can also change the restrictions to examine the pixels X, Y, Z above A in the figure 3.4: Pixel A on the staff line is kept if $X + Y + Z > 2$, otherwise it is removed. However, the result in figure 3.4 is unacceptable.

Range Analysis Method

Because of the staff line's thickness, the row number of a staff line is in a range $[x_1, x_2]$. We can remove the pixels in the range $[x_1 - \delta, x_2 + \delta]$ (where x_1 and x_2 are the vertical coordinate of the staff line boundary and δ is a small constant).

We can also only record the middle of a staff line and remove the pixels in the range $[x - \delta, x + \delta]$ (where x is the center vertical coordinate of the staff line and δ is a small constant). But more pixels may be deleted if δ is too large while unexpected pixels may remain if δ is too small. The result is similar to the first two examples from figure 3.3 in that sometimes either too many or too few pixels are removed.

Runs and Sections Method

In this subsection, we introduce a new method which avoids the weaknesses of former methods.

A Run is defined as a sequence of continuous vertical black pixels. The image is parsed vertically to generate vertical runs. If Runs in adjacent columns overlap, the relationship between the adjacent Runs are defined as parents and sons. That is: the left Run is the parent of the right Run while the right Run is the son of the left Run. As in figure 3.6, R_1 is the parent of R_3 and R_3 is the son of R_1. A section is a connected undirected graph of such Runs. For example, S_0 is composed of $R_0, R_1, ...R_{10}$.

After creating the Section, the Run queue of the Section is traversed. If a Run has two parents/children, this Section is split. Two types of Sections are defined here: Straight type and Cross type. A Section is Straight type if it has no more than one parent/son. Otherwise, it is Cross type. This is because if a Section belongs to two symbols, it should connect with no less than two other Sections. As a result, we can remove Straight type Sections which are on the staff lines. The result is shown in figure 3.9.

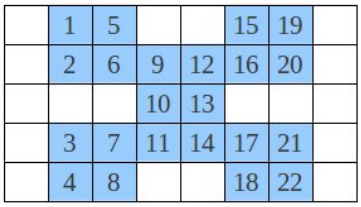

Figure 3.5: Runs & Section: Pixels

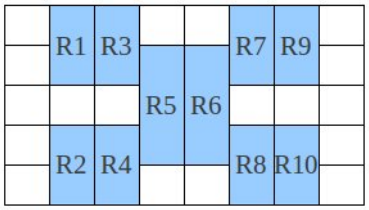

Figure 3.6: Runs & Section: Runs

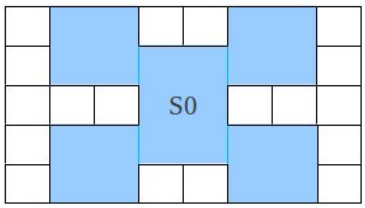

Figure 3.7: Runs & Section: Section

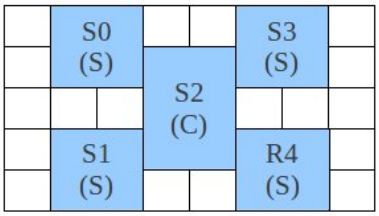

Figure 3.8: Runs & Section: Split Sections

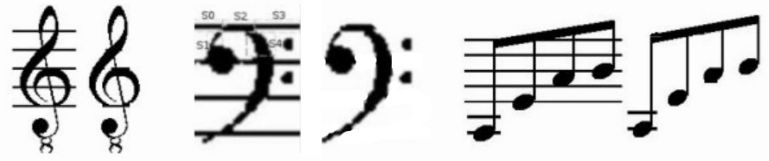

Figure 3.9: Runs & Section: Result

3.2 Stem Recognition

After the staff lines detection and removal procedure, interference is removed. If we want to further analyze the image, we have to remove the interference vertically in order to separate all the musical symbols, especially the note heads and beams.

Figure 3.3 contains a set of eighth notes. A beam connects these multiple consecutive eighth notes. The note heads are connected to the beam by stems. For the first note head, there are two additional stave leger lines which are used to notate pitches below the lines and spaces of the regular musical staff. To simplify recognizing the note heads and beam, the stems and additional stave leger lines should be removed.

For the stem detection, Run Length Encoding is used for encoding both in the vertical and horizontal direction. Let *VRun* be the run in the vertical direction, and *HRun* be the run in the horizontal direction. Every *VRun* is inspected to determine if it satisfies the conditions below:

1. Color: *VRun* is black;
2. Height: *VRun* length $\in$ *[stemMinHeight, stemMaxHeight]*;
3. Width: *HRun* length $\in$ *[1, stemMaxWidth]*;

For the variables above, the values below are used:

1. *stemMinLength*: *0.9*staffspace*;
2. *stemMaxLength*: *5*staffspace*;
3. *stemMaxWidth*: *2.0*staffthickness*.

If a *VRun* meets all three conditions above, every black pixel in this Run can be removed.

However, because the bar line which separates adjacent measures has similar features, it is also recognized as a stem in this step. The difference between stem

and measure bar line is that there is always a note head, beam or flag connected to the stem. In addition, the bar line's top and bottom always touches the first and fifth staff lines, but the stem does not. Because of their differences, the coordinates of the stems should be kept for distinguishing in the later procedures.

A new image with the same width and height as the original one is created. When the black pixel from the original image is removed in this step, it is added to the new image. After this process, all the stems besides the bars are in the new image. By using the stem position information, the bars can be distinguished and removed after the note heads and beams are recognized.

3.3 Musical Symbol Boundary Determination

After completion of the staff lines and stem removal procedure, some of the musical symbols are separated and can be recognized without staff lines' interference.

The first task now is to find the symbol's boundary and features.

3.3.1 Flood Fill Algorithm

In order to extract the features of a music symbol, its boundary should be detected. In this step, the image is scanned row by row. If there is a black pixel and it has not been checked, a new musical symbol is identified. The Flood Fill Algorithm shown below can be used to detect the musical symbol's boundary:

Flood-fill (pixel, pointer to box):

1. If the color of pixel is not black, return. Otherwise check the pixel;
2. If the pixel is not in the box, enlarge the box boundary;
3. Recursively perform Flood-Fill for this pixel's unchecked eight neighbors;
4. Return.

After using the Flood Fill Algorithm, we can identify the coordinates of each symbol's four boundaries. The four borders of each symbol are stored in *Box* type and the boxes are stored in a queue.

```
//use an 32 bit integer to store the pixel coordinate: (x, y)
//x = (p>>16), y = p&((1<<16)-1)
typedef int PIXEL;
//the upper left vertex (x0, y0) and the bottom right vertex (x1, y1)
//of the boundary are stored in a 64 bit unsigned long integer
//x0 = (p>>48), y0 = (p>>32)&((1<<16)-1)
//x1 = (p>>16)&((1<<16)-1), y1 = p&((1<<16)-1)
typedef unsigned long Box;
```

3.4 Noise Elimination

Although staff lines and stems are removed, some interference may remain. For example, some staff leger lines are connected to the note head. In addition, because of the poor quality of scanned images, there are speckles which are small separated black pixels. The Flood Fill Algorithm can be used to remove this noise. A bounding box can be drawn for each symbol and if the box is too small, all the pixels in this box are cleared. However, that method doesn't work for leger line removal, because they are connected to the note head.

Run Length Encoding is used to accomplish this task. After encoding the image vertically and horizontally, each run is checked. If the run's length is small either vertically or horizontally, it is a tiny segment such as a speckle which can be deleted. Because some useful segments could be removed, a copy of the original image is maintained. The stem removal result is show on the left of figure 3.10 while the noise elimination is on the right.

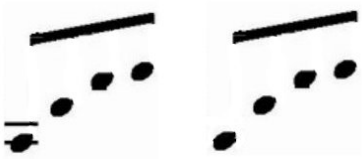

Figure 3.10: Noise Elimination Result

3.5 Note Head Recognition

After removing the stems, note heads and beams are no longer connected. In consequence, we can start to recognize musical symbols. Musical symbols can be identified by using a Neural Network or a Support Vector Machine, however, these algorithms are often time consuming. So time can be saved if most symbols can be distinguished first by using simple build-in rules.

3.5.1 Note Head Separation

There are two types of note heads: hollow ones which are whole notes or half notes, and solid ones such as quarter notes and eighth notes. In this step, we only recognize the solid notes.

A chord is any collections of notes that played simultaneously. In sheet music, these note heads often share the same stem.

If two note heads are located on different sides of the stem, as in figure 3.11, these two note heads can be separated by removing the stems. However, if note heads are on the same side of the stem and are connected vertically rather than horizontally (figure 3.13), another method is needed to separate them. If we draw a box for each symbol, there could be two cases for multiple note heads connected vertically according to the box' position. In figure 3.14, there are two multiple note head boxes; the left one contains two note heads and is the first type below, while the right one contains four note heads and is the second type below.

1. The box is placed with its center intersecting a staff line or ledger line;

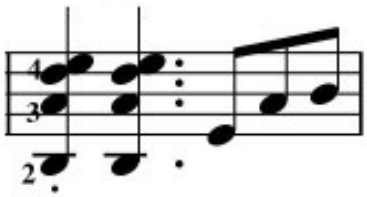

Figure 3.11: Case 1: Notes on Both Sides of the Stem

Figure 3.12: Case 1: Separation by Removing the Stems

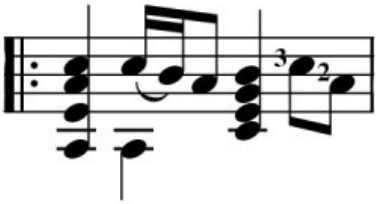

Figure 3.13: Case 2: Notes on One Side of the Stem

Figure 3.14: Case 2: Multi Note Heads with Boxes

Figure 3.15: Case 2: Multi Note Heads with Separation

2. The box is placed with its center between staff lines or ledger lines.

We identify the boundary of all symbols in the image, then check each symbol's box to see whether it meets the requirements below:

1. The box should meet one of the two cases above;
2. Since a single note head's height is almost the same as a staff space, box height is approximately a multiple of the staff space;
3. Box width is approximately 1.5 staff space, which is the width of a single note head;
4. For the solid note, the black pixel's percentage of the whole box is more than the white pixel's percentage (threshold 70%).

If a box meets these requirements, all the pixels on the staff line row or the middle of two consecutive staff lines should be deleted. If the note head is above or below a stave, it still follows these rules and is on or between the staff leger lines. Since the staff space is certain, staff leger lines' row number can be calculated and the pixels on or between the ledger lines can be deleted.

3.5.2 Note Head Recognition

Note heads can be recognized by checking bounding box's features.

1. The note head should be on a line or in a space, which is also applicable with respect to the leger lines;
2. The box height is approximately equal to staff space;
3. The box width is approximately equal to 1.5 staff space;
4. The percentage of black area is at least 70% of the total area;

After this step, all the solid note heads's bounding box can be identified and stored in the queue according to which stave set it belongs from top to bottom and its horizontal coordinate in the stave set from left to right.

3.6 Beam Recognition

In musical notation, beam is used to connect multiple note heads. The duration of the note head is related to the number of beams.

Normally, a beam's shape is determined by how many note heads are under it and the note heads' pitch. Though the shapes are not fixed, they have some commonalities. We can check features of the symbol and its bounding box to recognize the beam:

1. The height of each vertical slice of the symbol should be in the range: *[average height-1, average height+1]*;
2. The box width is larger than its height;
3. The box top left coordinate is not equal to its bottom right coordinate so it is not a single line.

After beam recognition, all beam boxes are stored in the beam queue.

3.7 Other Musical Symbols Recognition

After recognizing staff lines, stems, note heads and beams, they are all removed from the original image so that only other symbols remain. There are many methods to do recognition work. From each symbol's box, we can extract the characters from the symbol and then provide these characters as the input to classifiers. For example, a neural network or a support vector machine can be used as the classifier.

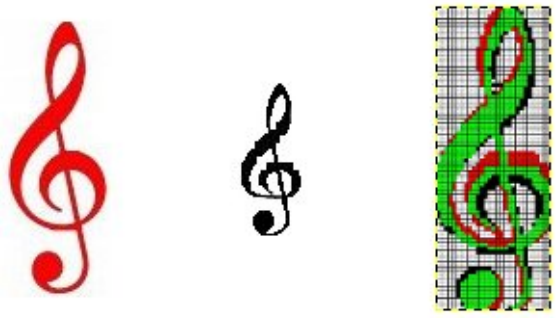

Figure 3.16: Compare symbol with template

A comparison method is used in this project. At first, a template catalog with different symbols is maintained. The symbols found in the music sheet are then compared with the template in the catalog one by one. If the symbol and template have different sizes, the symbol is stretched to be the same size as the template. The number of non-overlapped pixels are returned so we can find the closest match. As in the figure 3.16, the template on the left and the symbol in the middle are compared, then count the non-overlapped pixels on the right.

3.8 Conclusion

This section has described how music symbols are identified and stored in internal data structures. Some simple and common music symbols such as note heads and beams are recognized based on built-in rules, which avoids using complex recognition algorithms. Finally, other symbols are identified by a comparing method.

Some predefined constants are used as the criteria. For example, the note head box height is approximately equal to staff space. In the program, we use the requirements

```
0.9 * staff space < box height < 1.1 * staff space
```

to restrict the height of a box.

However, the average box height may not be the staff space, also two ratios 0.9 and 1.1 may not always suitable for all images. Normally, the distribution of the note

head height is a Gaussian distribution. After sorting the possible note head heights and finding the peak, we can choose a appropriate range for the note head heights.

Chapter 4

Image Reconstruction and Semantic Interpretation

After the recognition procedure, the symbols are stored in different queues. The spatial relationship between the music symbols is considered next in order to integrate and interpret the music symbols and reconstruct the music.

4.1 Spatial Relationship Between the Symbols

To verify the spatial relationship between different musical symbols is an important task in the musical symbol semantic interpretation procedure. These symbols interact with each other. For example, a beam is always attached to stems. With the information of how many beams or flags are attached to the stem, the duration of notes can be calculated.

The relationship between two musical elements can be presented as the following quad: $\langle E_1, E_2, R_v, R_h \rangle$, in which E_1, E_2 indicate the two musical elements while R_v and R_h means the interaction of these two elements in the vertical and horizontal direction.

In our project, several kinds of relations are considered:

1. Note Head and Stem;

2. Beam and Stem;

3. Augmentative Dots and Note Head;

4. Triplet and Note Head;

5. Tie and Note Head;

6. Sharp, Flat, Natural and Note Head.

4.1.1 Spatial Relationship Between Note Head and Stem

In order to find matches of the note head and stem, the vertical distance and horizontal distance between the two are taken into consideration. A spatial relationship evaluation system determines the normalized distance between the two symbols in these two directions. Here the horizontal normalized distance is discussed, vertical distance is calculated in an analogous way.

A perfect match occurs if the stem locates just beside the note head. The horizontal space is divided into several intervals, then situations corresponding to different intervals are analyzed. The evaluation system follows the rules below:

1. Positive/negative indicates relative position. In $\langle stem, notehead\ , R_v, R_h \rangle$, if R_v is positive, it means the note head is above the stem and if R_v is negative it means it is below the stem. In addition, if R_h is positive it means the note head is on the left side of the stem and if R_h is negative it means it is on the right side of stem;

2. If two symbols are too far from each other, the value is infinity;

3. If a perfect match occurs and the stem is on the left side of note head (1 pixel left), the value is -1;

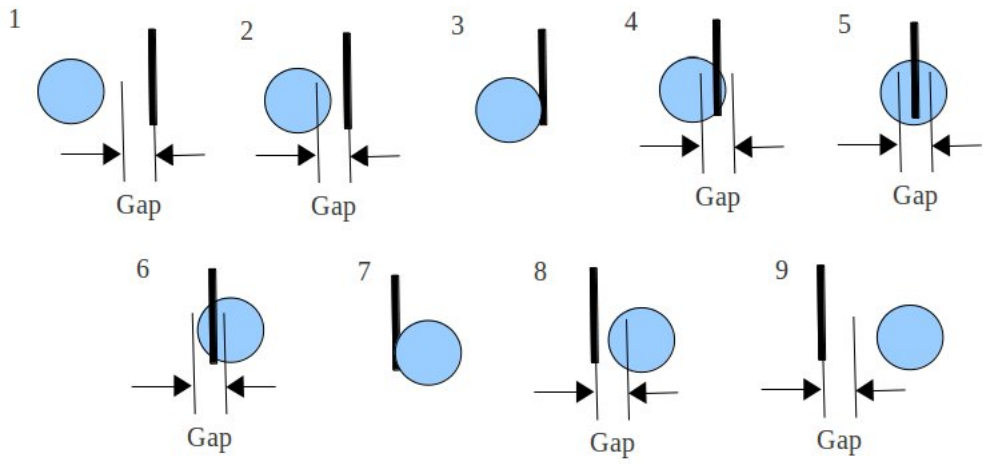

Figure 4.1: Horizontal Relationship between Stem and Note Head

4. If a perfect match occurs and the stem is on the right side of note head (1 pixel right), the value is 1;

5. All other matches except the two perfect matches are treated as acceptable matches. Their values are normalized between [-1, 1].

There are totally 9 cases as shown in figure 4.1 to consider. Suppose *gap* is a constant. If the note head is *gap* pixels or more far away from the stem, they do not interact. In the image, the distance of case 1, 5 and 9 is set to be ∞; the absolute distance of case 3 and 7 is 1 since these two cases are considered as a perfect match. For the rest cases, the distance is normalized to reflect the horizontal relationship between the note head and stem.

In addition, a similar method is used to calculate relative distance for the vertical direction. The stem is interpreted as being connected to the note head only if the values of both horizontal and vertical directions are not infinity. In this case they are stored in the following data structure and the data structure *Isn* is stored in a queue.

```
typedef struct Isn
{
    Box stem;
    Box nh;
    double Iv; //vertical relationship
    double Ih; //horizontal relationship
}Isn;
```

4.1.2 Spatial Relationship Between Beam and Stem

As in the former section, stems and beams are compared to calculate their relative distance in the horizontal and vertical directions. The stem is connected with a beam only if the distances of both direction are not infinity, the relationship is then stored in the data structure *Isb*.

```
typedef struct Isb
{
    Box stem, beam;
    double iv; //vertical interaction
    double ih; //horizontal interaction
}Isb;
```

However, there is a slight difference. When calculating the top and bottom of a note head, the coordinate of the box can be returned directly. But this is not the case for the top or bottom of a beam. The coordinate of its top should be decided by the horizontal position of the stem. For example, in figure 4.2 the top of the box is x_6, and the bottom of the box is x_1. We have to return different coordinates according to different stems, as the beam interacts with many stems and the interaction points are different from stem to stem. For example:

1. TOP(Stem, Beam) for the first stem, return x_0; for the second stem, return x_2.

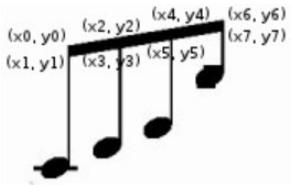

Figure 4.2: Beam's Top and Bottom

2. BOT(Stem, Beam) for the first stem, return x_1; for the second stem, return x_3.

The beam boundaries are used in counting the number of beams intersecting a note stem (see sub section 4.4.1).

4.2 Image Reconstruction

Based on the relationship between different symbols, several image data structures are introduced so that they can can be used to reconstruct the image and generate internal musical data structures. Finally, these internal musical data structures can be used to generate a midi audio file or a pdf format file.

Hierarchical image data structures are defined as follows: System, Stave, Measure, Chord.

The whole image can be considered a system which is a combination of one or more staves. The stave is a set of five horizontal staff lines and four spaces. There is sometimes a brace or bracket combining one or more staves which can be found on the left of the combined staves. The stave is formed by several measures and the measure corresponds to a segment of time defined by a given number of beats of a given duration. If one or more notes are played simultaneously, they make a chord. The hierarchical data structures can be defined:

```
typedef struct NoteHead
{
    Box b;              //note head boundary
    int duration, pitch;
    int position;       //note head's position
}NoteHead;

typedef struct Chord
{
    Box stem;           //Stem boundary
    LinkQueue * nh;     //NoteHead Queue
    int beginningTime;
    int duration;
}Chord;

typedef struct measure
{
    Box barline1;       //Left bar line
    Box barline2;       //Right bar line
    LinkQueue* QChord;  //Chord Queue
}MEASURE;

typedef struct staves
{
    CLEF *clefs;            //defined in the next section
    KEYSIGN *keysign;       //defined in the next section
    TIMESIGN *timesign;     //defined in the next section
    LinkQueue *QMeasures;   //Measure Queue
}STAVE;
```

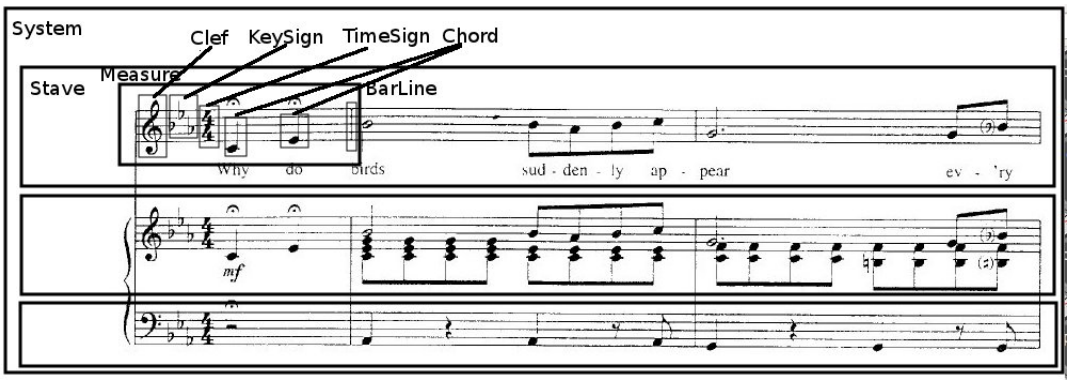

Figure 4.3: Music Hierarchical Data Structure

```
typedef struct system
{
    LinkQueue* QStaves;    //Stave Queue
}SYSTEM;
```

4.3 Note Pitch

Pitch is a basic property of sound, whose scale is determined by the sound frequency. There is a proportional relationship between the scale and the frequency: the pitch is high if the sound frequency is high, and vice versa.

In our project, we use SPN ("scientific pitch notation", first proposed by the Acoustical Society of America in 1939), a method that names the notes by combining a letter-name in the pitch class, accidentals, and a number identifying the pitch's octave. This is combined with MIDI notation, which is a single number in the range 0 to 127 which designates a pitch. In this notation, $C0$ (C in the first octave) is about 16 Hz. So counting up from zero: $C0$'s pitch is assigned the MIDI value 12, $C0\sharp$'s pitch is assigned the MIDI value 13, and so forth. So $G9$'s pitch has MIDI value 127 and $C4$ has MIDI value 60, which is middle C.

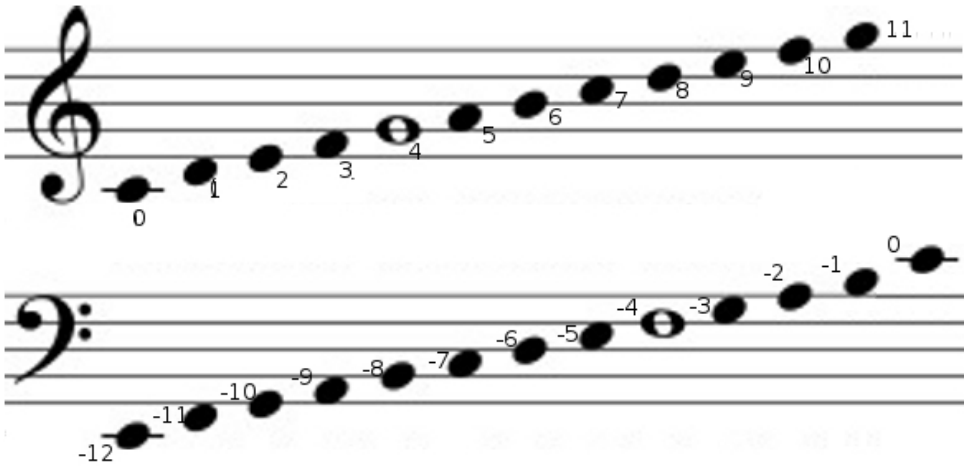

Figure 4.4: Bass and Treble clef

In the sheet music, note pitch is determined by many factors: the note head's position, accidental in front of the note head, clef sign, and key signature in the beginning of the staff set.

4.3.1 Note Head Position

Only considering the clef, the position is assigned 0 if the note head is middle C. For each position higher, subtract 1. For each position lower, add 1. (see figure 4.4)

4.3.2 Clef and Key Signature

Located on the leftmost beginning of the stave, a clef is used to determine the pitch of the following notes. Normally, there are two basic types of clef: treble clef and bass clef.

Generally placed right after the clef at the beginning of the stave, a collection of flat or sharp symbols constitute the key signature, which is used to indicate the key of a music.

Marked as a fractional number, the numerator of the time signature indicates how many beats per measure and the denominator specifies which type of note is a beat.

For instance, time signature 2/4 means there are 2 beats per measure and a quarter note constitutes one beat.

The data structure of clef, key signature and time signature is defined below:

```
typedef struct CLEF
{
    int type;          //treble or bass
    Box b;             //clef boundary
}CLEF;

typedef struct KEYSIGN
{
    int key;           //key
    Box b;             //key sign boundary
}KEYSIGN;

typedef struct TIMESIGN
{
    int top, bottom;   //two numbers of the time sign
    Box b;             //time sign boundary
}TIMESIGN;
```

In the midi file, pitch is indicated, ranging from 0 to 127. The lowest pitch C0 is assigned as 0 while the highest pitch G10 is 127, as explained above.

4.3.3 Accidental

Normally the accidental is placed on the left of a note in the music, which indicates the note is not a member of the scale specified by the key signature. There are three commonly used accidentals: sharp, flat and natural, which raise, lower and restore the note pitch separately.

4.3.4 Note Pitch Calculation

Based on the information such as a note's position, clef, key sign and accident, several rules are provided to calculate the note's pitch.

Step 1: There is a whole tone difference between most adjacent notes, but a semitone difference between B and C, E and F. The pitch difference is set to be 2 for the whole tone and 1 for the semitone. By using note's position, clef type and the pitch difference, the original pitch is calculated (see table 4.1).

Table 4.1: Note's Original Pitch Calculation

Clef	Note Head	Position	Pitch Difference	Pitch
Treble	C5	7	12	72
	B4	6	11	71
	A4	5	9	69
	G4	4	7	67
	F4	3	5	65
	E4	2	4	64
	D4	1	2	62
	C4	0	0	60
Bass	C4	0	0	60
	B3	-1	-1	59
	A3	-2	-3	57
	G3	-3	-5	55
	F3	-4	-7	53
	E3	-5	-8	52
	D3	-6	-10	50
	C3	-7	-12	48

Step 2: Key Signature is used to modify the notes it influences according to the rule of circle of fifths. Details could be found at [wik13b].

Step 3: If there is an accidental in front the note, the pitch of this note and all the same note throughout this entire measure are then adjusted by using table 4.2, unless there is a natural sign in front of the note.

Table 4.2: Accidental Adjusted Pitch Calculation

Accidental	Pitch
Sharp	Original pitch++
Nature	Original Pitch
Flat	Original pitch−−

4.4 Note Duration Value

In music notation, the note duration is determined by the shape of note head (solid or hollow), the number of stem/flags/beams/augmentative dot attached to the note head and the presence or absence of triplet/tie. In this section, the procedure of how these factors are involved in the note duration calculation is explained in detail.

4.4.1 Counting Beam's Number

If the several notes with flags appear in succession, a beam maybe used instead of these flags. The beams or flags connected to the stem are counted. If there are *nbeams* beams or flags, the duration could be determined by the note's type and beam number:

1. Hollow Note: $NoteDuration = \frac{1}{1+nstems}, nstem = 0, 1$
2. Solid Note: $NoteDuration = \frac{1}{4*2^{nbeams}}, nbeams = 0, 1, 2, 3...$

From figure 4.5, the first stem is intersected with two beams. Count how many black pixels on the right column of the stem are in a beam's box area for each of the two beams. The number of beams attached to the stem can be calculated by dividing the count by beam's average thickness.

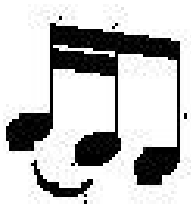

Figure 4.5: Beam Counting

4.4.2 Counting Augmentative Dots

A note's duration may also be augmented by the presence of a dot after the note. This dot makes it one and a half its original duration; *ndots* dots lengthen the note duration by a factor of $2 - 2^{-ndots}$ its original value, so two dots make a total of 1.75 times its original duration. Three dots make it 1.875 times the duration, and so on.

To count dots, the bounding boxes are checked one by one. Only if the height and width of the box is almost the same as the staff line width, the box is considered to be a dot. By using the dots' position information, the note in front of the dots are influenced.

4.4.3 Tuplet

In order to obtain irrational rhythm, tuplet is used to divide a beat into several parts. Triplet is the most common tuplet type.

Three triplet notes are the same duration as two standard notes, so the duration of a single triplet note is $\frac{2}{3}$ the duration of a note. There are two types of triplets (also see figure 4.6):

1. Notes are under triplet brackets;

2. Notes are beamed together and the number 3 is on the beam.

In the first triplet type, the paired bracket can be determined as the nearest other bracket. Paired bracket should also be at the same horizontal level. If there is a number 3 between these two brackets, a triplet is determined.

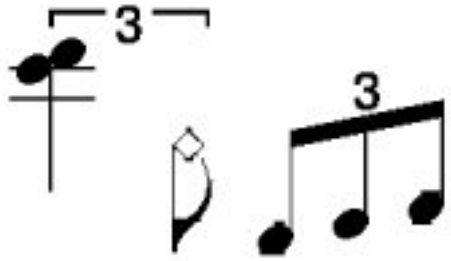

Figure 4.6: Two Triplet Types

For the second type, we check each box with a number 3 and check whether it is just above the beam.

After triplet recognition, the duration of the notes below the triplet are set to be $\frac{2}{3}$ of their original duration.

4.4.4 Slur/Tie

As a horizontal and curved line, a tie connects two same-pitch-notes. In this way, the first tied note duration is the sum of both note durations. On the other hand, the slur requires the music to be played smoothly.

Both slur and tie are curved lines. We check the notes near the end of a tie. If they have the same vertical position, the curved line is a tie and the durations of the tied notes are accumulated by the leftmost.

4.5 Starting Time Assignment

In general, each stave is assigned a track. For instance, there are 3 staves and 3 tracks in figure 4.3. Because all chords are played from left to right in each stave, and staves are played from top to bottom, the starting time of each chord in a track can be derived by sequence and duration: adding the duration of all chords which are before the current chord to get the starting time of the current chord.

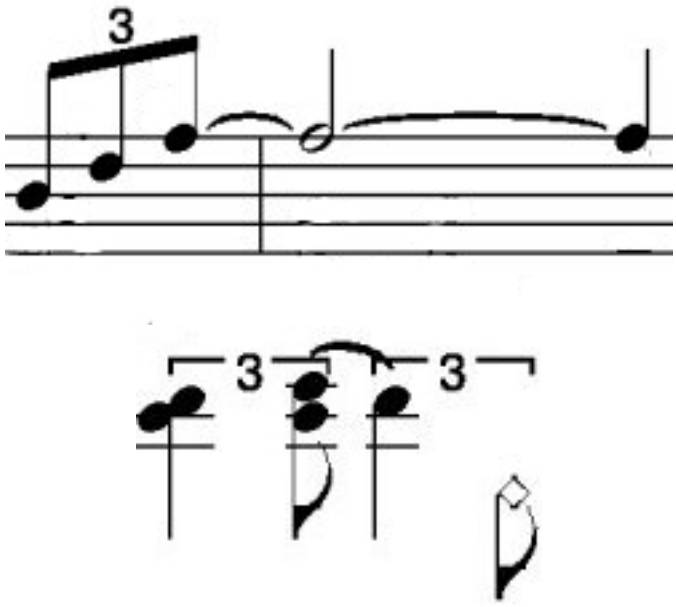

Figure 4.7: Slur and Tie

In figure 4.8, there are multiple tracks on a stave and the number of tracks can be reasonably interpreted as different for each measure: one track in the first measure, two tracks in the second measure, and three tracks in the last measure. Therefore we can't just add previous chord's duration together to obtain the starting time of the current chord, because chords in each track will be mixed. We must first separate the chords into tracks.

Several methods could be used for the time assignment. Since the total time of a measure is certain, Brute Force Search can be used to find an assignment of chords to tracks in which all tracks have durations that match the total time of a measure. An alternate method of how to assign the starting time of each chord is discussed in this section.

Music Rules

Two terms are defined here:

1. A chord is a single note or several notes with the same stem.

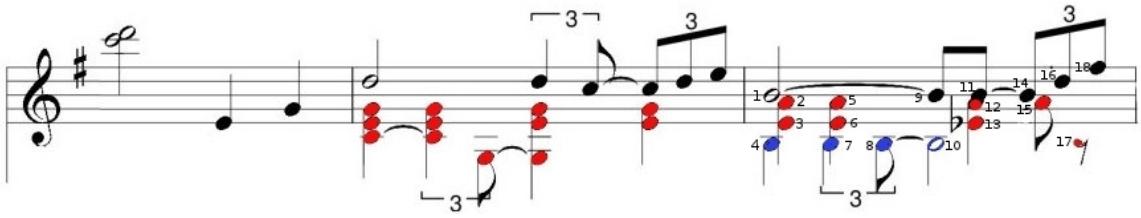

Figure 4.8: Multi Tracks in One System with the Note numbered

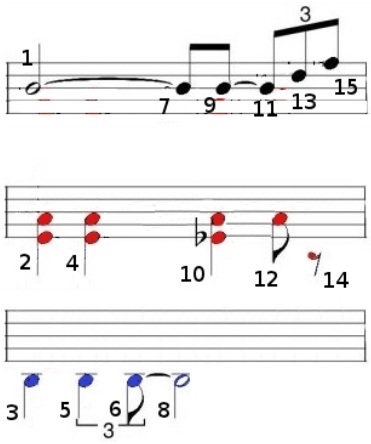

Figure 4.9: Chords in the Third Measure are Assigned to Three Tracks (here the numbers don't designate notes–as in figure 4.8– but indicate chords instead)

2. Chords overlapping in the vertical direction in the same stave belong to a chord set.

In figure 4.8, Note 1, Notes 2 and 3, Note 4 compose a chord separately. Then, these 3 chords compose a chord set.

Rules for when the notes are played:

1. Notes in the same chord start at the same time;

2. Notes in the same chord have same durations; if this is not initially true, the chord is split into parts such that each part is a chord whose notes have the same duration.

3. Each of the chord in the first chord set of a measure start at the same time;

4. The chords in other chord sets need not start simultaneously. e.g. The Chord 10 containing notes 12 and 13 doesn't start with the Chord 9 containing note 11, but with Chord 7 and Chord 8, the first containing note 9, and the second containing note 10.

5. The chord to the right starts no earlier than the chord to the left.

6. If several chords share a tie/slur, a beam, or a triplet, these chords should be played in the same track;

For the last rule, two more pointer fields *parent* and *child* are added to the `NoteHead` data structure. On the left part of the figure 4.10, note 1 is the parent of note 2, note 2 is the child of note 1. While on the right part (where a beam is shared by 5 notes), note 1 is set to be the parent of note 2, note 2 to be the parent of note 3, note 3 is the parent of note 4. By using the pointers, the notes on the same track can be traced easily and pushed into the same track.

According to the rules above, an array is used to compute the starting time of chords in each track. Each element corresponds to a track. The array size is set

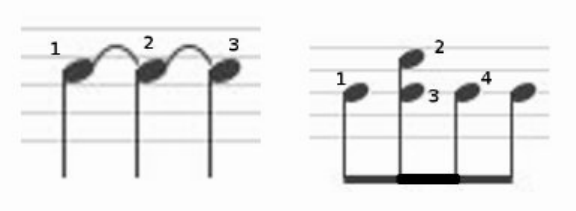

Figure 4.10: NoteHeads Relationship - Parent and Child

to be the maximum number of chords in the chord sets (there will be one track for each chord in a chord set). All the chords in the measure are sorted according to the coordinates, from left to right and from top to bottom (see the index of each chord in figure 4.9). In each step, the duration of the first unassigned chord or rest is added to the minimal array element; then pointer moves forward according to the *child* link to the next child and accumulate the child duration to the same array element until the *child* pointer is `NULL`. The position of that array element defines the track to which a chord or rest is assigned.

Take the third measure in figure 4.8 as an example. Suppose the total time of a measure is 480 (the duration is 240 for the half note, 120 for the quarter note, 60 for the eighth note; if the note is under the triplet, the duration is $\frac{2}{3}$ of the normal value). Because there are maximum 3 chords in a chord set, the array size is set to 3.

The duration of each chord (see figure 4.9) is listed in table 4.3:

Table 4.3: Duration of Each Chord

Chord 1: 240	Chord 4: 120	Chord 7: 60	Chord 10: 120	Chord 13: 40
Chord 2: 120	Chord 5: 80	Chord 8: 240	Chord 11: 40	Chord 14: 60
Chord 3: 120	Chord 6: 40	Chord 9: 60	Chord 12: 60	Chord 15: 40

Assuming a starting time of 0, the steps deriving the starting times for chords in the third measure are in the table 4.4. There are 15 steps, each step for each chord. The chord's duration is added to the smallest element of the array. At the end of the procedure, the measure duration 480 is derived for all three tracks. The table

elements with a slash show two numbers: the left number is the array element value and the right number is the chord index.

Table 4.4: Starting Time Calculation

Step	0	1	2	3	4	5
Track 1	0	240 / 1	300 / 7	360 / 9	400 / 11	440 / 13
Track 2	0	0	0	0	0	0
Track 3	0	0	0	0	0	0
Step	6	7	8	9	10	11
Track 1	480 / 15	480	480	480	480	480
Track 2	0	120 / 2	120	240 / 4	240	240
Track 3	0	0	120 / 3	120	200 / 5	240 / 6
Step	12	13	14	15		
Track 1	480	480	480	480		
Track 2	240	360 / 10	420 / 12	480 / 14		
Track 3	480 / 8	480	480	480		

4.6 Conclusion

In this section, both horizontally and vertically spatial relationships between different interrelated music symbols such as note head and stem, beam and stem are considered based on their coordinates. Two symbols could be combined only if they are placed close enough in the image. Then, the image is reconstructed hierarchically based on the combined results.

After the image reconstruction, the music meaning of different symbols is specified. For example, the note pitch is identified by note position, clef and key signature; the note duration value is determined by beam's number, augmentative dots and tuplet.

At last of the section, how to distribute the chords into different tracks is illustrated.

Chapter 5

Midi File Generation

5.1 Midi File Specification

A MIDI file is organized into a header chunk and several track chunks. Each chunk begins with an eight byte header: a four-byte-ID-string is used to specify the chunk type, then a four-byte-size number is used to identify the chunk's length (the number of bytes following the chunk's header).

For each track chunk, after the length field, is the track event data which contains a stream of events. Each event contains a delta time field and an event data field. Defined as a variable-length value, the event delta time determines the time gap between the current event and the previous event, which means the delta time is a relative time rather than absolute time. Delta time 0 denotes these two events happen simultaneously.

Two types of events are considered in our program: meta event that provides information about music description (i.e: time signature, key signature, set tempo, end of track) and midi event of how the note is played (i.e: note on event, note off event). Since the meta events which contain text messages are not taken into consideration, 4 integers are enough to store the time signature and key signature meta event contents. For midi event, both note on event and note off event have two

parameters which specify the midi key and the velocity (how fast/hard the key was released). Both two parameters are ranging from 0 to 127.

The midi file structure is summarized in the table 5.1 below:

Table 5.1: Midi File Structure

Chunk	8 Byte Header		Track Event Data
Header Chunk	MThd	6	$\langle type \rangle \langle tracks \rangle \langle division \rangle$
Track Chunk *1*	MTrk	$\langle Length \rangle$	$\langle delta\_time \rangle \langle event \rangle ...$
...			
Track Chunk n	MTrk	$\langle Length \rangle$	$\langle delta\_time \rangle \langle event \rangle ...$

A useful gateway to more detailed information concerning the MIDI file format is reference [dig13].

5.2 Hierarchical Midi Data Structures

Based on the Midi specification introduced above (see [dig13] for low-level details), it is easy to implement midi hierarchical data structures. The benefit is that each data structure in the hierarchy has a counterpart in the hierarchical image data structures, consequently the conversion between the two data structures is convenient.

The hierarchical midi data structures are designed as below:

```
typedef struct MetaEvent
{
    int type;                   //Meta Event type
    int len;                    //length of byte
    int iContent1;
    int iContent2;
    int iContent3;
    int iContent4;
}MetaEvent;
```

```
typedef struct MidiEvent
{
    int deltaTime;
    int type;                   //on or off
    int key;
    int velocity;
}MidiEvent;

typedef struct MidiHeaderChunk
{
    char   chunkID[4];          //"MThd"
    int    chunkSize;
    int    format;              //format: 0, 1, 2
    int    tracksNumber;
    int    deltaTimeTicks;      //how many ticks per minute
}MidiHeaderChunk;

typedef struct MidiTrackChunk
{
    char      chunkID[4];       //"MTrk"
    int       chunkSize;
    LinkQueue *QMetaEvent;      //meta event list
    LinkQueue *QMidiEvent;      //midi event list
}MidiTrackChunk;
```

```
typedef struct MidiFile
{
    MidiHeaderChunk *MHeader; //header chunk
    LinkQueue *QMTrackChunk; //track chunk queue
}MidiFile;
```

5.3 The Relationship between Midi File Structure and Image Data Structure

The relationship between midi file structure and image data structure is as follows: A midi file is organized into several track chunks which are played simultaneously. Also in the image data structure, a sheet music system is made of several simultaneously played staves. A track chunk consists of several midi events, while a stave consists many notes. Each element in the midi hierarchical data structure has a corresponding counterpart in the image data structure as in table 5.2 (the midi file and image data structures are in Section 5.2 and Section 4.2).

Table 5.2: Midi Hierarchical Data Structure vs. Image Hierarchical Data Structure

Midi File Data Structure	Image Data Structure
MidiFile	SYSTEM
MidiTrackChunk	STAVE
MidiEvent	NoteHead
MetaEvent - Time Signature	TIMESIGN
MetaEvent - Key Signature	KEYSIGN

5.4 Parallel Computing

Typical sheet music has many pages. Since the OMR work of each page is independent, parallel threads could be used to recognize these pages simultaneously. This project uses the Posix Pthread library [pos12]. Because of using two-core CPU

and hyper-threading technology, 4 threads are preferred in our program. However, the number of threads is a pre-defined constant and can be reset at compile time.

5.5 LilyPond Format File

Our program has two options to generate a midi file. One is to generate the midi file directly, by converting the musical data of each page into midi events and merging the midi events. However, sometimes due to music symbol mis-recognition, a way of editing is needed. LilyPond is a computer program and file format for music engraving, which provides a way for saving our recognition result. From a LilyPond format text file, LilyPond can generate a midi file and pdf format sheet music. In this way, by editing the Lilypond format text file, one may correct the recognition result [lil12a, lil12b].

Besides generating the midi file directly, our program also writes the recognition results into a LilyPond format file. In the LilyPond format file, the octave is raised by adding a single quote to the note name, and lowered by adding a comma to the note name. The duration of a note is specified by a number after the note name: 1 for a whole note, 2 for a half note, 4 for a quarter note and so on. The detailed format information can be found in [lil12b].

For example, the LilyPond format file below corresponds to sheet music in Figure 5.1.

```
\new Voice  {
\clef treble
\key g \major
\time 4/4
\partial 8
% 1th staves 1th measure: 1 track
<< { g''8 }\\ >> |
% 1th staves 2th measure: 2 tracks
```

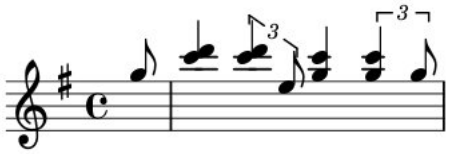

Figure 5.1: LilyPond Example

```
<< { < c''' d''' >4  \times 2/3 { < c''' d''' >4  e''8  }
     < g''  c''' >4  \times 2/3 { < g'' c''' >4 g''8 }  }\\ >>
}
```

5.6 Conclusion

In this chapter, the midi file specification is introduced. Since the midi hierarchical structures match the image hierarchical structures very well, the relationship mapping between the two structures is simple. Also due to the independence between pages in the image recognition phase, pthread parallelization is implemented to speedup the process. Our program also writes the recognition results into a LilyPond format file for saving and editing the music.

Chapter 6

Guitar Tablature Generation

6.1 Introduction

While not completely general — master guitar builder Keith Medley has a 27-string guitar [Med12], and Danelectro has made a 36-fret guitar [Dan12] — we limit consideration to music for typical six-string guitars having no more than thirty frets.

Unlike the piano, on which a note could only be played at one place, a note might be played at many places on a stringed instrument. For example, there may be six different positions for a note to be played on the six-string guitar. With standard tuning, for example, the open-string E note on the first string could be played at 6 different positions (see figure 6.1).

A musical chord consists of one or more notes playing simultaneously. We consider musical chords containing up to 6 notes, and attempt to find among their several frettings (or ways of playing), the best way to play the chords on a six-string guitar. The search space is very large — for instance in the music "Close to You" from the The Carpenters album [Car70], there are 0 note(rest) to 4 notes in each chord and 570 chords in total. The overall guitar tab possibility is 10^{838} — and not every fretting is feasible, due to constraints. The constraints we consider are: different notes can't be played on the same string and notes can't be on frets too far apart.

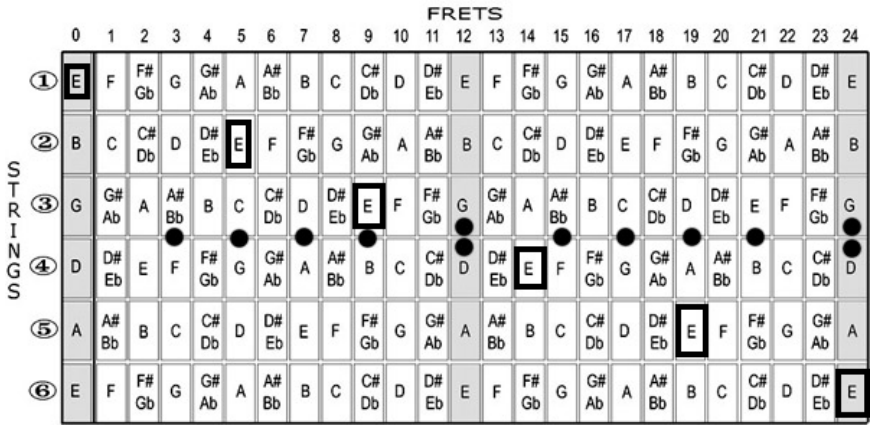

Figure 6.1: Guitar Finger Board

The major steps in our algorithm are:

1. Parse the input midi file and generate midi chords' internal data structure.
2. Based upon the guitar tuning, generate possible playing positions for each note (which could also be considered as one-note chord's playing position).
3. Generate the feasible playing positions for each chord in the song.
4. Generate tablature by making optimal choices from the feasible chord positions.

6.2 Representation

6.2.1 MIDI

Each pitch in a chord has a MIDI number in the range $\{0, \ldots, 127\}$ and is stored as a byte. Since the MIDI numbers require only seven bits, the high-order bit of each byte is available to indicate whether the note is tied. We limit consideration to a maximum of 6 notes per chord — there are six strings on a typical guitar — which allows a MIDI

chord to be stored as a 64-bit integer (unsigned long). An unsigned long contains 8 bytes, which allows for chords to be *variable size* — in the sense that they can be made up of differing numbers of pitches (with the zero byte serving as an end-of-array marker) — while each has a *fixed-length* physical representation. The pitches in a MIDI chord occur in sorted order to ensure chords have a unique representation as an unsigned long. Consequently in the chord's position generation step, we can compare two long integers to avoid calculating the chord's playing position if this chord has already appeared and the playing position has been calculated and saved before. This is particularly convenient, as it allows them to be used as unique keys to associate a MIDI chord with its various chord playing positions. Therefore, the result of converting the MIDI input into a sequence of MIDI chords comprised of MIDI pitches is efficiently represented as an array of unsigned longs.

6.2.2 Notes and Chords

In scientific pitch notation, the standard tuning of a six-string guitar is E_4(329.63 Hz), B_3(246.94 Hz), G_3(196.00 Hz), D_3(146.83 Hz), A_2(110.00 Hz), E_2(82.41 Hz) from the thinnest string (string 1) to the thickest (string 6) [BTSB12].

The pitch difference between two consecutive frets is a half-step interval on the chromatic scale, which means the pitch difference is 1. The fret position to play a note on a string may be obtained by the difference between the note pitch and the open string pitch. Each note playing position is a pair $\langle \mathit{string}, \mathit{fret} \rangle$, where *string* is in the interval $[0, 5]$ and where *fret* is limited to the interval $[0, 30]$. This allows the pair to be stored as a byte; the string number occupies the top 3 bits, while the fret number occupies the lower 5 bits. This representation is not only very memory efficient, it allows each byte to be encoded via $x \mapsto x + 1$ as a nonzero unsigned character, so that the zero byte is available as an end-of-sequence marker in an variable-sized

array of note playing positions (that is particularly convenient for a **while ... do** programming construct in a language — like C, for instance — which regards zero as false).

A chord position is a set of the note playing positions within the chord and is stored as a 64-bit integer (unsigned long) as described above.

6.2.3 Pitch and Playing Positions

The representation using unsigned longs described above also allows for efficient representation of the various positions a given pitch can be played. Placing those playing positions in the notes of a chord (as described above) means that a single unsigned long efficiently encodes all playing positions for a given pitch; hence an array of no more than 128 unsigned longs — corresponding to MIDI pitch 0 through 127 — is required, since each midi event uses 1 byte to store the note pitch [ton12, son12]. The note playing position are precomputed and then used to efficiently find all playing positions for a given pitch.

6.3 Generating MIDI Chords

As described in the last Section "MIDI Representation", a chord with maximum 6 notes is stored in an unsigned long integer and the whole music is stored in an unsigned long array. In our definition, notes in a chord have the same duration. If not, the longer duration note is split to meet others' duration like the example in figure 6.2: the half note is split into two quarter notes and also the high order tie bit of the second chord integer is set.

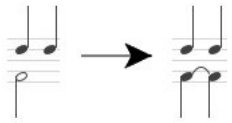

Figure 6.2: Note Splitting

The midi events of each track in the MIDI are parsed, merged, and split, and the notes are read into an long integer array.

6.4 Generating Chord Positions

A playing position for a chord can be regarded as a path from the root to a leaf in a "note tree" whose i_{th} level is comprised of playing positions for the i_{th} note of the chord. The resulting tree can be explored using depth-first search to generate the chord playing positions while Branch and Bound is used to prune infeasible positions. Cases which should be pruned include different notes on the same string, and notes that are too far apart (the maximum difference between frets in the chord exceeds 5, for instance). See figure 6.3. The resulting chord playing positions for a chord are stored as a linked list which is shared by the multiple instances of that chord in a song (see figure 6.4: chord 0 and chord 2 are the same).

6.5 Generating Guitar Tablature

Guitar tablature for a song can be regarded as a path from the root to a leaf in a "chord tree" whose i_{th} level is comprised of chord playing positions corresponding to the i_{th} chord of the song. The resulting tree can be explored using dynamic programming

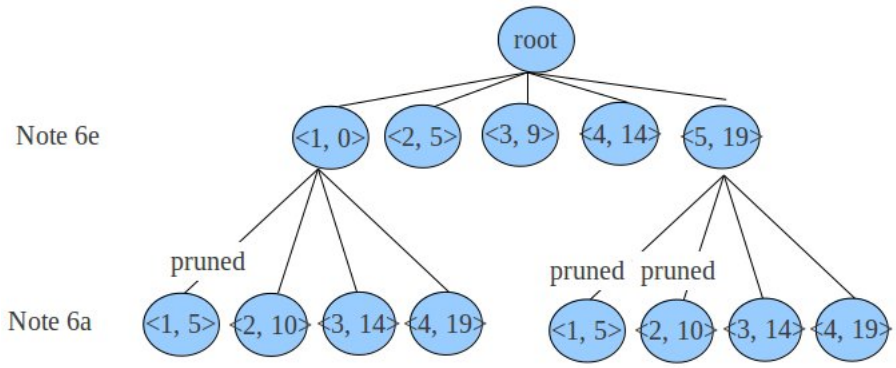

Figure 6.3: Pruning chord positions

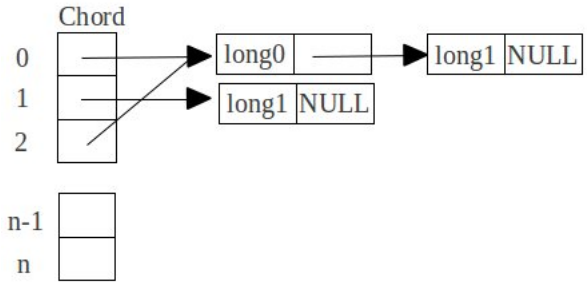

Figure 6.4: Sharing chord positions

to generate the Guitar tablature (from the sequence of chord playing positions along the path).

6.5.1 Fitness

Our guitar tablature generation method is not restricted to the particular fitness function we used. Our choice of fitness was chosen to suit the music type and playing style we happen to prefer, and to illustrate the method. Fitness has two parts: within-chord-fitness, and between-chord-fitness (our method will tolerate variation in the details of how they are computed). We use multiple criteria to judge the fitness of each playing possibility.

Within-chord-fitness

1. A measure of difficulty is the maximum distance between playing positions of notes in a chord (the distance between a fretted note and an open string is zero). For simplicity, we use a user-configurable array containing penalty values, indexed by the distance `d` measured in frets difference. However, since the *physical distance* between frets is not constant, it could be used for `d` instead.

2. A measure of difficulty is the left hand gesture which prefers small number of frets. For example, when a single-finger barre chord is played, only one finger is needed to press all the strings down at once on a single fret, which is simpler than using two or more fingers to press different frets. In this way the number of fret needs to be pressed is counted and a penalty `p` is given for each fret.

3. A measure of ease is the number of open strings. We use a user-configurable preference (Open_string) for having open strings.

4. When playing open-string harmonics, the string could be split into halves, thirds or fourths by plucking the string while touching it at a half, third, or fourth of its length. For example, playing at the 19th fret splits the string into thirds and playing a harmonic at the 7th or 19th fret is exactly the same note. Likewise, playing a harmonic at the 12th fret is the same as playing the open string note one octave higher. A chord which is a single note harmonic is given a reward as specified by the constant `Harmonic` in the configuration file. The use of harmonics can make an otherwise unplayable song playable. More generally, substituting some octave for a note in an unplayable chord can make it possible to play an otherwise unplayable song. Our program performs such substitution, controlled by the penalty constant substitution in the configuration file.

5. A measure of difficulty is the position — high or low on the neck of the guitar — where a chord is played (that is a preference for some guitar players). For simplicity, we use the square root of the average fret position as a penalty value.

6. A measure of preference is low string. For simplicity, we use the logarithm of one plus the sum of string numbers (strings are indexed from 0) as a penalty value.

Between-chord-fitness

1. By *ringing*, we mean a note may be sustained longer than required. Ringing notes necessarily prevent the string involved from being used to play a different note in the next chord. We use an user-configurable preference to reward ringing notes.

2. A measure of difficulty is the distance between the previous and current chord. A penalty value is used which increases with the distance `d` between chords (the position of a chord is the average of its nonzero fret numbers). If the current

position is nonzero but the previous position is zero, then previous chords are considered to locate the most recent nonzero position; that is used with the current positon to determine distance. This is important so that intervening open strings do not trivialize distance.

3. If there is a tie from the previous chord to the current chord, tied notes should have the same playing position. We use a user-configurable penalty value for each broken tie (preservation of ties is achieved by using a large penalty value).

4. A measure of consistency is the history. A guitar player may prefer to play a piece of music by reusing playing positions. History is maintained by the program to track whether a chord has appeared in the previous 8 chords. A reward (with decay rate) is given if a previously used playing position is chosen. Also, the past 8 average chord positions are treated as data points whose standard deviation is computed and then multiplied by by a user-configurable constant to get a penalty value. In this way, large persistent deviations or oscillations in hand position are penalized.

5. A measure of ease is the number of pivot fingers. A reward is given for each finger position shared between the previous playing position and the current playing position.

Rewards are negative, penalties are positive, and we seek to minimize fitness.

6.5.2 Pruning

Each chord at each level i has an associated *partial fitness*, which is the minimum fitness of paths from root to that chord; the fitness of a path is the sum of within-chord-fitness and between-chord-fitness over chords comprising the path (excluding

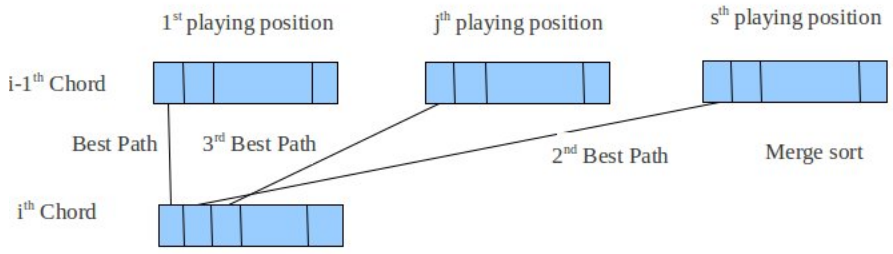

Figure 6.5: BFS Merge Sort

the root), and the minimum is taken over all such paths so-far encountered. Associated fitness is neither shared within nor between levels. We generalize this to find the best `Path` paths (`Path` is user configurable in the configuration file) and ignore the rest, using Dynamic Programming together with Breath-first Search of the chord tree.

6.5.3 Breadth First Search (Dynamic Programming)

In our breadth-first search using Dynamic Programming, a three dimensional matrix $m[s_1][s_2][s_3]$ is maintained to store the best paths. Here s_1 is the total number of chords, s_2 denotes the maximum number of chord playing positions for any chord, and s_3 paths are kept.

The matrix element $m[i][j][k]$ stores a triplet: $\langle pj, pk, f \rangle$. The matrix's indices denote the k_{th} best path for i_{th} chord's j_{th} playing position, while the stored triplet indicates the previous chord's playing position along the k_{th} path is represented by $m[i-1][pj][pk]$ with a partial fitness f.

Suppose (i, j) denotes the j_{th} playing position of the chord at level i. As the matrix at the $(i-1)_{th}$ level has been calculated, we choose paths to extend (see figure 6.5) by adding the within fitness of chord position $(i, 1)$ to the between fitness of $(i, 1)$

Table 6.1: Chord Playing Position and Within Fitness

$P_{11} = \langle 1, 3\rangle$	f_1
$P_{12} = \langle 2, 8\rangle$	f_2
$P_{13} = \langle 3, 12\rangle$	f_3
$P_{14} = \langle 4, 17\rangle$	f_4
$P_{21} = P_{31} = \langle 1, 10\rangle\langle 2, 13\rangle$	f_5
$P_{22} = P_{32} = \langle 2, 15\rangle\langle 3, 17\rangle$	f_6

playing position and the $(i-1, j)$ playing position to each path stored in the previous level (for each j). Then, merge the best `Path` paths. The program then continues the analogous process for $(i, 2)$, and so on.

When we finish the path calculation for all s_1 chord playing positions, the best `Path` paths can be derived by back tracing each path from the last chord using the triplet stored. For example, the best path is encoded by the triplet stored in $m[s_1 - 1][j][0]$ (there is only one playing position j=0 because the program ends the song with a rest). The stored triplet $\langle pj, pk, f\rangle$ indicates the index pj for the playing position of the penultimate chord, and the path is traced backwards using path pk of playing position pj of that chord.

Take the first 3 chords in figure 6.17 as an example. Suppose these three chords make up our song, and we have a 20-fret guitar. There are 4 playing positions for the first chord: $\langle 1, 3\rangle$, $\langle 2, 8\rangle$, $\langle 3, 12\rangle$ and $\langle 4, 17\rangle$; 2 play positions for the second and third chords: $\langle 1, 10\rangle\langle 2, 13\rangle$ and $\langle 2, 15\rangle\langle 3, 17\rangle$. Let P_{ij} be the j_{th} playing position for the i_{th} chord. The chords together with their within-chord fitness are shown in table 6.1, while the triplets stored in the matrix m are shown in figure 6.6.

For the first step, since there is no previous chord, the within chord fitness are stored: $m[0][0][0].f = f_1$, $m[0][1][0].f = f_2$, $m[0][2][0].f = f_3$ and $m[0][3][0].f = f_4$.

Next the second chord is considered. Paths in the first chord are extended by adding the within fitness of chord 2's first playing position, and the between chord fitness of each playing position of chord 1 and the first playing position of chord 2. Because there are 4 playing positions for the previous chord (and each playing position contains one path), 4 paths are sorted by fitness and stored in the matrix at positions $m[1][0][k]$ for k $\in$ [0, 4). The paths for the second playing positions of the second chord are extended analogously. Let $b_{ij,kl}$ be the between chord fitness for the playing positions P_{ij} and P_{kl}. According to the description above, the matrix elements fitness values are updated as shown in table 6.2:

Next the third chord is considered. Much the same as the former step, 8 paths are extended and stored sorted by fitness. Since the paths are sorted for the previous level, adding the same with-in and between fitness value preserves the ordering. This allows paths from the previous level to be merge sorted and stored in $m[2][0][k]$ (k $\in$ [0, 8)). Paths for the second playing positions of the third chord are constructed analogously. There are 16 paths for the two playing positions, as shown in table 6.4.

A rest is added at the end of the music. Since there is only one playing position for the rest (whose within fitness is zero), the path stored in $m[3][0][0]$ is the best path (with smallest fitness) as shown in table 6.4 (only the best 4 fitness are shown).

By back tracing using the indices stored in the triplets, the best path can be found (in figure 6.6, this corresponds to following arrows in reverse): $m[3][0][0]$, $m[2][0][0]$, $m[1][0][0]$, $m[0][2][0]$, which denotes the suggested guitar tablature: $\langle 3, 12\rangle$, $\langle 1, 10\rangle\langle 2, 13\rangle$ and $\langle 1, 10\rangle\langle 2, 13\rangle$.

Table 6.2: Updating Matrix Elements for the Second Chord

First Playing Position	Second Playing Position
$m[1][0][0].f = f_5 + b_{13,21} + m[0][2][0].f = -0.5$	$m[1][1][0].f = f_6 + b_{13,22} + m[0][2][0].f = 16$
$m[1][0][1].f = f_5 + b_{12,21} + m[0][1][0].f = 7.5$	$m[1][1][1].f = f_6 + b_{14,22} + m[0][3][0].f = 18$
$m[1][0][2].f = f_5 + b_{11,21} + m[0][0][0].f = 12.5$	$m[1][1][2].f = f_6 + b_{12,22} + m[0][1][0].f = 20$
$m[1][0][3].f = f_5 + b_{14,21} + m[0][3][0].f = 19.5$	$m[1][1][3].f = f_6 + b_{11,22} + m[0][0][0].f = 25$

Table 6.3: Updating Matrix Elements for the Third Chord

First Playing Position	Second Playing Position
$m[2][0][0].f = f_5 + b_{21,31} + m[1][0][0].f = 3.5$	$m[2][1][0].f = f_6 + b_{21,32} + m[0][0][0].f = 16$
$m[2][0][1].f = f_5 + b_{21,31} + m[1][0][1].f = 11.5$	$m[2][1][1].f = f_6 + b_{21,32} + m[0][0][1].f = 24$
$m[2][0][2].f = f_5 + b_{21,31} + m[1][0][2].f = 16.5$	$m[2][1][2].f = f_6 + b_{22,32} + m[0][1][0].f = 28$
$m[2][0][3].f = f_5 + b_{21,31} + m[1][0][3].f = 23.5$	$m[2][1][3].f = f_6 + b_{21,32} + m[0][0][2].f = 29$
$m[2][0][4].f = f_5 + b_{22,31} + m[1][1][0].f = 24.5$	$m[2][1][4].f = f_6 + b_{22,32} + m[0][1][1].f = 30$
$m[2][0][5].f = f_5 + b_{22,31} + m[1][1][1].f = 26.5$	$m[2][1][5].f = f_6 + b_{22,32} + m[0][1][2].f = 32$
$m[2][0][6].f = f_5 + b_{22,31} + m[1][1][2].f = 28.5$	$m[2][1][6].f = f_6 + b_{21,32} + m[0][0][3].f = 36$
$m[2][0][7].f = f_5 + b_{22,31} + m[1][1][3].f = 33.5$	$m[2][1][7].f = f_6 + b_{22,32} + m[0][1][3].f = 37$

Table 6.4: Updating Path Fitness for the Fourth Chord

First Playing Position	
$m[3][0][0].f = b_{31,41} + m[2][0][0].f = 3.5$	$m[3][0][1].f = b_{31,41} + m[2][0][1].f = 11.5$
$m[3][0][2].f = b_{32,41} + m[2][1][0].f = 16$	$m[3][0][3].f = b_{31,41} + m[2][0][2].f = 16.5$

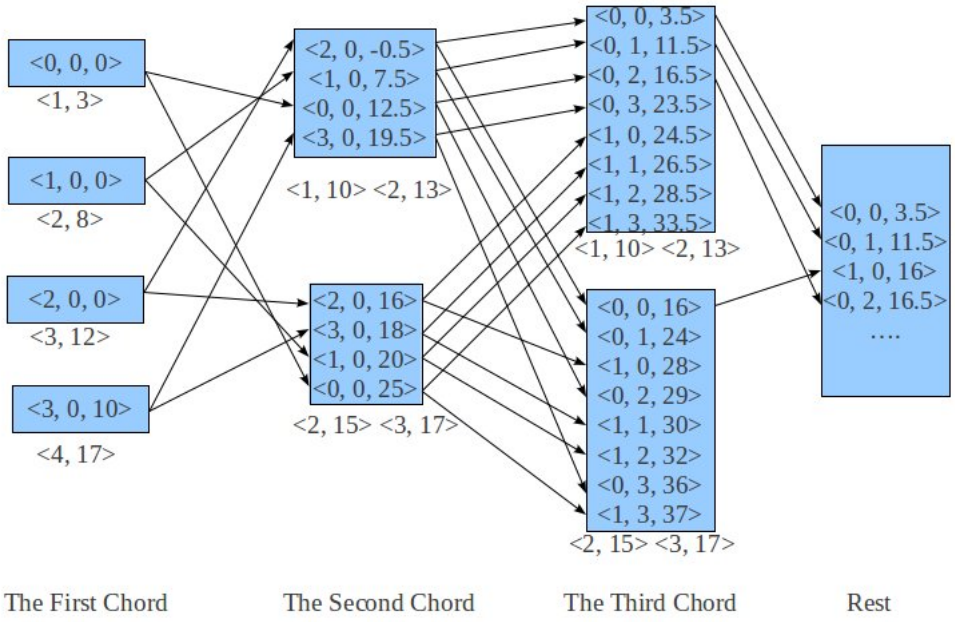

Figure 6.6: Breadth First Search Example

6.6 Guitar Tunings

Guitar tunings adjust the open string pitches to the guitar players' preference [wik12]. The standard tuning from low to high defines the string pitches as E, A, D, G, B, E. Sometimes it is difficult to play the music if the standard tuning is used. In that case, an alternate tuning might help.

In our method, a list of guitar tunings is maintained such as DADGAD and CGCDGA. A histogram of the midi pitches is calculated first, then the program shifts the midi pitches (and thereby shifts the histogram) so that most of the transposed midi pitches (as measured by the histogram) could be played in the given tuning. If the guitar tuning is not specified by the user, all the tunings maintained in the list are traversed and for each a suitable transposition maximizing playable notes is found. After the fitness value of each tuning is calculated, the best guitar tuning with the minimum fitness value is suggested and the corresponding transposition is reported.

6.7 Examples

In Section 1.2.2, TuxGuitar, TablEdit, Guitar Pro and PowerTab were tested on three midi files. Here our application called cTab uses the same files for testing purposes. A qualitative assessment of how the generated tablatures compare follows.

6.7.1 Example Analysis

What_a_wonderful_world.mid

Compared with Guitar Pro and tuxGuitar, cTab shows a more playable result with a minimized hand movement and more open strings (see figures 6.7, 6.8, 6.9, 6.10 and 6.11). The playing position suggested by TuxGuitar for the fifth chord in the second measure is unplayable because of the large stretch; cTab arrives the minimized hand movement among all the software for the second measure. Also, cTab prefers open strings, as the first chord of the third measure, the second chord and the fifth chord of the fourth measure compared with Guitar Pro.

Alone_again.mid

In this midi file: the fifth chord in the third measure is unplayable. cTab shows exactly what the note is (figure 6.12), while TuxGuitar, TablEdit and PowerTab show a wrong note, and Guitar Pro leaves out the note (see figures 6.13, 6.14, 6.15 and 6.16). In cTab, the use of octave substitution makes the unplayable chord playable; the notes on the twelfth fret should be played as open strings.

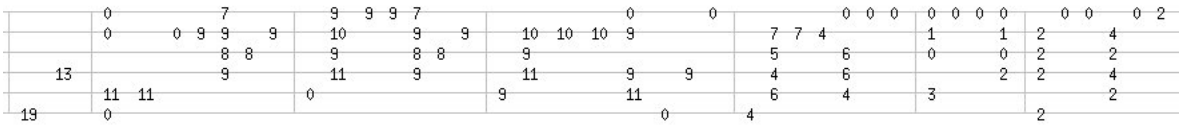

Figure 6.7: cTab Result: what_a_wonderful_world.mid

Moreover, cTab prefers ringing as for the second and fifth chords in the second measure, and the sixth chord in the fifth measure.

Close_to_you.mid

The results for Close_to_you.mid are given in figures 6.17 and 6.18. The first tablature is by Muriel Anderson, who is a world famous composer and guitar performer, and a winner of the National Fingerpicking Guitar Championship. The tablature generated by cTab matches Muriel Anderson's published tablature much better than the others (see figures 6.19, 6.20, 6.21, 6.22). The other software doesn't properly consider hand movement. Also, the harmonic in the fourth chord allows ringing, but the other software produces tablature which does not allow such ringing.

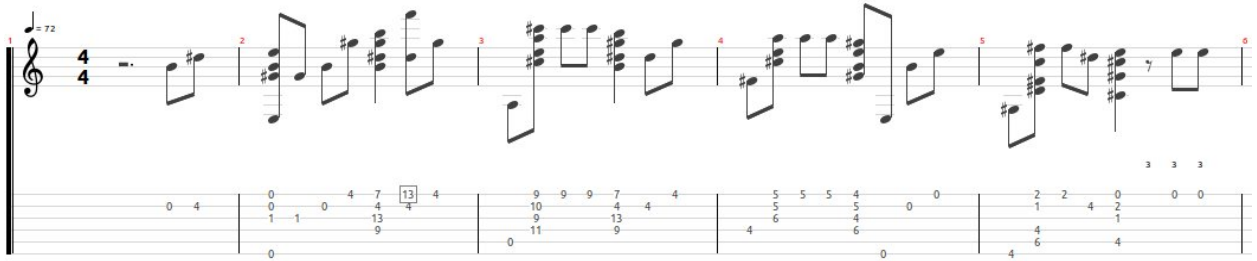

Figure 6.8: TuxGuitar Results: what_a_wonderful_world.mid

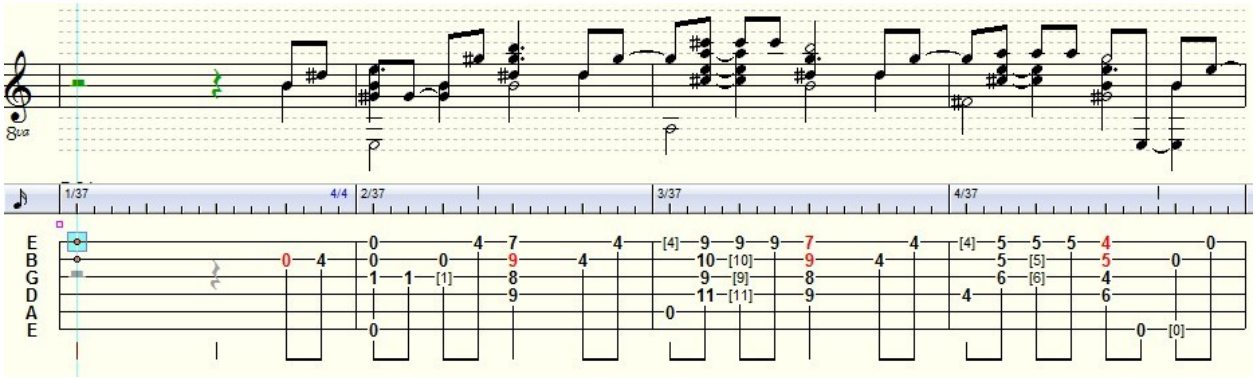

Figure 6.9: TablEdit Results: what_a_wonderful_world.mid

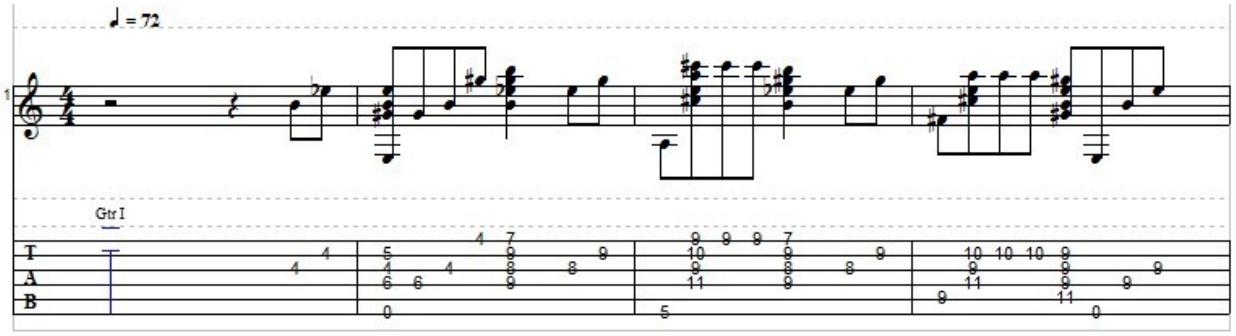

Figure 6.10: Power Tab Editor Results: what_a_wonderful_world.mid

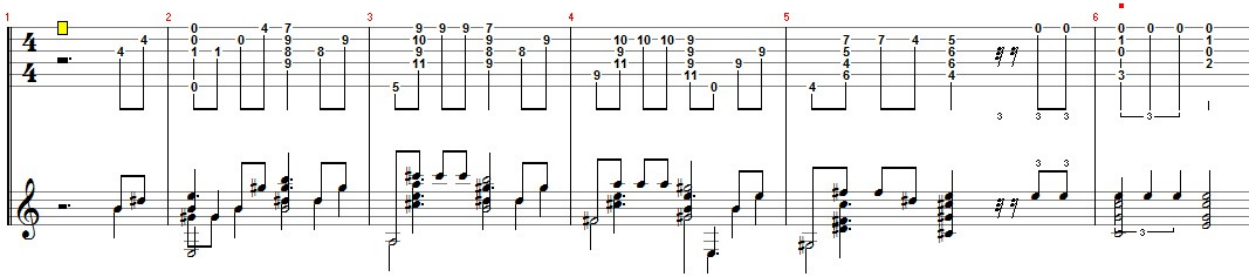

Figure 6.11: Guitar Pro Results: what_a_wonderful_world.mid

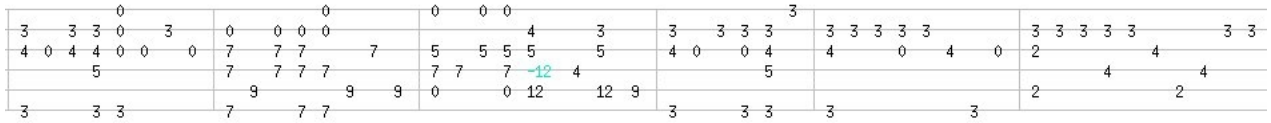

Figure 6.12: cTab Result: alone_again.mid

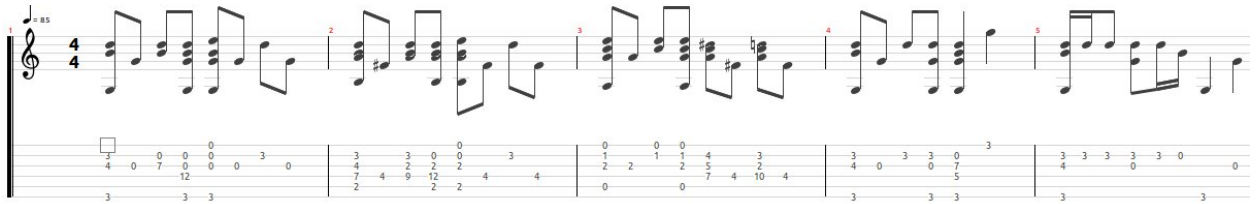

Figure 6.13: TuxGuitar Results: alone_again.mid

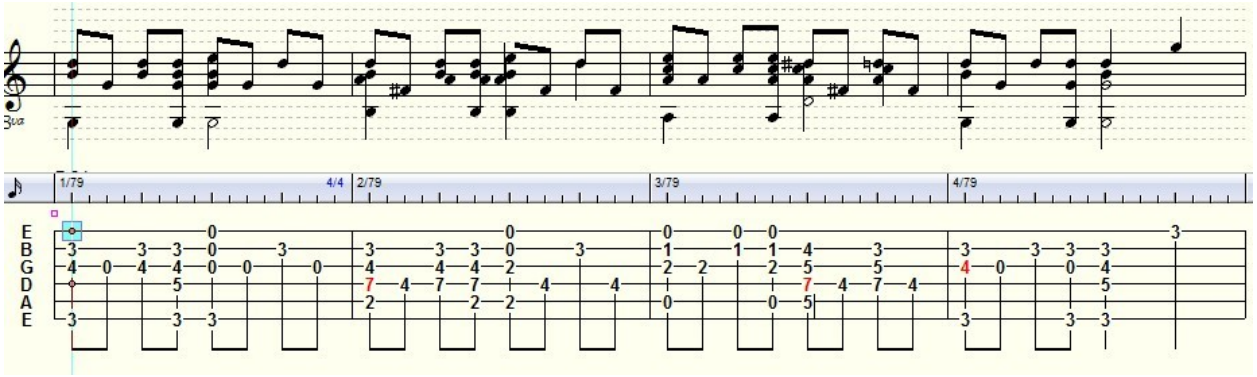

Figure 6.14: TablEdit Results: alone_again.mid

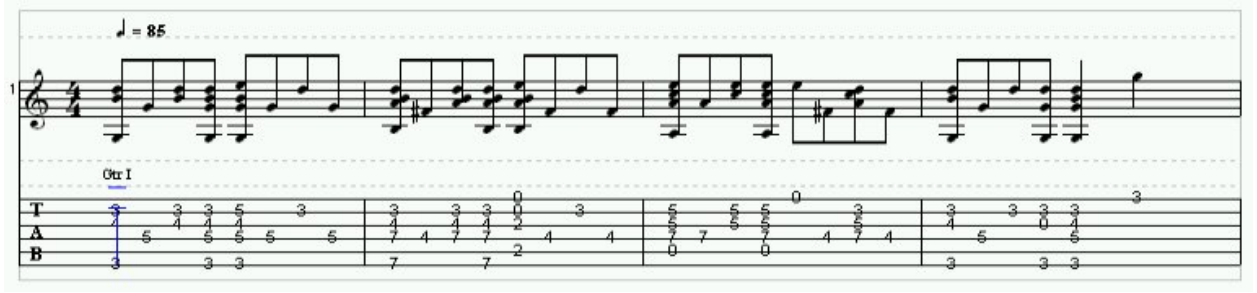

Figure 6.15: Power Tab Editor Results: alone_again.mid

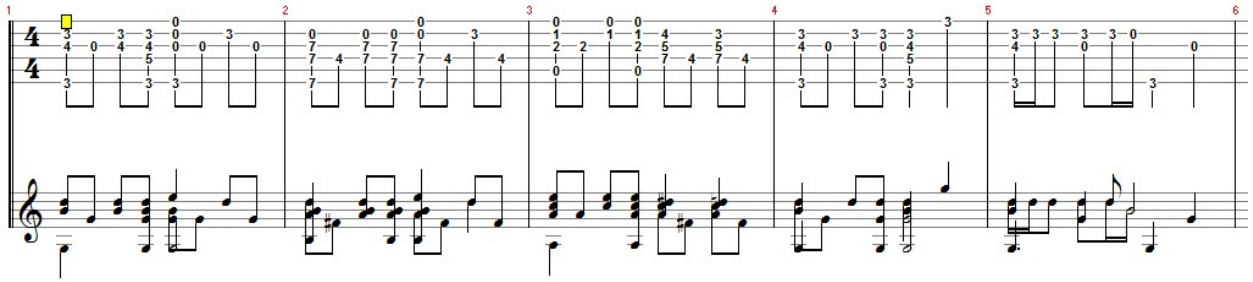

Figure 6.16: Guitar Pro Results: alone_again.mid

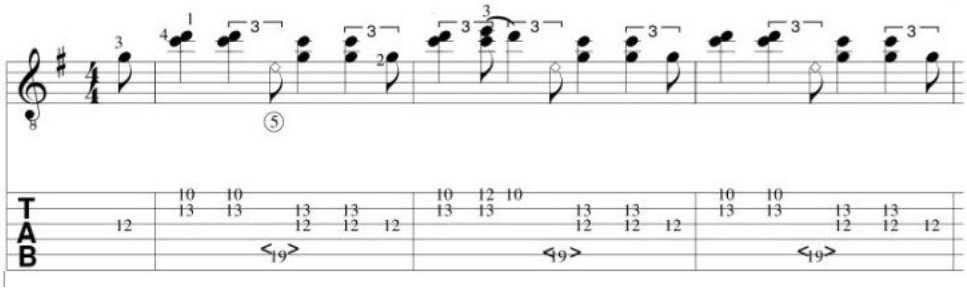

Figure 6.17: cTab Result: close_to_you.mid

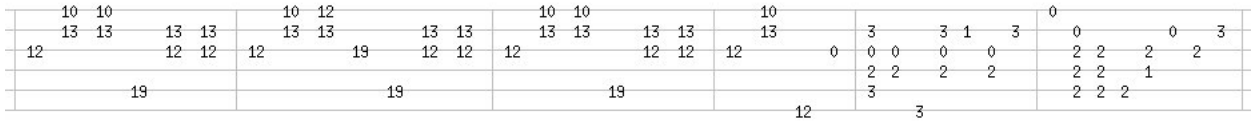

Figure 6.18: cTab Result: close_to_you.mid

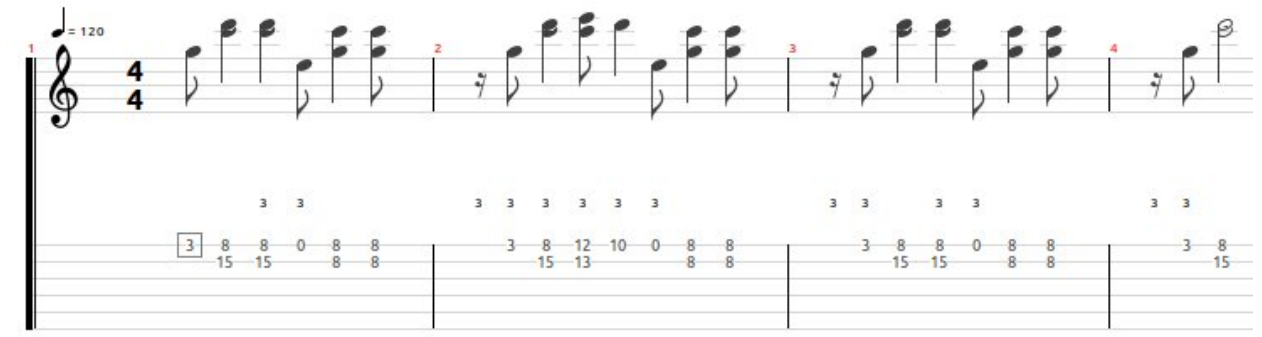

Figure 6.19: TuxGuitar Results: close_to_you.mid

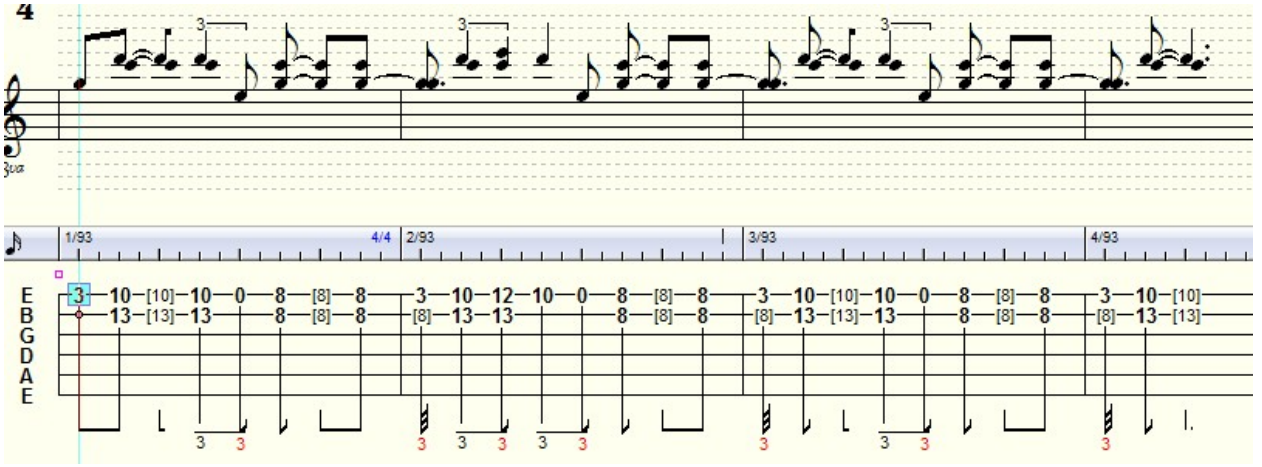

Figure 6.20: TablEdit Results: close_to_you.mid

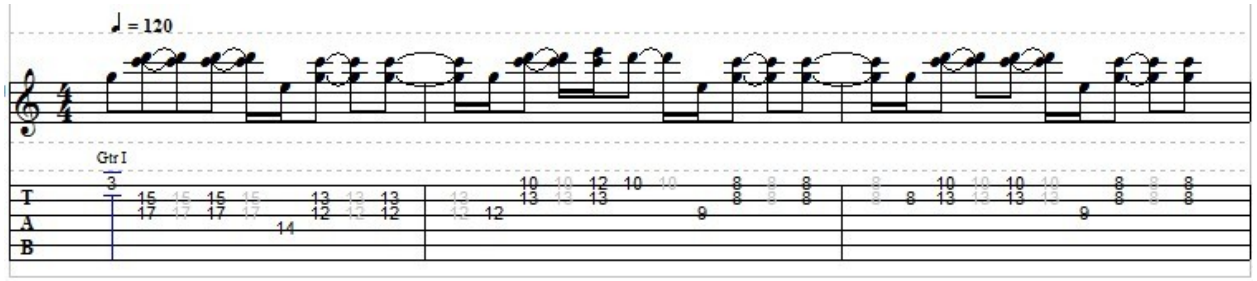

Figure 6.21: Power Tab Editor Results: close_to_you.mid

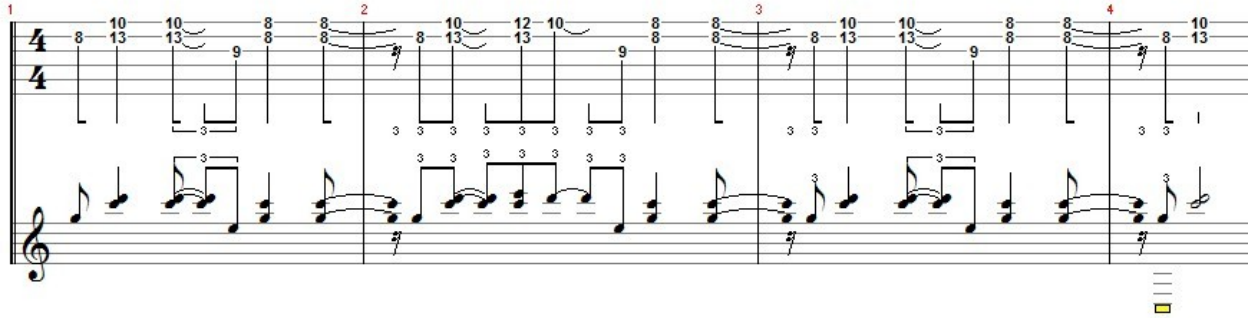

Figure 6.22: Guitar Pro Results: close_to_you.mid

6.7.2 Time Complexity

For the example "Close to You" (in figure 6.17), there are 10^{838} playing possibilities. In chord tree generation, only 1500 complete guitar tabs are explored by BFS and the rest are pruned. At last the best 100 results are derived in 0.01 seconds, if history is not used. With chord history enabled, the time increases to 0.4 seconds. The dynamic programming dominates the time complexity of parsing the MIDI file and generating feasible playing positions for the chords. If the song contains N Chords (containing replications) and there are at most M playing positions per chord, then computing the best k results is $O(kM^2N)$.

6.7.3 Memory Usage

In our program, the long integer type is utilized to store note and chord playing possibilities. This is memory efficient and also makes memory operations convenient. Here *valgrind* is used for memory usage testing, and it shows our program allocates about 1 MB memory [val12] for close_to_you.mid.

6.8 Conclusion

In this project, the depth-first Branch and Bound method is used to explore the search space of chord playing positions to find feasible ways to play chords. This method consists of a systematic enumeration of candidate solutions, where subsets of fruitless candidates are discarded. After obtaining the chord positions, a breadth-first dynamic programming approach is used to find an optimal way to play the song by choosing from the chord position possibilities.

Our algorithm is guaranteed to produce optimal solutions, which can be tailored according to user preference by modifying the configuration file. More generally, criteria could be added or removed from the computation of fitness.

The advantage of our algorithm over previous work includes:

- The optimum is guaranteed to be found, since the search space is completely enumerated. That is in contrast with heuristic methods (like genetic algorithms or neural networks), which are not guaranteed to return an optimal result.
- The music is not restricted to one-note chords, unlike the reference [RAL04], it also considers the whole midi music in contrast with the paper [fWL97] which only chooses the main channel in a MIDI file for processing.
- Our algorithm is time efficient: For example, by pruning the search tree to avoid considering all 10^{838} possibilities, only 0.01 seconds is needed to generate the guitar tablature for "Close to You"[Car70] when not using history, and only 0.4 seconds when using history.
- Our implementation is memory efficient, unlike the genetic algorithm which keeps 300 population in the mating pool [TP05]; only a path storage matrix is maintained.
- Consistency: In music, there are often repeated sections. Our program prefers the tablature to reuse playing positions which previously were found to be appropriate.
- Our program can automatically tune the guitar and transpose the music to find good tablature;
- Both hand's stretch and hand movement can be minimized. Preference can be given to harmonics, open strings, ringings, pivots, and tie preservation.

- Users can configure preferences – both for or against items listed above – so as to taylor the guitar tablature generation process by editing the configuration file.
- Users can move notes to a different string in the graphic user interface.
- Since the best 100 results are generated, user could choose from among many possibilities.

Chapter 7

Optical Music Recognition Result Evaluation

In presenting performance and recognition rates, OMR results are sometimes based on favorable evaluation methods. A standard methodology is necessary for the objective evaluation and comparison OMR systems.

7.1 Related Research

Nowadays, almost all objective OMR performance evaluation methods are based on the three metrics (described by the paper [BBN07b]) which are as follows:

7.1.1 Basic Symbols Recognition Evaluation

As mentioned in the reference, the first metric is the "basic symbol recognition" evaluation, which judges the recognition correctness of basic symbols such as beam, flag and note head.

The symbols used in the evaluation set are:

1. C: Correct basic music symbol.
2. F: False basic music symbol.
3. M: Missing basic music symbol.
4. T: Total basic music symbol: $T = C + F + M$;
5. R: Recognition rate.

Let $C_i = 1$ if the i_{th} music symbol is correct, otherwise C_i is 0. F_i and M_i are analogously defined. The basic symbol recognition rate is defined as:

$$R = \frac{\sum\limits_{0<i<T} C_i}{T} \tag{7.1}$$

in which $C_i + F_i + M_i = 1$

7.1.2 Complete Symbols Recognition and Relationships Reconstruction Evaluation

The goal of "complete music symbols and relationships reconstruction is to evaluate the capacity to recognize complete music symbols and music syntax". For example,

the identification of a note head does not imply the complete note recognition; it is characterized by its pitch, duration, and the presence of an accidental. The complete symbol recognition evaluation equation is the same as equation 7.1, but considers the complete music symbol.

7.1.3 Costs Needed to Correct Mistakes

Finally, recognition costs measures the extra work needed to correct mistakes. For example, "to re-format a measure, to rebuild music symbols at the end of recognition, and to insert or to create a new symbol or relationship by means of a music editor".

7.2 Midi File Evaluation

The three metrics mentioned in the last section consider the image recognition and image reconstruction results, but they fail to analyze the result of music semantic interpretation from the music side, i.e, the correctness of the resulting midi file. For example, several ingredients are ignored: whether the software can interpret several parallel voices in a single stave effectively by distributing them into corresponding tracks; whether the notes in the midi file are in the right sequence, and whether the key signatures in the music are correct.

In our evaluation method, another metric which analyzes the music midi file result is included with the metrics mentioned above. The correctness of the music midi events is judged based on values of the following symbols.

1. C: Correct midi events: the note's pitch, duration and sequence are correct.

2. F: False midi events: the note's pitch, duration or sequence are wrong.

3. M: Missing midi events.

4. T: Total midi events: $T = C + F + M$;

5. R: Correctness Rate.

The evaluation set most appropriate for generating guitar tablature is:

1. Note pitch in the midi event;

2. Note duration in the midi event;

3. Note sequence in the midi event;

4. Key signature in the meta event which influences all notes' pitch;

5. Time signature in the meta event which influences all notes' duration;

Let $C_i = 1$ if the i_{th} midi event is correct, otherwise C_i is 0. F_i and M_i are analogously defined. The midi file correctness rate evaluation equation is defined as below:

$$R = \frac{\sum\limits_{0<i<T} C_i}{T} \tag{7.2}$$

in which $C_i + F_i + M_i = 1$

7.3 OMR Evaluation Results

The third metric which estimates the extra work needed to correct mistakes is not used to evaluate the result, since we consider the definition of "extra work" as ambiguous,

subjective, and different from user to user. With the test image 1.2, the evaluation results of the OMR software mentioned in Section 1.1.4 and our application are presented below.

7.3.1 Basic Symbol Recognition Evaluation Results

The software recognition results are evaluated by counting by hand the correct, false, missing numbers of note heads (including standard solid notes, hollow notes and harmonics), flags, rests, ties, triplets, beams and key signatures, which are the most common and important symbols in the sheet music for generating guitar tablature. Results are summized in the table 7.1, 7.2, 7.3 and 7.4. Table 7.5 shows the recognition rate for all the basic symbols.

7.3.2 Complete Symbol Recognition Evaluation Results

Chord is the only element evaluated because it is the building block for guitar tablature. It is composed by note head, stem, flag, beam, triplet, tie, and augmentation dot. False recognition of any part results in miss-recognition of the chord. We summarize the correct, false, missing and total number of chord in table 7.6.

Table 7.1: PhotoScore Basic Symbol Recognition Evaluation Results

Symbol / Category	NoteHead	Flag	Rest	Tie	Triplet	Beam	KeySign
Correct(C)	243	35	8	23	4	19	10
False(F)	1	4	0	0	1	1	0
Missing(M)	10	0	1	10	14	0	0
Total(T)	254	39	9	33	19	20	10
Recognition Rate(R)	95.67%	89.74%	88.89%	69.7%	21.05%	95%	100%

Table 7.2: SharpEye Basic Symbol Recognition Evaluation Results

Symbol / Category	NoteHead	Flag	Rest	Tie	Triplet	Beam	KeySign
Correct(C)	176	11	5	1	1	13	5
False(F)	0	0	0	0	0	0	0
Missing(M)	78	28	4	32	18	7	5
Total(T)	254	39	9	33	19	20	10
Recognition Rate(R)	69.29%	28.21%	55.56%	3.03%	5.26%	65%	50%

Table 7.3: Audiveris Basic Symbol Recognition Evaluation Results

Symbol / Category	NoteHead	Flag	Rest	Tie	Triplet	Beam	KeySign
Correct(C)	223	38	9	0	19	20	10
False(F)	26	1	0	33	0	0	0
Missing(M)	5	0	0	0	0	0	0
Total(T)	254	39	9	33	19	20	10
Recognition Rate(R)	87.80%	97.44%	100%	0%	100%	100%	100%

Table 7.4: Our Application Basic Symbol Recognition Evaluation Results

Category \ Symbol	NoteHead	Flag	Rest	Tie	Triplet	Beam	KeySign
Correct(C)	253	35	9	21	18	20	10
False(F)	0	4	0	0	0	0	0
Missing(M)	1	0	0	12	1	0	0
Total(T)	254	39	9	33	19	20	10
Recognition Rate(R)	99.60%	89.74%	100%	63.63%	94.74%	100%	100%

Table 7.5: All Basic Symbol Recognition Evaluation Results

Category \ Software	PhotoScore	SharpEye	Audiveris	Our Application
Correct(C)	342	212	339	366
False(F)	7	0	40	4
Missing(M)	35	172	5	14
Total(T)	384	384	384	384
Recognition Rate(R)	89.06%	55.21%	88.28%	95.31%

Table 7.6: Complete Symbol Recognition Evaluation Results

Category \ Software	PhotoScore	SharpEye	Audiveris	Our Application
Correct(C)	131	57	121	153
False(F)	42	73	58	0
Missing(M)	6	49	0	26
Total(T)	179	179	179	179
Recognition Rate(R)	73.18%	31.84%	67.60%	85.47%

7.3.3 Midi File Evaluation Results

The test image 1.2 is a music sheet with a guitar tablature in it. Some software such as SharpEye tries to recognize guitar tab and write the guitar tablature recognition result into the midi file, which results in a low quality midi file although its recognition rate is higher than other commercial software. In order to prevent this disadvantage, all the guitar tablatures are removed before the recognition procedure.

Because the PhotoScore and SmartScore Demo versions have a restriction which prevents saving midi files, and OpenOMR doesn't provide functionality to play or save the midi result, only SharpEye could be used to evaluate the midi files as in table 7.7 (this project was not funded at a level to provide for the purchase of unrestricted versions of commercial software).

Table 7.7: Midi File Evaluation Results

Software / Category	SharpEye	Our Application
Correct(C)	82	182
False(F)	174	74
Missing(M)	0	0
Total(T)	256	256
Correctness Rate(R)	32%	71%

7.3.4 Conclusion

Based on the basic symbol recognition evaluation, complete symbol recognition evaluation and midi file evaluation results calculated above, the conclusions we make are the following:

PhotoScore can identify all the standard solid and hollow note heads, but miss all the harmonics. Also it can only find the second type of triplet mentioned in Section 4.4.3.

Audiveris is able to find all the triplets. However, for the second type of triplet, it only takes the number 3 near the notes into consideration but ignores the two corresponding brackets. This is the reason why some staccatissimos are also considered as triplets too. It also has trouble in finding the note head with a ledger line.

Our application can find almost all the basic symbols such note heads and beams using building-in rules. However, overlapped symbols may not be recognized well by our template comparing method. For example, some flags are overlapped with a tie in the test image. This problem could be solved by adding some build-in rules. (by considering the spacial relationship and shape characteristics, almost all flags could be recognized). Anther way to handle an overlapped symbol is to add additional templates to the catalog list. The mis-recognition of ties influences the midi file correctness rate since midi event durations are determined by factors which include the ties. However, since our program distributes the notes into different tracks effectively, our program still has a higher midi file correctness rate compared to other software.

7.4 Guitar Tablature Evaluation

Unlike the recognition task whose results could be analyzed quantitatively, it is hard to define a quantitative analysis criterion for guitar tablature evaluation since each guitar player's preference is different. However, a qualitative analysis was made in Section 6.7.

Figure 7.1: Alone Again cTab Result (Open G Tuning)

This section concludes with a final example which demonstrates the effectiveness of our cTab program. When run without specifying any particular tuning, ctab explores tunings and transpositions to generate optimal tablature. The result (which most guitar players would find pleasing) is the following arrangement in Open G tunning (DGDGBD) for Gilbert O'Sullivan's "Alone Again" (see figure 7.1).

Chapter 8

Future Work and Conclusion

Optical Music Recognition is a comprehensive topic in computer science which is related to music knowledge, image processing, pattern recognition, artificial intelligence, syntax and semantics analysis, data structures and programming. Improvement in any part could result in the improvement of the whole system.

8.1 Finer Level Parallelization

Coarse grain Pthread parallelization is implemented in our application. However, it only works at the page level, rather than at the stave level or music symbol level. Moreover, only one thread is used for image recognition if our program takes a single page of sheet music as an input. However, the staves in this single page could be recognized by several threads, one for each stave. Hence if threads work in parallel at the stave level rather than the page level, better performance could be achieved.

Our Pattern Recognition method mentioned in Section 3.7 is particularly well suited for efficient music symbol level parallelization because each music symbol's recognition is relatively independent, and there could be hundreds of music symbols on each page. It also makes sense to look into mapping our technique to a Graphical Processing Unit (GPU) architecture.

8.2 Conclusion

In this dissertation, the concepts and related implementation technology of OMR is introduced, and image format conversion is discussed. If the image is tilted or distorted due to poor scanned quality, the image is restored to a more ideal alignment by a method which can handle curvature as well as rotation.

In the image processing procedure, so as to eliminate symbol interference, all staff lines and stems are located and removed. Afterwards, the music symbols are located and identified: some simple music symbols are recognized based on the built-in rules while other symbols are identified by a comparing method. After the recognition procedure, music symbols are sorted and stored into different queues.

The analysis and design of image reconstruction and semantic interpretation is also important for OMR. The spatial relationship between the music symbols is considered next in order to integrate the music symbols. Then the calculation of each note's pitch, duration and starting time is done according to the results of semantic interpretation. Based on the results, from the former procedures, the midi file is generated.

The guitar tablature generation process is the focus of the second part of this dissertation. At first note pitch, chord pitches and their corresponding playing positions are computed and represented in a memory-saving manner. After importing

midi files into a sequence of chords, the chord playing positions are calculated by depth first search and branch and bound. Then, using with-in and between fitness functions which evaluate the chord positions, the entire chord tree is traversed using effective pruning based on dynamic programming. Preference is also given to chords that have historically been found to be suitable. Finally, different guitar tunings and transpositions are explored and the best possibilities are suggested to the user.

A comparison between our application and other available software is made using three objective metrics. From the results of basic recognition rate, complete recognition rate, and midi file correctness, we can see the performance of our system is superior to the other open source software and the commercial counterparts.

Bibliography

Bibliography

[ACD00] É. Anquetil, B. Couasnon, and F. Dambreville. A symbol classifier able to reject wrong shapes for document recognition systems. *Graphics Recognition Recent Advances*, pages 209–218, 2000. 8

[Bai94] D. Bainbridge. A complete optical music recognition system: Looking to the future. 1994. 5

[Bai96] D. Bainbridge. Optical music recognition: A generalised approach. In *Second New Zealand Computer Science Graduate Conference*, 1996. 5

[Bai97] D. Bainbridge. Extensible optical music recognition. *Unpublished doctoral dissertation, Department of Computer Science, University of Canterbury, Canterbury, New Zealand*, 1997. 5

[BB01] D. Bainbridge and T. Bell. The challenge of optical music recognition. *Computers and the Humanities*, 35(2):95–121, 2001. 5

[BB03] D. Bainbridge and T. Bell. A music notation construction engine for optical music recognition. *Software: Practice and Experience*, 33(2):173–200, 2003. 5

[BBN01] P. Bellini, I. Bruno, and P. Nesi. Optical music sheet segmentation. In *Web Delivering of Music, 2001. Proceedings. First International Conference on*, pages 183–190. IEEE, 2001. 7

[BBN03] P. Bellini, I. Bruno, and P. Nesi. Multilingual lyric modeling and management, 2003. 7

[BBN04] P. Bellini, I. Bruno, and P. Nesi. An off-line optical music sheet recognition. *Visual Perception of Music Notation: On-Line and Off-Line Recognition. Hershey, Pennsylvania: Idea Group*, pages 40–77, 2004. 7

[BBN07a] P. Bellini, I. Bruno, and P. Nesi. Assessing optical music recognition tools. *Computer Music Journal*, 31(1):68–93, 2007. 7

[BBN07b] Pierfrancesco Bellini, Ivan Bruno, and Paolo Nesi. Assessing optical music recognition tools. *Comput. Music J.*, 31(1):68–93, March 2007. 99

[BBN08] P. Bellini, I. Bruno, and P. Nesi. Optical music recognition: Architecture and algorithms. *Interactive multimedia music technologies*, page 80, 2008. 7

[BC97] D. Bainbridge and N. Carter. Automatic reading of music notation, 1997. 5

[Bit11] Herv Bitteur. Audiveris music scanner, Auguest 2011. 11

[BNMW+99] D. Bainbridge, C.G. Nevill-Manning, I.H. Witten, L.A. Smith, and R.J. McNab. Towards a digital library of popular music. In *Proceedings of the fourth ACM conference on Digital libraries*, pages 161–169. ACM, 1999. 5

[BTSB12] I. Barbancho, L.J. Tardon, S. Sammartino, and A.M. Barbancho. Inharmonicity-based method for the automatic generation of guitar tablature. *Audio, Speech, and Language Processing, IEEE Transactions on*, 20(6):1857 –1868, aug. 2012. 18, 75

[Car70] Carpenters. Carpentersclosetoyou, Aug 1970. 73, 97

[Car89] N.P. Carter. Automatic recognition of printed music in the context of electronic publishing. phd thesis, University of Surrey, 1989. 6

[Car92] N. P. Carter. Conversion of the haydn symphonies into electronic form using automatic score recognition: A pilot study. *Proceedings of SPIE 2181*, pages 279–290, 1992. 6

[Car94] N. P. Carter. Music score recognition: Problems and prospects. *Computing in Musicology*, page 152, 1994. 6

[CBSB95] B. Couasnon, P. Brisset, I. Stephan, and C.P. Brisset. Using logic programming languages for optical music recognition. In *In Proceedings of the Third International Conference on The Practical Application of Prolog*. Citeseer, 1995. 8

[CC94] B. Couasnon and J. Camillerapp. Using grammars to segment and recognize music scores. In *International Association for Pattern Recognition Workshop on Document Analysis Systems*, pages 15–27, 1994. 8

[CC95] B. Couasnon and J. Camillerapp. A way to separate knowledge from program in structured document analysis: application to optical music recognition. In *icdar*, page 1092. Published by the IEEE Computer Society, 1995. 8

[CDD+00] G.S. Choudhury, T. DiLauro, M. Droettboom, I. Fujinaga, B. Harrington, and K. MacMillan. Optical music recognition system within a large-scale digitization project. In *Proceedings ISMIR 00*. Citeseer, 2000. 4

[Cou06] B. Couasnon. Dmos, a generic document recognition method: application to table structure analysis in a general and in a specific

way. *International Journal on Document Analysis and Recognition*, 8(2):111–122, 2006. 8

[CRUE95] B. Couasnon, B. Rétif, C. Uasnon, and B.R. Etif. Using a grammar for a reliable full score recognition system. 1995. 8

[Dan12] Danelectro. Danelectroguitar, Dec 2012. 73

[DDPF07] C. Dalitz, M. Droettboom, B. Pranzas, and I. Fujinaga. A comparative study of staff removal algorithms. *IEEE transactions on pattern analysis and machine intelligence*, pages 753–766, 2007. 4

[Des06] Arnaud F. Desaedeleer. Reading sheet music. diploma thesis, Imperial College, University of London, September 2006. 13

[dig13] digitalpreservation. Midi file format, Mar 2013. 68

[FAPB89] I. Fujinaga, B. Alphonce, B. Pennycook, and N. Boisvert. Optical recognition of music notation by computer. *Computers in music research*, 1:161–164, 1989. 4

[FAPH91] I. Fujinaga, B. Alphonce, B. Pennycook, and K. Hogan. Optical music recognition: Progress report. In *Proceedings of the International Computer Music Conference*, pages 66–66. INTERNATIONAL COMPUTER MUSIC ACCOCIATION, 1991. 4

[FMS98] I. Fujinaga, S. Moore, and D.S. Sullivan. Implementation of exemplar-based learning model for music cognition. In *Proc. of the International Conference on Music Perception and Cognition*, pages 171–179. Citeseer, 1998. 4

[FP97] I. Fujinaga and B. Pennycook. Adaptive optical music recognition. *McGill University, Montreal, Que., Canada*, 1997. 4

[Fuj88] I. Fujinaga. Optical music recognition using projections. *Master's thesis, McGill University, Faculty of Music, Montreal, Canada*, 1988. 4

[Fuj96] I. Fujinaga. Exemplar-based learning in adaptive optical music recognition system. In *Proceedings of the International Computer Music Conference*, pages 55–56. Citeseer, 1996. 4

[Fuj04] I. Fujinaga. Staff detection and removal. *Visual Perception of Music Notation*, pages 1–39, 2004. 4

[fWL97] Jeng feng Wang and Tsai-Yen Li. Generating guitar scores from a midi source, 1997. 17, 97

[Gui13] GuitarPro. Guitarpro, Mar 2013. 18, 25

[KOS+87] I. Kato, S. Ohteru, K. Shirai, T. Matsushima, S. Narita, S. Sugano, T. Kobayashi, and E. Fujisawa. The robot musician [] wabot-2'(waseda robot-2). *Robotics*, 3(2):143–155, 1987. 4

[lil12a] lilypond.org. Lilypond, Dec 2012. 71

[lil12b] lilypond.org. Lilypond manual, Dec 2012. 71

[MACdR05] S. Macé, É. Anquetil, B. Couasnon, and I.I. de Rennes. A generic method to design pen-based systems for structured document composition: Development of a musical score editor. In *Proc. of the 1st Workshop on Improving and Assessing Pen-Based Input Techniques*, pages 15–22. Citeseer, 2005. 8

[Mat88] Toshiaki Matsushima. Printed music to braille. *Journal of information processing*, 11(4):249, 1988. 4

[Mat89] T. Matsushima. Automatic printed-music-to-braille translation system. *Journal of information processing*, 11(4):249–257, 1989. 4

[MDF02] K. MacMillan, M. Droettboom, and I. Fujinaga. Gamera: Optical music recognition in a new shell. In *Proceedings of the International Computer Music Conference*, pages 482–485. Citeseer, 2002. 4

[Med12] Keith Medley. keithmedleymusic, Dec 2012. 73

[MHHY04] M. Miura, I. Hirota, N. Hama, and M. Yanagida. Constructing a system for finger-position determination and tablature generation for playing melodies on guitars. *Systems and Computers in Japan*, 35(6):10–19, 2004. 15, 17

[MHS+85] T. Matsushima, T. Harada, I. Sonomoto, K. Kanamori, A. Uesugi, Y. Nimura, S. Hashimoto, and S. Ohteru. Automated recognition system for musical score–the vision system of wabot-2. *Bulletin of Science and Engineering Research Laboratory, Waseda University*, 112:25–52, 1985. 4

[NB92] KC Ng and RD Boyle. Segmentation of music primitives. In *British Machine Vision Conference*, pages 472–480, 1992. 5

[NB96a] K.C. Ng and R.D. Boyle. Recognition and reconstruction of primitives in music scores. *Image and Vision computing*, 14(1):39–46, 1996. 5

[NB96b] KC Ng and RD Boyle. Reconstruction of music scores from primitive sub-segmentation. *Image and Vision Computing*, pages 39–46, 1996. 5

[NBC95a] KC Ng, RD Boyle, and D. Cooper. Automated optical musical score recognition and its enhancement using high-level musical knowledge. In *Proceedings of the XI Colloquium on Musical Informatics*, pages 167–170, 1995. 5

[NBC95b] KC Ng, RD Boyle, and D. Cooper. Low-and high-level approaches to optical music score recognition. In *Document Image Processing and*

Multimedia Environments, IEE Colloquium on, pages 3–1. IET, 1995. 5

[net13] netpbm.sourceforge.net. Netpbm, Apr 2013. 28

[Ng95] KC Ng. Automated computer recognition of music scores. *School of Computer Studies. Leeds, GB, University of Leeds*, 1995. 5

[Ng01] KC Ng. Optical music recognition: Stroke tracing and reconstruction of handwritten manuscripts. In *Joint International Conference of the Association for Computers and the Humanities and the Association for Literary and Linguistic Computing. New York, New York*, volume 13, page 16, 2001. 5

[Ng02] K. Ng. Music manuscript tracing. *Graphics Recognition Algorithms and Applications*, pages 330–342, 2002. 5

[Ng11] Dr. Kia Ng. Optical manuscript analysis, June 2011. 5

[NJ03] K.C. Ng and A. Jones. A quick-test for optical music recognition systems. In *2nd MUSICNETWORK Open Workshop, Workshop on Optical Music Recognition System, Leeds*, 2003. 5

[PBF07] L. Pugin, J.A. Burgoyne, and I. Fujinaga. Map adaptation to improve optical music recognition of early music documents using hidden markov models. In *Proceedings of the 8th International Conference on Music Information Retrieval*, pages 513–6, 2007. 4

[pos12] posix. Posix thread, Dec 2012. 70

[Pow13] PowerTab. Powertab, Mar 2013. 18, 25

[Pre70] David Stewart Prerau. Computer pattern recognition of standard engraved music notation. Ph. d. dissertation, Massachusetts Institute of Technology, 1970. 3

[Pru66] Dennis Howard Pruslin. Automatic recognition of sheet music. Sc. d. dissertation, Massachusetts Institute of Technology, 1966. 3

[RAL04] Daniele Radicioni, Luca Anselma, and Vincenzo Lombardo. A segmentation-based prototype to compute string instruments fingering. In *Proceedings of the Conference on Interdisciplinary Musicology*, 2004. 17, 97

[RD04a] A. Radisavljevic and P. Driessen. Path difference learning for guitar fingering problem. In *Proceedings of the International Computer Music Conference*. Citeseer, 2004. 15

[RD04b] Er Radisavljevic and Peter Driessen. Path difference learning for guitar fingering problem. In *In Proceedings of the International Computer Music Conference. International Computer Music Association*, 2004. 18

[Ree95] T. Reed. Optical music recognition. 1995. 8

[Rot93] M. Roth. *OMR Optical Music Recognition*. PhD thesis, NCR Stiftung, 1993. 7

[Rot94] M. Roth. *An approach to recognition of printed music*. Eidgen "ossische Technische Hochschule Departement Informatik Institut f "ur theoretische Informatik, 1994. 7

[RP96] K.T. Reed and JR Parker. Automatic computer recognition of printed music. In *Pattern Recognition, 1996., Proceedings of the 13th International Conference on*, volume 3, pages 803–807. IEEE, 1996. 8

[Rut09] N. Rutherford. Fingar, a genetic algorithm approach to producing playable guitar tablature with fingering instructions. *Undergraduate project dissertation, Dept. of Computer Sci., Univ. of Sheffield*, 2009. 15

[son12] sonicspot. Midi file format, Dec 2012. 76

[TA12] Camille Goudeseune Terry Allen. Topological considerations for tuning and fingering stringed instruments. 2012. 17

[Tab13] TablEdit. Tabledit, Mar 2013. 18, 19

[TDSR04] Ernesto Trajano, Mrcio Dahia, Hugo Santana, and Geber Ramalho. Automatic discovery of right hand fingering in guitar accompaniment. In *In Proceedings of the International Computer Music Conference (ICMC04*, pages 722–725, 2004. 17

[ton12] tonalsoft. Midi note number, Dec 2012. 76

[TP05] D. Tuohy and WD Potter. A genetic algorithm for the automatic generation of playable guitar tablature. In *Proceedings of the International Computer Music Conference*, pages 499–502. Citeseer, 2005. 15, 16, 97

[TP06a] D. Tuohy and W. Potter. Generating guitar tablature with lhf notation via dga and ann. *Advances in Applied Artificial Intelligence*, pages 244–253, 2006. 15, 16

[TP06b] DR Tuohy and WD Potter. Creating guitar tablature with neural networks and distributed genetic search. In *Proc. International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems Annecy, France*, 2006. 15, 16

[TP06c] D.R. Tuohy and W.D. Potter. Ga-based music arranging for guitar. In *Evolutionary Computation, 2006. CEC 2006. IEEE Congress on*, pages 1065–1070. IEEE, 2006. 15, 16

[TPC06] D.R. Tuohy, WD Potter, and A.I. Center. An evolved neural network/hc hybrid for tablature creation in ga-based guitar arranging. In

Proceedings of the International Computer Music Conference. Citeseer, 2006. 15, 16

[Tux13] TuxGuitar. Tuxguitar, Mar 2013. 18, 19

[val12] valgrind.org. valgrind, Dec 2012. 96

[wik11] wikipedia. Music ocr, June 2011. 1

[wik12] wiki. guitartuning, Dec 2012. 86

[wik13a] wiki. guitartabsoftware, Mar 2013. 18

[wik13b] wikipedia. Key$_s$*ignature*, *Apr*2013.57

[wik13c] wikipedia. Mmap, Apr 2013. 29

[wik13d] wikipedia. Music notation, Apr 2013. 2

[wik13e] wikipedia. Runlengthencoding, Apr 2013. 33

Vita

Chuanjun HE was born and raised in Ganzhou, P.R.China on Jan 21, 1985. He received a B.S. in computer science from Nanchang University, P.R.China in 2002 and a M.S. in computer science from the Graduate University, Chinese Academy of Sciences in 2009.

He completed his doctorate in computer science at the University of Tennessee in May, 2013.