Computer-aided Learning Of Music

Zervopoulos Thanassis

17 November 2006

Contents

Chapter 1

Acknowledgements

I dedicate this thesis to my father *Aristeidis Zervopoulos*, my mother *Aggeliki Zervopoulou*, my brother *Yannis Zervopoulos* and my beloved friends who supported me all these years.

Chapter 2

Abstract

In this thesis we present an application for computer-aided learning of music. The application is based on the comparison of two musical scores. We investigate two alternative algorithms for the comparison: a simple matching algorithm and the longest common subsequence algorithm, which is based on dynaming programming. The Music Match (MM) application that we developed includes additional functionalities, such as playing and recording a song, showing the sheet notation of a song and also representing the results of the comparison between two musical scores in a sheet notation, using color coding to present the errors to the user.

Chapter 3

Introduction

3.1 Motivation

As the division of e-learning is rapidly developing in many areas, in the department of music great efforts have been made. There are a lot of programs such as composing music, playing music, writing a music sheet notation, but there is nothing like an e-teacher of music. It would be nice for those who know how to play an instrument to get deeper in a music analysis, to compare their songs with the original songs played by famous musicians.

The main idea was to create an electronic teacher of music. A student reads a music sheet notation and tries to play that in a keyboard. While playing there is a great possibility of errors. Those errors could be errors of four types: a note name error, a time error according to the tempo, a duration error of a note and an amplitude error of a note. The teacher should correct the student according to his-her level and present him-her the errors. There are many kinds of teachers in real life. There are strict teachers, but there are also patient and tolerant teachers. In contrast to real life, the student chooses his-her teacher any time he-she wants to. Thus, any time that a song is played, the student is able to compare it with the original song without the aid of a teacher...at least not a real one.

3.2 Prior Art

This paper cites two algorithms which are able to find relevant occurences between two musical scores. Both algorithms are based on the main idea of a string pattern matching algorithm. Instead of strings, musical sequences have been used. Those thesis below, proposed also some algorithms which are able to find similarities and dissimilarities between the query musical pattern sequence and the target music.

- **A Pattern Extraction Algorithm For Abstract Melodic Represantations That Allow Partial Overlapping Of Intervallic Categories**

  Emilios Cambouropoulos, Maxime Crochemore, Costas Iliopoulos, Manal Mohamed, Marie-France Sagot

  An efficient pattern extraction algorithm is proposed in this paper that can be applied on melodic sequences. Thoses sequences are represented as strings of abstract intervallic symbols and can find maximal repeating pairs and repetitions with two matching subsequences separated with an intervening non-matching symbol.

- **An Approximate String Matching Algorithm For Content-Based Music Data Retrieval**

  Chih Chin Liu, Jia-Lien Hsu, Arbee L. P. Chen

  This paper presents an approach for content-based music data retrieval by transforming the problem into the string matching problem and providing a new approximate string matching algorithm which provides fault tolerance ability according to the music characteristics.

- **Audio Indexing For Efficient Music Information Retrieval**

  Ioannis Karydis, Alexandros Nanopoulos, Apostolos N. Papadopoulos, Yannis Manolopoulos

  A novel algorithm is recommended in this paper. The algorithm efficiently retrieves audio data similar to an audio query. Moreover, a novel false alarm resolution method is outlined that utilises a reverse order schema while calculating the distance of the query and results, in order to avoid costly operations.

- **Time-Warped Longest Common Subsequence Algorithm For Music Retrieval**

  AnYuan Guo, Hava Siegelmann

  This paper proposes the Time-Warped Longest Common Subsequence algorithm which deals with singing errors involving rhythmic distortions. The algorithm is employed in song retrieval tasks, where its performance is compared to the longest common subsequence algorithm.

- **Efficient Algorithms For The δ-Approximate String Matching Problem In Musical Sequences**

  Domenico Cantone, Salvatore Cristofaro, Simone Faro

  This paper proposes two new algorithms upon the δ-approximate string matching problem. These algorithms achieve the best performances in the case of large alphabets and short patterns, which typically occur in practical situations in music retrieval.

- **String Matching And Geometric Algorithm For Melodic Similarity**

  Kjell Lemström, Niko Mikkilä, Veli Mäkinen, Esko Ukkonen

  This paper cites two algorithms which are able to find musically relevant occurences of the query pattern in the target music. The first one is suitable for assessing the similarity of monophonic sequences based on the string matching framework and the other one is suitable for polyphonic sequences based on geometric matching.

- **Comparison Of Musical Sequences**

  Marcel Mongeau, David Sankoff

  This paper suggests a dynamic programming algorithm that measure the overall similarity and dissimilarity between two musical scores. The measure of comparison is defined so as to detect similarities in melodic line despite gross differences in key, mode or tempo. The key element is the notion of consolidation and fragmentation, which differ both from the deletions and insertions familiar in sequence comparison, and from the compressions and expansions of time warping in automatic speech recognition.

- **Finding Maximum-Length Repeating Patterns In Music Databases**

Ioannis Karydis, Alexandros Nanopoulos, Yannis Manolopoulos

This paper proposes a novel algorithm which discovers all maximum-length repeating patterns using an "aggressive" accession during searching, by avoiding costly repetition frequency calculation and by examining as few as possible repeating patterns in order that maximum-length repeating pattern(s) could be reached.

- **O-Generator World Music (African and Latin)**

  EducationGuardian.co.uk E-learning

  O-Generator World Music is an education program. It demonstrates African and Latin American rhythms and enables students to create their own tracks that can be represented either in O-Generator mode or as traditional notation. It supports a record, save and export music utility.

- **Neuratron AudioScore Professional 3**

  www.audioscore.com

  AudioScore allows the users to create musical scores, using a MIDI keyboard or a microphone and uses highly sophisticated pitch recognition and the most advanced artificial intelligence technology to determine the users intended rhythm and create accurate and musical score notation.

- **Cakewalk**

  www.cakewalk.com

  Cakewalk is the world's leading developer of powerful and easy to use products for music creation and recording. These products include award-winning digital audio workstations and sequencers, fully-integrated music software and hardware solutions, and innovative virtual instruments.

3.3 Thesis Contribution

This thesis presents an application according to which two songs could be compared, using a brute-force algorithm and – from biological applications

– a dynamic-programming algorithm named LCS (Longest Common Subsequence).Those algorithms have been extended with some additional properties to support fault tolerance ability. The results of the comparison are demonstrated in a sheet notation.

The above application supports the followings:

The user could listen to a midi file, create a midi file by using a midikeyboard and save it to hard disk, extract a sheet notation from a midi file, compare their midi file with the original one and see the results in a new sheet notation by using either a brute-force algorithm or a dynamic-programming algorithm. They could also make a transporto in a song, open a sheet notation and save it.

3.4 Thesis Outline

A quick look at the table of contents shows that there are 6 chapters represented below.

Chapter 3: *covers the functional and technical analysis of the MM application. Discusses the algorithms that are being used and also describes how a user could handle this application.*

Chapter 4: *is devoted to the tools that are being used to implement the MM application.*

Chapter 5: *contains all the useful data, for someone, to implement the MM application.*

Chapter 6: *demonstrates the evaluation stage of the MM application.*

Chapter 7: *represents some conclusions and what is next to be done.*

Chapter 8: *its the appendix. It also displays the precious source code.*

Chapter 9: *its the reference guide.*

Chapter 4

The MM (Music Match) Application

4.1 Functional Description

The user of an electronic teacher of music should be able to listen to a song. After understanding the melody, the user should see the sheet notation of this song. Then they should try to play this song and record it through a midi keyboard. After the recording process they should compare the original song with the recorded one. Finally the user should be able to see the result of the comparison in a new sheet notation and a percentage of success.

MM application features

- It supports any ".mid" or ".midi" file. That means that the user could read any ".mid" or ".midi" file of type0 and type1 and also record ".mid" files of type0.
- It loads or saves a sheet notation as a ".cmn" file, which easily becomes a ".mid" file.
- A ".mid" or ".midi" file could be transformed into a ".cmn" file.
- A sheet notation could be shown in the x server, or become a postscript file, which could be easily transmitted to a printer.

Running the application

In order for someone to load all the packages that the application depends on, they should execute the command load-system with argument music-match in the Lisp interpreter.

If someone wishes to run the application, then, they can simply type the command (music-match) in the interpreter.

The table below presents the commands that are to be used for the MM application.

Command	Arguments	Description
Play	*filename*	Plays a midi file using timidity.
Change Sound	*1 or 2 & value*	Changes between soundcards. 1 is for the default soundcard and 2 is for an external soundcard. The *value* indicates the sound volume.
Quit	–	Exit the application.
Clear	–	Clears the screen window.
Clear History	–	Clears the history window.
Sheet Notation	*filename*	Presents a sheet notation of the specified midi file, by transforming the midi file in a cmn file.
Close Sheet	–	Closes an open sheet notation.
Discard Track	–	Flushes the global variable *myseq* and sets the application ready for a new recording.
Transporto	*num*	Change the global variable *transpose-value* by such semitones as indicated by *num*. *num* should be an integer between $[-12, 12]$
TempoChange	*num*	Change the global variable *mytempo*.*num* indicates the new tempo value.
Reset tt & dt	–	Sets the global variables *time-tolerance* and *duration-tolerance* to default values, which are 0.1 and 0.2 respectivly.
OpenNotation	*filename*	Opens the specified cmn file.

SaveNotation	*filename*	Saves a sheet notation as a cmn file, named *filename*.
Music Match	*num1, num2, filename1, filename2, symbol*	Two midi files are compared using a brute-force algorithm and returns a percentage of how similar the two files are. *filename1* stands for the original midi file while *filename2* stands for the midi file played by the user. *num1* indicates the time tolerance value and *num2* indicates the duration tolerance value. *symbol* is either *yes or no* value that enables a time process report.
LcsMatch	*num1, filename1, filename2, symbol*	Two midi files are compared using the LCS algorithm and returns a percentage of how similar the two files are. *filename1* stands for the original midi file while *filename2* stands for the midi file played by the user. *num1* indicates the duration tolerance value. *symbol* is either *yes or no* value that enables a time process report.
Show Sheet After Compare	*filename1, filename2*	A sheet notation is created with the results of the comparison between the two midi files. The brute-force algorithm is used for the comparison.
Show Sheet After Lcs Compare	*filename1, filename2*	A sheet notation is created with the results of the comparison between the two midi files. The LCS algorithm is used for the comparison.

Next there are some examples that demonstrate how the MM application works.

Lets assume we have the following midi file in *figure 2.1*

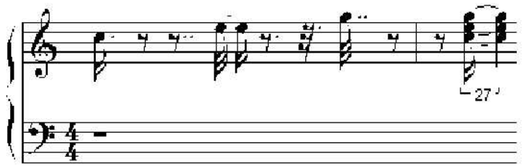


Figure 4.1: A sheet notation from the original midi file

And we produce the midi file in *figure 2.2* below

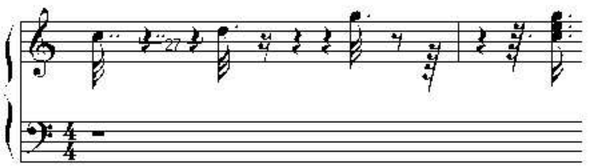


Figure 4.2: A sheet notation from a student's midi file

In *figure 2.3*, using the command *Show Sheet After Compare*, results...

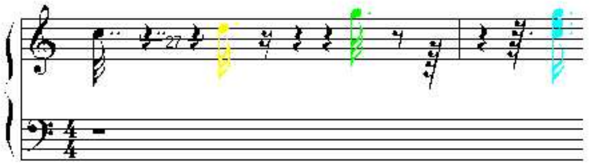
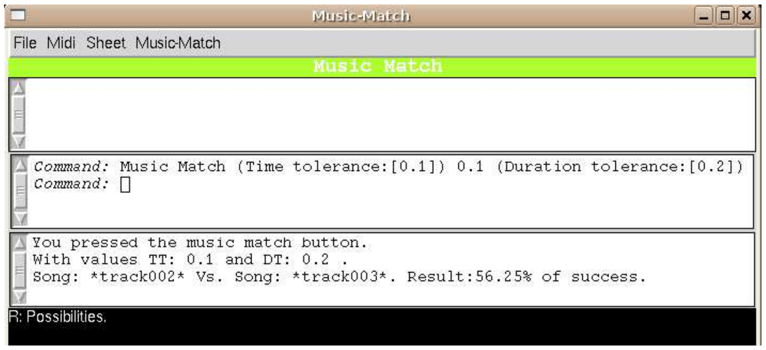


Figure 4.3: After the comparison with tt= 0.1 and dt= 0.2, we see that the 1*st* note is equal, the 2*nd* has only the duration parameter equal, the 3*rd* is equal except for the time parameter and the 4*th* is equal except for the time and duration parameter.

In *figure 2.4* we see a percentage of how similar the two songs are, using the command *Music Match*



```
Command: Music Match (Time tolerance:[0.1]) 0.1 (Duration tolerance:[0.2])
Command:
```

```
You pressed the music match button.
With values TT: 0.1 and DT: 0.2 .
Song: *track002* Vs. Song: *track003*. Result:56.25% of success.
```

Figure 4.4: The history window shows the percent of how similar are.

Now, the same example will be used but with the Longest Common Subsequence algorithm. Using the command *Show Sheet After Lcs Compare* the results are shown in *figure 2.5*

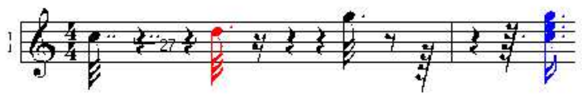
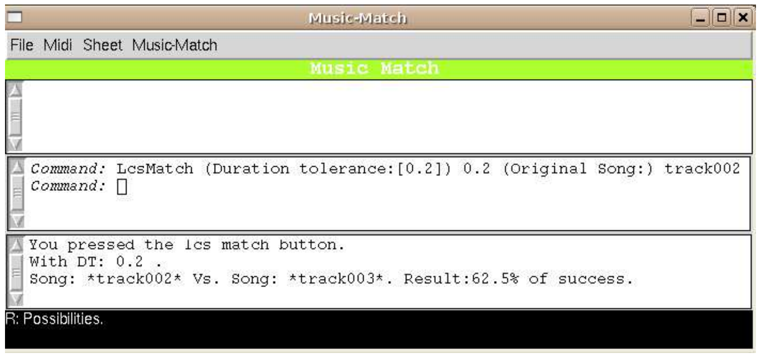


Figure 4.5: After the comparison with dt= 0.2, we see that the 1*st* note is equal, the 2*nd* is not equal, the 3*rd* is equal and the 4*th* is equal except for the duration parameter.

And if we want to see a percent of how similar the two songs are we utilize the command *LcsMatch* in *figure 2.6*

If we change the dt (duration tolerance) value in the same example, then, we will get different results as shown below in *figures 2.7-2.10*.



```
Command: LcsMatch (Duration tolerance:[0.2]) 0.2 (Original Song:) track002
Command:
```

```
You pressed the lcs match button.
With DT: 0.2 .
Song: *track002* Vs. Song: *track003*. Result:62.5% of success.
```

Figure 4.6: The history window shows the percentage of how similar the two songs are. Also we have 1 deletion and 1 insertion.

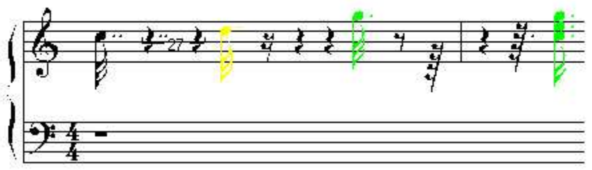
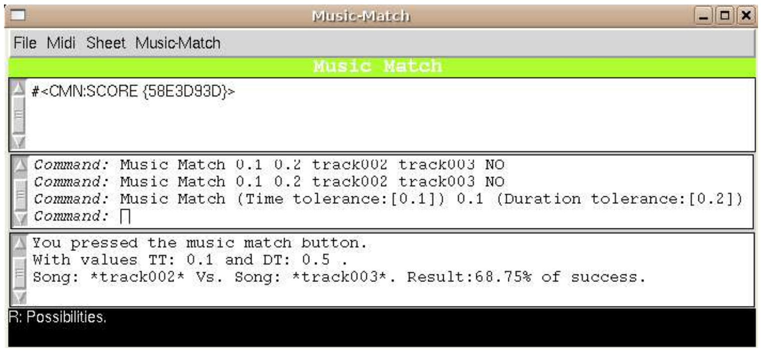


Figure 4.7: After the comparison with tt= 0.1 and dt= 0.5, we can perceive that the 1*st* note is equal, the 2*nd* has only the duration parameter equal, the 3*rd* is equal except for the time parameter and the 4*th* is equal now, except for the time parameter.



```
#<CMN:SCORE {58E3D93D}>
```

```
Command: Music Match 0.1 0.2 track002 track003 NO
Command: Music Match 0.1 0.2 track002 track003 NO
Command: Music Match (Time tolerance:[0.1]) 0.1 (Duration tolerance:[0.2])
Command:
```

```
You pressed the music match button.
With values TT: 0.1 and DT: 0.5 .
Song: *track002* Vs. Song: *track003*. Result:68.75% of success.
```

Figure 4.8: The history window shows the percentage of how similar the two songs are with dt= 0.5

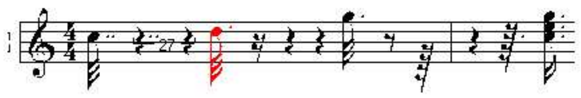
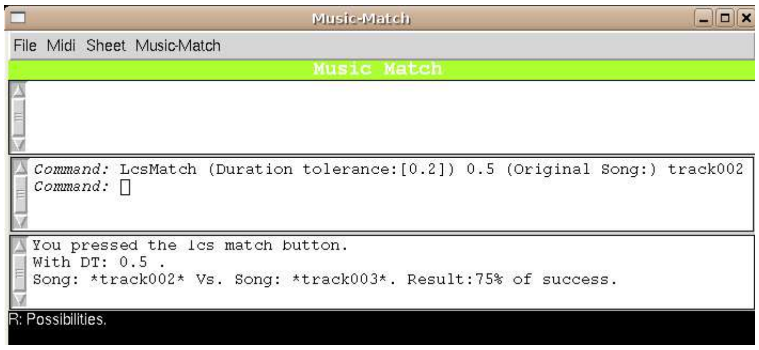


Figure 4.9: After the comparison with dt= 0.5, we see that the 1*st* note is equal, the 2*nd* is not equal, the 3*rd* is equal and the 4*th* is equal now.



```
Command: LcsMatch (Duration tolerance:[0.2]) 0.5 (Original Song:) track002
Command:
```

```
You pressed the lcs match button.
With DT: 0.5 .
Song: *track002* Vs. Song: *track003*. Result:75% of success.
```

R: Possibilities.

Figure 4.10: The history window shows the percentage of how similar the two songs are. Also we have 1 deletion and 1 insertion.

Another example is our making a transporto to a midi file. Lets assume we have the following midi file in *figure 2.11*

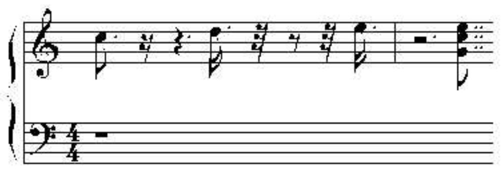


Figure 4.11: Before transporto...

Using the command *transporto* −3, we decrease each note by 3 semitones and the result is shown in *figure 2.12*

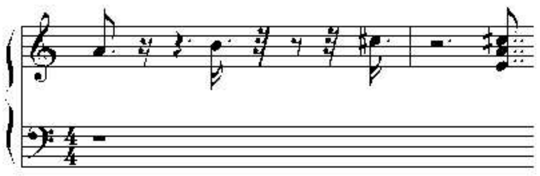


Figure 4.12: After transporto −3 semitones...

Another example is our changing the tempo to a midi file. So, lets assume we have the following midi file in *figure 2.13*

If we reduce the tempo, we will get the result below, in *figure 2.14*

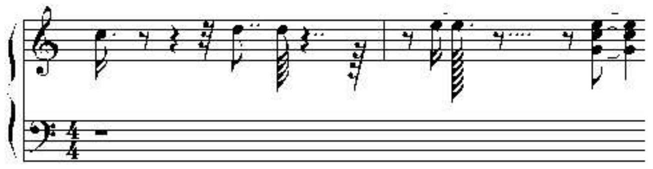


Figure 4.13: The default tempo is 60.

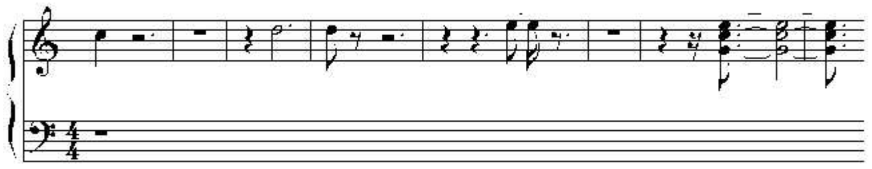


Figure 4.14: The tempo now is 16.

4.2 Technical Description

In this section, our aim is to formulate the music match problem by providing two different algorithms that will solve this problem. The first one is a simple match algorithm the second one is a brute-force algorithm and the last one is a dynamic-programming algorithm.

4.2.1 Problem Formulation

In Music Match, we wish to compose two different pieces of music. A piece of music consists of instances of notes or chords. Representing each of an instance, a note can be expressed as a list of (n, t, d), where $n = n_1$ for the note name, $t = t_1$ for the time note and $d = d_1$ for the note duration, as for the chord $(\overrightarrow{n}, \overrightarrow{t}, \overrightarrow{d})$, where $\overrightarrow{n} = (n_1, n_2, n_3)$, $\overrightarrow{t} = (t_1, t_2, t_3)$ and $\overrightarrow{d} = (d_1, d_2, d_3)$.

For example, a music piece may be:

$M_1 = \{(n_1, t_1, d_1), (n_2, t_2, d_2), (n_3, t_3, d_3), ..., (n_n, t_n, d_n)\}$

while another music piece may be:

$M_2 = \{(n'_1, t'_1, d'_1), (n'_2, t'_2, d'_2), (n'_3, t'_3, d'_3), ..., (n'_n, t'_n, d'_n)\}$

One goal of comparing two music pieces is to determine how "similar" the two music pieces are, providing some measure of how closely related the two music pieces are.

We formalize this last notion of similarity as the longest-common-subsequence problem. In the **longest-common-subsequence problem (LCS)**, we are given two sequences $M_1 = \{m_1, m_2, ..., m_m\}$ and $M_2 = \{m'_1, m'_2, ..., m'_m\}$ and we wish to find a maximum-length common subsequence of M_1 and M_2. Longest Common Subsequence of two sequences $M_1, M_2, (LCS(M_1, M_2))$ is defined as the largest set of symbols that fullfill the following properties:

i. LCS(M_1, M_2) is a subset of both M_1, M_2

ii. The symbols in LCS(M_1, M_2) follow the same ordering as in both M1,M2

The LCS problem can be solved efficiently by our using dynaming programming.

4.2.2 Simple Match Approach

In this approach we do not try to find the LCS of M_1, M_2, but we assume that there is a 1 to 1 correspondence of the symbol of the shortest sequence to the first symbols of the longest sequence. So, the alignment for $M_1 \rightarrow M_2$ contributes to minimize the $D(M_1, M_2)$. Lets assume that an instance of a note from M_1 can be expressed as $N : (n, t, d)$ and from M_2 as $N' : (n', t', d')$ and both lengths of M_1 and M_2 are equal.

The $D(M_1, M_2)$ would be equal to the sum of each difference between two instances.

$$D(M_1, M_2) = \sum_{i=1}^{n} d(N_i, N_i')$$

The $d(N_i, N_i')$ would be equal to the sum of the differences between the note name, time note and note duration.

$$d(N_i, N_i') = \lambda_1 * d(n_i, n_i') + \lambda_2 * d(t_i, t_i') + \lambda_3 * d(d_i, d_i')$$

where each difference could be expressed as the equations below:

- $d(n_i, n_i') = c * (1 - \delta(n_i - n_i'))$

where c is a constant value and the function δ where

$$\delta(n_i - n_i') = \begin{cases} 1 & , n_i = n_i' \\ 0 & , n_i \neq n_i' \end{cases}$$

- $d(t_i, t_i') = |t_i - t_i'|$

where

$$|t_i - t_i'| = \begin{cases} 0 & , t_i - T < t_i' < t_i + T \\ MAXTHRES & \end{cases}$$

and T is the time tolerance.

- $d(d_i, d_i') = |d_i - d_i'|$

where

$$|d_i - d'_i| = \begin{cases} 0 & , d_i - D < d'_i < d_i + D \\ MAXTHRES & \end{cases}$$

and D is the duration tolerance.

4.2.3 Brute-Force Approach

A brute-force approach is established in order to enumerate all subsequences of M_1 and to check each subsequence so that we could see if it is also a subsequence of M_2. Each subsequence of M_1 corresponds to a subset of the indices $\{1, 2, ..., m\}$ of M_1. There are 2^m subsequences of M_1, so this approach requires exponential time, making it impractical for long sequences.

4.2.4 Dynamic Programming Solution

That was a brute-force approach to solve the LCS problem. However the LCS problem has an optimal-substructure property, as the following theorem shows. As we shall see, the natural classes of subproblems correspond to pairs of "prefixes" of the two input sequences. To be more precise, given a sequence $M_1 = \{m_1, m_2, ..., m_m\}$, we define the ith **prefix** of M_1, for $i = 0, 1, 2, ...m$, as $M_1^{(i)} = \{m_1, m_2, ...m_i\}$. For example, if $M_1 = \{A, B, C, B, D, A, B\}$, then $M_4 = \{A, B, C, B\}$ and M_0 is the empty sequence.

Theorem: Optimal substructure of an LCS

Let $M_1 = \{m_1, m_2, ..., m_m\}$ and $M_2 = \{m'_1, m'_2, ..., m'_n\}$ be sequences, and let $M_3 = \{m''_1, m''_2, ..., m''_k\}$ be any LCS of M_1 and M_2.

i. $m_m = m'_n$,then $m''_k = m_m = m'_n$ and $M_3^{(k-1)}$ is an LCS of $M_1^{(m-1)}$ and $M_2^{(n-1)}$.

ii. $m_m \neq m'_n$,then $m''_k \neq m_m$ implies that M_3 is an LCS of $M_1^{(m-1)}$ and M_2.

iii. $m_m \neq m'_n$,then $m''_k \neq m'_n$ implies that M_3 is an LCS of M_1 and $M_2^{(n-1)}$.

Proof:

(1) If $m''_k \neq m_m$, then we could append $m_m = m'_n$ to M_3 to obtain a common subsequence of M_1 and M_2 of length $k+1$, contradicting the supposition that M_3 is a longest common subsequence of M_1 and M_2. Thus, we must have $m''_k = m'_n = m_m$. Now, the prefix $M_3^{(k-1)}$ is a length$-(k-1)$ common subsequence of $M_1^{(m-1)}$ and $M_2^{(n-1)}$. We wish to show that it is an LCS. Suppose for the purpose of contradiction that there is a common subsequence M_4 of $M_1^{(m-1)}$ and $M_2^{(n-1)}$ with length greater than $k-1$. Then, appending $m_m = m'_n$ to M_4 it produces a common subsequence of M_1 and M_2 whose length is greater than k, which is a contradiction.

(2) If $m''_k \neq m_m$, then M_3 is a common subsequence $M_1^{(m-1)}$ and M_2. If were there a common subsequence M_4 of $M_1^{(m-1)}$ and M_2 with length greater than k, then M_4 would also be a common subsequence $M_1^{(m)}$ and M_2, contradicting the assumption that M_3 is an LCS of M_1 and M_2.

(3) The proof is symetric to (2).

The characterization of the theorem above shows that an LCS of two sequences incorporates an LCS of prefixes of the two sequences. Thus, the LCS problem has an optimal-substructure property. A recursive solution has also the overlapping-subproblems property, as we shall see later on.

A recursive solution

The theorem below implies that there are either one or two subproblems to be examined when seeking an LCS of $M_1 = \{m_1, m_2, ...m_m\}$ and $M_2 = \{m'_1, m'_2, ..., m'_n\}$. If $m_m = m'_n$, we must find an LCS of $M_1^{(m-1)}$ and $M_2^{(n-1)}$. Appending $m_m = m'_n$ to this LCS, then an LCS of M_1 and M_2 is yielded. If $m_m \neq m'_n$, then we must solve two subproblems: an LCS of $M_1^{(m-1)}$ and M_2 and an LCS of M_1 and $M_2^{(n-1)}$ must be found. Whichever of these two LCS's is longer, then this is an LCS of M_1 and M_2. Because these cases exhaust all possibilities, we can be convinced that one of the optimal subproblem solutions must be applied within an LCS of M_1 and M_2.

We can readily observe the overlapping-subproblems property in the LCS problem. In order for us to find an LCS of M_1 and M_2, we may need to find the LCS's of M_1 and $M_2^{(n-1)}$ and $M_1^{(m-1)}$ and M_2. But each of these subproblems have the subsubproblem of finding the LCS of $M_1^{(m-1)}$ and $M_2^{(n-1)}$. However, there are many other subproblems which share subsubproblems.

As in the matrix-chain multiplication problem, our recursive solution to the LCS problem involves establishing a reccurence for the value of an optimal solution. Let us define c[i, j] to be the length of an LCS of the sequences $M_1^{(i)}$ and $M_2^{(j)}$. If either $i = 0$ or $j = 0$, one of the sequences has length 0, so the LCS has length 0. The optimal substructure of the LCS problem gives the recursive formula.

$$c(i,j) = \begin{cases} 0 & , i = 0 \text{ or } j = 0 \\ c(i-1, j-1) + 1 & , i, j > 0 \text{ and } M_1^{(i)} = M_2^{(j)} \\ max(c(i, j-1), c(i-1, j)) & , \text{otherwise} \end{cases}$$

Observe that in this recursive formulation, a condition in the problem poses restrictions upon which subproblems should be considered. When $m_i = m'_j$, we ought to take into consideration the subproblem of finding the LCS of $M_1^{(i-1)}$ and $M_2^{(j-1)}$. Otherwise, we should consider the two subproblems of finding the LCS of $M_1^{(i)}$ and $M_2^{(j-1)}$ and of $M_1^{(i-1)}$ and $M_2^{(j)}$ in stead.

Based on the above equation, we could easily write an exponential-time recursive algorithm to compute the length of an LCS of two sequences. However since there are only $O(mn)$ distinct subproblems, we can use dynamic programming to compute the solutions bottom up.

Procedure LCS-LENGTH takes two sequences $M_1 = \{m_1, m_2, ..., m_m\}$ and $M_2 = \{m'_1, m'_2, ..., m'_n\}$ as inputs. It stores the c[i, j] values in a table c[$0 \rightarrow m, 0 \rightarrow n$] whose entries are computed in a row-major order. (That is, the first row of c is filled in from left to right, then the second row, is too and so on.) It also maintains the table b[$1 \rightarrow m, 1 \rightarrow n$] to simplify construction of an optimal solution. Intuitively, b[i, j] points to the table entry corresponding to the optimal subproblem solution chosen when computing c[i, j]. The procedure returns the b and c tables; c[m, n] contains the length of an LCS of M_1 and M_2.

```
LCS-LENGTH($M_1, M_2$)
1 $m \leftarrow$ length[$M_1$]
2 $n \leftarrow$ length[$M_2$]
3 for $i \leftarrow 0$ to $m$
4     do c[$i, 0$]$\leftarrow 0$
5 for $j \leftarrow 0$ to $n$
6     do c[$0, j$]$\leftarrow 0$
7 for $i \leftarrow 1$ to $m$
```

```
8   do for $j \leftarrow 1$ to $n$
9      do if $M_1^{(i)} = M_2^{(j)}$
10          then c[$i, j$]$\leftarrow$ c[$i-1, j-1$]+1
11                b[$i, j$]$\leftarrow$ "$\nwarrow$"
12        else if c[$i-1, j$]$\geq$ c[$i, j-1$]
13          then c[$i, j$]$\leftarrow$ c[$i-1, j$]
14                b[$i, j$]$\leftarrow$ "$\uparrow$"
15        else c[$i, j$]$\leftarrow$ c[$i, j-1$]
16                b[$i, j$]$\leftarrow$ "$\leftarrow$"
17 return b and c
```

- "$\nwarrow$" : it implies that both elements of the two subsequences are equal, also moves the pointer one position forward to both subsequences.
- " $\uparrow$ ": corresponds to a deletion, also moves the pointer of the first subsequence one position forward.
- "$\leftarrow$" : corresponds to an insertion, also moves the pointer of the second subsequence on position forward.

The figure below shows the tables produced by LCS-LENGTH on the sequences $M_1 = \{A, B, C, B, D, A, B\}$ and $M_2 = \{B, D, C, A, B, A\}$. The running time of the procedure is $O(mn)$, since each table entry takes $O(1)$ time to compute.

	j	0	1	2	3	4	5	6
i		$M_2^{(j)}$	B	D	C	A	B	A
0	$M_1^{(i)}$	0	0	0	0	0	0	0
1	A	0	$\uparrow$ 0	$\uparrow$ 0	$\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\nwarrow$ 1
2	B	0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
3	C	0	$\uparrow$ 1	$\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2	$\uparrow$ 2	$\uparrow$ 2
4	B	0	$\nwarrow$ 1	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\leftarrow$ 3
5	D	0	$\uparrow$ 1	$\nwarrow$ 2	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\uparrow$ 3
6	A	0	$\uparrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\nwarrow$ 3	$\uparrow$ 3	$\nwarrow$ 4
7	B	0	$\nwarrow$ 1	$\uparrow$ 2	$\uparrow$ 2	$\uparrow$ 3	$\nwarrow$ 4	$\uparrow$ 4

The c and b tables are computed by LCS-LENGTH on the sequences $M_1 = \{A, B, C, B, D, A, B\}$ and $M_2 = \{B, D, C, A, B, A\}$. The square in row i and column j contains the value of c[i, j] and the appropriate arrow for the value of b[i, j]. The entry 4 in c[$7, 6$] – the lower right-hand corner of the table – is the length of an LCS $\{B, C, B, A\}$ of M_1 and M_2. For $i, j > 0$, entry c$[i,j]$ depends only on whether $M_1^{(i)} = M_2^{(j)}$ and the values in entries c$[i-1,j]$,c$[i,j-1]$, and c$[i-1,j-1]$, which are computed before c$[i,j]$. In order to reconstruct the elements of an LCS, we should follow the b$[i,j]$ arrows from the lower right-hand corner. Each "$\nwarrow$" on the path corresponds to an entry for which $M_1^{(i)} = M_2^{(j)}$ is a member of an LCS.

The b table returned by LCS-LENGTH can be used to promptly construct an LCS of $M_1 = \{m_1, m_2, ..., m_m\}$ and $M_2 = \{m'_1, m'_2, ..., m'_n\}$. We simply begin at b$[m,n]$ and trace through the table following the arrows. Whenever we encounter a $\nwarrow$ in entry b$[i,j]$, it implies that $M_1^{(i)} = M_2^{(j)}$ is an element of the LCS. The elements of the LCS are encountered in reverse order according to this method. The following recursive procedure prints out an LCS of M_1 and M_2 in the proper, forward order. The initial invocation is PRINT-LCS(b, M_1, length$[M_1]$, length$[M_2]$).

```
PRINT-LCS($b, M_1, i, j$)
1 if $i = 0$ or $j = 0$
2    then return
3 if b$[i,j]$= "$\nwarrow$"
4    then PRINT-LCS($b, M_1, i-1, j-1$)
5         print $M_1^{(i)}$
6 else if b$[i,j]$= "$\uparrow$"
7         then PRINT-LCS($b, M_1, i-1, j$)
8 else PRINT-LCS($b, M_1, i, j-1$)
```

The functions LCS-LENGTH and PRINT-LCS will be described in chapter 4.4.2 *page 42*.

4.2.5 Complexity

The MM algorithm is $O(n^2)$ which is a non linear complexity. See more details on *Appendix*.

Chapter 5

Tools

5.1 MIDI

5.1.1 What is MIDI

Musical Instrument Digital Interface, or MIDI, is an industry-standard electronic communications protocol that defines each musical note or event in an electronic musical instrument. Furthermore, it shows a device such as a synthesizer, precisely and concisely, allowing electronic musical instruments, computers and some other show equipment to exchange data in real time. MIDI does not transmit audio - it simply transmits real time digital data providing information such as the type and intensity of the musical notes and technical cues played during a performance.

The MIDI standard consists of a communications messaging protocol designed for use with musical instruments, as well as a physical interface standard. It consists physically of a one-way (simplex) digital current loop serial communications electrical connection signaling at 31,250 bits per second. One start bit (must be 0), eight data bits, no parity bit and one stop bit (must be 1) are used.

Each one-way connection (called a port) can transmit or receive standard musical messages, such as note-on, note-off, controllers (which include volume, pedal, modulation signals, etc.), pitch bend, program change, aftertouch, channel pressure, and system-related messages. Those signals are sent along with one of the 16 channel identifiers. The channels are used to separate ”voices” or ”instruments”, somewhat like tracks in a multi-track mixer.

The ability to multiplex 16 ”channels” onto a single wire makes it possible to control several instruments at once using a single MIDI connection. When a MIDI instrument is capable of producing several independent sounds simultaneously (a multitimbral instrument), MIDI channels are used so that those sections could be addressed independently. (This should not be confused with ”polyphonic”; the ability to play several notes simultaneously in the same ”voice”.)

MIDI messages (along with timing information) can be collected and stored in a computer file system, in what is commonly called a MIDI file, or more formally, a Standard MIDI File (SMF). The SMF specification was developed and is maintained by the MIDI Manufacturers Association (MMA). MIDI files are typically created by the individual's using desktop/laptop computer-based sequencing software (or sometimes a hardware-based MIDI instrument or workstation) that organizes MIDI messages into one or more parallel ”tracks” for independent recording and editing. In most but not all sequencers, each track is assigned a specific MIDI channel and/or a specific General MIDI instrument patch. Although most current MIDI sequencer softwares use proprietary ”session file” formats rather than SMF. Yet, almost all sequencers provide export or ”Save As...” support for the SMF format.

An SMF consists of one header chunk and one or more track chunks. There are three SMF formats; the format is encoded in the file header. Format 0 contains a single track and represents a single song performance. Format 1 may embody any number of tracks, thus, enabling preservation of the sequencer track structure, and also representing a single song performance. Format 2 may have any number of tracks, each representing a separate song performance. Nonetheless, Format 2 is not commonly supported by sequencers.

5.1.2 Why Midi

Almost all music recordings today utilize MIDI as a key enabling technology for recording music. In addition, MIDI is also used as a means to control hardware including recording devices as well as live performances equipment such as stage lights and effects pedals.

A number of music file formats have been based on the MIDI bytestream. Those formats are very compact; a file of only 10 kilobytes can often produce a full minute of music. This is advantageous for applications such as musical ringtones in mobile phones and some video games.

MIDI messages are extremely compact, due to the low bandwidth of the connection, and the need for real-time accuracy. Most messages consist of a status byte (channel number in the low 4 bits, and an opcode in the high 4 bits), followed by one or two data bytes. However, the serial nature of MIDI messages entail long strings of MIDI messages which consume an appreciable period of time to be sent. At times even audible delays are triggered, especially when someone is dealing with dense musical information or when many channels are particularly active.

The data stream, ”Running status”, could be further optimized by a convention that allows the status byte to be omitted if it is to be the same as that of the previous message and therefore, to mitigate bandwidth issues somehow.

Large collections of SMFs can be found on the web, most commonly with the extension .mid. Those files are most frequently authored with the assumption that they will be played on General MIDI players. But not always will the files do so, as the result will be an occasional unintended bad-sounding playback.

A proposal for High-Definition MIDI (HD-MIDI) extension is now being discussed by members of the MMA. This major update to MIDI would provide greater resolution in data values, increase the number of MIDI Channels, and support the creation of entirely new kinds of MIDI messages. [2] [3] This work involves representatives from all sizes and types of companies, ranging from the smallest specialty show control operations to the largest musical equipment manufacturers.

5.2 Lisp

5.2.1 What is Lisp

Lisp is a family of computer programming languages with a long history and a distinctive, fully-parenthesized syntax. Originally specified in 1958, Lisp is the second oldest, high-level programming language in widespread use today; only Fortran is older. Like Fortran, Lisp has changed a great deal since its early days, and a number of dialects have existed throughout its history. Today, the most widely-known, general-purpose Lisp dialects are Common Lisp and Scheme.

Lisp was originally created as a practical mathematical notation for computer programs, based on Alonzo Church's lambda calculus. It quickly became the favored programming language for artificial intelligence research. As one of the earliest programming languages, Lisp pioneered many ideas in computer science, including tree data structures, automatic storage management, dynamic typing, object-oriented programming, and the self-hosting compiler.

The name Lisp derives from "List Processing". Linked lists are one of Lisp languages' major data structures, and Lisp source code is itself made up of lists. As a result, Lisp programs can manipulate source code as a data structure, giving rise to the macro systems that allow programmers to create new syntax or even new "little languages" embedded in Lisp.

The interchangeability of code and data also offer Lisp its instantly recognizable syntax. The whole program code is written as s-expressions, or parenthesized lists. A function call or syntactic form is written as a list with the function or operator's name first, and the following arguments: (f x y z).

5.2.2 Why Lisp

The MM application required a platform that could handle MIDI connection, record, edit, compose, synthesize sound and be able to describe a musical notation. All these features provided by CM, CMN and PORTMIDI, are packages created on a Lisp language and are opensources. That is the main reason why Lisp has been chosen to serve the needs of the MM application.

Lisp was the first homoiconic programming language: the primary representation of program code is the same type of list structure that is also used for the main data structures. As a result, Lisp functions can be manipulated, altered or even created within a Lisp program without extensive parsing or manipulation of binary machine code. This is generally considered one of the primary advantages of the language with regard to its expressiveness, and makes the language amenable to metacircular evaluation.

Lisp is an expression-oriented language. Unlike most other languages, no distinction is made between ”expressions” and ”statements”; all code and data are written as expressions. When an expression is evaluated, it produces a value (or list of values), which then can be embedded into other expressions.

The reliance on expressions enables the language to acquire great flexibility. Because Lisp functions are themselves written as lists, they can be processed exactly like data: thus, allowing easy writing of programs which manipulate other programs (metaprogramming). Many Lisp dialects exploit this feature using macro systems, which enables extension of the language almost without limits.

5.2.3 CMUCL

CMUCL is a free, high performed implementation of the Common Lisp programming language which runs on most major Unix platforms. It mainly conforms to the ANSI Common Lisp standard. CMUCL provides the following: a sophisticated native code compiler, a powerful foreign function interface, an implementation of CLOS, the Common Lisp Object System; which includes multimethods, a metaobject protocol, a source-level debugger, a code profiler and an Emacs-like editor implemented in Common Lisp. CMUCL is maintained by a team of volunteers collaborating over the Internet, and it mostly appears in the public domain.

5.2.4 CLIM

The Common Lisp Interface Manager (CLIM) is a powerful Lisp-based system that provides a layered set of facilities for building user interfaces. These facilities incorporate a portable layer called Silica that includes basic windowing, input, output services, and mechanisms for constructing window types. Moreover, user interface components, stream-oriented input and output facilities extended with presentations and context sensitive input similar to the work pioneered in the Genera UI system are provided. Another useful facility presented by the system is a gadget-oriented toolkit similar to those found in the X world extended with support for look and feel adaptiveness.

The Common Lisp Interface Manager (CLIM) is specifying an interface to a broad range of services necessary or useful for developing graphical user interfaces. These services include low level facilities like geometry, graphics, event-oriented input, and windowing. Also, intermediate level facilities like support for Common Lisp stream operations, output recording, and advanced output formatting are displayed. Last but not least, those services provide high level facilities like context sensitive input, an adaptive toolkit, and an application building framework. CLIM will eventually support a large number of window environments including Genera[16], X[12], the Macintosh[1], Microsoft Windows[5], SunView[13], and NextStep[6]. CLIM supports the incorporation of toolkits written in other languages (e.g. C-based toolkits that implement Motif[7] or OpenLook[15]).

5.2.5 CM

Common Music (CM) is an object-oriented music composition environment. It produces sound by transforming a high-level representation of musical structure into a variety of control protocols for sound synthesis and display. Common Music defines an extensive library of compositional tools and an API through which the composer can easily modify and extend the system.

Common Music began in 1989 as a response to the proliferation of different audio hardware, software and computers that resulted from the introduction of low cost computers. As choices increased it became clear that composers would be benefitted from a portable, powerful and consistent interface in the face of the myriad sound rendering possibilities. Work on Common Music began in 1989 when the author was a guest composer at CCRMA, Stanford University. Much of the system as it exists today was implemented at the Institut fur Musik und Akustik at the Zentrum fur Kunst und Medientechnologie in Karlsruhe, Germany, where the author worked for five years. Common Music continues to evolve today at the University of Illinois at Urbana-Champaign, where the author is now an associate professor of music composition. In 1996 Common Music received First Prize in the computer-assisted composition category at the 1er Concours International de Logiciels Musicaux in Bourges, France.

CM is released under the LLGPL (Lisp Lesser General Public License).

5.2.6 CMN

Common Music Notation is an independent subsystem of Common Music. It provides a way of describing musical notation in terms of Lisp programmes using CLOS, so that not only can Lisp programmers typeset Music with it, but they can also use the power of Lisp to aid in composing. Common Music Notation is known to run in ACL, MCL, CLISP and CMUCL.

5.2.7 PORTMIDI

Common Music supports reading and writing MIDI messages to and from Portmidi, an open-source, cross-platform MIDI device library. CM supports Portmidi on OS X and Linux in OpenMCL, SBCL and CMUCL. The support consists of a portmidi-stream class so that different Portmidi input/output devices could be managed. This kind of support also incorporates a handful of auxiliary functions both for querying Portmidi about its current configuration and for accessing its millisecond timer.

These modules are to be loaded:

```
$ modprobe snd_seq_midi snd_seq_oss snd_seq_midi_emul
```

portmidi-stream manages resources and connections between CM and Portmidi input and output devices. This class is automatically chosen when one specifies a stream with a .pm extension. The convenience functions portmidi-open and portmidi-close are provided for the typical case of a single input/output pair. Opening a portmidi-stream initializes the library and starts the millisecond timer if they have not been initialized.

The fact that CM, CMN, PORTMIDI are opensource packages facilitates the study and change of the source code, in that the time complexity of the MM algorithm can be found.

Chapter 6

Implementation

6.1 Application Overview

Represents the basic data structures.

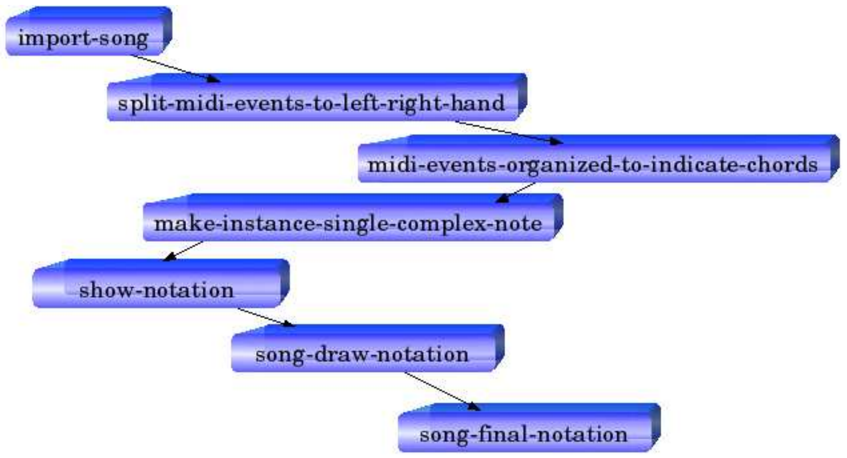

import-song
split-midi-events-to-left-right-hand
midi-events-organized-to-indicate-chords
make-instance-single-complex-note
show-notation
song-draw-notation
song-final-notation

Figure 6.1: The block diagram of the process that represents the sheet notation of a song.

In order to extract the sheet notation of a musical score, we should call the functions above. The arrows indicate the right order, those functions should be called.

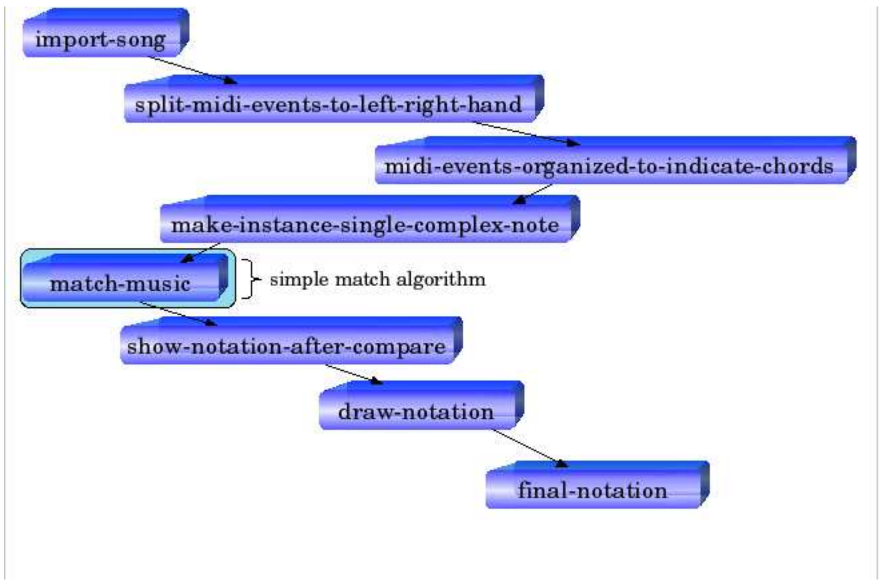

import-song
split-midi-events-to-left-right-hand
midi-events-organized-to-indicate-chords
make-instance-single-complex-note
match-music
simple match algorithm
show-notation-after-compare
draw-notation
final-notation

Figure 6.2: The block diagram of the process that compares two songs with a simple match algorithm and represents the result in a sheet notation.

In order to extract the sheet notation of the results of the comparison between two musical scores, we should call the functions above. The arrows indicate the right order, those functions should be called. A simple match algorithm takes place for the comparison.

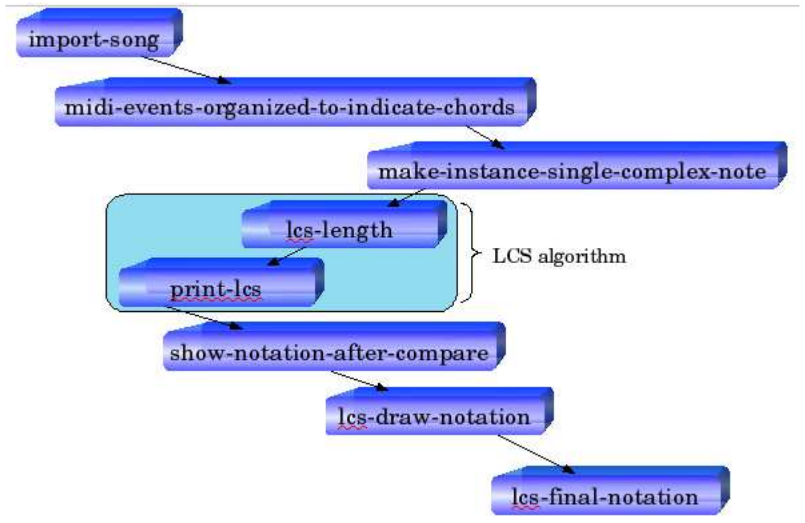

import-song
midi-events-organized-to-indicate-chords
make-instance-single-complex-note
lcs-length
print-lcs
LCS algorithm
show-notation-after-compare
lcs-draw-notation
lcs-final-notation

Figure 6.3: The block diagram of the process that compares two songs with the LCS algorithm and represents the result in a sheet notation.

In order to extract the sheet notation of the results of the comparison between two musical scores, we should call the functions above. The arrows indicate the right order, those functions should be called. The LCS algorithm takes place for the comparison.

6.2 Classes

The basic class is the “music-element” which is an abstract class.

- (defclass music-element ()())

In order to handle better the single notes and the chords of a song, two subclasses are being used the “sinlge-note” and the “complex-note” with 3 slots each of them.

- (defclass single-note (music-element))

The first slot is the “note-name” which keeps the name of the single note. The second slot is the “time” which keeps the time when the note has been pressed down. The third slot is the “duration” which keeps the duration of a note being pressed down.

- (defclass complex-note (music-element))

The first slot is the “note-name” which keeps the name of the chords. The second slot is the “time” which keeps the time when the chords have been pressed down. The third slot is the “duration” which keeps the duration of a chord being pressed down.

- (defclass container nil)

The class “container” from the package “cm” has one slot named “name” which stores the name of an object.

- (defclass seq (container))

The class “seq” from the package “cm” has two slots named “time” which keep the time of an object and “subobjects” which keep the subobjects of an object. The second slot is mostly being used, because it keeps the midi events of a song.

6.3 Variables

- (defparameter *myseq* nil)

The global variable ***myseq*** is being used to keep the “seq” object that the user will play from the midi keyboard, saving some time for the comparison.

Therefore, the user will not have to revoke the file from the hard disk since it is on memory already.

- (defparameter *time-chords* .0234375)

The global variable ***time-chords*** is being utilized to define which midi events will compose a chord. According to the package cm, the value of this variable corresponds to a 128th note. The lower value which is 0.015625 corresponds to a 256th note. But it is quite difficult to encounter on a sheet notation, notes with r values of 128th. So, in order for someone to define if two or more midi events will compose a chord, they must have equal time slot values or at least a time slot value lower than the *time-chords* variable, each one. Otherwise the midi events will be considered as sinlge notes of a 128th note.

- (defparameter *my-tempo* 60)

The global variable ***my-tempo*** contributes to the definition of the global tempo. It defaults to 60, because the package cm and cmn use that value as default to write midi messages and to demonstrate midi messages on a sheet notation respectively.

- (defparameter *time-tolerance* .1)

The global variable ***time-tolerance*** aims to define the tolerance that will be held while comparing two time slot values, in case they are not equal. It is set to 0.1 which means something less than a 256th note value. Thus, the MM algorithm initiates based on very strict conditions as far as the time is concerned. The more this variable increases, the less the restrictions of an error are.

- (defparameter *duration-tolerance* .2)

The global variable ***duration-tolerance*** is being used to define the tolerance that will be held while comparing two duration slot values, in case they are not equal. It is set to 0.2 which means something less than a 128th note value. Thus, the MM algorithm initiates with very strict conditions regarding the duration. By our increasing this variable, the restrictions of an error lessen.

- (defparameter *transpose-value* 0)

The global variable ***transpose-value*** is being applied to make possible the *trasporto* on a song. The default value is set to 0, which means that the song will be demonstrated as it is. By our increasing or decreasing the value of this variable, each note of a song will increase or decrease by the value of a semitone, respectively. As a result, it makes a *transporto* to the song.

- (setf *cmn-output-type* :x)

The global variable ***cmn-output-type*** is a global variable of the cmn package. It is used to define the output type of a sheet notation that could be a postscript, x or quickdraw type. It is set to demonstrate the sheet notation using the x server.

- *midi-player*

The global variable ***midi-player*** is a global variable of the package cm. It is utilized to define the player according to which midi files will be played. The default value is *timidity*.

- (defparameter *seq-name* ”my-song”)

The global variable ***seq-name*** is being applied to determine a name for the song the user will record. The default value is *“my_song”*.

- (defparameter *pm* nil)

The global variable ***pm*** is useful to remember the ports for the connection between the computer and the midi keyboard. Default value nil.

- size

The global variable **size** is a global variable of the package cmn. It is used to define the size of the sheet notation.

- page-width

The global variable **page-width** is a global variable of the package cmn. It aims to estimate the width of each page of the sheet notation.

- automatic-page-numbers

The global variable **automatic-page-numbers** is a global variable of the package cmn. It is used to record a number for each page of the sheet notation automatically.

6.4 Functions

This section is divided into three subsections, describing the functions which compose the MM algorithm, the functions that make use of the MM algorithm so as to demonstrate its result into a sheet notation and finally the functions that build the user interface of the MM application.

6.4.1 MM (Music Matching) Algorithm Functions

- (defun compare-music (filename1 &optional (filename2 nil)))

The final function of the algorithm MM is the **compare-music**. It takes as an argument two files ".mid" or ".midi" and returns a list of lists. Each incorporated list contains two elements, a code indicating the result of the comparison of a note and the appropriate instance of this note. The **compare-music** consist of the functions **import-song**,**midi-events-organized-to-indicate-chords**,**make-instance-single-complex-note** and **match-music**. Each of these functions will be described below.

- (defun import-song (filename))

The function **import-song** takes as an argument a file ".mid" or ".midi" and returns a "seq" object, which holds a list of time sorted subobject, which are typically musical events. This invokes the function **load-song** and if the ".mid" or ".midi" file is type0 then it will return a "seq" object as #<seq track0>, but if the file is type1, it now will return a list of "seq" object as (#<seq track0> #<seq track1> #<seq track2>), so the function **transform-midi-to-seq** is spawned so that it could get the track that contains the melody.

- (defun load-song (filename &key (meta cm::true) (key-note ':note)))

The function **load-song** takes as an argument a file ".mid" or ".midi" and returns a "seq" object, only if the file is of type0 and a list of "seq" objects, or if the file is of type1. It also takes two keyword arguments, the :meta and the :key-note with default values *true* and *':note* respectively. That means when the function **import-events** is invoked from package cm, which returns a seq containing midi objects from one or more tracks of file, all meta messages will be excluded (such as information about the rhythm, meter, scala) and MIDI key number values will be interpreted as notes (i.e for the note mi in 4th octave is e4). Each midi object generates MIDI note on and off pairs from a more general representation of key number, duration and amplitude (i.e i#(midi :time 0 :keynum 60 :duration .5 :amplitude .1 :channel 3)).[]

- (defun transform-midi-to-seq (object))

The function **transform-midi-to-seq** takes as an argument a list of “seq” objects and returns the “seq” object that contains the melody. By calling the slot subobjects, from the class seq for each “seq” object, it is able to check which “seq” object has subobjects. As it has already been suggested, the function above has been called with the meta keyword argument set to *true*, so all meta messages will be excluded which means that only one “seq” object will have subobjects, the one which has the melody.

- (defun remove-non-midi-object (object))

The function **remove-non-midi-object** takes as an argument a “seq” object and returns the same “seq” object, but it only applies subobjects which are now midi events. The subobjects of a “seq” object could be midi events,midi-control-change events,midi-program-change events and many others. So, it checks all the events of the subobject, one by one to find which one is a midi event and add them on a list. Hence, by using the functions **new** and **events** from the package cm, it constructs a “seq” object with the same name as the previous one and the only difference lies in that the new subobjects consist of midi events only.

- (defun midi-events-organized-to-indicate-chords (object))

The function **midi-events-organized-to-indicate-chords** takes as an argument a “seq” object and returns a list. This function attempts to find out the chords in a song and return them in a list along with the single notes. Two or more midi events will be considered as a chord and they will be pushed in a list, only if they have the same time slot value or at least a time slot value lower than the global variable *time-chords*. Yet, single notes are still midi events and chords (complex notes) are lists of midi events. What’s more, it supports 3 cases, a song to commence with a chord, a song to end with a chord and finally a song to have chords in the middle. For example, a song with 6 midi events 3 of which compose a chord should look like the midi events below.

Before the function:

```
(#i(midi
time 2.7895834 keynum c5 duration 1.6375 amplitude 0.93700784 channel 0)
```

```
 #i(midi
time 2.7812502 keynum e5 duration 1.6437501 amplitude 0.8031496 channel 0)
 #i(midi
time 2.7812502 keynum g4 duration 1.6354167 amplitude 0.5748032 channel 0)
 #i(midi
time 1.7875001 keynum e5 duration 0.29375002 amplitude 0.72440946 channel 0)
 #i(midi
time 0.8416667 keynum d5 duration 0.33333334 amplitude 0.78740156 channel 0)
 #i(midi
time 0.0 keynum c5 duration 0.31458336 amplitude 0.56692916 channel 0))
```

After the function:

```
((#i(midi
time 2.7895834 keynum c5 duration 1.6375 amplitude 0.93700784 channel 0)
  #i(midi
time 2.7812502 keynum e5 duration 1.6437501 amplitude 0.8031496 channel 0)
  #i(midi
time 2.7812502 keynum g4 duration 1.6354167 amplitude 0.5748032 channel 0))
 #i(midi
time 1.7875001 keynum e5 duration 0.29375002 amplitude 0.72440946 channel 0)
 #i(midi
time 0.8416667 keynum d5 duration 0.33333334 amplitude 0.78740156 channel 0)
 #i(midi
time 0.0 keynum c5 duration 0.31458336 amplitude 0.56692916 channel 0))
```

- (defmacro first-subobject (obj start end))

The macro function **first-subobject** takes as an argument a “seq” object with two numbers indicating a position and returns the appropriate subobject. Using the function **subobjects** from the package cm and specifying a specific position, it is able to get the desirable midi event from the “seq” object.

- (defmacro substitute-times (obj start1 end1 start2 end2))

The macro function **substitute-times** takes as an argument a “seq” object with four numbers indicating a position and returns a number. That number is the difference between two time slot values of the midi events specified by the function **first-subobject**.

- (defun make-instance-single-complex-note (midi-events-list))

The function **make-instance-single-complex-note** takes as an argument a list and returns a list as well. The input of this function is the result of the function **midi-events-organized-to-indicate-chords**, which means a list of midi events. Some of the midi events stand alone and some other gathered on lists. Those midi events which stand alone are registered as instances of the class single-note and those which gathered on a list are registered as instances of the class complex-note. So, the output of this function is a list of single-note - complex-note instances. Using the function **transpose** from the package cm and before proceeding into any kind of instances, it could transpose a note, depending on the global variable *transpose-value*. Also, the function **note** from the package cm is being used to transform the key numbers of each midi event to notes (i.e the keynum of middle do is 60 and become c4), so that the package cmn will use them to reveal a sheet notation. Finally, when the user wants to write MIDI messages from the midi keyboard, it could make use of the portmidi package. But, the problem is that when the user tries to play the same chord more than ones, the first time could pass through the midi-stream as (C4 E4 G4) and the second as (C4 G4 E4). So, should those chords be compared, they will resault error. Thus, the function **sort** appears each time a complex-note instance is to be constructed to avoid such errors. An example applying to the same track of the 6 midi events which could adjust that function would be:

```
(#<COMPLEX-NOTE {581CCBDD}> #<SINGLE-NOTE {58217915}> #<SINGLE-NOTE {58217A5D}>
 #<SINGLE-NOTE {58217A8D}>)
```

- (defun match-music (oldsong newsong))

The function **match-music** takes as an argument two lists and returns a list. The input of this function is the result of the function above for each list and corresponds to a different ".mid" or ".midi" file. The output is a list of lists and each sublist holds a code and an instance of the single-note or complex-note class. The first file is the original song from the internet and the second file is the song performed by the user. Actually, this function is the last level of the comparison and the method **match-note**, which will be described below, is the key method, because along with the function **mapcar** it triggers a comparison among each instances. That means, the MM algorithm is a kind of brute force algorithm which compares the first instance of the first file with the first instance of the second file and the same occurs for the rest instances. Some examples are shown below:

```
((011 #<COMPLEX-NOTE {582F7C35}>) (010 #<SINGLE-NOTE {582F7C65}>)
 (011 #<SINGLE-NOTE {582F7C95}>) (T #<SINGLE-NOTE {582F7CC5}>))

((T #<COMPLEX-NOTE {58311F65}>) (T #<SINGLE-NOTE {58311F95}>)
 (T #<SINGLE-NOTE {58311FC5}>) (T #<SINGLE-NOTE {58311FF5}>))

((010 #<COMPLEX-NOTE {5832A75D}>) (010 #<SINGLE-NOTE {5832A78D}>)
 (011 #<SINGLE-NOTE {5832A7BD}>) (001 #<SINGLE-NOTE {5832A7ED}>))
```

Before the **match-note** methods are described below, it is essential that the meaning of the code be explained. Two instances are considered equal only if they have the same key number slot value, the same time slot value and the same duration slot value or at least if they satisfy the conditions on each **match-note** methods, depending on the global variables *time-tolerance* and *duration-tolerance*. Only then the code becomes *t* (true). If they are not equal, the matrix below will demonstrate the value of the code for each mistake. For example, if the time slot and the duration slot value are equal but the key number slot value is not, then the code value will be (1 0 0). That means that the note will be presented in the sheet notation as a "red" note.

keynum	Time	Duration	Chromatic Code
	t		(000) : black
0	0	1	(001) : blue
0	1	0	(010) : green
0	1	1	(011) : light blue
1	0	0	(100) : red
1	0	1	(101) : purple
1	1	0	(110) : yellow
1	1	1	(111) : white

- (defmethod match-note ((oldsong single-note) (newsong single-note) &key (note-only nil)))

This method **match-note** takes as an argument two instances and returns a list. The input includes two instances of the same class single-note and the result is a list with two elements. The first one is the code which indicates the kind of error that occured, depending on the matrix above and the second one is the instance of the second file.

- (defmethod match-note ((oldsong complex-note) (newsong complex-note) &key (note-only nil)))

This method **match-note** takes as an argument two instances and returns a list. The input is two instances of the same class complex-note and the result is a list with two elements. The first one is the code which indicates the kind of error made, depending on the matrix above and the second one is the instance of the second file. As it compares complex notes, the comparison between the time slot value of the chords is estimated by the average value of each time slot value of the midi events that compose a chord. The comparison between the duration slot value of the chords is estimated by the macro **duration-check** that will be described below.

- (defmacro duration-check(old new))

The macro **duration-check** takes as an argument two numbers and returns a number. The input of this macro is the two values of the duration slot value of each complex-note instance that is being compared. The output is a number indicating the number of midi events which are not equal to the duration slot value.

6.4.2 LCS (Longest-Common-Subsequence) Functions

- (defun zero-column(array))

The function **zero-column** takes as an argument an array and returns that array with the first column filled with zero's.

- (defun zero-row(array))

The function **zero-row** takes as an argument an array and returns that array with the first row filled with zero's.

- (defun lcs-length (seq1 seq2))

The function **lcs-length** takes as an argument two music sequences and according to the theory of the LCS algorithm, it returns the b table. The only difference is that when the $M_{1i} = M_{2j}$, there is a "when" condition that checks if the note duration of each element is equal or not.

- (defun print-lcs (b-array seq i j &key (del 0) (insert 0) (music-result '())))

The function **print-lcs** takes as an argument the b table, a music sequence and the lengths of the two music sequences. It returns a list of single-note and complex-note instances as well as the number of deletions and insertions that took place during the dynamic-proccess. Each time that an up-arrow is encountered, we increase the deletion counter and each time a left-arrow is encountered, we increase the insertion counter.

- (defun lcs-match (filename1 &optional (filename2 nil)))

The function **lcs-match** takes as an argument two ".mid" or ".midi" files and returns a list of single-note and complex-note instances using the function below.

6.4.3 Utility Functions

- (defun percent-of-success (list-obj))

The function **percent-of-success** takes as an argument a list and return a number. The input of the function is the result from the comparison of two songs. The output of the function is a number indicating the percentage of success of that comparison. Since there are three parameters taking part in the process (keynum, time, duration), there are eight situations that could occur. Each case depends on a coefficient factor analogous to that case. For example, if a pair of instances are equal, then the coefficient factor is 2 and if the keynum is the only mistake then the coefficient factor is 1.

- (defun lcs-percent-of-success (list-objects))

The function **lcs-percent-of-success** takes as an argument a list, which actually is the result of the lcs-match function and returns a number indicating the percentage of the success of the comparison between two music sequences. Using the LCS algorithm, only two parameters could take part in the procedure, the keynum and the duration.

Besides the function above, a series of functions that make a bridge between the cm and cmn package are going to follow, so as to demonstrate the result of a comparison on a sheet notation. Hence, each note is going to have a specific color, depending on the chromatic code, so the user would be able to check what kind of mistakes have been made.

- (defun split-midi-events-to-left-right-hand (object))

The function **split-midi-events-to-left-right-hand** takes as an argument a “seq” object and returns a list. That list consists of two “seq” objects. The first object contains all the midi events whose key number is greater than 60 (middle C) and it will be considered as the *right hand* on the sheet notation. The second object contains all the other midi events whose key number is lower than 60 (middle C) and it will be considered as the *left hand* on the sheet notation.

- (defun prepare-midi-for-notation (&optional (filename nil)))

The function **prepare-midi-for-notation** takes as an argument a “.mid” or “.midi” file and returns a list. This function is an assistant function which makes use of the function **import-song** and **split-midi-events-to-left-right-hand**.

- (defun compare-seperated-hands (hand1 hand2))

The function **compare-seperated-hands** takes as an argument two “seq” objects and returns a list. This function is an assistant function which makes use of the function **midi-events-organized-to-indicate-chords**, **make-instance-single-complex-note** and **match-music**, to compare the right or the left hands of two songs.

- (defun draw-notation (clef meter-scala midi-instances1 midi-instances2))

The function **draw-notation** takes as an argument two variables and two “seq” objects and returns a cmn object of type #staff. The first variable indicates the kind of the clef (i.e treble,tenor,bass). The second variable indicates the meter of the song (i.e 3/4, 6/8, 4/4) and the key signature (i.e C major, G minor). This function utilizes the function **compare-seperated-hands** and along with the function **show-notation-after-compare** produces all the data required to create the sheet notation.

- (defun show-notation-after-compare (list-obj))

The function **show-notation-after-compare** takes as an argument a list and returns a list. The input is the result from a comparison between the hands of a song and the output is a list of cmn objects, such as notes and rests. This function along with the functions **add-color-to-note**, **describe-rests**, **get-rest-value**, transform the instances into cmn notes and rests, so that the instances could be presented on a sheet notation.

- (defmethod add-color-to-note (code (song single-note)))

This method **add-color-to-note** takes as an argument a variable and an instance of a single-note class and returns a cmn note. The variable is the code indicating the kind of error regarding that instance, so an appropriate color is registered to that note.

- (defmethod add-color-to-note (code (song complex-note)))

This method **add-color-to-note** takes as an argument a variable and an instance of a complex-note class and returns a cmn chord. The variable is the code indicating the kind of error regarding that instance, so an appropriate color is registered to that chord.

- (defmethod add-color-to-note (code object))

This method **add-color-to-note** is used in case there is no object or code available and returns a rest of a whole measure.

- (defmethod note-to-notation ((song single-note)))

This method **note-to-notation** takes as an argument an instance of a single-note class and returns a cmn object. It transforms a single-note instance into a note object of cmn package, using its name and duration.

- (defmethod note-to-notation ((song complex-note)))

This method **note-to-notation** takes as an argument an instance of a complex-note class and returns a cmn object. It transforms a complex-note instance into a chord object of cmn package, using its name and duration.

- (defmethod note-to-notation (object))

This method **note-to-notation** is used in case there is no object available and returns a rest of a whole measure.

- (defmethod get-note-time ((song single-note)))

This method **get-note-time** takes as an argument an instance of a single-note class and returns a number indicating the time slot value of that instance.

- (defmethod get-note-time ((song complex-note)))

This method **get-note-time** takes as an argument an instance of a complex-note class and returns a number pointing out the time slot value of the first note of that instance.

- (defmethod get-note-time (object))

This method **get-note-time** is used in case there is no object to be used and returns 0.

- (defmethod get-note-duration ((song single-note)))

This method **get-note-duration** takes as an argument an instance of a single-note class and returns a number suggesting the duration slot value of that instance.

- (defmethod get-note-duration ((song complex-note)))

This method **get-note-duration** takes as an argument an instance of a complex-note class and returns a number depicting the duration slot value of the first note of that instance.

- (defmethod get-note-duration (object))

This method **get-note-duration** is used in case there is no object available and returns 0.

- (defun get-rest-value (note note1))

The function **get-rest-value** takes as an argument two instances and returns a number. The input of this function could be instances either of the single-note or complex-note class. The output is a number indicating a rest between two notes. This number refers to the difference of the time slot value of the second note from which the sum of the time and duration slot values of the first note are substracted.

- (defun describe-notes (var-duration))

The function **describe-notes** takes as an argument a number and returns a number. The input is a number indicating the duration slot value of an instance single-note or a complex-note. The output is a number indicating the rhythmic value of a note of that instance, which also depends on the tempo of the song. For example, if the tempo is 60, then a quarter note will be between 0.9375 and 1, but if the tempo becomes 120, the quarter note will be between 0.46875 and 0.5.

- (defun describe-rests (rest-scanner))

The function **describe-rests** takes as an argument a number and returns a cons. The input is a number indicating the rest value between two instances either a single-note or a complex-note. The output is a cons indicating the rhythmic value of a rest between those instances, which also depends on the tempo of the song. For example, if the tempo is 60, then a quarter rest will be between 0.9375 and 1, but if the tempo becomes 120, the quarter rest will be between 0.46875 and 0.5.

- (defun find-scala (key mode))

The function **find-scala** takes as an argument two numbers and returns the key signature of a song. The first number "mode" determines if the song is in minor or major scale and the "key" determines the key signature. For example, if the mode is 0 and the key is also 0 then a C major symbol is returned.

- (defun midi-info (object))

The function **midi-info** takes as an argument a "seq" object and returns two symbols. The first symbol indicates the key signature of a song and the second symbol the time signature (i.e 4/4, 6/8). Also it sets the global variable *my-tempo* to the original tempo of the song.

- (defun get-midi-info(&optional (filename nil)))

The function **get-midi-info** takes as an argument a ".mid" or ".midi" file and returns the key and the time signature of that file. If the file is type0, the global variable *my-tempo* is set to 60 and a 4/4 time singature is returned, but if the file is type1, then, along with the function above it returns the appropriate key and time signature of the song.

- (defun final-notation (filename1 &optional (filename2 nil)))

The function **final-notation** takes as an argument two files ".mid" or ".midi" and returns a sheet notation. The sheet notation consists of the melody of the second file which is the result of the comparison with the first file and if there are any differences between those files, then the notes on the sheet notation will be colored.

Proceeding with a couple of changes of some functions above, it is possible for the user to produce a sheet notation for a file ".mid" or ".midi". Those functions are shown below.

- (defun prepare-hand (hand))

The function **prepare-hand** takes as an argument a "seq" object and returns a list of single-note - complex-note instances. It is used instead of the function **compare-seperated-hands**.

- (defun song-draw-notation (clef meter-scala midi-instances))

The function **song-draw-notation** takes as an argument two variables and a “seq” object and returns a cmn object named <#staff>. It’s a resemblance of the function **draw-notation** and the only difference is that it makes use both of the function above and the **show-notation**.

- (defun show-notation (list-obj))

The function **show-notation** takes as an argument a list and returns a list. It resembles of the function **show-notation-after-compare**, but the only difference is that it accepts a list of instances that are produced from one file and not from the comparison of two files. Also, the function **add-color-to-note** is not necessary. So the function **note-to-notation** is used instead. It returns a list of notes and rests ready to be translated into the package cmn.

- (defun song-final-notation (&optional (filename nil)))

The function **song-final-notation** takes as an argument a “.mid” or “.midi” file and returns a sheet notation. It makes use of all the functions above and produce the sheet notation of that file.

- (defun lcs-prepare-song (&optional (filename nil)))

The function **lcs-prepare-song** takes as an argument a music file “.mid” or “.midi” and returns a list of both single-note and complex-note instances.

- (defun lcs-draw-notation (clef meter-scala midi-instances))

The function **lcs-draw-notation** takes as an argument two variables and two “seq” objects and returns a cmn object of type <#staff>. The first variable indicates the kind of the clef (i.e treble,tenor,bass). The second variable indicates not only the meter of the song (i.e 3/4, 6/8, 4/4) but also the key signature (i.e C major, G minor). This function makes use of the function **show-notation-after-compare** to produce all the data required so that the sheet notation could be created.

- (defun lcs-final-notation (filename1 &optional (filename2 nil)))

The function **lcs-final-notation** takes as an argument a “.mid” or “.midi” file and returns a sheet notation. The sheet notation consist of the melody of the second file which is the result of the comparison with the first file, (using the lcs algorithm) and if there are any differences between those files then the notes on the sheet notation will be colored.

6.4.4 GUI Functions

- (defun music-match (&key (new-process nil)))

The function **music-match** is the main function that calls the MM application.

- (define-application-frame music-match ())

The function **define-application-frame** implements the layout of the MM application. There are four basic areas. The header area where the title is, the screen area where the sheet notation is going to be revealed, the interactor area where the user could interact with the MM application and finally the actions area which acts as a history channel.

- (defun music-match-title (frame pane))

The function **music-match-title** takes as an argument a specific pane and returns the labels that will be presented as the title of the MM application in the header area.

- (defun output-pane (pane))

The function **output-pane** takes as an argument a pane and sets the *application-frame* variable to that pane, so that the output would be presented to that pane.

- (define-command (com-record :name ”Start Recording” :command-table midi-commands))

The function **com-record** takes as an argument a string pointing out a name for a song. That function allows the user to record a song by means of a midi keyboard. Using the function **portmidi-open** from the package cm, it opens the ports of communication between the computer and the midi keyboard.

- (defun start-recording (&key (new-process nil)))

The function **start-recording** uses the function **portmidi-record!** from the package cm to write midi messages.

- (defun get-my-input-device-id ())

By using the functions **countdevices**, **deviceinfo.input**, **pm-get-device-info** and **pm-get-default-input-device-id** from the package portmidi, the function **get-my-input-device-id** returns the input id that corresponds to the midi keyboard.

- (defun get-my-output-device-id ())

The function **get-my-output-device-id** uses the functions **countdevices**, **deviceinfo.input**, **pm-get-device-info** and **pm-get-default-output-device-id** from the package portmidi and, as a result, the output id that corresponds to the midi keyboard is returned.

- (define-command (com-stop :name ”Stop Recording” :command-table midi-commands))

The function **com-stop** allows the user to stop the procedure of recording and therefore, the song is saved as a “.mid” file to disk.

- (defun sort-by-time-if-needed (object))

The function **sort-by-time-if-needed** takes as an argument a “seq” object and returns a “seq” object. The reason this function is being used is that the function **portmidi-record!** from the package cm is not perfectly implemented. When the user tries to play a melody with the right hand and the left hand keeps holding a note, then the melody which has already been produced from the right hand has been red as midi messages, but they are not registered as midi events. Also, when the user stops pressing the note with their left hand, that particular note is passed as a midi event but in a wrong time order. Thus, this function categorises in the right order the midi events by sorting all the midi events of a “seq” object according to the time slot value.

- (define-command (com-play :name ”Play” :command-table file-commands))

The function **com-play** takes as an argument a string indicating the name of a “.mid” file. It allows the user to play that song by using timidity.

- (define-command (com-open-sheet :name ”OpenNotation” :command-table file-commands))

The function **com-open-sheet** takes as an argument a string showing the name of a “.cmn” file. It allows the user to open that sheet notation.

- (define-command (com-save-sheet :name ”SaveNotation” :command-table file-commands))

The function **com-save-sheet** takes as an argument a string indicating a name for the “.cmn” file. It allows the user to save a sheet notation.

- (define-command (com-sound :name ”Change Sound” :command-table file-commands))

The function **com-sound** takes as n argument two integers. The first indicates which sound card is going to be used and the second one indicates the volume. It allows the user to change both sound cards and volume.

- (defun change-sound-volume (soundcard volume))

The function **change-sound-volume** takes as an argument two numbers and sets the global variable *midi-player* so that the specified sound card along with the appropriate volume could be used.

- (define-command (com-reset :name ”Reset tt & dt” :command-table file-commands))

The function **com-reset** is used should the user have already changed the values of the global variables *time-tolerance* and *duration-tolerance*, but they want to see the results of a comparison in a sheet notation. Thus, in order that the user could avoid the new values of the global variables, which influence the result of the comparison, they set to the default value to each of them.

- (define-command (com-quit :name ”Quit” :command-table file-commands))

The function **com-quit** allows the user to exit the MM application.

- (define-command (com-clear :name ”Clear” :command-table file-commands))

The function **com-clear** allows the user to clear the layout of the screen area.

- (define-command (com-clear-history :name ”Clear History” :command-table file-commands))

The function **com-clear-history** allows the user to clear the layout of the actions area.

- (define-command (com-show :name ”Show Sheet After Compare” :command-table sheet-commands))

The function **com-show** takes as an argument two strings and returns a sheet notation. The strings indicate which files are going to be compared and the result of the comparison is presented to a sheet notation.

- (define-command (com-show-notation :name ”Sheet Notation” :command-table sheet-commands))

The function **com-show-notation** takes as an argument a string and returns a sheet notation. It allows the user to enter a ".mid" file and watch it's sheet notation.

- (define-command (com-close :name "Close Sheet" :command-table sheet-commands))

The function **com-close** allows the user to close the sheet notation.

- (define-command (com-discard :name "Discard Track" :command-table midi-commands))

The function **com-discard** sets the global variable *myseq* into nil. It allows the user to discard a previous song and record a new one.

- (define-command (com-transpose :name "Transporto" :command-table midi-commands))

The function **com-transpose** takes as an argument an integer and sets the global variable *transpose-value* to that number. That integer indicates semitones. It also, allows the user to make a *transporto* to a song.

- (define-command (com-music-match :name "Music Match" :command-table music-match-commands))

The function **com-music-match** takes as an argument five values and returns the result of a comparison between two songs. The first and second value indicate the *time-tolerance* and *duration-tolerance* global variable respectively. The third and fourth value, which are to be compared, represents the original song and the user's song respectively. The last value indicates whether the user would like to see the result of the comparison as a percentage of success or just the time period that the MM algorithm requires in order to act.

- (define-command (com-tempo-change :name "TempoChange" :command-table midi-commands))

The function **com-tempo-change** takes as an argument an integer and sets the global variable *my-tempo* to that number. It allows the user to change the tempo of a song.

- (define-command (com-lcs-music-match :name "LcsMatch" :command-table music-match-commands))

The function **com-lcs-music-match** takes as an argument a value for the variable *duration-tolerance*, two music files and a variable that offers the user the option to check or not the time report of the above function. It rerurns the percentage of success, after the comparison of the two music sequences has occured. Using the functions **lcs-match** and **lcs-percent-of-success**. Hence, if the time report variable is yes, then it will also return the time that was necessary so that the two sequences could be compared with the aid of the lcs algorithm.

- (define-command (com-lcs-show :name ”Show Sheet After Lcs Compare” :command-table sheet-commands))

The function **com-lcs-show** takes as an argument two music files, it compares them and returns the result in a music sheet notation, showing each note with the apropriate color.

Chapter 7

Evaluation

In support of the efficiency of the provided application, we present in this section a number of experiments that have been performed. A description of the experimentation platform and data sets is also cited followed by a performance analysis.

The application has been implemented and performed on a personal computer with 1, 7 M Intel Centrino processor, 512 MByte RAM, operating system Ubuntu Linux, while the developing packages utilised, were CM version 2.7.0, CMN version 10-11-05, PORTMIDI 1.1.1.1, CLIM 0.9, CMUCL 19c.

The experiment is conducted upon the Beethoven's "Ode to Joy" sheet notation. "Ode to Joy" will be played by 3 pianists, who are at a different level, and our goal concentrates on our attempt to spot similarities between the original "Ode to Joy" and the three music pieces represented by each one of the three pianists. Firstly, our intention will be concentrated on the implementation of the simple match algorithm and then we will proceed with the use of the LCS algorithm.

The clef of the bass is not depicted in the sheet notations, which is the left hand, because of an error in this version of the CM package. Despite that error, the MM application supports both the recording and demostrating utility of the left hand.

The percentages which are written on the following experiment, depict the percentage of success of the comparisons. Since there are three parameters taking part in the process (keynum, time, duration), there are eight situations that could occur. Each case depends on a coefficient factor analogous to that case. For example, if a pair of instances are equal, then the coefficient factor is 2 and if the keynum is the only mistake then the coefficient factor is 1.

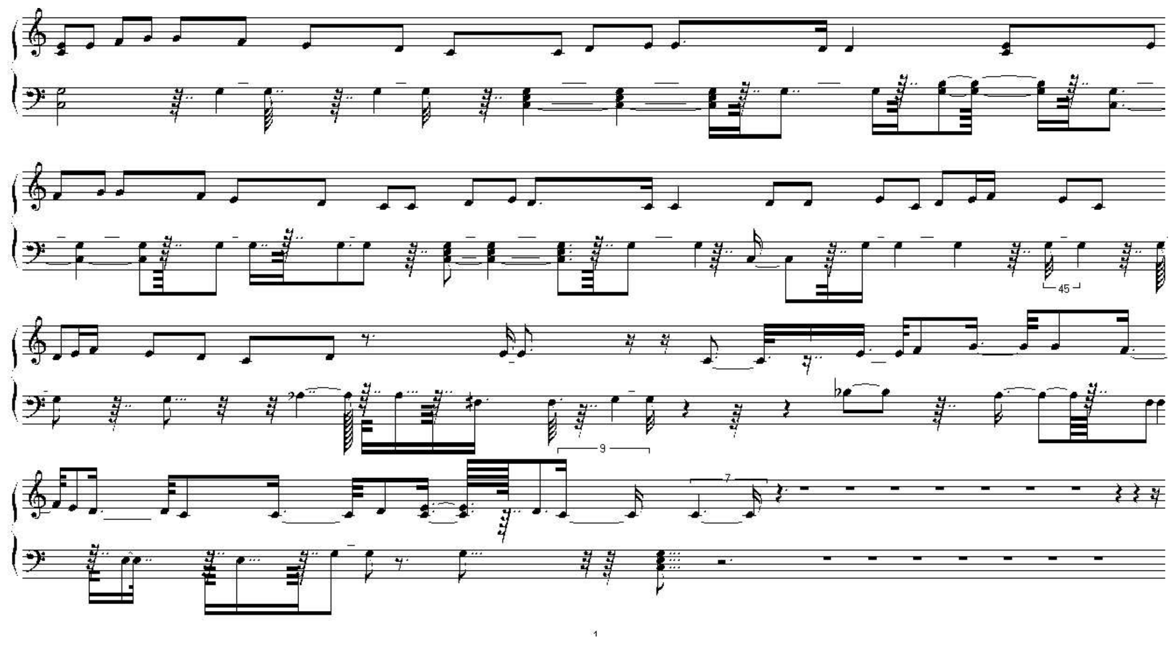


Figure 7.1: This is the original "Ode to Joy" sheet notation.

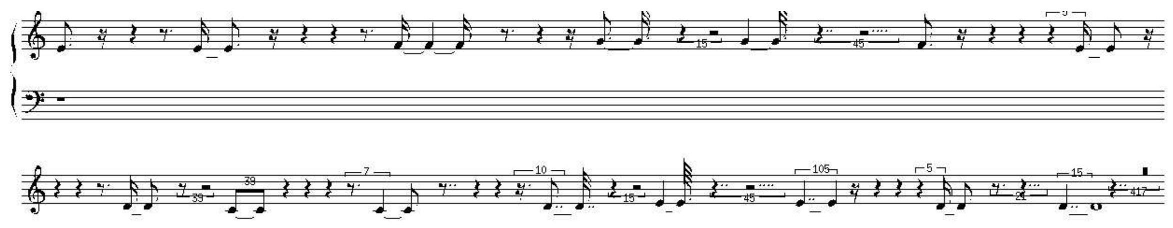


Figure 7.2: The 1*st* pianist.

This sheet notation extracted by a beginer pianist but steady and concentrated.

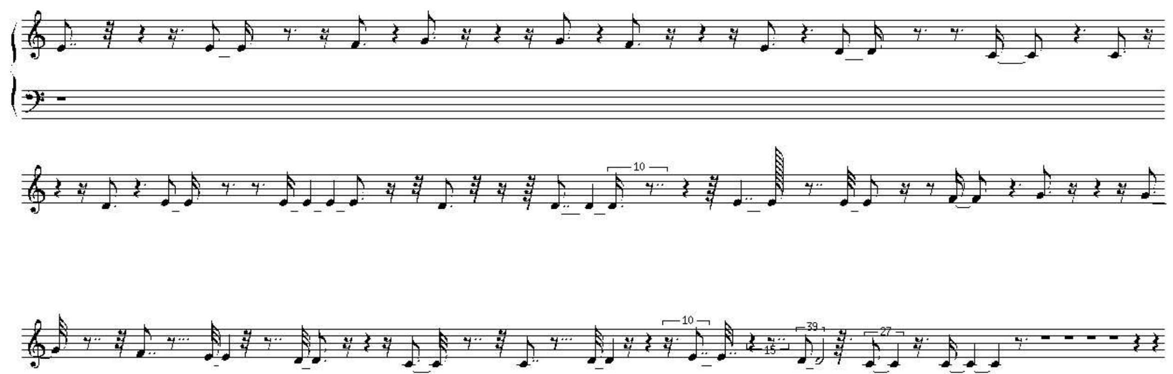


Figure 7.3: The 2*nd* pianist.

This sheet notation extracted by a medium-level pianist.

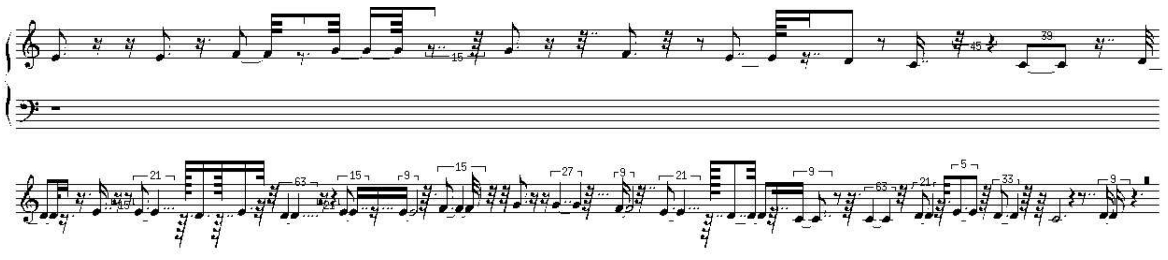


Figure 7.4: The *3rd* pianist.

This sheet notation extracted by a pianist with good technique but unsteady and very enthousiastic.

If note errors occur while comparing two songs, each error corresponds to a specific color according to the chromatic code below. We have 0, if the criteria of the comparison are satisfied and 1, if not.

keynum	Time	Duration	Chromatic Code
	t		(000) : black
0	0	1	(001) : blue
0	1	0	(010) : green
0	1	1	(011) : light blue
1	0	0	(100) : red
1	0	1	(101) : purple
1	1	0	(110) : yellow
1	1	1	(111) : white

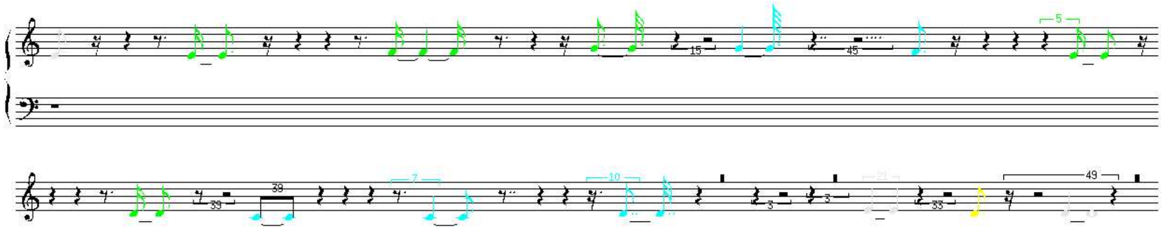


Figure 7.5: Simple match algorithm: Comparing the original song with the song of the 1*st* pianist with duration-tolerance= 0.2 and time-tolerance= 0.1, we have 18.333334% similarities.

The 1*st* pianist played a part of the *Ode to joy* with a restrict electronic teacher and the results were not good.

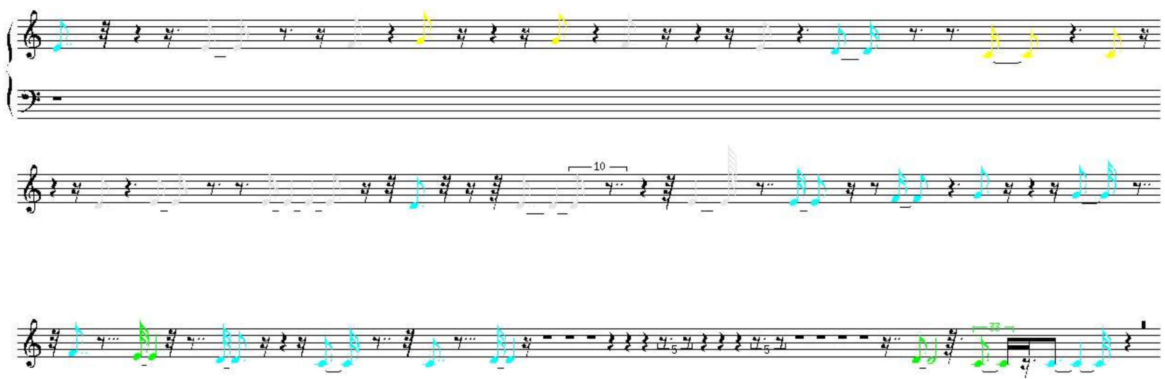


Figure 7.6: Simple match algorithm: Comparing the original song with the song of the 2*nd* pianist with duration-tolerance= 0.2 and time-tolerance= 0.1 we have 10.0% similarities.

The 2*nd* pianist played also a part of the *Ode to joy* with the same restrict electronic teacher and the results were not good as the 1*st* pianist, because he was out of the tempo some times.

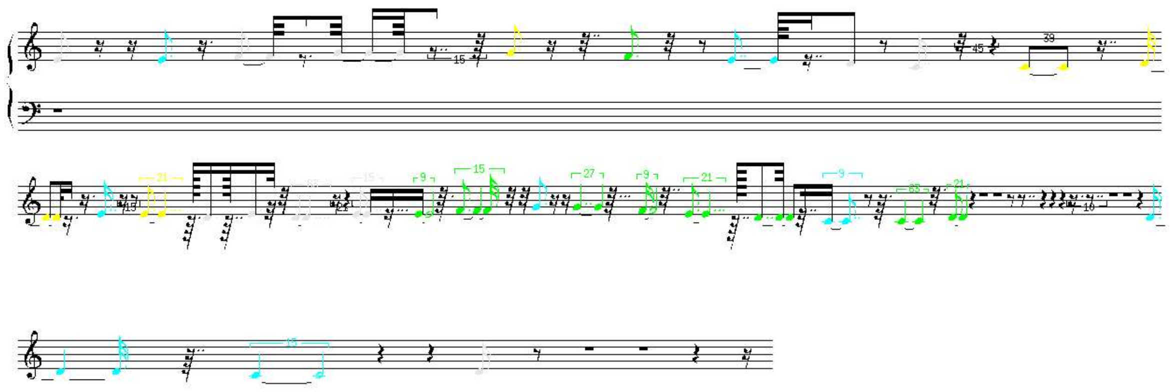


Figure 7.7: Simple match algorithm: Comparing the original song with the song of the 3*rd* pianist with duration-tolerance= 0.2 and time-tolerance= 0.1, we have 12.903225% similarities.

The 3*rd* pianist played a part of the *Ode to joy* with also the same restrict electronic teacher and the results were not good, but better than the 2*nd* pianist and worse than the 1*st* one, because some times he was out of the tempo and some ohters out of the duration.

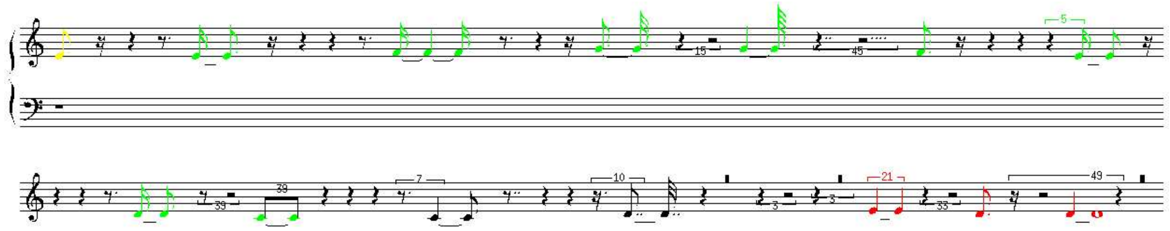


Figure 7.8: Simple match algorithm: Comparing the original song with the song of the 1*st* pianist with duration-tolerance= 20 and time-tolerance= 20, we have 25.0% similarities.

By increasing the time and duration tolerance we have better results, but yet not good.

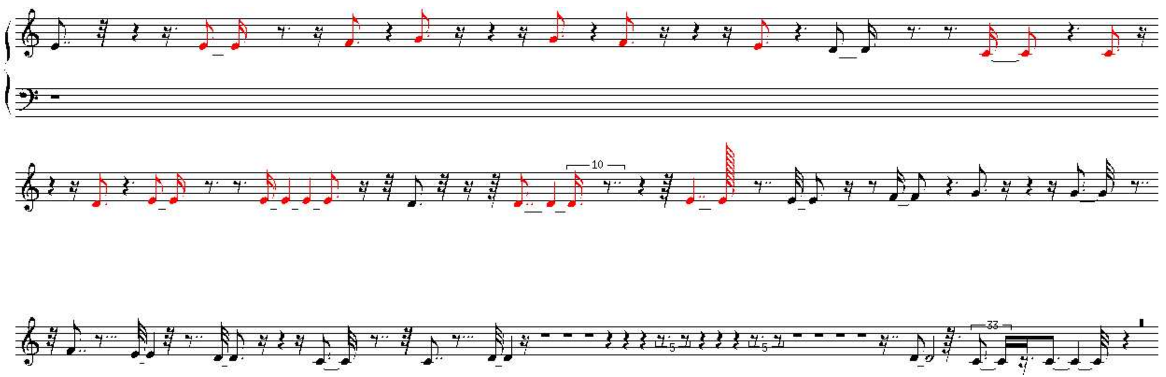


Figure 7.9: Brute-force algorithm: Comparing the original song with the song of the $2nd$ pianist with duration-tolerance= 20 and time-tolerance= 20. We have 35.0% similarities.

The $2nd$ pianist becomes better than the $1st$ one.

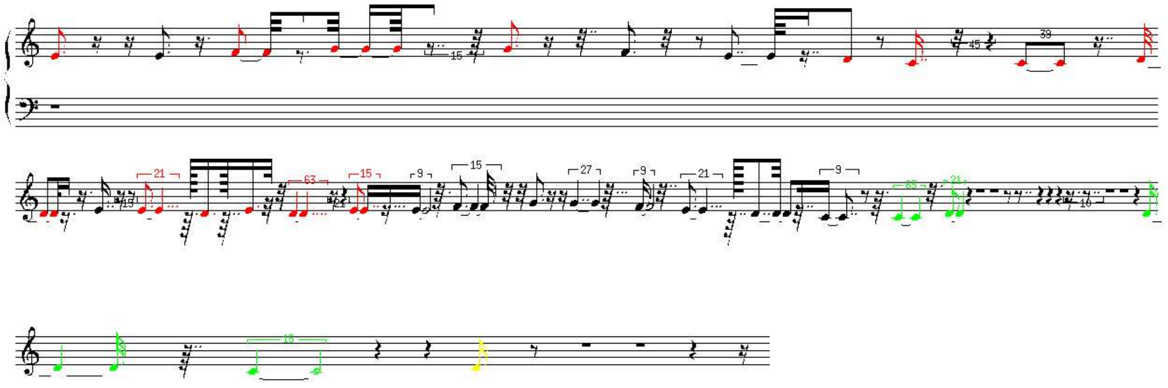


Figure 7.10: Brute-force algorithm: Comparing the original song with the song of the $3rd$ pianist with duration-tolerance= 20 and time-tolerance= 20, we have 34.677418% similarities.

The $3rd$ pianist also becomes better than the $1st$ one.

The outcome of our estimation regarding the evaluation time with the contribution of the brute-force algorithm for each comparison are the following:

Using only the simple match algorithm

- For the 1*st* comparison, the evaluation lasted: 0.03 seconds of real time, 0.024996 seconds of user run time, 9.99e-4 seconds of system run time, 21,427,271 CPU cycles, 0 page faults and 402,112 bytes consed.
- For the 2*nd* comparison, the evaluation occupied: 0.03 seconds of real time, 0.027996 seconds of user run time, 0.001 seconds of system run time, 22,581,305 CPU cycles, 0 page faults and 456,872 bytes consed.
- For the 3*rd* comparison, the evaluation consumed: 0.03 seconds of real time, 0.027996 seconds of user run time, 0.0 seconds of system run time, 22,504,620 CPU cycles, 0 page faults and 460,376 bytes consed.

Using the simple match algorithm and demonstrating the results in a sheet notation

- For the 1*st* comparison, the evaluation lasted: 0.07 seconds of real time, 0.06899 seconds of user run time, 0.0 seconds of system run time, 55,142,910 CPU cycles, 0 page faults and 1,930,680 bytes consed.
- For the 2*nd* comparison, the evaluation occupied: 0.12 seconds of real time, 0.102984 seconds of user run time, 0.002999 seconds of system run time, 93,067,968 CPU cycles, 0 page faults and 3,345,248 bytes consed.
- For the 3*rd* comparison, the evaluation consumed: 0.12 seconds of real time, 0.098985 seconds of user run time, 0.005 seconds of system run time, 90,134,574 CPU cycles, 0 page faults and 3,260,976 bytes consed.

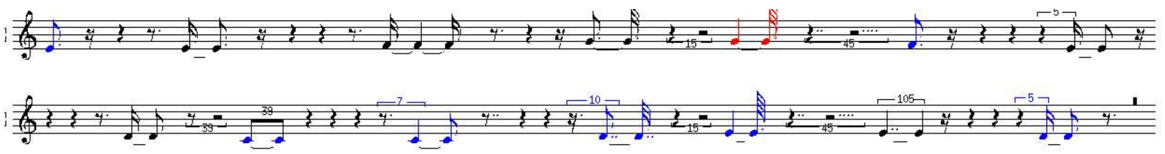


Figure 7.11: LCS algorithm: Comparing the original song with the song of the 1*st* pianist with duration-tolerance= 0.2, we have 50 deletions, 1 insertion and 67.85714% similarities.

The 1*st* pianist has good results but not perfect with a restrict electronic teacher.

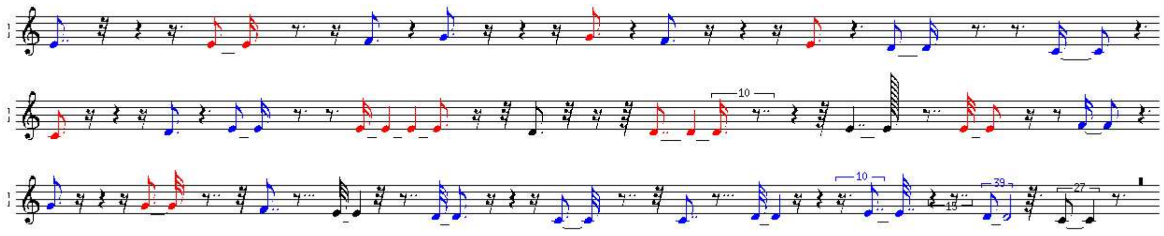


Figure 7.12: LCS algorithm: Comparing the original song with the song of the 2*nd* pianist with duration-tolerance= 0.2, we have 42 deletions, 8 insertions and 43.103447% similarities.

The 2*nd* pianist is not as good as the 1*st* one because of some durations errors.

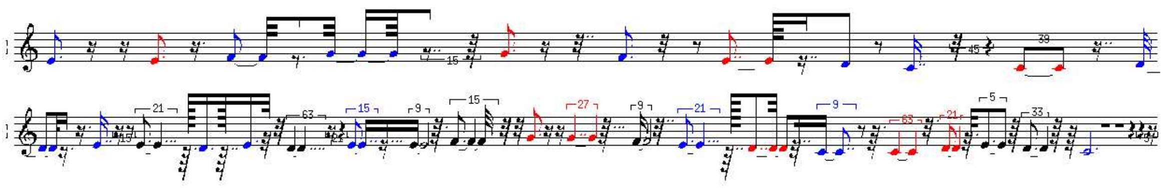


Figure 7.13: LCS algorithm: Comparing the original song with the song of the 3*rd* pianist with duration-tolerance= 0.2, we have 42 deletions, 9 insertions and 46.666668% similarities.

The 3*rd* pianist is better than the 2*nd* one, but because of some duration errors is not as good as the 1*st* one.

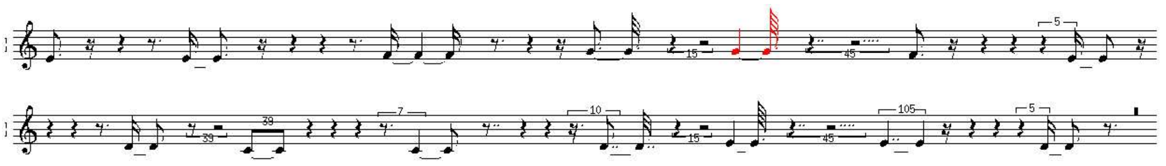


Figure 7.14: LCS algorithm: Comparing the original song with the song of the 1*st* pianist with duration-tolerance= 1, we have 50 deletions, 1 insertion and 92, 85714% similarities.

The 1*st* pianist with a more tolerable electronic teacher is almost perfect.

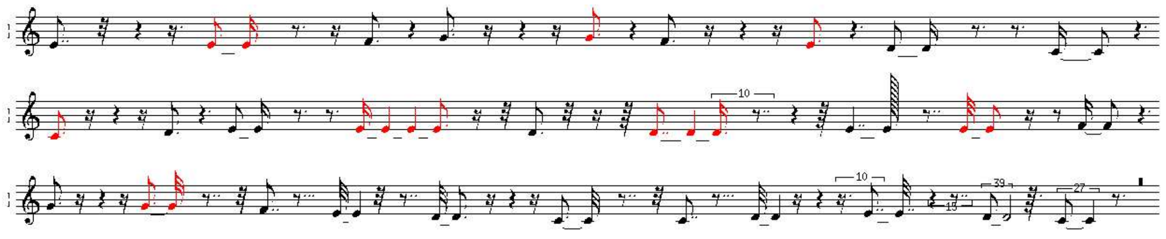


Figure 7.15: LCS algorithm: Comparing the original song with the song of the 2*nd* pianist with duration-tolerance= 1, we have 42 deletions, 8 insertions and 72.413795% similarities.

The 2*nd* pianist with a more tolerable electronic teacher is not as good as the 1*st* one, because of some duration errors.

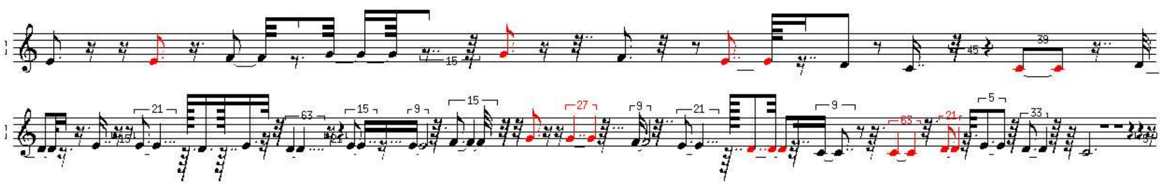


Figure 7.16: LCS algorithm: Comparing the original song with the song of the 3*rd* pianist with duration-tolerance= 1, we have 42 deletions, 9 insertions and 70.0% similarities.

The 3*rd* pianist, despite his good technique has more duration errors than the 2*nd* and the 1*st* pianist.

As far as the evaluation time is concerned and with the use of the LCS algorithm, we have the following results for each comparison:

Using only the LCS algorithm

- For the 1*st* comparison, the evaluation lasted: 0.03 seconds of real time, 0.025996 seconds of user run time, 0.001 seconds of system run time, 21,474,265 CPU cycles, 0 page faults and 413,424 bytes consed.

- For the 2*nd* comparison, the evaluation consumed: 0.03 seconds of real time, 0.030995 seconds of user run time, 0.002 seconds of system run time, 26,801,183 CPU cycles, 0 page faults and 478,840 bytes consed.

- For the 3*rd* comparison, the evaluation occupied: 0.03 seconds of real time, 0.029995 seconds of user run time, 0.002 seconds of system run time, 25,459,815 CPU cycles, 0 page faults and 501,616 bytes consed.

Using the LCS algorithm and demonstrating the results in a sheet notation

- For the 1*st* comparison, the evaluation lasted: 0.07 seconds of real time, 0.06399 seconds of user run time, 0.005999 seconds of system run time, 55,363,188 CPU cycles, 0 page faults and 1,917,520 bytes consed.

- For the 2*nd* comparison, the evaluation consumed: 0.11 seconds of real time, 0.097986 seconds of user run time, 0.003 seconds of system run time, 86,922,774 CPU cycles, 0 page faults and 3,154,568 bytes consed.

- For the 3*rd* comparison, the evaluation occupied: 0.1 seconds of real time, 0.093986 seconds of user run time, 0.007999 seconds of system run time, 84,326,730 CPU cycles, 0 page faults and 3,021,776 bytes consed.

	$1st$ pianist	$2nd$ pianist	$3rd$ pianist
Human teacher	A beginer pianist which keep the right tempo	A medium level pianist	A pianist with good technique but unsteady.
Electronic teacher, using the simple match algorithm with tt=0.1 and dt=0.2	$18,333334\%$	$10,0\%$	$12,903225\%$
Electronic teacher, using the simple match algorithm with tt=20 and dt=20	$25,0\%$	$35,0\%$	$34,677418\%$
Electronic teacher, using the LCS algorithm with tt=$0,1$ and dt=$0,2$	$65,85714\%$	$43,103447\%$	$46,666668\%$
Electronic teacher, using the LCS algorithm with tt=20 and dt=20	$92,85714\%$	$72,413795\%$	$70,0\%$

As we see in the table above, even a pianist with good technique as the $3rd$ one, might be not so good with electronic teacher if he does not follow the tempo and improvise some parts. But, a beginer pianist as the $1st$ one, might be a very good executant with electronic teacher, as he follows the music rules.

Chapter 8

Conclusions & Future Work

Conclusions:

This application is an attempt for potential users to perceive and represent the possibilities and limitations of an approach to music pattern matching and music representation. Using the brute-force algorithm, it is hard for the application to find similarities between two scores, unless the user increases to a great extent the time-tolerance and duration-tolerance parameters. It is also runs in time $O(n^2)$ making it impractical for long sequences. Nonetheless, despite any human errors that may occur, it is easy for the application to find similarities between two songs – the original and the user's song – with the aid of the LCS algorithm while they are playing the piano and without increasing the duration-tolerance parameter.The LCS algorithm runs in time $O(nm)$ which is propably very hard to improve. But, a time-tolerance parameter could be added, which is more relevant to a computer-aided learning of music, than our trying to improve the $O(nm)$ factor.

As far as future work is concerned, there are some technical issues that need to be changed and some feature issues that should be added.

Technical Issues:

The function **portmidi-record!** from package cm needs some changes. When a note parses a midi-note-on message and is pressed down for a period of time, a counter starts counting until the note parses a midi-note-off message. That counter indicates the duration of that note. The problem occurs when someone tries to play something at that period of time. The new midi-note-on's and midi-note-of's messages are parsed correctly but it is impossible for the function to write down those messages as midi events. In the end, only the first note could become a midi event whereas the other notes are lost.

The function **midi-events-organized-to-indicate-chords** needs to be redefined. Instead of a list, a hash table should be used to hold the chords.

The function **duration-check** should change. If two instances of complex-note are to be compared, then at least the notes which are equal should be returned and not the whole chord.

Also, the process of presenting the left hand in a sheet notation should be subject to certain changes. The "seq" object, that holds the left hand, should be transposed upto 4 semitones down to satisfy the bass clef.

The sheet notation should be demonstrated inside the screen area of the MM application.

The function **insert-bars-ties-and-rests** in package CMN in file cmn4.lisp requires some changes on how the bars assignment on an unmeasured score are handled.

The LCS algorithm supports a duration false tolerance ability but a time false tolerance ability could also be added.

Feature Issues:

The MM algorithm is able to compare two songs among three parameters, which are the note, time, duration. An amplitude parameter should be added, so that the user would be more careful on how hard (*forte*) or how soft (*piccolo*) they press the notes according to the sheet notation.

There is also a package named gsharp, which allows the user to create a sheet notation by typewriting in real time. A nice idea would be if the MM application adapted this package, so that the user could play on a midi keyboard and thus, a sheet notation would be created in real time. In the past, an effort was made so that this achievment could be accomplished, but the existing knowledge at that time was insufficient and an e-mail from the creator of the package gsharp has made clear that this idea could only be applied without the time and duration parameters, or at least until now its too dificult for it to occur. If someone wishes to acquire further information on the topic, they should contact Robert Strandh. Alternatively, if they wish to receive email information about CCRMA's family of Lisp music programs (CM, CLM and CMN) they should e-mail cmdist-request@ccrma.stanford.edu. By using the jack with qsynth and the appropriate connections between the midi keyboard and the sound card, the MM application will be able to support sound production. But that implies that some changes in the CM package should be achieved, in the first place.

Last but not least, a print utility function should be added to facilitate the print of a sheet notation.

Chapter 9

Appendix

9.1 Musical Terms

A few usefull musical terms.

Transposition

Music may be transposed when the original key is changed, a process all too necessary in accompanying singers and for whom a transposition of the music down a tone or two may be necessary. Some instruments are known as transposing instruments because the written notes for them sound higher or lower than the apparent written pitch, when they are played.

Forte

Forte (Italian: loud) is used in directions to performers. It appears in the superlative form fortissimo, very loud. The letter f is an abbreviation of forte, ff an abbreviation of fortissimo, with fff or more rarely ffff even louder.

Piano

Piano (Italian: soft) is generally represented by the letter p in directions to performers. Pianissimo, represented by pp, means very soft. Addition of further letters p indicates greater degrees of softness, as in Tchaikovsky's Sixth Symphony, where an excessive pppppp is used.

Mezzo

Mezzo (Italian: half) is found particularly in the compound words mezzo-forte, half loud, represented by the letters mf, and mezzo-piano, half soft, represented by the letters mp. Mezzo can serve as a colloquial abbreviation for mezzo-soprano, the female voice that employs a generally lower register than a soprano and consequently is often, in opera, given the parts of confidante, nurse or mother, secondary roles to the heroine, usually a soprano. The instruction mezza voce directs a singer to sing with a controlled tone. The instruction can also occur in instrumental music.

Notation

Notation is the method of writing music down, practices of which have varied during the course of history. Staff notation is the conventional notation that makes use of the five-line staff or stave, while some recent composers have employed systems of graphic notation to indicate their more varied requirements, often needing detailed explanations in a preface to the score. Notation is inevitably imprecise, providing a guide of varying accuracy for performers, who must additionally draw on stylistic tradition.

Octave

The octave is an interval of an eighth, as for example from the note C to C or D to D. The first note can have a sharp or flat providing the last note has the corresponding sharp or flat (i. e. C sharp to C sharp).

Score

A musical score is written music that shows all parts. A conductor's score, for example, may have as many as thirty different simultaneous instrumental parts on one page, normally having the woodwind at the top, followed below by the brass, the percussion and the strings. A distinction is made between a vocal score, which gives voice parts with a simplified two-stave version of any instrumental parts, and a full score, which includes all vocal and instrumental parts generally on separate staves. To score a work is to write it out in score. A symphony, for example, might be sketched in short score, on two staves, and later orchestrated or scored for the required instruments.

Scale

A scale is a sequence of notes placed in ascending or descending order by step.

Staff

The staff or stave (plural: staves) indicates the set of lines used for the notation of notes of different pitches. The five-line stave is in general use, with a four-line stave used for plainchant. Staves of other numbers of lines were once used. The system, with coloured lines for C and for F, followed principles suggested first by Guido of Arezzo in the 11th century. Staff notation is the system of notation that uses the stave.

9.2 Classes

```
(defclass music-element ()())



(defclass single-note (music-element)
  ((note-name :accessor single-note-name
              :initarg :note-name
              :initform nil)
   (time      :accessor single-note-time
              :initarg :time
              :initform 0)
   (duration  :accessor single-note-duration
              :initarg :duration
              :initform 0)))



(defclass complex-note (music-element)
  ((note-name :accessor complex-note-name
              :initarg :note-name
              :initform nil)
   (time      :accessor complex-note-time
              :initarg :time
              :initform 0)
   (duration  :accessor complex-note-duration
              :initarg :duration
              :initform 0)))
```

```
(progn
  (defclass container nil
    ((name :initform nil
           :accessor object-name
           :initarg :name)))
  (defparameter <container> (find-class 'container))
  (finalize-class <container>)
  (values))
```

```
(progn
  (defclass seq (container)
    ((time :accessor object-time
           :initarg :time
           :initform 0)
     (subobjects :initform '()
                 :accessor container-subobjects
                 :initarg :subobjects)))
  (defparameter <seq> (find-class 'seq))
  (finalize-class <seq>)
  (values))
```

9.3 Variables

```
(defparameter *myseq* nil)
```

```
(defparameter *time-chords* .0234375)
```

```
(defparameter *my-tempo* 60)
```

```
(defparameter *time-tolerance* .1)
```

```
(defparameter *duration-tolerance* .2)
```

```
(defparameter *transpose-value* 0)
```

```
(setf *cmn-output-type* :x)
```

```
*midi-player* timidity -A 300
```

```
(defparameter *seq-name* "my_song")
```

```
(defparameter *pm* nil)
```

```
size 30
```

```
page-width 45
```

```
automatic-page-numbers t
```

9.4 Functions

9.4.1 MM (Music Matching) Algorithm Functions

```
(defun compare-music (filename1 &optional (filename2 nil))
  (match-music
   (make-instance-single-complex-note
    (midi-events-organized-to-indicate-chords
     (import-song filename1)))
   (make-instance-single-complex-note
    (midi-events-organized-to-indicate-chords
     (or (import-song filename2)
         cl-user::*myseq*)))))



(defun import-song (filename)
  (let ((song (load-song filename :key-note ':keynum)))
    (if (atom song)
        song
      (transform-midi-to-seq song))))



(defun load-song (filename &key (meta cm::true) (key-note ':note))
  (cond ((null filename) nil)
        (t (cm::import-events
            (namestring
             (make-pathname :directory '(:absolute "home" "pontikis" "Lisp")
                            :name filename
                            :type (or "mid" "midi")))
            :meta-exclude meta :keynum-format key-note))))



(defun transform-midi-to-seq (object)
  (remove-non-midi-object
   (dolist (track object)
     (if (cm::container-subobjects track)
         (return track)))))



(defun remove-non-midi-object (object)
  (let ((midi-ev nil))
    (dolist (midievent (cm::container-subobjects object))
      (typecase midievent
        (cm::MIDI (push midievent midi-ev))))
    (cm::events (reverse midi-ev)
                (new cm::seq :name (object-name object)))))
```

```
(let ((midi-events)
      (midi-complex))
  (defun midi-events-organized-to-indicate-chords (object)
   (let ((len (length (cm::container-subobjects object))))
    (cond ((null object)
           nil)
          (t
           (setq midi-events nil)
           (setq midi-complex nil)
           (dotimes (i len)
             (cond ((= len 1)
                      (push (first-subobject object 0 1)
                            midi-events))
                    ((equal i (- len 1))                       ;;1st case
                     (when (<= 0 (substitute-times object i (+ i 1)
                                                    (- i 1) i)
                                 *time-chords*)
                      (push (first-subobject object i (+ i 1))
                            midi-complex)
                      (push midi-complex
                            midi-events))
                     (unless (<= (substitute-times object i (+ i 1)
                                                    (- i 1) i)
                                   *time-chords*)
                      (push (first-subobject object i (+ i 1))
                            midi-events)))
                    ((<= 0 (substitute-times object i (+ i 1)        ;;2nd case
                                             (+ i 1) (+ i 2))
                        *time-chords*)
                     (push (first-subobject object i (+ i 1))
                           midi-complex))
                    ((>= (substitute-times object i (+ i 1)          ;;3rd case
                                           (+ i 1) (+ i 2))
                        *time-chords*)
                    (when (equal (- i 1) -1)
                      (push (first-subobject object i (+ i 1))
                            midi-events)
                      (setf i 1))
                    (when (<= 0 (substitute-times object i (+ i 1)
                                                         (- i 1) i)
                              *time-chords*)
                      (push (first-subobject object i (+ i 1))
                            midi-complex)
                      (push midi-complex
                            midi-events)
                      (setq midi-complex nil))
                    (when (and (equal i 1)
                               (<= 0 (substitute-times object i (+ i 1)
                                                        (+ i 1) (+ i 2))
                                   *time-chords*))
                      (push (first-subobject object i (+ i 1))
                            midi-complex))
                    (unless (or (equal (- i 1) -1)
                                (<= 0 (substitute-times object i (+ i 1)
                                                               (- i 1) i)
                                    *time-chords*)
                                (<= 0 (substitute-times object i (+ i 1)
                                                         (+ i 1) (+ i 2))
                                    *time-chords*))
                      (push (first-subobject object i (+ i 1))
                            midi-events)))))
           midi-events)))))


(defmacro first-subobject (obj start end)
  ‘(first (cm::subobjects ,obj :start ,start :end ,end)))


(defmacro substitute-times (obj start1 end1 start2 end2)
  ‘(cond ((null (first-subobject ,obj ,start2 ,end2))
          (sv (first-subobject ,obj ,start1 ,end1) :time))
         (t
          (abs (- (sv (first-subobject ,obj ,start1 ,end1) :time)
                  (sv (first-subobject ,obj ,start2 ,end2) :time))))))



(defun make-instance-single-complex-note (midi-events-list)
  (mapcar #'(lambda (midis)
              (if (atom midis)
               (make-instance 'single-note
                              :note-name (cm::note
                                          (cm::transpose
                                           (sv midis :keynum)
                                            *transpose-value*))
                              :time (sv midis :time)
                              :duration (sv midis cm::duration))
               (make-instance 'complex-note
                              :note-name (cm::note
                                          (sort
                                           (keynum
                                            (mapcar
                                              #'(lambda (x)
                                                  (cm::transpose
                                                    (sv x :keynum)
                                                      *transpose-value*))
                                               midis))
                                           #'<))
                              :time (mapcar #'(lambda (x)
                                                 (sv x :time))
                                           midis)
                              :duration (mapcar #'(lambda (x)
                                                     (sv x cm::duration))
                                               midis))))
        midi-events-list))


(defun match-music (oldsong newsong)
  (mapcar #'match-note oldsong newsong))
```

```
(let ((res))
  (defmethod match-note ((oldsong single-note)
                         (newsong single-note)
                         &key (note-only nil))
    (setf res nil)
    (cond (note-only
           (setf res (equal (single-note-name oldsong)
                            (single-note-name newsong))))
          ((and (equal (single-note-name oldsong)
                       (single-note-name newsong))
                (<= 0
                    (abs (- (single-note-time oldsong)
                            (single-note-time newsong)))
                    *time-tolerance*)
                (<= 0
                    (abs (- (single-note-duration oldsong)
                            (single-note-duration newsong)))
                    *duration-tolerance*))
           (setf res (list t newsong)))
          ((and (equal (single-note-name oldsong)
                       (single-note-name newsong))
                (<= 0
                    (abs (- (single-note-time oldsong)
                            (single-note-time newsong)))
                    *time-tolerance*)
                (>
                 (abs (- (single-note-duration oldsong)
                         (single-note-duration newsong)))
                 *duration-tolerance*))
           (setf res (list 001 newsong)))
          ((and (equal (single-note-name oldsong)
                       (single-note-name newsong))
                (>
                 (abs (- (single-note-time oldsong)
                         (single-note-time newsong)))
                 *time-tolerance*)
                (<= 0
                    (abs (- (single-note-duration oldsong)
                            (single-note-duration newsong)))
                    *duration-tolerance*))
```

```
 (setf res (list 010 newsong)))
((and (equal (single-note-name oldsong)
             (single-note-name newsong))
      (>
       (abs (- (single-note-time oldsong)
               (single-note-time newsong)))
       *time-tolerance*)
      (>
       (abs (- (single-note-duration oldsong)
               (single-note-duration newsong)))
       *duration-tolerance*))
 (setf res (list 011 newsong)))
((and (not
       (equal (single-note-name oldsong)
              (single-note-name newsong)))
      (<= 0
          (abs (- (single-note-time oldsong)
                  (single-note-time newsong)))
          *time-tolerance*)
      (<= 0
          (abs (- (single-note-duration oldsong)
                  (single-note-duration newsong)))
          *duration-tolerance*))
 (setf res (list 100 newsong)))
((and (not
       (equal (single-note-name oldsong)
              (single-note-name newsong)))
      (<= 0
          (abs (- (single-note-time oldsong)
                  (single-note-time newsong)))
          *time-tolerance*)
      (>
       (abs (- (single-note-duration oldsong)
               (single-note-duration newsong)))
       *duration-tolerance*))
 (setf res (list 101 newsong)))
((and (not
       (equal (single-note-name oldsong)
              (single-note-name newsong)))
      (>
                 (abs (- (single-note-time oldsong)
                         (single-note-time newsong)))
                 *time-tolerance*)
                (<= 0
                    (abs (- (single-note-duration oldsong)
                            (single-note-duration newsong)))
                    *duration-tolerance*))
           (setf res (list 110 newsong)))
          ((and (not
                 (equal (single-note-name oldsong)
                        (single-note-name newsong)))
                (>
                 (abs (- (single-note-time oldsong)
                         (single-note-time newsong)))
                 *time-tolerance*)
                (>
                 (abs (- (single-note-duration oldsong)
                         (single-note-duration newsong)))
                 *duration-tolerance*))
           (setf res (list 111 newsong))))
    res))



(let ((res))
  (defmethod match-note ((oldsong complex-note)
                         (newsong complex-note)
                         &key (note-only nil))
    (cond (note-only
           (setf res (equal (complex-note-name oldsong)
                            (complex-note-name newsong))))
          ((and (equal
                 (complex-note-name oldsong)
                 (complex-note-name newsong))
                (<= 0
                    (abs (-
                          (/ (apply #'+ (complex-note-time oldsong))
                             (length (complex-note-time oldsong)))
                          (/ (apply #'+ (complex-note-time newsong))
                             (length (complex-note-time newsong)))))
                    *time-tolerance*)
                (equal 0
                       (duration-check (complex-note-duration oldsong)
                                         (complex-note-duration newsong))))
           (setf res (list t newsong)))
          ((and (equal (complex-note-name oldsong)
                       (complex-note-name newsong))
                (<= 0
                    (abs (-
                          (/ (apply #'+ (complex-note-time oldsong))
                             (length (complex-note-time oldsong)))
                          (/ (apply #'+ (complex-note-time newsong))
                             (length (complex-note-time newsong)))))
                    *time-tolerance*)
                (>
                 (duration-check (complex-note-duration oldsong)
                                 (complex-note-duration newsong)) 0))
           (setf res (list 001 newsong)))
          ((and (equal (complex-note-name oldsong)
                       (complex-note-name newsong))
                (>
                 (abs (-
                       (/ (apply #'+ (complex-note-time oldsong))
                          (length (complex-note-time oldsong)))
                       (/ (apply #'+ (complex-note-time newsong))
                          (length (complex-note-time newsong)))))
                 *time-tolerance*)
                (equal 0
                       (duration-check (complex-note-duration oldsong)
                                       (complex-note-duration newsong))))
           (setf res (list 010 newsong)))
          ((and (equal (complex-note-name oldsong)
                       (complex-note-name newsong))
                (>
                 (abs (-
                       (/ (apply #'+ (complex-note-time oldsong))
                          (length (complex-note-time oldsong)))
                       (/ (apply #'+ (complex-note-time newsong))
                          (length (complex-note-time newsong)))))
                 *time-tolerance*)
                (>
                 (duration-check (complex-note-duration oldsong)
                                 (complex-note-duration newsong)) 0))
           (setf res (list 011 newsong)))
          ((and (not
                 (equal (complex-note-name oldsong)
                        (complex-note-name newsong)))
                (<= 0
                    (abs (-
                          (/ (apply #'+ (complex-note-time oldsong))
                             (length (complex-note-time oldsong)))
                          (/ (apply #'+ (complex-note-time newsong))
                             (length (complex-note-time newsong)))))
                    *time-tolerance*)
                (equal 0
                       (duration-check (complex-note-duration oldsong)
                                       (complex-note-duration newsong))))
           (setf res (list 100 newsong)))
          ((and (not
                 (equal (complex-note-name oldsong)
                        (complex-note-name newsong)))
                (<= 0
                    (abs (-
                          (/ (apply #'+ (complex-note-time oldsong))
                             (length (complex-note-time oldsong)))
                          (/ (apply #'+ (complex-note-time newsong))
                             (length (complex-note-time newsong)))))
                    *time-tolerance*)
                (>
                 (duration-check (complex-note-duration oldsong)
                                 (complex-note-duration newsong)) 0))
           (setf res (list 101 newsong)))
          ((and (not
                 (equal (complex-note-name oldsong)
                        (complex-note-name newsong)))
                (>
                 (abs (-
                       (/ (apply #'+ (complex-note-time oldsong))
                          (length (complex-note-time oldsong)))
                       (/ (apply #'+ (complex-note-time newsong))
                          (length (complex-note-time newsong)))))
                 *time-tolerance*)
                (equal 0
                       (duration-check (complex-note-duration oldsong)
                                       (complex-note-duration newsong))))
           (setf res (list 110 newsong)))
          ((and (not
                 (equal (complex-note-name oldsong)
                        (complex-note-name newsong)))
                (>
                 (abs (-
                       (/ (apply #'+ (complex-note-time oldsong))
                          (length (complex-note-time oldsong)))
                       (/ (apply #'+ (complex-note-time newsong))
                          (length (complex-note-time newsong)))))
                 *time-tolerance*)
                (>
                 (duration-check (complex-note-duration oldsong)
                                 (complex-note-duration newsong)) 0))
           (setf res (list 111 newsong))))
    res))
```

```
(defmacro duration-check(old new)
  `(count nil
          (mapcar (lambda (x y)
                    (when (<= (abs (- x y)) *duration-tolerance*)
                      t)) ,old ,new)))
```

9.4.2 LCS (Longest-Common-Subsequence) Functions

```
(defun zero-column(array)
  (do ((i 0 (+ i 1))
       (m (car (array-dimensions array))))
      ((= i m))
    (setf (aref array i 0) 0)))
```

```
(defun zero-row(array)
  (do ((j 0 (+ j 1))
       (n (second (array-dimensions array))))
      ((= j n))
    (setf (aref array 0 j) 0)))



(defun lcs-length (seq1 seq2)
  (let* ((len1 (length seq1))
         (len2 (length seq2))
         (c (make-array (list (+ len1 1) (+ len2 1))
                        :initial-element nil))
         (b (make-array (list (+ len1 1) (+ len2 1))
                        :initial-element nil)))
    (zero-row c)
    (zero-column c)
    (do ((i 1 (+ i 1)))
        ((= i (+ len1 1) ))
      (do ((j 1 (+ j 1)))
          ((= j (+ len2 1) ))
        (cond ((equal (get-note-name
                       (nth (- i 1) seq1))
                      (get-note-name
                       (nth (- j 1) seq2)))
               (when (> (abs (- (get-note-duration
                                 (nth (- i 1) seq1))
                                (get-note-duration
                                 (nth (- j 1) seq2))))
                        *duration-tolerance*)
                 (setf (aref c i j)
                       (+ (aref c (- i 1) (- j 1))
                          1))
                 (setf (aref b i j)
                       'diagonal-up-no-duration))
               (unless (> (abs (- (get-note-duration
                                   (nth (- i 1) seq1))
                                  (get-note-duration
                                   (nth (- j 1) seq2))))
                          *duration-tolerance*)
                 (setf (aref c i j)
                       (+ (aref c (- i 1) (- j 1))
                          1))
                 (setf (aref b i j)
                       'diagonal-up)))
              ((>= (aref c (- i 1) j)
                   (aref c i (- j 1)))
               (setf (aref c i j)
                     (aref c (- i 1) j))
               (setf (aref b i j)
                     'up))
              ((< (aref c (- i 1) j)
                  (aref c i (- j 1)))
               (setf (aref c i j)
                     (aref c i (- j 1)))
               (setf (aref b i j)
                     'left)))))
    b))



(defun print-lcs (b-array seq i j
                          &key
                          (del 0)
                          (insert 0)
                          (music-result '()))
  (cond ((or (= i 0)
             (= j 0))
         (when (null music-result)
           (dolist (midi-instance seq)
             (push (list 100 midi-instance) music-result))
           (setq music-result (reverse music-result)))
         (list "deletions:" del "insertions:" insert music-result))
        ((equal (aref b-array i j)
                'diagonal-up)
         (push (list t (nth (- j 1) seq)) music-result)
         (print-lcs b-array seq (- i 1) (- j 1)
                    :del del
                    :insert insert
                    :music-result music-result))
        ((equal (aref b-array i j)
                'diagonal-up-no-duration)
         (push (list 001 (nth (- j 1) seq)) music-result)
         (print-lcs b-array seq (- i 1) (- j 1)
                    :del del
                    :insert insert
                    :music-result music-result))
        ((equal (aref b-array i j)
                'up)
         (print-lcs b-array seq (- i 1) j
                    :del (+ del 1)
                    :insert insert
                    :music-result music-result))
        ((equal (aref b-array i j)
                'left)
         (push (list 100 (nth (- j 1) seq)) music-result)
         (print-lcs b-array seq i (- j 1)
                    :del del
                    :insert (+ insert 1)
                    :music-result music-result))))



(defun lcs-match (filename1 &optional (filename2 nil))
       (let* ((midi-ev1 (lcs-prepare-song filename1))
              (midi-ev2 (lcs-prepare-song filename2))
              (len1 (length midi-ev1))
              (len2 (length midi-ev2)))
         (print-lcs (lcs-length midi-ev1 midi-ev2)
                    midi-ev2
                    len1
                    len2)))
```

9.4.3 Utility Functions

```
(defun percent-of-success (list-obj)
  (let ((i 0)
        (len (length list-obj))
        (percent 0))
    (dolist (notes-instances list-obj)
      (case (first notes-instances)
        ((t) (incf i 2))
        (1 (incf i 1.5))
        (10 (incf i 1.5))
        (11 (incf i .5))
        (100 (incf i 1))
        (101 (incf i .5))
        (110 (incf i .5))
        (111 (incf i 0))))
    (setq percent (float (* (/ i (* 2 len))
                     100)))
    percent))



(defun lcs-percent-of-success (list-objects)
  (let* ((i 0)
        (list-obj (fifth list-objects))
        (len (length list-obj))
        (percent 0))
    (dolist (notes-instances list-obj)
      (case (first notes-instances)
        ((t) (incf i 2))
        (001 (incf i 1))
        (100 (incf i 0))))
    (setq percent (float (* (/ i (* 2 len))
                     100)))
    percent))



(defun split-midi-events-to-left-right-hand (object)
  (cond ((null object) nil)
        (t (list
            (new seq :name (string-append
                             (object-name object) "-right-hand")
```

```
                 :subobjects
                 (subobjects object :test
                                  (lambda(x) (>= (sv x :keynum) 60))))
            (new seq :name (string-append
                            (object-name object) "-left-hand")
                 :subobjects
                 (subobjects object :test
                                  (lambda(x) (< (sv x :keynum) 60))))))))



(defun prepare-midi-for-notation (&optional (filename nil))
  (split-midi-events-to-left-right-hand
   (or (import-song filename)
       cl-user::*myseq*)))




(defun compare-seperated-hands (hand1 hand2)
  (match-music
   (make-instance-single-complex-note
    (midi-events-organized-to-indicate-chords hand1))
   (make-instance-single-complex-note
    (midi-events-organized-to-indicate-chords hand2))))




(defun draw-notation (clef meter-scala midi-instances1 midi-instances2)
  (cm::cmn-eval
   (append
    '(staff) '(bar) (list clef) (list meter-scala)
    (show-notation-after-compare
     (compare-seperated-hands midi-instances1 midi-instances2)))))




(defun show-notation-after-compare (list-obj)
  (cond ((null list-obj)
         (push (add-color-to-note nil nil)
               oi))
        ((null (cdr list-obj))
         (push (add-color-to-note (first (first list-obj))
                                  (second (first list-obj)))
               oi)
         (when (> (get-note-time (second (first list-obj))) 0)
           (push (describe-rests
                  (get-note-time (second (first list-obj))))
                 oi))
         oi)
        (t
         (push (add-color-to-note (first (first list-obj))
                                  (second (first list-obj)))
               oi)
         (push (describe-rests
                (get-rest-value (second (first list-obj))
                                (second (second list-obj))))
               oi)
         (show-notation-after-compare (cdr list-obj)))))


(defmethod add-color-to-note (code (song single-note))
  (case code
    ((t) (note-to-notation song))
    (1 (cmn::note (note-to-notation song) (color '(0 0 1))))
    (10 (cmn::note (note-to-notation song) (color '(0 1 0))))
    (11 (cmn::note (note-to-notation song) (color '(0 1 1))))
    (100 (cmn::note (note-to-notation song) (color '(1 0 0))))
    (101 (cmn::note (note-to-notation song) (color '(1 0 1))))
    (110 (cmn::note (note-to-notation song) (color '(1 1 0))))
    (111 (cmn::note (note-to-notation song) (gray-scale .9)))))


(defmethod add-color-to-note (code (song complex-note))
  (case code
    ((t) (note-to-notation song))
    (1 (cmn::chord (note-to-notation song) (color '(0 0 1))))
    (10 (cmn::chord (note-to-notation song) (color '(0 1 0))))
    (11 (cmn::chord (note-to-notation song) (color '(0 1 1))))
    (100 (cmn::chord (note-to-notation song) (color '(1 0 0))))
    (101 (cmn::chord (note-to-notation song) (color '(1 0 1))))
    (110 (cmn::chord (note-to-notation song) (color '(1 1 0))))
    (111 (cmn::chord (note-to-notation song) (gray-scale .9)))))



(defmethod add-color-to-note (code object)
  (cond ((null (or code
                   object))
         (rest (rq 4)))))



(defmethod note-to-notation ((song single-note))
  (note (cm::cmn-eval (single-note-name song))
        (quarters
         (describe-notes
          (single-note-duration song)))))



(defmethod note-to-notation ((song complex-note))
  (let ((notes-list nil))
    (dolist (note-ev (complex-note-name song))
      (push (cm::cmn-eval note-ev)
            notes-list))
    (chord (cm::cmn-eval
            (apply #'list
                   (append '(notes)
                           notes-list)))
           (quarters
            (describe-notes
             (get-note-duration song))))))



(defmethod note-to-notation (object)
  (cond ((null object)
         (rest (rq 4)))))



(defmethod get-note-time ((song single-note))
  (single-note-time song))
```

```
(defmethod get-note-time ((song complex-note))
  (car (complex-note-time song)))


(defmethod get-note-time (object)
  (cond ((null object)
         0)))


(defmethod get-note-duration ((song single-note))
  (single-note-duration song))


(defmethod get-note-duration ((song complex-note))
  (car (complex-note-duration song)))


(defmethod get-note-duration (object)
  (cond ((null object)
         0)))


(defun get-rest-value (note note1)
  (abs (-
        (+ (get-note-time note)
           (get-note-duration note))
        (get-note-time note1))))


(defun describe-notes (var-duration)
  (let ((tempo (/ *my-tempo* 60)))
    (cond ((< var-duration (* .0234375 tempo))
           '1/64)
          ((<= (* .0234375 tempo) var-duration (* .03125 tempo))
           '1/32)
          ((<= (* .03125 tempo) var-duration (* .046875 tempo))
           '3/64)
          ((<= (* .046875 tempo) var-duration (* .0546875 tempo))
           '7/128)
          ((<= (* .0546875 tempo) var-duration (* .05859375 tempo))
           '15/256)
          ((<= (* .05859375 tempo) var-duration (* .0625 tempo))
           '1/16)
          ((<= (* .0625 tempo) var-duration (* .09375 tempo))
           '3/32)
          ((<= (* .09375 tempo) var-duration (* .109375 tempo))
           '7/64)
          ((<= (* .109375 tempo) var-duration (* .1171875 tempo))
           '15/128)
          ((<= (* .1171875 tempo) var-duration (* .125 tempo))
           '1/8)
          ((<= (* .125 tempo) var-duration (* .1875 tempo))
           '3/16)
          ((<= (* .1875 tempo) var-duration (* .21875 tempo))
           '7/32)
          ((<= (* .21875 tempo) var-duration (* .234375 tempo))
           '15/64)
          ((<= (* .234375 tempo) var-duration (* .25 tempo))
           '1/4)
          ((<= (* .25 tempo) var-duration (* .375 tempo))
           '3/8)
          ((<= (* .375 tempo) var-duration (* .4375 tempo))
           '7/16)
          ((<= (* .4375 tempo) var-duration (* .46875 tempo))
           '15/32)
          ((<= (* .46875 tempo) var-duration (* .5 tempo))
           '1/2)
          ((<= (* .5 tempo) var-duration (* .75 tempo))
           '3/4)
          ((<= (* .75 tempo) var-duration (* .875 tempo))
           '7/8)
          ((<= (* .875 tempo) var-duration (* .9375 tempo))
           '15/16)
          ((<= (* .9375 tempo) var-duration (* 1 tempo))
           '1)
          ((<= (* 1 tempo) var-duration (* 1.5 tempo))
           '3/2)
          ((<= (* 1.5 tempo) var-duration (* 1.75 tempo))
           '7/4)
          ((<= (* 1.75 tempo) var-duration (* 1.875 tempo))
           '15/8)
          ((<= (* 1.875 tempo) var-duration (* 2 tempo))
           '2)
          ((<= (* 2 tempo) var-duration (* 3 tempo))
           '3)
          ((<= (* 3 tempo) var-duration (* 3.5 tempo))
           '7/2)
          ((<= (* 3.5 tempo) var-duration (* 3.75 tempo))
           '15/4)
          ((<= (* 3.75 tempo) var-duration (* 4 tempo))
           '4)
          ((<= (* 4 tempo) var-duration (* 6 tempo))
           '6)
          ((<= (* 6 tempo) var-duration (* 8 tempo))
           '8)
          ((> var-duration (* 8 tempo))
           var-duration))))



(defun describe-rests (rest-scanner)
  (let ((tempo (/ *my-tempo* 60)))
    (cond ((< rest-scanner (* .0234375 tempo))
           '(rest (rq 1/64)))
          ((<= (* .0234375 tempo) rest-scanner (* .03125 tempo))
           '(rest (rq 1/32)))
          ((<= (* .03125 tempo) rest-scanner (* .046875 tempo))
           '(rest (rq 3/64)))
          ((<= (* .046875 tempo) rest-scanner (* .0546875 tempo))
           '(rest (rq 7/128)))
          ((<= (* .0546875 tempo) rest-scanner (* .05859375 tempo))
           '(rest (rq 15/256)))
          ((<= (* .05859375 tempo) rest-scanner (* .0625 tempo))
           '(rest (rq 1/16)))
```

```
((<= (* .0625 tempo) rest-scanner (* .09375 tempo))
 '(rest (rq 3/32)))
((<= (* .09375 tempo) rest-scanner (* .109375 tempo))
 '(rest (rq 7/64)))
((<= (* .109375 tempo) rest-scanner (* .1171875 tempo))
 '(rest (rq 15/128)))
((<= (* .1171875 tempo) rest-scanner (* .125 tempo))
 '(rest (rq 1/8)))
((<= (* .125 tempo) rest-scanner (* .1875 tempo))
 '(rest (rq 3/16)))
((<= (* .1875 tempo) rest-scanner (* .21875 tempo))
 '(rest (rq 7/32)))
((<= (* .21875 tempo) rest-scanner (* .234375 tempo))
 '(rest (rq 15/64)))
((<= (* .234375 tempo) rest-scanner (* .25 tempo))
 '(rest (rq 1/4)))
((<= (* .25 tempo) rest-scanner (* .375 tempo))
 '(rest (rq 3/8)))
((<= (* .375 tempo) rest-scanner (* .4375 tempo))
 '(rest (rq 7/16)))
((<= (* .4375 tempo) rest-scanner (* .46875 tempo))
 '(rest (rq 15/32)))
((<= (* .46875 tempo) rest-scanner (* .5 tempo))
 '(rest (rq 1/2)))
((<= (* .5 tempo) rest-scanner (* .75 tempo))
 '(rest (rq 3/4)))
((<= (* .75 tempo) rest-scanner (* .875 tempo))
 '(rest (rq 7/8)))
((<= (* .875 tempo) rest-scanner (* .9375 tempo))
 '(rest (rq 15/16)))
((<= (* .9375 tempo) rest-scanner (* 1 tempo))
 '(rest (rq 1)))
((<= (* 1 tempo) rest-scanner (* 1.5 tempo))
 '(rest (rq 3/2)))
((<= (* 1.5 tempo) rest-scanner (* 1.75 tempo))
 '(rest (rq 7/4)))
((<= (* 1.75 tempo) rest-scanner (* 1.875 tempo))
 '(rest (rq 15/8)))
((<= (* 1.875 tempo) rest-scanner (* 2 tempo))
 '(rest (rq 2)))
```

```
          ((<= (* 2 tempo) rest-scanner (* 3 tempo))
           '(rest (rq 3)))
          ((<= (* 3 tempo) rest-scanner (* 3.5 tempo))
           '(rest (rq 7/2)))
          ((<= (* 3.5 tempo) rest-scanner (* 3.75 tempo))
           '(rest (rq 15/4)))
          ((<= (* 3.75 tempo) rest-scanner (* 4 tempo))
           '(rest (rq 4)))
          ((<= (* 4 tempo) rest-scanner (* 6 tempo))
           '(rest (rq 6)))
          ((<= (* 6 tempo) rest-scanner (* 8 tempo))
           '(rest (rq 8)))
          ((> rest-scanner (* 8 tempo))
           (rest (rq rest-scanner))))))



(defun find-scala (key mode)
  (case mode
    (0
     (case key
       (0 'c-major)
       (1 'g-major)
       (2 'd-major)
       (3 'a-major)
       (4 'e-major)
       (5 'b-major)
       (6 'fs-major)
       (7 'cs-major)
       (-1 'f-major)
       (-2 'bf-major)
       (-3 'ef-major)
       (-4 'af-major)
       (-5 'df-major)
       (-6 'gf-major)
       (-7 'cf-major)))
     (1
      (case key
        (0 'a-minor)
        (1 'e-minor)
        (2 'b-minor)
        (3 'fs-minor)
        (4 'cs-minor)
        (5 'gs-minor)
        (6 'ds-minor)
        (7 'as-minor)
        (-1 'd-minor)
        (-2 'g-minor)
        (-3 'c-minor)
        (-4 'f-minor)
        (-5 'bf-minor)
        (-6 'ef-minor)
        (-7 'af-minor)))))



(defun midi-info (object)
  (let ((scala nil)
        (meter nil))
    (dolist (info (cm::container-subobjects object))
      (typecase info
        (cm::MIDI-KEY-SIGNATURE (setq scala
                                      (find-scala (sv info cm::key)
                                                  (sv info cm::mode))))
        (cm::MIDI-TIME-SIGNATURE (setq meter
                                       (list 'meter (sv info :numerator)
                                             (sv info :denominator))))
        (cm::MIDI-TEMPO-CHANGE (if (= (sv info cm::time) 0)
                                   (setq *my-tempo* (* (/ 60
                                                          (sv info cm::usecs))
                                                       1000000))))))
    (values scala meter)))



(defun get-midi-info(&optional (filename nil))
  (let ((song (load-song filename :meta cm::false)))
    (multiple-value-bind (sc mt)
        (typecase song
          (null (meter 4 4))
          (list (midi-info (car song)))
          (seq (meter 4 4)))
      (values sc mt))))



(defun final-notation (filename1 &optional (filename2 nil))
  (let* ((meter-scala (get-midi-info filename1))
         (both-hand1 (prepare-midi-for-notation filename1))
         (both-hand2 (prepare-midi-for-notation filename2))
         (right-hand1 (first both-hand1))
         (left-hand1 (second both-hand1))
         (right-hand2 (first both-hand2))
         (left-hand2 (second both-hand2)))
    (cmn (size 30)
         (page-width 45)
         (automatic-page-numbers t)
         (system brace
                 (draw-notation 'treble meter-scala right-hand1 right-hand2)
                 (setf oi nil)
                 (draw-notation 'bass meter-scala left-hand1 left-hand2)
                 (setf oi nil)))))



(defun prepare-hand (hand)
  (make-instance-single-complex-note
   (midi-events-organized-to-indicate-chords hand)))



(defun song-draw-notation (clef meter-scala midi-instances)
  (cm::cmn-eval
   (append
    '(staff) '(bar) (list clef) (list meter-scala)
    (show-notation
     (prepare-hand midi-instances)))))



(defun show-notation (list-obj)
  (cond ((null list-obj)
         (push (note-to-notation nil)
               io))
        ((null (cdr list-obj))
         (push (note-to-notation (first list-obj))
               io)
         (when (> (get-note-time (first list-obj)) 0)
           (push (describe-rests
                  (get-note-time (first list-obj)))
                 io))
         io)
        (t
         (push (note-to-notation (first list-obj))
               io)
         (push (describe-rests
                (get-rest-value (first list-obj)
                                (first (cdr list-obj))))
               io)
         (show-notation (cdr list-obj)))))



(defun song-final-notation (&optional (filename nil))
  (let* ((meter-scala (get-midi-info filename))
         (both-hand (prepare-midi-for-notation filename))
         (right-hand (first both-hand))
         (left-hand (second both-hand)))
    (cmn (size 30)
         (page-width 45)
         (automatic-page-numbers t)
         (system brace
                 (song-draw-notation 'treble meter-scala right-hand)
                 (setf io nil)
                 (song-draw-notation 'bass meter-scala left-hand)
                 (setf io nil)))))



(defun lcs-prepare-song (&optional (filename nil))
  (make-instance-single-complex-note
   (midi-events-organized-to-indicate-chords
    (or (import-song filename)
        cl-user::*myseq*))))
```

```
(defun lcs-draw-notation (clef meter-scala midi-instances)
  (cm::cmn-eval
   (append
    '(staff) '(bar) (list clef) (list meter-scala)
    (show-notation-after-compare midi-instances))))



(defun lcs-final-notation (filename1 &optional (filename2 nil))
  (let* ((meter-scala (get-midi-info filename1))
         (midi-events1 (lcs-prepare-song filename1))
         (midi-events2 (lcs-prepare-song filename2))
         (len1 (length midi-events1))
         (len2 (length midi-events2))
         (del-ins-music (print-lcs
                         (lcs-length midi-events1 midi-events2)
                         midi-events2
                         len1
                         len2)))
    (format t "~%~A Deletions" (second del-ins-music))
    (format t "~%~A Insertions" (fourth del-ins-music))
    (cmn (size 30)
         (page-width 45)
         (automatic-page-numbers t)
         (system brace
                 (lcs-draw-notation 'treble meter-scala (fifth del-ins-music))
                 (setf oi nil)))))
```

9.4.4 GUI Functions

```
(defun music-match (&key (new-process nil))
  (let ((frame (make-application-frame 'music-match :width 700 :height 500)))
    (flet ((run ()
                (run-frame-top-level frame)))
      (if new-process
          (clim-sys:make-process #'run :name "music-match")
      (run)))))



(define-application-frame music-match ()
  ()
  (:panes
   (screen :application
          :display-time nil
          :text-style (make-text-style :sans-serif :roman :normal))
   (interactor :interactor)
   (actions :application
           :display-time nil)
   (doc :pointer-documentation)
   (header (make-pane 'application-pane
                     :background +green-yellow+
                     :display-function 'music-match-title
                     :scroll-bars nil)))
  (:command-table (music-match
                  :inherit-from (menubar-commands
                                 file-commands
                                 midi-commands
                                 sheet-commands
                                 music-match-commands
                                 file-command-table
                                 midi-command-table
                                 sheet-command-table
                                 music-match-command-table)
                  :menu (("File" :menu file-command-table)
                         ("Midi" :menu midi-command-table)
                         ("Sheet" :menu sheet-command-table)
                         ("Music-Match" :menu music-match-command-table))))
  (:layouts
   (default (vertically ()
             (1/10 header)
             (8/10 screen)
             (2/10 (spacing (:thick-lines 5 :thickness 2)
                     interactor))
             (1/10 (spacing (:thick-lines 5)
                      actions))
              (1/10 doc))))
  #+nil
  (:top-level (menutest-frame-top-level)))



(defun music-match-title (frame pane)
  (declare (ignore frame))
  (let ((out pane))
    (with-drawing-options (out :text-size 20 :text-face :bold :ink +white+)
      (format out "                          Music Match~%"))
    (with-drawing-options (out :text-size :normal :ink +white+)
      (format out "                        Author: Zervopoulos Thanassis~%"))))



(defun output-pane (pane)
  (get-frame-pane *application-frame* pane))



(define-command (com-record :name "Start Recording"
                            :command-table midi-commands)
    ((name 'string :prompt "Name your song:" :default (setq *seq-name* "my_song")))
  (setf *seq-name* name)
  (setf *myseq* nil)
  (setf *pm*
        (cm::portmidi-open :latency 0
                           :input (get-my-input-device-id)
                           :output (get-my-output-device-id)))
  (setf *myseq* (cm::new cm::seq))
  (start-recording)
  (format *standard-output* "You pressed the start recording button.~%")
  (finish-output *standard-output*))



(defun start-recording (&key (new-process nil))
  (flet ((run ()
              (cm::portmidi-record! *myseq*)))
    (if new-process
        (clim-sys:make-process #'run :name "start-recording")
      (run))))



(defun get-my-input-device-id ()
  (let ((my-input nil))
    (dotimes (i (pm:countdevices))
      (if (and (pm::deviceinfo.input
                 (pm::pm-get-device-info i))
                (not (eql
                      (pm::pm-get-default-input-device-id)
                      i)))
          (setq my-input i)))
    my-input))



(defun get-my-output-device-id ()
  (let ((my-output nil))
    (dotimes (i (pm:countdevices))
      (if (and (not (pm::deviceinfo.input
                      (pm::pm-get-device-info i)))
                (not (eql
                      (pm::pm-get-default-output-device-id)
                      i)))
          (setq my-output i)))
    my-output))



(define-command (com-stop :name "Stop Recording"
                          :command-table midi-commands)
    ()
  (cm::portmidi-record! nil)
  (cm::portmidi-close *pm*)
  (cm::events (cmn::sort-by-time-if-needed *myseq*)
              (concatenate 'string
                           *seq-name*
                           ".mid")
              :play nil)
  (format *standard-output* "You pressed the stop recording button.~%")
  (finish-output *standard-output*))



(defun sort-by-time-if-needed (object)
  (let ((seq-track (cm::new cm::seq)))
    (dolist (midev (cm::container-subobjects object))
      (cm::insert-object midev seq-track))
    seq-track))



(define-command (com-play :name "Play"
                          :command-table file-commands)
    ((song-name 'string :prompt "Track"))
  (cm::play (concatenate 'string song-name ".mid"))
  (format *standard-output* "You pressed the play button.~%")
  (finish-output *standard-output*))



(define-command (com-open-sheet :name "OpenNotation"
                                :command-table file-commands)
    ((name 'string :prompt "Sheet Name"))
  (load (concatenate 'string name ".cmn"))
  (format *standard-output* "You pressed the open notation button.~%")
  (finish-output *standard-output*))



(define-command (com-save-sheet :name "SaveNotation"
                                :command-table file-commands)
    ((name 'string :prompt "Sheet Name"))
  (cmn::cmn-store cmn::*cmn-score* (concatenate 'string name ".cmn"))
  (format *standard-output* "You pressed the save notation button.~%")
  (finish-output *standard-output*))



(define-command (com-sound :name "Change Sound"
                           :command-table file-commands)
    ((sd 'integer :prompt "Sound Card" :default 1)
     (v 'integer :prompt "Volume" :default 400))
  (change-sound-volume sd v)
  (format *standard-output* "You pressed the change sound button.~%")
  (finish-output *standard-output*))



(defun change-sound-volume (soundcard volume)
  (setf cm::*midi-player*
        (concatenate 'string
                     "timidity -ig -"
                     (if (eql soundcard 1)
                         "A"
                       "Os")
                     " "
                     (cm::number->string volume))))



(define-command (com-reset :name "Reset tt & dt"
                           :command-table file-commands)
    ()
  (setf cmn::*time-tolerance* .1)
  (setf cmn::*duration-tolerance* .2)
  (format *standard-output* "You pressed the reset tt & dt.~%")
  (finish-output *standard-output*))



(define-command (com-quit :name "Quit"
                          :command-table file-commands)
    ()
  (frame-exit *application-frame*))



(define-command (com-clear :name "Clear"
                           :command-table file-commands)
    ()
  (window-clear (output-pane 'screen)))
```

```
(define-command (com-clear-history :name "Clear History"
                                   :command-table file-commands)
    ()
  (window-clear (output-pane 'actions)))



(define-command (com-show :name "Show Sheet After Compare"
                          :command-table sheet-commands)
    ((track1 'string :prompt "Original Song")
     (track2 'string :prompt "Your Song"))
  (format (output-pane 'screen) "~S~%"
    (if (equal track2 *seq-name*)
      (cmn::final-notation track1)
    (cmn::final-notation track1 track2)))
  (format *standard-output* "You pressed the show sheet button.~%")
  (finish-output *standard-output*))



(define-command (com-show-notation :name "Sheet Notation"
                                   :command-table sheet-commands)
    ((track 'string :prompt "Original Song"))
  (if (equal track *seq-name*)
      (cmn::song-final-notation)
    (cmn::song-final-notation track))
  (format *standard-output* "You pressed the sheet notation button.~%")
  (finish-output *standard-output*))



(define-command (com-close :name "Close Sheet"
                           :command-table sheet-commands)
    ()
  (cmn::x-close)
  (format *standard-output* "You pressed the close sheet button.~%")
  (finish-output *standard-output*))



(define-command (com-discard :name "Discard Track"
                            :command-table midi-commands)
    ()
  (setf *myseq* nil)
  (format *standard-output* "You pressed the discard track button.~%")
  (finish-output *standard-output*))



(define-command (com-transpose :name "Transporto"
                               :command-table midi-commands)
    ((semitone 'integer :prompt "No of Semitones"))
  (setf cmn::*transpose-value* semitone)
  (format *standard-output* "You pressed the transporto button.~%")
  (finish-output *standard-output*))



(define-command (com-music-match :name "Music Match"
                                 :command-table music-match-commands)
    ((tt 'real :prompt "Time tolerance:"
              :default (setq cmn::*time-tolerance* .1))
     (dt 'real :prompt "Duration tolerance:"
              :default (setq cmn::*duration-tolerance* .2))
     (track1 'string :prompt "Original Song:")
     (track2 'string :prompt "Your Song:")
     (func-time 'symbol :prompt "Time Report"))
  (let ((result nil))
    (setq cmn::*time-tolerance* tt)
    (setq cmn::*duration-tolerance* dt)
    (case func-time
      (no (setq result (if (equal track2 *seq-name*)
                             (cmn::compare-music track1)
                           (cmn::compare-music track1 track2)))
       (format *standard-output* "You pressed the music match button.~%")
       (format *standard-output* "With values TT: ~A and DT: ~A .~%"tt dt)
       (format *standard-output* "Song: *~A* Vs. Song: *~A*.
                                  Result:~A% of success.~%"
              track1 track2 (cmn::percent-of-success result)))
      (yes (if (equal track2 *seq-name*)
                (time (cmn::compare-music track1))
              (time (cmn::compare-music track1 track2)))))
    (finish-output *standard-output*)))



(define-command (com-tempo-change :name "TempoChange"
                                  :command-table midi-commands)
    ((tempo 'integer :prompt "New Tempo"))
  (setf cmn::*my-tempo* tempo)
  (format *standard-output* "You pressed the tempo change button.~%")
  (finish-output *standard-output*))



(define-command (com-lcs-music-match :name "LcsMatch"
                                     :command-table music-match-commands)
    ((dt 'real :prompt "Duration tolerance:" :default (setq cmn::*duration-tolerance* .2))
     (track1 'string :prompt "Original Song:")
     (track2 'string :prompt "Your Song:")
     (func-time 'symbol :prompt "Time Report"))
  (let ((result nil))
    (setq cmn::*duration-tolerance* dt)
    (case func-time
      (no (setq result (if (equal track2 *seq-name*)
                           (cmn::lcs-match track1)
                         (cmn::lcs-match track1 track2)))
          (format *standard-output* "You pressed the lcs match button.~%")
          (format *standard-output* "With DT: ~A .~%"dt)
          (format *standard-output* "Song: *~A* Vs. Song: *~A*. Result:~A% of success.~%"
                  track1 track2 (cmn::lcs-percent-of-success result)))
      (yes (if (equal track2 *seq-name*)
               (time (cmn::lcs-match track1))
             (time (cmn::lcs-match track1 track2)))))
    (finish-output *standard-output*)))



(define-command (com-lcs-show :name "Show Sheet After Lcs Compare"
                          :command-table sheet-commands)
    ((track1 'string :prompt "Original Song")
     (track2 'string :prompt "Your Song"))
  (format (output-pane 'screen) "~S~%" (if (equal track2 *seq-name*)
                                                (cmn::lcs-final-notation track1)
                                              (cmn::lcs-final-notation track1 track2))
  (format *standard-output* "You pressed the show sheet button.~%")
  (finish-output *standard-output*))
```

9.5 Algorithm Complexity Analysis

Before analysing the complexity of the MM algorithm, it is important that we present the functions which take part in the MM implementation and are considered to have time stability. Those functions are depicted below:

```
let
if
cond
case
eq
equal
<
<=
>
=
>=
push
pop
first
second
when
unless
and
or
not
null
oddp
let*
flet
setf
```

```
setq
+
-
/
abs
progn
with-open-io
make-instance
sv
lambda
integerp
consp
list
format
typep
error
```

It is necessary that we examine which functions influence the amount of complexity, such as recursive functions or loops so as to be able to find the complexity of each function. For example dolist, dopairs, dotimes are instances of those kind of functions.

The time complexity of the function **load-song** is:

```
(defun load-song (filename &key (meta cm::true) (key-note ':note))
  (cond ((null filename) nil)
        (t (cm::import-events
            (namestring
             (make-pathname :directory '(:absolute "home" "pontikis" "Lisp")
                            :name filename
                            :type (or "mid" "midi")))
            :meta-exclude meta :keynum-format key-note))))
```

This function takes as an argument a file, so it is more important for our research to find the time complexity of the function **import-events**.

```
(defmethod import-events ((io midi-file)
                          &key (tracks t) seq meta-exclude
                          channel-exclude (time-format ':beats) tempo
                          exclude-tracks (keynum-format ':keynum)
                          (note-off-stack t))
  (let ((results '())
        (notefn nil)
        (result nil)
        (class nil)
        (root nil)
        (tempo-map nil)
        (num-tracks nil))
    (if exclude-tracks
        (if tracks
            (if (eq tracks t)
                (setf tracks nil)
                (error
                 ":tracks and :exclude-tracks are exclusive keywords."))
            (cond
              ((integerp exclude-tracks)
               (setf exclude-tracks (list exclude-tracks)))
              ((consp exclude-tracks))
              (t
               (error
                ":exclude-tracks value '~s' not number, list or ~s."
                exclude-tracks
                t))))
        (cond
          ((eq tracks t))
          ((consp tracks))
          ((integerp tracks)
           (setf tracks (list tracks)))
          (t
           (error ":tracks value '~s' not number, list or ~s."
                  tracks
                  t))))
    (case time-format
      ((:ticks) t)
      ((:beats) t)
      (t
       (error ":time-format value ~s is not :beats or :ticks."
              time-format)))
```

```
(case keynum-format
  ((:keynum nil) t)
  ((:note) (setf notefn #'note))
  ((:hertz) (setf notefn #'hertz))
  (t
   (error
    ":keynum-format value '~s' not :keynum, :note or :hertz."
    keynum-format)))
(unless (member channel-exclude '(t nil))
  (unless (consp channel-exclude)
    (setf channel-exclude (list channel-exclude)))
  (dolist (e channel-exclude)
    (unless
        (and (integerp e)
             (<= +ml-note-off-opcode+ e +ml-pitch-bend-opcode+))
      (error
       ":channel-exclude value '~s' not a channel message opcode."
       e))))
(unless (member meta-exclude '(t nil))
  (unless (consp meta-exclude)
    (setf meta-exclude (list meta-exclude)))
  (dolist (e meta-exclude)
    (unless
        (and (integerp e)
             (or
              (<= +ml-file-text-event-opcode+
                  e
                  +ml-file-cue-point-opcode+)
              (<= +ml-file-tempo-change-opcode+
                  e
                  +ml-file-sequencer-event-opcode+)))
      (error
       ":meta-exclude value '~s' not a meta message opcode."
       e))))
(with-open-io (file io :input)
 (cond
   ((= 1 (midi-file-format file))
    (setf num-tracks (midi-file-tracks file))
    (if tracks
        (if (eq tracks t)
            (setf tracks
                    (loop for i below num-tracks collect i)))
        (setf tracks
                (loop for i below num-tracks
                      unless (member i exclude-tracks)
                        collect i))))
   (t
    (setf num-tracks 1)
    (if (eq tracks t) (setf tracks (list 0)))))
 (cond
   ((typep seq <seq>))
   ((or (not seq) (find-class seq))
    (setf class (or seq <seq>))
    (setf root
            (format nil
                    "~a-track"
                    (filename-name (file-output-filename io))))
    (setf seq nil))
   (t
    (setf root (format nil "~s" seq))
    (setf class <seq>)
    (setf seq nil)))
 (when (and (eq time-format ':beats) (not tempo))
   (setf tempo-map (parse-tempo-map file)))
 (dolist (track tracks)
   (when (consp track)
     (setf channel-exclude
             (if (numberp (second track))
                 (list (second track))
                 (second track)))
     (setf meta-exclude
             (if (numberp (third track))
                 (list (third track))
                 (third track)))
     (setf track (first track)))
   (cond
     ((<= 0 track (- num-tracks 1))
      (unless seq
        (setf seq
                (make-instance class
                                 :name (format nil
                                              "~a-~s"
                                              root
                                              track))))
         (setf result
               (midi-file-import-track file track seq notefn
                note-off-stack channel-exclude meta-exclude))
         (push result results)
         (setf seq nil))
        (t
         (error "track '~s' out of range. Maximum track is ~s."
                track
                (- num-tracks 1)))))
    (setf results (reverse results))
    (cond
      ((and (eq time-format ':beats)
            (not tempo)
            (consp (cdr tempo-map)))
       (let ((div (midi-file-divisions file)))
         (dolist (tr results) (apply-tempo-map div tempo-map tr))))
      ((not (eq time-format :ticks))
       (let ((div (midi-file-divisions file))
             (scaler
              (if tempo
                  (/ 60.0 tempo)
                  (if (consp tempo-map)
                      (* (tc-scaler (car tempo-map)) 1.0)
                      0.5))))
         (dolist (tr results) (apply-tempo-scaler div scaler tr)))))
    (if (null results)
        nil
        (if (null (cdr results)) (car results) results)))))
```

The function **import-events** imports musical events in a file according to the type of the file specified. If the file is a ".mid" file, then **import-events** returns a seq containing midi objects from one or more tracks of the file. As the function uses a dolist function to import each midi event, the time complexity is $O(n)$. Thus, it could be considered that **load-song** is also $O(n)$.

The time complexity of the function **remove-non-midi-object** is:

```
(defun remove-non-midi-object (object)
  (let ((midi-ev nil))
    (dolist (midievent (cm::container-subobjects object))
      (typecase midievent
        (cm::MIDI (push midievent midi-ev))))
    (cm::events (reverse midi-ev) (new cm::seq :name (object-name object)))))
```

This function uses a dolist function to check each midi event, so, that gives a $O(n)$. Also, the reverse function which has linear complexity $O(n)$ is used. Moreover, the function **events** is applied from the package cm which is:

```
(defun events (object &rest args)
  (if (scheduling?)
      (progn
        (format t
                "Can't schedule events in non-real time while RTS is running.")
        nil)
      (let* ((to
              (if
               (and (consp args)
                    (or (stringp (car args))
                        (eq (car args) nil)
                        (typep (car args) <object>)))
               (pop args)
               (current-output-stream)))
             (ahead
              (if
               (and (consp args)
                    (or (consp (car args)) (numberp (car args))))
               (pop args)
               0))
             (err? ':error))
        (when (oddp (length args))
          (error "Uneven initialization list: ~s." args))
        (unwind-protect
            (progn
              (flet ((getobj (x)
                       (if (not (null x))
                           (if (or (stringp x) (and x (symbolp x)))
                               (find-object x)
                               x)
                           (error "Not an object specification: ~s."
                                  x))))
                (if (not to)
                    (setf *out* nil)
                    (progn
                      (setf *out*
                              (open-io (apply #'init-io to args)
                               ':output))
                      (initialize-io *out*)))
                (schedule-events *out*
                 (if (consp object)
                     (mapcar #'getobj object)
                     (getobj object))
                 ahead)
                (setf err? nil)))
          (when *out*
            (deinitialize-io *out*)
            (close-io *out* err?)))
        (if (or err? (not *out*))
            nil
            (if (typep *out* <event-file>)
                (let ((path (file-output-filename *out*))
                      (args (event-stream-args *out*))
                      (hook (io-class-output-hook (class-of *out*))))
                  (when hook
                    (apply hook path args))
                  path)
                *out*)))))
```

The function **events** records events ranging from objects to destination. Objects can be either a single object (i.e. an event, seq or process) or a list of objects. Destination can be a file, port, seq or plotter window.[] As the function uses a mapcar function to write each midi event into a "seq" object, the time complexity is $O(n)$. Thus, the function **remove-non-midi-object** has $O(n)$ (from dolist) + $O(n)$ (from reverse) + $O(n)$ (from events) that gives $O(3n)$,which means a linear complexity $O(n)$.

The time complexity of the function **transform-midi-to-seq** is:

```
(defun transform-midi-to-seq (object)
  (remove-non-midi-object
   (dolist (track object)
     (if (cm::container-subobjects track)
         (return track)))))
```

This function takes as an argument a list of “seq” objects of a “.mid” file type1. However, two hypothetical cases could appear regarding the “.mid” file type1. The worst case would be, if the list had displayed n elements, so the complexity would become $O(n)$ (from remove-non-midi-object) + $O(n)$ (from dolist) that would give $O(2n)$. The best case would be, if the list contained one element $n = 1$, the complexity would become $O(n) + O(1)$ that would give $O(n)$. The best case happens to be the average case, because only one “seq” object contains the melody.

The time complexity of the function **import-song** is:

```
(defun import-song (filename)
  (let ((song (load-song filename :key-note ':keynum)))
    (if (atom song)
        song
      (transform-midi-to-seq song))))
```

This function takes as an argument a file, so the internal functions will specify the time complexity. From the functions above, **load-song** has $O(n)$ and **transform-midi-to-seq** has also $O(n)$. The worst case would be if both functions were to be called, so the complexity would become $O(n)$ (from load-song) + $O(n)$ (from transform-midi-to-seq) that would give $O(2n)$. The best case would be if the function **transform-midi-to-seq** were not to be called, so the complexity would become $O(n)$. The **import-song** is going to be called exactly twice for the needs of the MM algorithm. The first time, it will satisfy the best case and the second time will satisfy the worst case.

The time complexity of the function **map-objects** from package cm is:

```
(defun map-objects
       (fn objs
        &key (start 0) end (step 1) (width 1) at test class key slot
        slot! arg2
        &aux doat indx this)
  (if (not (listp objs)) (setf objs (container-subobjects objs)))
  (if (and slot slot!)
      (error ":slot and slot! are exclusive keywords."))
  (when (or slot slot!)
    (if key
        (error ":slot[!] and :key are exclusive keywords.")
        (setf key
                (if slot!
                    (lambda (x) (slot-value x slot!))
                    (lambda (x) (slot-value x slot))))))
  (when at
    (unless (and (eq start 0) (not end) (eq step 1))
      (error ":at excludes use of :start step and :end"))
    (unless (apply #'< at)
      (error ":at values not in increasing order."))
    (setf doat t)
    (setf start (pop at)))
  (setf indx start)
  (do ((tail (nthcdr start objs) (nthcdr step tail))
       (data nil)
       (done nil)
       (func (cond
               ((not arg2)
                fn)
               ((eq arg2 ':object)
                (lambda (x) (funcall fn x this)))
               ((eq arg2 ':position)
                (lambda (x) (funcall fn x indx)))
               (t
                (error ":arg2 not :object or :position")))))
      ((or (null tail) done (and end (not (< indx end)))) (values))
    (cond
      ((> width 1)
       (setf this
               (loop for i below width
                     for x = (nthcdr i tail)
                     until (null x)
                     collect (car x)))
       (when
           (or (not class)
               (loop for x in this always (typep x class)))
         (if key (setf data (mapcar key this)) (setf data this))
         (if
          (or (not test)
              (loop for x in data always (funcall test x)))
          (if slot!
              (loop for x in (funcall func data)
                    for y in this
                    do (setf (slot-value y slot!) x))
              (funcall func data)))))
      (t
       (setf this (car tail))
       (when (or (not class) (typep this class))
         (if key (setf data (funcall key this)) (setf data this))
         (if (or (not test) (funcall test data))
             (if slot!
                 (setf (slot-value this slot!) (funcall func data))
                 (funcall func data))))))
    (when doat
      (if (null at)
          (progn
            (setf done t)
            (setf step 0))
          (progn
            (setf step (- (pop at) indx)))))
    (setf indx (+ indx step))))
```

This function maps a function of one argument over a list or seq of objects according to the specified keyword arguments. As it makes use of the do function the complexity becomes $O(n)$.

The time complexity of the function **subobjects** from package cm is:

```
(defun subobjects (object &rest args)
  (if (null args)
      (container-subobjects object)
      (let* ((head (list nil)) (tail head))
        (if (member ':slot! args)
            (error "Illegal keyword argument :slot!"))
        (apply #'map-objects
               (lambda (a x)
                 a
                 (rplacd tail (list x))
                 (setf tail (cdr tail)))
               object
               :arg2 ':object
               args)
        (cdr head))))
```

This function returns a list of subobjects from a list or seq of objects. It makes use of the **map-object** above, so the complexity becomes $O(n)$ as well.

The complexity of the macros **first-subobject**, **substitute-times** is:

```
(defmacro first-subobject (obj start end)
  `(first (subobjects ,obj :start ,start :end ,end)))


(defmacro substitute-times (obj start1 end1 start2 end2)
  `(cond ((null (first-subobject ,obj ,start2 ,end2))
          (sv (first-subobject ,obj ,start1 ,end1) :time))
         (t
          (abs (- (sv (first-subobject ,obj ,start1 ,end1) :time)
                  (sv (first-subobject ,obj ,start2 ,end2) :time))))))
```

Those macros utilize the function **subobjects** above, so the complexity for them becomes $O(n)$.

The time complexity of the function **midi-events-organized-to-indicate-chords** is:

```
(let ((midi-events)
      (midi-complex))
  (defun midi-events-organized-to-indicate-chords (object)
   (let ((len (length (cm::container-subobjects object))))
    (cond ((null object)
           nil)
          (t
           (setq midi-events nil)
           (setq midi-complex nil)
           (dotimes (i len)
             (cond ((equal i (- len 1))    ;;1st case
                    (when (<= 0 (substitute-times object i (+ i 1)
                                                           (- i 1) i)
                              *time-chords*)
                      (push (first-subobject object i (+ i 1))
                            midi-complex)
                      (push midi-complex
                            midi-events))
                    (unless (<= (substitute-times object i (+ i 1)
                                                           (- i 1) i)
                                *time-chords*)
                      (push (first-subobject object i (+ i 1))
                            midi-events)))
                   ((<= 0 (substitute-times object i (+ i 1)     ;;2nd case
                                            (+ i 1) (+ i 2))
                        *time-chords*)
                    (push (first-subobject object i (+ i 1))
                          midi-complex))
                   ((>= (substitute-times object i (+ i 1)       ;;3rd case
                                          (+ i 1) (+ i 2))
                        *time-chords*)
                    (when (equal (- i 1) -1)
                      (push (first-subobject object i (+ i 1))
                            midi-events)
                      (setf i 1))
                    (when (<= 0 (substitute-times object i (+ i 1)
                                                           (- i 1) i)
                              *time-chords*)
                      (push (first-subobject object i (+ i 1))
                            midi-complex)
                      (push midi-complex
                            midi-events)
                      (setq midi-complex nil))
                    (when (and (equal i 1)
                               (<= 0 (substitute-times object i (+ i 1)
                                                        (+ i 1) (+ i 2))
                                   *time-chords*))
                      (push (first-subobject object i (+ i 1))
                            midi-complex))
                    (unless (or (equal (- i 1) -1)
                                (<= 0 (substitute-times object i (+ i 1)
                                                               (- i 1) i)
                                    *time-chords*)
                                (<= 0 (substitute-times object i (+ i 1)
                                                         (+ i 1) (+ i 2))
                                    *time-chords*))
                      (push (first-subobject object i (+ i 1))
                            midi-events)))))
           midi-events)))))
```

The function **container-subobjects** from package cm, has access to the memory to get the slot subobjects from the instance object of the class “seq” of which the class “container” is a subclass. This access needs a constant time, so the complexity becomes $O(1)$. The function length along with the container-subobjects, returns the amount of that instance object, so if n is the number of all the subobjects, then, complexity will become $O(n)$. As the function uses the dotimes function along with the macros above, **first-subobjects** - **substitute-times** the complexity becomes $O(n)$ (from length) + $O(n)$ (from dotimes) *$O(n)$ (from macros), which means that according to the equation $O(n) + O(n^2) = O(n^2)$, the complexity becomes a non linear one.

The time complexity of the function **note** from package cm is:

```
(defun note (freq &rest args)
  (let ((oper nil)
        (hz? nil)
        (scale nil)
        (test? nil)
        (tuning nil)
        (mode nil)
        (acci nil))
    (unless freq
      (error
       "~s is not a note name, key number, Hertz value or list."
       false))
    (when
        (and (consp args)
             (member (first args) %hertz)
             (oddp (length args)))
      (setf args (list* ':hz true (rest args))))
    (dopairs (sym val args)
     (case sym
       ((:hz) (setf hz? val))
       ((:in :in? :through)
        (when oper
          (error
           "Found more than one of :in, :in? or :through in ~s."
           (cons 'note args)))
        (if (eq sym ':in?) (setf test? t))
        (setf oper sym)
        (setf scale val))
       ((:accidental) (setf acci val))
       (t
        (error "~s not a valid keyword: :hz :in :in? :through"
               sym))))
    (unless scale
      (setf scale *scale*))
    (if (typep scale <mode>)
        (progn
          (if (eq oper ':in) (setf oper ':from))
          (setf tuning (mode-tuning scale))
          (setf mode scale))
        (progn
          (if (eq oper ':through)
              (error "Not a mode: ~s ~s." oper scale))
          (setf tuning scale)))
    (when (and (eq oper ':from) (or hz? (and freq (symbolp freq))))
      (error ":from conversion: ~s is not a key number in ~s."
              freq
              mode))
    (cond
      ((consp freq)
       (with-default-octave tuning
        (loop for f in freq collect (apply #'note f args))))
      ((and freq (symbolp freq))
       (let ((ref (tuning-note->note tuning freq acci (not test?))))
         (if (not mode)
             ref
             (if (not ref)
                 nil
                 (let* ((key (tuning-note->keynum tuning ref t))
                        (out
                         (mode->tuning mode (tuning->mode mode key t)
                          ':note)))
                   (if test? (if (eq out ref) ref nil) out))))))
      (t
       (let ((keyn (if hz? (tuning-hertz->keynum tuning freq) freq)))
         (if (not mode)
             (tuning-keynum->note tuning keyn acci (not test?))
             (let* ((in
                     (if (eq oper ':from)
                         keyn
                         (tuning->mode mode keyn (not test?))))
                    (out (and in (mode->tuning mode in ':note))))
               (if test?
                   out
                   (or out
                       (error "No note for keynum ~s in mode ~s."
                              keyn
                              mode))))))))))
```

This function returns the note name of a note name, a key number, a Hertz value or a list of the same. As it uses the dopairs function, the complexity becomes $O(n)$.

The time complexity of the function **keynum** from package cm is:

```
(defun keynum (freq &rest args)
  (let ((hz? nil)
        (scale nil)
        (mode nil)
        (tuning nil)
        (oper nil)
        (test? nil))
    (unless freq
      (error
       "~s is not a note name, key number, Hertz value or list."
       false))
    (when
        (and (consp args)
             (member (first args) %hertz)
             (oddp (length args)))
      (setf args (list* ':hz true (rest args))))
    (dopairs (sym val args)
     (case sym
       ((:hz) (setf hz? val))
       ((:in :in? :through :to :from)
        (when oper
          (error "More than one of: :in, :in? or :through in ~s."
                 (cons 'note args)))
        (if (eq sym ':in?) (setf test? t))
        (setf oper sym)
        (setf scale val))
       (t
        (error "~s not a valid keyword: :hz :in :in? :through"
               sym))))
    (unless scale
      (setf scale *scale*))
    (if (typep scale <mode>)
        (progn
          (setf tuning (mode-tuning scale))
          (setf mode scale))
        (progn
          (setf tuning scale)))
    (if (consp freq)
        (with-default-octave tuning
         (loop for f in freq collect (apply #'keynum f args)))
        (let ((key
               (if hz?
                   (tuning-hertz->keynum tuning freq)
                   (if (and freq (symbolp freq))
                       (tuning-note->keynum tuning freq (not test?))
                       freq))))
          (if mode
              (let ((in
                     (if (eq oper ':from)
                         (if (integerp key)
                             key
                             (error
                              "keynum:  :from value ~s not integer."
                              key))
                         (tuning->mode mode key
                          (not (eq oper ':in?))))))
                (if (not in)
                    nil
                    (if (eq oper ':to)
                        in
                        (mode->tuning mode in ':keynum))))
              (if (eq oper ':from)
                  (if (integerp key)
                      (tuning-hertz->keynum *chromatic-scale*
                       (tuning-keynum->hertz scale key))
                      (error "keynum:  :from value ~s not integer."
                             key))
                  (if (eq oper ':through)
                      (values (round key))
                      key)))))))
```

This function returns the key number of a note name, a key number, a Hertz value or a list of the same. As it utilizes the function dopairs, the complexity becomes $O(n)$.

The time complexity of the function **transpose** from package cm is:

```
(defun transpose (data &key (from-key c-major) (to-key c-major) (octave 0))
  (let* ((from-sig (map-signature from-key))
 (to-sig (map-signature to-key))
 (cclass-inc-1 (- (key-base to-key) (key-base from-key)))
 (cclass-inc (+ (if (minusp cclass-inc-1)
                          (+ cclass-inc-1 7) cclass-inc-1) (* octave 7)))
 (pclass-inc-1 (- (key-pitch to-key) (key-pitch from-key)))
 (pclass-inc (+ (if (minusp pclass-inc-1)
                          (+ pclass-inc-1 12) pclass-inc-1) (* octave 12)))
 (saved-from-sig (map-signature from-key))
 (saved-to-sig (map-signature to-key)))

    (flet ((transpose-obj (obj)
     (if (audible-p obj)
 (if (note-p obj)
     (update-note obj cclass-inc pclass-inc from-sig to-sig)
   (loop for nobj in (chord-data obj) do
     (update-note nobj cclass-inc pclass-inc from-sig to-sig)))
       (if (key-p obj)
   (update-key obj from-sig to-sig))))
   (reset-original-sigs ()
     (loop for i from 0 below 6 do
       (setf (aref from-sig i) (aref saved-from-sig i))
       (setf (aref to-sig i) (aref saved-to-sig i)))))

      (if (not (eq (key-mode from-key) (key-mode to-key)))
  (warn "changing modes is not fair --
              results are unlikely to be convincing..."))
      (if (score-p data)
  (loop for sys in (systems data) do
    (loop for stf in (staves sys) do
      (reset-original-sigs)
      (loop for obj in (staff-data stf) do
(transpose-obj obj))))
(if (system-p data)
    (loop for stf in (staves data) do
      (reset-original-sigs)
      (loop for obj in (staff-data stf) do
(transpose-obj obj)))
  (if (staff-p data)
      (loop for obj in (staff-data data) do
(transpose-obj obj))
```

```
    (loop for obj in data do
      (transpose-obj obj)))))

      data)))
```

This function returns the transposed value of a note name, a key number or a list of the same, in scale. As it makes use of the loop function, the complexity becomes $O(n)$.

The time complexity of the function **make-instance-single-complex-note** is:

```
(defun make-instance-single-complex-note (midi-events-list)
  (mapcar #'(lambda (midis)
             (if (atom midis)
              (make-instance 'single-note
                            :note-name (cm::note
                                         (cm::transpose
                                          (sv midis :keynum)
                                           *transpose-value*))
                            :time (sv midis :time)
                            :duration (sv midis cm::duration))
              (make-instance 'complex-note
                            :note-name (cm::note
                                         (sort
                                          (keynum
                                            (mapcar
                                              #'(lambda (x)
                                                  (cm::transpose
                                                    (sv x :keynum)
                                                      *transpose-value*))
                                                midis))
                                           #'<))
                             :time (mapcar #'(lambda (x)
                                                (sv x :time))
                                            midis)
                             :duration (mapcar #'(lambda (x)
                                                    (sv x cm::duration))
                                                midis))))
```

```
      midi-events-list))
```

This function makes use of the mapcar function, so the complexity becomes $O(n)$. Also, when a single-note instance is to be created the complexity of the **note**,**keynum**,**transpose** becomes $O(1)$ as there is only one note. When a complex-note instance is to be formed, the complexity for those functions is $O(n)$. But, it is known that a chord consists of 2 to 5 notes, which is a constant number. So, as this is an average case and despite the fact that is $O(n)$, the complexity of the **note**, **keynum**, **transpose** functions, now becomes $O(1)$. Finally the best case would be, if all instances were single-note instances, so the complexity would become $O(n)$. The worst case would be, if all instances were comple-note instances, thus each chord would have an unlimited number of notes m. The complexity becomes $O(nm)$, because of the internal mapcar. In an average case, the complexity becomes $O(n)$ because each chord contains up to 4 notes and the time of the internal mapcar could be considered upper limited.

The time complexity of the function **match-music** is:

```
(defun match-music (oldsong newsong)
  (mapcar #'match-note oldsong newsong))
```

This function makes use of the mapcar function, so the complexity becomes $O(n)$.

The time complexity of the method **match-note** is:

```
(let ((res))
  (defmethod match-note ((oldsong single-note)
                         (newsong single-note)
                         &key (note-only nil))
    (setf res nil)
    (cond (note-only
           (setf res (equal (single-note-name oldsong)
                            (single-note-name newsong))))
          ((and (equal (single-note-name oldsong)
                       (single-note-name newsong))
                (<= 0
                    (abs (- (single-note-time oldsong)
                            (single-note-time newsong)))
                    *time-tolerance*)
                (<= 0
                    (abs (- (single-note-duration oldsong)
                            (single-note-duration newsong)))
                    *duration-tolerance*))
           (setf res (list t newsong)))
          ((and (equal (single-note-name oldsong)
                       (single-note-name newsong))
                (<= 0
                    (abs (- (single-note-time oldsong)
                            (single-note-time newsong)))
                    *time-tolerance*)
                (>
                 (abs (- (single-note-duration oldsong)
                         (single-note-duration newsong)))
                 *duration-tolerance*))
           (setf res (list 001 newsong)))
          ((and (equal (single-note-name oldsong)
                       (single-note-name newsong))
                (>
                 (abs (- (single-note-time oldsong)
                         (single-note-time newsong)))
                 *time-tolerance*)
                (<= 0
                    (abs (- (single-note-duration oldsong)
                            (single-note-duration newsong)))
                    *duration-tolerance*))
           (setf res (list 010 newsong)))
          ((and (equal (single-note-name oldsong)
                       (single-note-name newsong))
                (>
                 (abs (- (single-note-time oldsong)
                         (single-note-time newsong)))
                 *time-tolerance*)
                (>
                 (abs (- (single-note-duration oldsong)
                         (single-note-duration newsong)))
                 *duration-tolerance*))
           (setf res (list 011 newsong)))
```

```
((and (not
       (equal (single-note-name oldsong)
              (single-note-name newsong)))
      (<= 0
          (abs (- (single-note-time oldsong)
                  (single-note-time newsong)))
          *time-tolerance*)
      (<= 0
          (abs (- (single-note-duration oldsong)
                  (single-note-duration newsong)))
          *duration-tolerance*))
 (setf res (list 100 newsong)))
((and (not
       (equal (single-note-name oldsong)
              (single-note-name newsong)))
      (<= 0
          (abs (- (single-note-time oldsong)
                  (single-note-time newsong)))
          *time-tolerance*)
      (>
       (abs (- (single-note-duration oldsong)
               (single-note-duration newsong)))
       *duration-tolerance*))
 (setf res (list 101 newsong)))
((and (not
       (equal (single-note-name oldsong)
              (single-note-name newsong)))
      (>
       (abs (- (single-note-time oldsong)
               (single-note-time newsong)))
       *time-tolerance*)
      (<= 0
          (abs (- (single-note-duration oldsong)
                  (single-note-duration newsong)))
          *duration-tolerance*))
 (setf res (list 110 newsong)))
((and (not
       (equal (single-note-name oldsong)
              (single-note-name newsong)))
      (>
                (abs (- (single-note-time oldsong)
                        (single-note-time newsong)))
                *time-tolerance*)
               (>
                (abs (- (single-note-duration oldsong)
                        (single-note-duration newsong)))
                *duration-tolerance*))
          (setf res (list 111 newsong))))
   res))
```

This method takes as an argument single-note instances. It makes use of functions that are considered time constants. So, the complexity becomes $O(1)$.

The time complexity of the method **match-note** is:

```
(let ((res))
  (defmethod match-note ((oldsong complex-note)
                         (newsong complex-note)
                         &key (note-only nil))
    (cond (note-only
           (setf res (equal (complex-note-name oldsong)
                            (complex-note-name newsong))))
          ((and (equal
                 (complex-note-name oldsong)
                 (complex-note-name newsong))
                (<= 0
                    (abs (-
                          (/ (apply #'+ (complex-note-time oldsong))
                             (length (complex-note-time oldsong)))
                          (/ (apply #'+ (complex-note-time newsong))
                             (length (complex-note-time newsong)))))
                    *time-tolerance*)
                (equal 0
                       (duration-check (complex-note-duration oldsong)
                                       (complex-note-duration newsong))))
           (setf res (list t newsong)))
          ((and (equal (complex-note-name oldsong)
                       (complex-note-name newsong))
```

```
      (<= 0
          (abs (-
                (/ (apply #'+ (complex-note-time oldsong))
                   (length (complex-note-time oldsong)))
                (/ (apply #'+ (complex-note-time newsong))
                   (length (complex-note-time newsong)))))
          *time-tolerance*)
      (>
       (duration-check (complex-note-duration oldsong)
                       (complex-note-duration newsong)) 0))
 (setf res (list 001 newsong)))
((and (equal (complex-note-name oldsong)
             (complex-note-name newsong))
      (>
       (abs (-
             (/ (apply #'+ (complex-note-time oldsong))
                (length (complex-note-time oldsong)))
             (/ (apply #'+ (complex-note-time newsong))
                (length (complex-note-time newsong)))))
       *time-tolerance*)
      (equal 0
             (duration-check (complex-note-duration oldsong)
                             (complex-note-duration newsong))))
 (setf res (list 010 newsong)))
((and (equal (complex-note-name oldsong)
             (complex-note-name newsong))
      (>
       (abs (-
             (/ (apply #'+ (complex-note-time oldsong))
                (length (complex-note-time oldsong)))
             (/ (apply #'+ (complex-note-time newsong))
                (length (complex-note-time newsong)))))
       *time-tolerance*)
      (>
       (duration-check (complex-note-duration oldsong)
                       (complex-note-duration newsong)) 0))
 (setf res (list 011 newsong)))
((and (not
       (equal (complex-note-name oldsong)
              (complex-note-name newsong)))
      (<= 0
          (abs (-
                (/ (apply #'+ (complex-note-time oldsong))
                   (length (complex-note-time oldsong)))
                (/ (apply #'+ (complex-note-time newsong))
                   (length (complex-note-time newsong)))))
          *time-tolerance*)
      (equal 0
             (duration-check (complex-note-duration oldsong)
                             (complex-note-duration newsong))))
 (setf res (list 100 newsong)))
((and (not
       (equal (complex-note-name oldsong)
              (complex-note-name newsong)))
      (<= 0
          (abs (-
                (/ (apply #'+ (complex-note-time oldsong))
                   (length (complex-note-time oldsong)))
                (/ (apply #'+ (complex-note-time newsong))
                   (length (complex-note-time newsong)))))
          *time-tolerance*)
      (>
       (duration-check (complex-note-duration oldsong)
                       (complex-note-duration newsong)) 0))
 (setf res (list 101 newsong)))
((and (not
       (equal (complex-note-name oldsong)
              (complex-note-name newsong)))
      (>
       (abs (-
             (/ (apply #'+ (complex-note-time oldsong))
                (length (complex-note-time oldsong)))
             (/ (apply #'+ (complex-note-time newsong))
                (length (complex-note-time newsong)))))
       *time-tolerance*)
      (equal 0
             (duration-check (complex-note-duration oldsong)
                             (complex-note-duration newsong))))
 (setf res (list 110 newsong)))
((and (not
                (equal (complex-note-name oldsong)
                       (complex-note-name newsong)))
               (>
                (abs (-
                      (/ (apply #'+ (complex-note-time oldsong))
                         (length (complex-note-time oldsong)))
                      (/ (apply #'+ (complex-note-time newsong))
                         (length (complex-note-time newsong)))))
                *time-tolerance*)
               (>
                (duration-check (complex-note-duration oldsong)
                                (complex-note-duration newsong)) 0))
          (setf res (list 111 newsong))))
    res))
```

This method takes as an argument complex-note instances. It deals with functions that are considered time constants. So, the complexity becomes $O(1)$.

The time complexity of the function **compare-music** is:

```
(defun compare-music (filename1 &optional (filename2 nil))
  (match-music
   (make-instance-single-complex-note
    (midi-events-organized-to-indicate-chords
     (import-song filename1)))
   (make-instance-single-complex-note
    (midi-events-organized-to-indicate-chords
     (or (import-song filename2)
         cl-user::*myseq*)))))
```

This function takes as an argument two files, so it is of paramount importance for us to see the complexity of the internal functions. The more one function is being called, the more it influences the time complexity. The problem is that the **compare-music** consists of functions that are called equal times. So, the complexity will be determined from the function that requires more time to act, which is the function **midi-events-organized-to-indicate-chords** with time complexity $O(n^2)$. As a result, the time complexity of the MM algorithm is $O(n^2)$ which is a non linear complexity.

It could become linear, if though, instead of lists hash tables, were used and some functions from the package cm were redefined.

Chapter 10

References

[1] A Guided Tour Of The Common Lisp Interface Manager *Tour Of CLIM*

[2] http://www-ccrma.stanford.edu/software/cmn/ *Common Music Notation*

[3] Wikipedia *Lisp (Programming Language)*

[4] Wikipedia *Open-source software*

[5] Wikipedia *Musical Instrument Digital Interface*

[6] Wikipedia *CLOS (Common Lisp Object System)*

[7] www.cm.html *Common Music*

[8] Introduction To Algorithms 2nd Edition *Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein*

[9] Algorithms *Panagiotis D. Bozanis*

[10] Speech And Language Processing 1999 *Daniel Jurafsky, James H. Martin*

[11] A. Amir, E. Porat, and M. Lewenstein. Approximate subset matching with don't cares. In Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms, pages 305-306. Society for Industrial and Applied Mathematics, 2001. ISBN 0-89871-490-7.

[12] T. Crawford, C. Iliopoulos, and R. Raman. String matching techniques for musical similarity and melodic recognition. Computing in Musicology,

11:73-100, 1998.

[13] C. S. Iliopoulos, M. Mohamed, L. Mouchard, K. Perdikuri, W. F. Smyth, and A. Tsakalidis. String regularities with don't cares. Nordic Journal of Computing, 10(1):40-51, 2003.

[14] K. Lemstrom and P. Laine. Musical information retrieval using musical parameters. In Proc. International Computer Music Conference (ICMC '98), pages 341-348, 1998.

[15] B. Schieber and U. Vishkin. On finding lowest common ancestors: Simplification and parallelization. SIAM Journal on Computing, 17(6):1253-1262, 1988.

[16] Hjelsvold, R. and R. Midstraum, "Modelling and Querying Video Data" in Proc. o VLDB Conf, pp. 686-694, 1994.

[17] Hsu J. L., C. C. Liu, and A. L. P. Chen, "Efficient Repeating Pattern Finding in Music Databases" in Proc. of Seventh International Conference on Information and Knowledge Management (CIKM' 98), 1998.

[18] Jain, R., "Metadata in Video Databases" SIGMOD RECORD, Vol. 23, No. 4, pp. 27-33, Dec. 1994.

[19] Jones, M. R. and S. Holleran, Cognitive bases of musical communication, American Psychological Association, 1991.

[20] Knuth D. E., I. H. Morris, V. R. Pratt, "Fast Pattern Matching 4. Conclusion in Strings" SIAM J. Comput.,pp.323-350, 1977.

[21] Lerdahl, F. and R. Jackendoff, A generative rheory of tonal music, The MIT Press, 1983.

[22] Liu, C. C., J . L Hsu, and A. L. P. Chen, "Efficient Near Neighbor Searching Using Multiple Indexes for Content-Based Multimedia Retrieval" Multimedia Tools and Applications (to be appeared).

[23] Liu C. C., J. L. Hsu and A. L. P. Chen, "Efficient Theme and Non-Trivial Repeating Pattern Discovering in Music Databases" in Proc. o IEEE International ConJ on Data Engineering, 1999.

[24] Boyer, R. S. and J . S. Moore, "A Fast String Searching Algorithm" CACM, Vol. 20, Oct. 1977.

[25] Chiueh, T. C., "Content-Based Image Indexing" in Proc. of the 20th VLDB Conf, pp. 582-593, 1994.

[26] Chou, T. C., A. L. P. Chen, and C. C. Liu, “Music Databases: Indexing Techniques and Implementation” in Proc. of IEEE

[27] A. Uitdenbogerd and J. Zobel. Manipulation of music for melody matching. In B. Smith and W. Effelsberg, editors, Proceedings of the ACM Multimedia Conference, pages 235-240, Bristol, UK, Sept. http://www.intelliscore.net/. 1998.

[28] Innovative music systems, inc. wav to midi, mp3 to midi converter - intelliscore.

[29] T. Cormen, C. Leiserson, and R. Rivest. Introduction to Algorithms. The MIT Press, Cambridge, MA, 1990.

[30] F. Damerau. A technique for computer detection and correction of spelling errors. Comm. of the ACM, 7(3):171-176, 1964.

[31] R. Wagner and M. Fisher. The string to string correction problem. Journal of the ACM, 21:168-178, 1974.

[32] D. Deutsch. Grouping mechanisms in music. In Psychology of Music, Orlando, 1982. Academic Press.

[33] A. L. Uitdenbogerd and J. Zobel. Matching techniques for large music databases. In D. Bulterman, K. Jeffay, and H. J. Zhang, editors, Proceedings of the ACM Multimedia Conference, pages 57-66, Orlando, Florida, Nov. 1999.