CONTENT-BASED RETRIEVAL OF MUSIC USING MONOPHONIC QUERIES ON A DATABASE OF POLYPHONIC, MIDI INFORMATION

By

RAVIKUMAR NIDADAVOLU

Bachelor of Science in Information Technology

Andhra University

Hyderabad, Andhra Pradesh, India

2004

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2008

CONTENT-BASED RETRIEVAL OF MUSIC USING MONOPHONIC QUERIES ON A DATABASE OF POLYPHONIC MIDI, INFORMATION

Thesis Approved:

Dr. Blayne E. Mayfield

Thesis Adviser

Dr. John P. Chandler

Dr. Venkatesh Sarangan

Dr. A. Gordon Emslie

Dean of the Graduate College

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The use of multimedia data has spread throughout the world with the availability of high performance, low-cost computers, and this has led to a need for accurate, efficient retrieval methods for large multimedia databases. Many users experience difficulty in formulating a query in words when they want to retrieve something from a multimedia database. Content-based retrieval is seen as a solution to this problem [1]. Content-based retrieval enables users to represent what they want directly; it makes the retrieval system easier to use. For a music retrieval system, content-based query can be either a hummed query or played through a digital piano.

1.1 Motivation

In spite of the growth of digital music libraries, or generically speaking, any multimedia database, it is really useful only if users can find what they are seeking in an efficient manner. Presently, whether it is the case of a digital music library, the Internet or any music database, search and retrieval is carried out mostly in a textual manner, based on metadata such as author, title or genre. Normally, metadata related to music is created manually. Some metadata information provided is meaningless and ambiguous. For instance, the track number of a CD album does not provide any useful information for musical data retrieval. A natural way of searching for music is to identify it by its content

rather than its secondary information (e.g., title), because the content is usually more memorable and a more robust feature of the musical work. Many people remember a short tidbit of a song but fail to recall a song's name. Information such as title, artist, and genre often is learned at a much later stage of people's relationship with a song. Furthermore, as the human brain sometimes forgets this information, while the melody remains fresh in mind, it seems obvious that an application capable of retrieving music from humming would be very useful. A query-by-humming system allows a user to find a song even if he merely knows the tune from part of the melody. The user simply hums the tune into a computer microphone, and the system searches through a database of songs containing the tune and returns a ranked list of search results.

1.2 Music Terminology

Each time a key pressed on a piano or air is blown across a flute mouthpiece to make a sound, we say that a note has been played. Each type of musical instrument has its own method of playing notes within a certain range. This range refers to frequency range of sounds that can be produced from the musical instrument. Musicians do not usually discuss the frequencies of notes, but refer to them as having a particular *pitch*. Every note has a certain *pitch*. *Pitch* is how the listener perceives the tone. A note with higher frequency is said to have a higher pitch than one of a lower frequency. The pitch distance between notes is called an *interval*. If one note has double the frequency of another, then it is said to be one *octave* above the lower note, i.e. the interval between the notes is one octave. Notes that are an octave apart sound very similar to each other. All notes that are a whole number octave apart have the same note name, which is usually a

letter, or a letter combined with a sharp (♯) or flat (♭) symbol. The notes used in western music result from dividing an octave into 12 notes that are a *semitone* apart. The note names used for notes within one octave starting at C and going up are:

C C♯ D D♯ E F F♯ G G♯ A A♯ B C.

C♯ is read as 'C sharp'. It is the same note as 'D flat' (D♭). Similarly D♯, F♯, G♯, and A♯ can be referred to as E♭, G♭, A♭, and B♭ respectively. Note that a sharp symbol is used to raise the pitch of a note by one semitone, and a flat is used to lower the pitch of a note by one semitone.

Music is often divided into three categories based on the amount of concurrency present. *Monophonic* ('Mono' means one and 'phony' means sound) means 'one sound'. In monophonic music only one note sounds at a time; two notes cannot sound simultaneously. The simplest type of monophony would be one person singing alone. *Homophonic* ('Homo' means same and 'phony' means sound) means that different notes may sound simultaneously but must start and end at the same time. For example, a singer accompanied by a guitar picking can be homophonic. *Polyphonic (*'Poly' means many and 'phony' means sound) means many sounds at once. In polyphony, notes may sound simultaneously with different start and end times. Generally most music by large instrumental groups such as bands or orchestras is polyphonic. This research deals with searching of a monophonic music (query) in a polyphonic database.

1.3 Music data Representation

Musical data can be represented in a computer in two different ways: symbolic representation and acoustic representation. Formats such as *mp3*, *wav* and *au* are popular

for acoustic representation. Symbolic representation is based on musical scores, with one entry per note. Each entry keeps track of information such as pitch, duration, type of instrument and so on. Examples of this representation include MIDI (Musical Instrument Digital Interface), Humdrum, and sheet music with MIDI being the most popular format. Score-based representations are much more structured and easier to handle than raw audio data. A MIDI file contains the performance-related information of a piece of music; it contains all the necessary information to recreate that piece of music. As MIDI is the widely used music format this research concentrates in this format.

1.4 Thesis Overview

In this research, content-based music retrieval is performed on a polyphonic MIDI music database where the query is a hummed tune. After a hummed tune is translated to MIDI format, it is matched against the database of 250 MIDI files which may contain either monophonic or polyphonic MIDI music files. The similarity is based on the intuitive notion of similarity perceived by humans: two musical files are similar if they are fully or partially based on the same score, even if they are performed in a different key or at a different tempo. Retrieval results are ranked based on the computed similarity estimate. Two approximate string matching algorithms, LCTS and Myers algorithms are modified, applied to the problem, and retrieval performance is calculated. Response times of the algorithms are calculated by altering the values of some of the interesting parameters such as the query length, degree of polyphony and size of the database.

CHAPTER 2

LITERATURE REVIEW

2.1 MIDI

When music technology moved into the 1980s, keyboards with internal microprocessors had been introduced and become popular. Manufacturers began to experiment with interfaces that would transfer digital information among their own various devices. By the beginning of the decade (1980), it dawned on the electronics industry that, if each manufacturer developed its own interface, it would not be beneficial to the growth of the entire industry and it would ultimately prove frustrating to musicians. In an attempt to find a way forward from this situation, audio engineer and synthesizer designer Dave Smith of Sequential Circuits, Inc. proposed the MIDI standard in 1981 in a paper to the Audio Engineering Society**.** The proposal received widespread enthusiasm within the industry and the MIDI Specification 1.0 was published in August 1983[2]. It is a standard that manufacturers of electronic musical instruments have agreed on. It is a set of specifications that they can use in building their instruments so that instruments of one manufacturer can, without difficulty, communicate musical information with instruments of another manufacturer. MIDI was originally designed for keyboards, but it is flexible enough that it could be easily adapted to various other kinds of electronic instruments.

MIDI is an acronym for the *Musical Instrument Digital Interface*. The most fundamental aspect of MIDI is that it does not deal with musical sound at all; it only

deals with musical data which can be transmitted among musical computing devices, including keyboards, drum machines, sequencers and computers running music software. Two musical instruments communicating using the MIDI standard send information as a series of numbers (binary) over connecting cables. The MIDI standard specifies that the numbers sent as data be sent in groups called *MIDI messages*. Each MIDI message communicates one musical event. These musical events usually are actions that a performer makes while playing a musical instrument, actions such as pressing keys, setting switches etc. MIDI messages can be collected and stored in a computer file commonly called a MIDI file, or more formally, a Standard MIDI File.

Standard MIDI files provide a common file format used by most musical software and hardware devices. This format stores the standard MIDI messages plus a time-stamp for each message (i.e. a series of bytes that represent how many clock pulses to wait before "playing" the event). MIDI messages usually are two or three bytes. The first byte of any MIDI message is called the status byte. It tells what kind of message it is. The bytes that follow the status byte are called data bytes. Many types of MIDI messages use two bytes to carry additional information, but some need only one byte. MIDI uses MIDI channels to allow communication between individual devices in a MIDI system. MIDI contains 16 channels and each of these channels can be used to set up an instrument, which gives the potential to program for 16 instruments playing at the same time.

All MIDI messages are divided into two types: *system messages* which carry information that is not channel specific, such as timing signal for synchronization, and *channel messages* which are transmitted on individual channels rather than globally to all devices. For example, the MIDI *Note-On* message is a channel message that instructs a

synthesizer to begin playing a note. This note will continue playing until a corresponding *Note-Off* message is received. The four most significant bits of the status byte of all Note-

Table 2.1: Format of Note-on and Note-off events

Event-Type	Byte 0		Byte 1	Byte2
	Most significant Nibble (4 bits)	**Least significant Nibble (4 bits)**		
Note-On	1001	MIDI channel [0-15]	Key Number [0-127]	Velocity [0-127]
Note-Off	1000	MIDI channel [0-15]	Key Number [0-127]	Velocity [0-127]

On messages must be 1001 and, as with all channel voice messages, the four least significant bits specify the channel on which the note should be played. *Note-On* messages have two data bytes. The first specifies pitch, from 0 to 127, and the second specifies velocity, also from 0 to 127. Pitch is numbered in semitone increments, with note 60 being designated as middle C. The velocity value specifies how hard a note is struck, which most synthesizers map to initial volume. *Note-Off* is the channel message used for turning off a currently playing MIDI note. The *Note-Off* message has an identical format to a *Note-On* message, except that the four most significant bits of the status byte are 1000. The pitch value specifies the pitch of the note that is to be stopped on the given channel and the velocity value specifies how quickly a note is released. When the note is turned off, it is just like an instrumentalist releasing a note its sound should start decaying. The basic format of the *Note-On* and *Note-Off* message is as shown in Table 2.1. There are many other channel messages such as *Channel Pressure Message*,

Program Change Message, etc. This research mainly concentrates on the *Note-On* and *Note-Off* messages.

All standard MIDI files consist of groups of data called *chunks*, each of which consist of a four-character identifier, a thirty-two bit value indicating the length in bytes of the chunk and the chunk data itself. There are two types of chunks: header chunks and track chunks. The header chunk is found at the beginning of the file and includes the type of file format, number of tracks etc. The header chunk starts with four bytes representing ASCII symbols for the letters "Mthd" followed by four bytes specifying the length of the header chunk. Track chunks contain all of the information and MIDI messages specific to individual tracks. The Track chunk begins with four bytes representing the ASCII symbols for the letters "Mtrk" followed by four bytes specifying the length (in bytes) of the track data to follow, followed by the data. The data differ from the normal MIDI message only in that a *delta time* precedes each one. The delta time specifies how much time has elapsed since the previous track event. The MIDI message and their associated delta times are called *Track events*. Track events can be a MIDI events or *meta-events*. Meta-events provide the ability to include information such as lyrics, key signatures, time signatures, tempo changes and track names in files.

The note names and different pitch values in standard MIDI are as shown in Table 2.2. The numbers used for note values are from 0 to 127. The lowest note is a C (C0) and this is assigned note number 0. The C# (C#0) above it would have a note number of 1. The D (D0) note above that would have a note number of 2.

Table 2.2: Midi Note Numbers for different Octaves

Octave	Note Numbers											
	C	**C#**	**D**	**D#**	**E**	**F**	**F#**	**G**	**G#**	**A**	**A#**	**B**
0	0	1	2	3	4	5	6	7	8	9	10	11
1	12	13	14	15	16	17	18	19	20	21	22	23
2	24	25	26	27	28	29	30	31	32	33	34	35
3	36	37	38	39	40	41	42	43	44	45	46	47
4	48	49	50	51	52	53	54	55	56	57	58	59
5	**60**	61	62	63	64	65	66	67	68	69	70	71
6	72	73	74	75	76	77	78	79	80	81	82	83
7	84	85	86	87	88	89	90	91	92	93	94	95
8	96	97	98	99	100	101	102	103	104	105	106	107
9	108	109	110	111	112	113	114	115	116	117	118	119
10	120	121	122	123	124	125	126	127				

2.2 String Matching

The string matching problem is to find one or more occurrences of a given pattern $P_{1...m}$ with length m in a large text $T_{1...n}$ with length n $(m \leq n)$, both being sequences of characters drawn from a finite character set. This problem is fundamental in computer science and is a basic need of many applications, such as text retrieval, music retrieval, computational biology, data mining etc. It has been studied extensively and numerous techniques and algorithms have been designed to solve this problem. String matching can be divided mainly into two types, exact and inexact (approximate) string matching.

2.2.1 Exact Matching

The exact string matching problem can be defined as follows:

Given a text $T_{1...n}$ of n characters and the pattern $P_{1...m}$ of m characters ($m \leq n$) over a finite alphabet Σ, the exact string matching problem is to find an integer s called the valid shift where $0 \leq s \leq n\text{-}m$ and $T_{s+1...s+m} = P_{1...m}$.

The simplest approach for exact string matching is a brute force method, also called as the naïve method [3]. In this method, the first character of the pattern is aligned with the first character of the text. Comparison of characters of the aligned region is made until a mismatch is found. If the end of the pattern is reached without a mismatch, an occurrence of the pattern is reported. In either case, the pattern is shifted one position to the right and the process is repeated. In worst case the number of comparisons made by this method is O (*mn*) where *m* and *n* are length of pattern and text respectively.

The Boyer-Moore algorithm [3] provides marked improvement over the brute force method through the implementation of two heuristics, the *bad-character-heuristic* and the *good-suffix-heuristic*. The idea of the Boyer-Moore algorithm is to skip those parts of the text that cannot match the pattern. At the beginning of the matching process the pattern is left-aligned with the text. The algorithm starts the comparison between text and pattern from right to left by checking if $T_m = P_m$ (where *m* is the length of pattern). If the characters match, the algorithm continues with $T_{m-1} = P_{m-1}$, and so on. If a mismatch occurs, the two heuristics calculate how far the pattern can be moved to the right, thus skipping over some text positions that cannot yield a match.

The first heuristic is the bad-character-heuristic. In this heuristic, as shown in Figure 2.1, if a mismatch occurs at position *k* in text T, i.e. $P_c \neq T_k$, then pattern P is shifted right by **Max** {1, c - B (T_k)} positions. B (T_k) (bad-character) is the position of the right-most occurrence of T_k in the pattern P. If T_k does not occur in P then the value of B (T_k) is 0. The shaded portion in the figure represents the matched portion of the text and pattern.

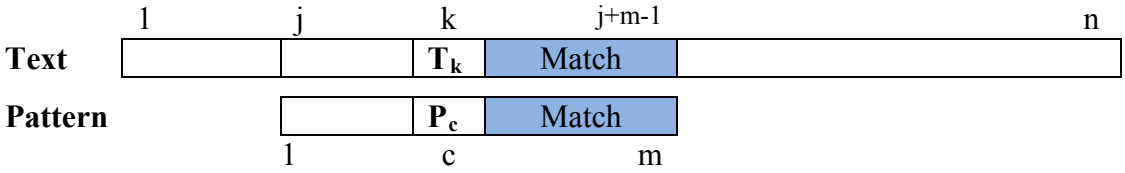


Figure 2.1: Bad-character-heuristic explanation

For example consider the text and the pattern as shown in Figure 2.2. Initially the first character of pattern is aligned with the first character of the text. The right-to-left comparison starts, and a mismatch occurs between the characters 'O' and 'E' at position 5. The pattern is searched for the rightmost occurrence of character 'O' (i.e. T_k) which is found at position 2. So, the pattern is moved 3 (5-2) positions to the right (Step1). This process continues until an exact match of pattern is found or until the end of text is reached. In this example exact match is found in step2.

Position	1	2	3	4	5	6	7	8	9	10	11
Text	R	A	C	D	O	G	M	O	U	S	E
Pattern	M	O	U	S	E						
Step1				M	O	U	S	E			
Step2							M	O	U	S	E

Figure 2.2: bad-character-heuristic

The second heuristic is the *good-suffix-heuristic*. While comparing the pattern with the text from right to left, and the mismatch is at pattern position k, then a suffix of the pattern $P_{k+1\ldots m}$ must match; otherwise, a mismatch would have occurred earlier. As shown in Figure 2.3, the shaded portion 'u' in the text and pattern is the matched portion and a mismatch is between characters T_k and $P_{k.}$. The shaded portion 'u' in the pattern (Step1) is the matched suffix of the pattern.

When a mismatch occurs, the pattern $P_{1...k}$ is searched for the right-most occurrence of the 'u'. If 'u' is present in $P_{1...k}$, then the pattern is moved to the right so that the right-most occurrence of 'u' aligns to the text. In figure 2.3, step2 shows how the pattern is moved to the right if 'u' is present.

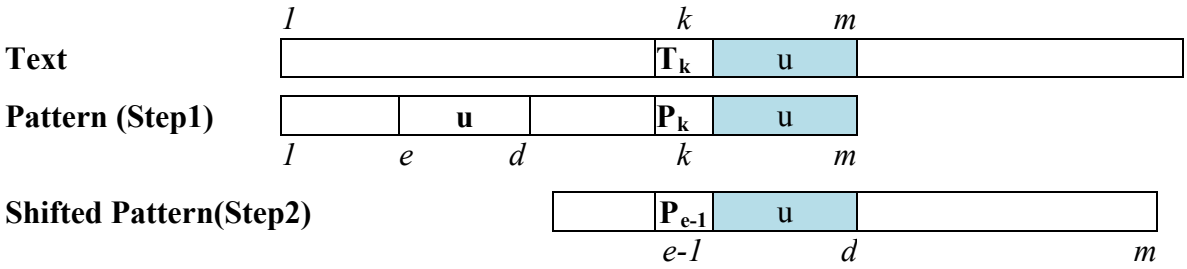


Figure 2.3: Good-Suffix rule illustrations 1

If no such occurrence of 'u' is found in $P_{1...k}$, then the pattern is shifted to the right so that the longest suffix of $T_{k+1...m}$ aligns with a matching prefix of $P_{1...m}$. In Figure 2.4, the portion 'v' in the pattern i.e. prefix of the pattern is assumed to match the suffix of $T_{k+1...m}$. If no such portion 'v' exists then the pattern is moved its whole length to the right.

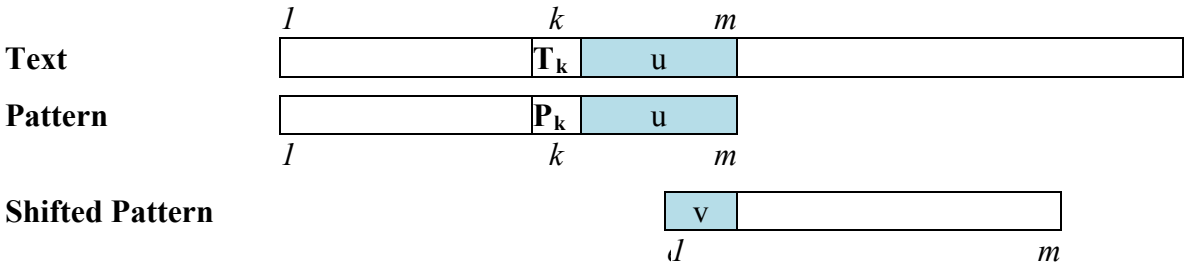


Figure 2.4: Good-Suffix rule illustrations 2

For example, consider the text and the pattern as shown in Figure 2.5. Initially, the first character of pattern is aligned with the first character of the text. The comparison begins from right to left. A mismatch occurs at position 3.The matched suffix is of the

pattern is ‘ab’. The pattern can be shifted until the next occurrence of ‘ab’ in the pattern is aligned to the text symbols ‘ab’, i.e. to position 3.

Position	1	2	3	4	5	6	7	8	9	10
Text	A	B	A	A	B	A	B	B	A	C
Pattern	C	A	B	A	B					
Step1			C	A	B	A	B			

Figure 2.5: good-suffix-heuristic

The worst case complexity is still O (*mn*) which is the same as the naïve method, where *m* and *n* are length of pattern and text respectively. The algorithm can achieve an average case running time of O (*n*) by preprocessing the pattern. Considering the bad-character-heuristic, in the preprocessing, for each different character ‘c’ in the pattern, B[c] (bad-character) is calculated in the following way:

- Initially the B[c] is set to zero for all characters.
- Then, the processing of the pattern starts from the right, if the value of B[c] is zero, and then B[c] is set to a value, which is the position of the character in the pattern.

For example, for the pattern ‘xyzyz’, initially B[x], B[y], B[z] are set to 0. Processing starts from right and B[z] is set to 5, and then B[y] is set to 4, and then B[z] whose value is 5, which is not zero, so the value is not modified. This process continues until the end of the pattern. Finally the values of B[x], B[y], and B[z] are 0, 4 and 5. Preprocessing of pattern considering good-suffix-heuristic also calculates the amount of shift the pattern can be moved to the right when a mismatch occurs. (Complete explanation is discussed in [3]).

The Rabin-Karp algorithm [4] presented by Michael Rabin and Richard Karp achieves an average run time of O (m+n) using hashing. The Rabin-Karp algorithm is based on the fact that if two strings are equal, their hash values are also equal. But since two hash values can be equal even if the underlying strings differ, the algorithm has to verify every match of hash values. The hash values are calculated based on pre-assigned values for each character, generally prime numbers. In a preprocessing phase, the hash value of the pattern P is calculated. Then, for every text position $i = 1...n-m+1$ the hash value of $T_i \ldots T_{i-1+m}$ is compared to the hash value of the pattern. If the hash values match, the pattern is compared to the text character-by-character beginning at position i to verify that there is indeed a match at this position. A good hash function would yield the same value only when there is a match of strings. For smaller patterns hash value can be calculated considering the m character sequence as an m-digit number in base b, where b is the size of the alphabet.

The hash value for the text $T_{i \ldots i+m-1}$ is calculated using the formula:

$$\text{Hash}(i) = (T_i)(b^{m-1}) + (T_{i+1})(b^{m-2}) \ldots\ldots\ldots (T_{i+m-1})$$

Furthermore, given Hash(i), Hash(i+1) for the next subsequence $T_{i+1 \ldots i+m}$ can be computed in constant time using the formula below.

$$\text{Hash}(i+1) = (\text{Hash}(i) * b) \quad \text{[shift left by one digit]}$$
$$- (T_i * b^m) \quad \text{[subtract leftmost digit]}$$
$$+ (T_{i+m}) \quad \text{[add new right most digit]}$$

In this way the new hash value is computed by adjusting the previous hash value, resulting in a time savings. If m is large, then the resulting value b^m will be enormous. For this reason, the hash value is calculated by taking mod of prime number q.

For example let Σ = { a,b,c,d,e,f,g,h,i,j} be the alphabet. Let us say character 'a' corresponds to 1, character 'b' corresponds to 2 and so on. The hash value for the string 'cah' would be

$$(3*10^2)+(1*10^1)+8 = 318.$$

Consider the text T= 'abcahd' and the pattern P= 'cah',

Figure 2.6 shows the hash values for different positions of the text. Then the algorithm checks for every text position if the current text hash value matches the pattern hash value. A match is found at the text position 3.

Position	1	2	3	4
Text	abc	bca	cah	ahd
Hash	123	231	318	184

Figure 2.6: Hash value calculations

Baeze-Yates and Gonnet [5] devised a simple, bit-oriented method that solves exact matching problem very efficiently for relatively small patterns. They called their method the *Shift-And* method. The algorithm is based on bit comparison rather than character comparison. The Shift-And method has an advantage over character comparison because bit comparisons are performed much more quickly by processors. The Shift-And algorithm simulates the operation of non-deterministic automata that search text for the pattern.

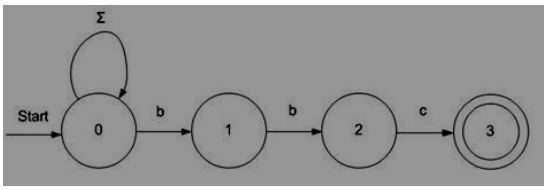


Figure 2.7: Non-deterministic automata to search text for pattern P = 'bbc'

Notation: A bit mask of length *n* (a sequence of bits) is denoted by $b_{n...1}$. Superscripts are used to denote bit repetition; for example $0^3 = 000$. Σ denotes an alphabet and *m* denotes pattern length.

In the preprocessing step of the algorithm a table B that stores a bit mask B[c] = $b_{m...1}$ for each character $c \in \Sigma$ is created. The bit mask $B[c]$ has the *i*-th bit set to 1 if P_i = c and all other bits are set to 0. For example let Σ = {a,b,c} , pattern P = 'bbc' and Text T = 'acbbc'. Bit masks for each character will be B[a] = 000, B[b] = 011, B[c] = 100.

The state of the search is kept in another bit mask $D = d_{m...1,}$ where $d_i = 1$ if and only if the state numbered *i* in Figure 2.7 is active. Initially $D = 0^m$, which represents state numbered 0. When the text position *j* is scanned, $d_i = 1$ whenever $P_{1...i} = T_{j-i+1...j}$ ($1 \leq j \leq n$). A match is reported whenever bit d_m is 1. Initially $D = 0^m$ and, for each character T_j, update D using the formula below

$$D \leftarrow ((D << 1) \mid 0^{m-1}1) \ \& \ B[T_j]$$

From the above formula, the *i*-th bit is set to 1 if and only if the (*i*-1)-th bit was set for the previous text character and the new text character matches the pattern at position *i*. In other words, $T_{j-i+1...j} = P_{1...i}$ if and only if $T_{j-i+1...j-1} = P_{1...i-1}$ and $T_j = P_i$. For the above example the state of search varies as shown in Figure 2.8 (initially $D = 0^m$).

On reading character 'a'	D ← (000 | 001) & 000, i.e. D = 000.
On reading character 'c'	D ← (000 | 001) & 100, i.e. D = 000.
On reading character 'b'	D ← (000 | 001) & 011, i.e. D = 001.
On reading character 'b'	D ← (010 | 001) & 011, i.e. D = 011.
On reading character 'c'	D ← (110 | 001) & 100, i.e. D = 100.

Figure 2.8: State of search pattern P = 'bbc', text T = 'acbbc'

If the length m of the pattern $\leq w$, *(w* is the size of machine word) the state of search can be represented as a single machine word. This approach has a best performance of O (*n*) when the length of the pattern is no longer smaller than the word-size of the machine.

Lemstrom and Tarhio [6] developed a variation of the Baeza-Yates and Gonnet Shift-And algorithm for distributed string matching that is defined as follows:

> Given a collection T containing *h* strings $t^i = t^i_1 t^i_2 \ldots t^i_n$ $(1 \leq i \leq h)$ of equal length *n,* we say that some pattern $P_{1\ldots m}$ occurs at position *j* of T if $p_s \in \{ t^i_{j+s-1} \mid 1 \leq i \leq h\}$ for all $1 \leq s \leq m$.

For example consider the pattern P = 'cat' and the collection T= {'abcde', 'bcdef', 'abade', 'xyztz'}. Table 2.3 shows the distributed matching of the pattern in T. A match is found at position 2.

Table 2.3: Distributed Matching of pattern 'CAT'

$\mathbf{t^1}$	A	B	C	D	E
$\mathbf{t^2}$	B	C	D	E	F
$\mathbf{t^3}$	A	B	A	D	E
$\mathbf{t^4}$	X	Y	Z	T	Z

In their approach, bit-masks for each character are calculated similar to that of shift-and method. But the state of search at position j in the text is calculated using the formula below.

$$\mathbf{D \leftarrow ((D << 1) \mid 0^{m-1}1) \;\&\; \{B[t_j^1] \mid B[t_j^2] \mid \ldots B[T_j^h] \}}$$

A musical score can be viewed as string; the symbols in the string can be simply a set of notes or a set of intervals. The music retrieval task is to find occurrences of a query

pattern within a music database. Both the pattern and database entries can be treated as strings. Some music information retrieval techniques [7] use exact matching for finding a pattern in a music database. Smith and Medina [8] used exact string matching to discover themes in music. However, inexact (approximate) matching generally is more useful for music information retrieval as a query pattern may contain errors.

2.2.2 Approximate String Matching

Given a text string T and a pattern P, the *approximate string matching* problem is to find substrings of T that approximately match P. To decide if a pattern approximately matches a text with a limited number of errors a metric for measuring those errors must be employed. A metric frequently used in approximate string matching is *edit distance*. The use of edit distance was proposed first by Levenshtein and is known as the *Levenshtein distance* [8]. The Levenshtein distance between two strings is the minimum number of insertions, deletions and substitutions of characters needed to transform one string into another.

The dynamic programming method generally is used to calculate the edit distance measure. Let $A_{1\ldots m}$ and $B_{1\ldots n}$ be two strings. The dynamic programming method computes a (m+1) × (n+1) matrix C, where $C_{i,j}$ corresponds to the minimum number of edit steps needed to transform $A_{1..i}$ to $B_{1\ldots j}$. The recurrence relation for the edit distance is defined as follows:

$$\mathbf{C_{i,0} = i \text{ and } C_{0,j} = j \text{ for } 0 \le i \le m \text{ and } 0 \le j \le n}$$

$$\mathbf{C_{i,j}} = \boldsymbol{Min}\begin{cases} \boldsymbol{C_{i-1,j-1}} + \boldsymbol{\delta} \;\; \textbf{if } \mathbf{P_i} = \boldsymbol{T_j} \;\; \boldsymbol{\delta = 0} \textbf{ else } \boldsymbol{\delta = 1} & [\textbf{Substitution}] \\ \boldsymbol{C_{i-1,j}} + \mathbf{1} & [\boldsymbol{Deletion}] \\ \boldsymbol{C_{i,j-1}} + \mathbf{1} & [\boldsymbol{Insertion}] \end{cases}$$

The relation accounts for the three allowed operations, insertion, deletion and substitution. The last entry in the matrix i.e. $C_{m,n}$ evaluates to the edit distance between the two strings. For example consider the strings 'PARK' and 'SPAKE' as shown in Table 2.4; the edit distance is calculated using the above formula, and the value of the edit distance is 3.The complexity of the algorithm is O (*mn*).

Table 2.4: Edit distance Matrix

	(*i*)	P	A	R	K
(*j*)	0	1	2	3	4
S	1	1	2	3	4
P	2	1	2	3	4
A	3	2	1	2	3
K	4	3	2	2	2
E	5	4	3	3	3

Needleman and Wunsch were first to use the dynamic programming algorithm to perform inexact matching on biological sequences [9]. Mongeau and Sankoff [10] were among the first to adapt this technique to musical scores. They used the algorithm to compare the similarity of entire pieces of music. McNab [11] incorporated these ideas into the MELDEX (MELody inDEX) system, which searches for relatively short user queries within entire songs.

During the last decade, algorithms based on bit-parallelism have emerged as the fastest approximate string matching algorithms [12]. Wu and Manber [13] devised an algorithm that amplifies the Shift-And method by finding inexact occurrences of a pattern in text. The bit mask that represents the state of search in the Shift-And method can be expressed as a table M of size $m \times n$, where each column represents the bit mask at character T_j in text. For example the contents of Figure 2.5 can be represented in tabular form as shown in Table 2.5. The shaded cell represents a match at that position.

Table 2.5: State of search in tabular form

M		-- j--				
		A	**C**	**B**	**B**	**C**
-- i--	**B**	0	0	1	1	0
	B	0	0	0	1	0
	C	0	0	0	0	1

In the Wu-Manber method, the pre-processing step is similar to that of the Shift-And algorithm. A bit mask is calculated for each character, and instead of one state of search, *k+1* different states of search M^s ($0 \leq s \leq k$) are used, where *k* is at most number of mismatches allowed. M^s denotes in tabular form the state of search with at most *s* mismatches. $M^s(i, j) = 1$ if and only if $P_{1\ldots i} = T_{j-i+1\ldots j}$ with $\leq$ s errors. There are four possibilities for the condition $P_{1\ldots i} = T_{j-i+1\ldots j}$ with $\leq$ s errors to be true:

- $P_{1\ldots i-1} = T_{j-i+1\ldots j-1}$ with $\leq$ s errors, $T_j = P_i$ [Matching]
- $P_{1\ldots i-1} = T_{j-i+1\ldots j-1}$ with $\leq$ s-1 errors, $T_j \neq P_i$ [Substitution]
- $P_{1\ldots i-1} = T_{j-i+1\ldots j}$ with $\leq$ s-1 errors [Deletion of P_i]
- $P_{1\ldots i} = T_{j-i+1\ldots j-1}$ with $\leq$ s-1 errors [Inserting T_j]

Using the above four conditions, the formula below is derived which is used to calculate M^s (where $s = 1\ldots k$) at each text position *j*.

$$M^s(j) = (M^s(j-1)<<1) \,\&\, B[T_j] \mid M^{s-1}(j-1)]<<1 \mid M^{s-1}(j) << 1 \mid M^{s-1}(j-1)$$

(*matching*) (*substitution*) (*deletion*) (*insertion*)

The initial state of search, i.e. the first column of M^s, will be 1^s0^{m-s}, and the remaining columns are calculated using the above formula. If $M^s(m, j) = 1$ then there is an occurrence of P in T ending at position *j* (in the text) with at most *s* mismatches. For

example, consider pattern P = 'bbc', text T = 'acbbc' and *k=1*. M^1, shown in Table 2.6, can be calculated using the above formula. The two shaded cells in M^1 represent the two matches, when the number of errors allowed is 1.

In this method *k+1* different tables (M^0, M^1….M^k) are calculated to find whether the pattern is in the text with at most *k* mismatches. Hence the complexity is *k* times that of the Shift-And method i.e. O (*knm/w*).

Table 2.6: Allowed number of errors 1

M^1		--j--				
		A	**C**	**B**	**B**	**C**
·i·	**B**	1	0	0	0	1
	B	0	0	1	1	1
	C	0	0	0	1	1

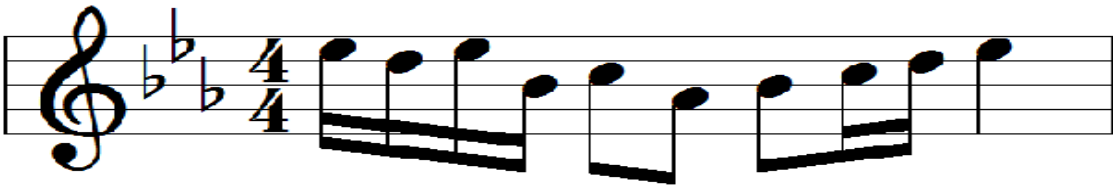
In 1998 Myers [14] presented a fast, bit-parallel implementation for approximate string matching with at most *k* differences. This approach is based on a bit-parallel simulation of the dynamic programming array discussed in section 2.5. The parallelization has optimal speedup, and the time complexity is O (*mn*/*w*) in the worst case, that is *k* times better than the Wu-Manber algorithm.

2.3 Monophonic Music Retrieval

String matching is the method used most often for content-based music retrieval from a monophonic music database. This is because music can be represented as strings of symbols, where the symbols are taken from an alphabet corresponding to some particular attribute of music, such as the duration or note pitch. In some music retrieval systems [7][11], only the pitch information for a piece of music is encoded as a string

and exact string matching techniques such as Boyer-Moore, Rabin-Karp and Shift-And methods are used for matching. *Pitch direction* often is used instead of the pitch itself for music retrieval. Music is transformed into a string that consists of three symbols, 'U', 'D', and 'S', which represent whether a note is higher than, lower than, or the same as the previous note, respectively [15][16] (Figure 2.9). The main reason for this representation is that when a hummed tune is input, it realizes robust retrieval even though the hummed tune may have differences in tone and tempo compared to the database tune it should match. However, for a large database, retrieval using only this information cannot provide sufficient resolution. Rhythm information and pitch intervals can be used to improve resolution.

When music retrieval is done by humming a tune, errors in the hummed tune may include not only variations in tone and tempo, but also insertions, deletions and errors in note values. Approximate string matching algorithms should be used to allow these errors. Similarity is calculated by edit distance in DP-matching (Dynamic Programming).



Absolute Pitch values	*63*	*62*	*63*	*58*	*60*	*56*	*57*	*60*	*62*	*63*
UDS Representation		*D*	*U*	*D*	*U*	*D*	*U*	*U*	*U*	*U*
Relative Pitches		*-1*	*1*	*-5*	*2*	*-4*	*1*	*3*	*2*	*1*

Figure 2.9: UDS Notation and Relative pitch values.

The *Note-Interval* system developed by Pardo [17][18] uses both pitch and rhythm information of musical pieces for retrieval. The Note-Interval system combines a *relative pitch* with a *logarithm of IOI Ratio* (LogIOIR) to form a <Pitch, Rhythm> pair. The relative pitch is the difference between the pitches of two adjacent notes. The *Inter-onset-interval* (IOI) is the time difference between the onset times of a note and the note that follows it. The final note of the piece has the same IOI value as its previous note. LogIOIR is used to represent rhythm. Both the target music database and the query are transformed to <Pitch, Rhythm> pairs. These pairs are encoded as strings and dynamic programming methods are used to find the similarity between the strings. The advantage of relative pitch is that transposition, (i.e. shifting a tune to a different key or octave) has no effect on matching, and the IOI ratio is invariant with respect to tempo. But, this technique is good only when the music database is monophonic. In polyphonic cases, two or more notes can be sounded at the same time, so relative pitches will not be useful as the query can be formed from different parts of the music (Figure 2.10).

Pardo and Sanghi [17] used probabilistic string matching techniques for monophonic music retrieval. They used the longest common subsequence approach to calculate the similarity. A matrix M of size (m+1)×(n+1) is computed using the following recurrence relation.

$$\mathbf{M_{i,0} = 0 \text{ and } M_{0,j} = 0 \text{ for } 0 \le i \le m \text{ and } 0 \le j \le n}$$

$$\mathbf{M_{i,j}} = \boldsymbol{Max}\begin{cases} \boldsymbol{M_{i-1,j-1}} + \mu(\mathbf{P_i}, \boldsymbol{T_j}) & [\textbf{Substitution}] \\ \boldsymbol{M_{i-1,j}} + \mu(\mathbf{P_i}, -) & [\boldsymbol{Deletion}] \\ \boldsymbol{M_{i,j-1}} + \mu(-, \boldsymbol{T_j}) & [\boldsymbol{Insertion}] \end{cases}$$

μ(x, y) represents the match score function where x and y (note values) are symbols of the pattern and text respectively. In the above formula, the '-'character is a

blank, which is added to the alphabet of note values. Matching to a blank can be thought of as skipping a character matched to the blank. For example pattern P be 'BCD' and text T be 'ABCD'. If character '-' is added to pattern at the beginning, the pattern will be '–BCD'. Now this is matched with 'ABCD' with skipping the first character in the text, which is matched to blank.

μ (x, y) gives the numeric value that corresponds to goodness of match between characters x and y. The match score function value for matching a blank is set to the penalty for skipping the element matched to the blank. Skipping a character of the pattern assumes the pattern inserted an extra character that was not in the text string. Skipping a character of the text string assumes that the pattern deleted that character of the text string. This function returns a negative value when the probability of a meaningful match is below that of a random co-occurrence. Similarly, the value is positive when a meaningful match is more likely than random chance. The similarity value between P and T is defined as the highest valued element of matrix M**.**

2.4 Polyphonic Music Retrieval

Two approaches commonly have been used when searching for a monophonic pattern in a polyphonic database. The first approach is to convert polyphony into monophony and then apply DP-matching techniques on the monophonic data. Uitdenbogerd and Zobel [19] transform polyphonic music into monophonic music by using the note with the highest pitch at any given moment. One drawback of this technique is the assumption that polyphonic music can be represented in this way, which is not always true [20].

Another technique is to extract the melody from the polyphonic data and then compare the melody against the monophonic pattern [20]. But it is quite possible that a query will be based on some part of the music other than the melody, such as a bass line [21]. In this case all of the parts of the music should be considered (Figure 2.10).

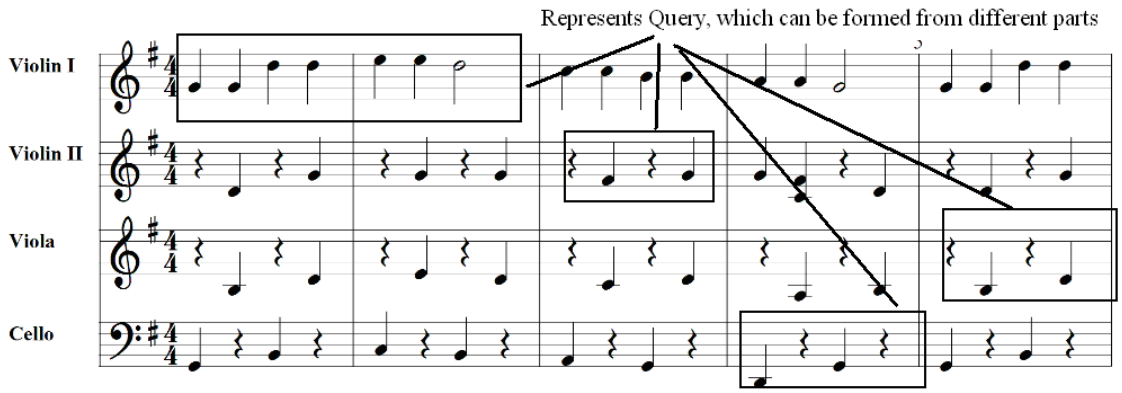


Figure 2.10: Sheet music representation of 'Twinkle Twinkle Little Star' song

The second approach is to match the monophonic pattern with all parts (i.e. voices or staff lines) of the polyphonic music. Figure 2.10 shows the sheet music representation of a polyphonic musical piece, in which four different voices are played at the same time. The boxed areas of the figure illustrate that a user's hummed query pattern can be matched to different voices; i.e. a pattern may skip from one part to another.

In their approach to polyphonic music retrieval, Iliopoulos and Kurokawa [15] considered all parts of a polyphonic music piece to match a pattern. They used the Shift-And algorithm for exact distributed matching, i.e. to match a pattern across different parts. For example Table 2.7 shows sequences of (pitch, duration) pairs from three voices of a polyphonic music piece and a pattern that will be matched against the voices. Pairs in which the first value is zero represents rests.

Table 2.7: Different parts of polyphonic music

Part1	(72, 2), (70, 2), (69, 4)
Part2	(0,1),(67,2),(67,1)(0,1)(67,1),(65,1),(64,1)
Part3	(52, 2), (48, 2), (53, 2), (48, 2)
Pattern	(72, 1) (52, 1) (70, 1) (67, 1)(69, 1) (53, 1) (65, 1) (64, 1)

The approach used by Iliopoulos and Kurokawa replaces the (pitch, duration) pairs by sequences of pitch values, such that the length of each sequence corresponds to the duration. For example the (pitch, duration) in the above example (72, 2) is modified to (72, 1) (72, 1). Table 2.8 shows the modified parts of the polyphonic piece based on shortest duration, and how the pattern matches different parts of the polyphonic musical piece. The value -1 in Table 2.8 represents rest value i.e. no note is played at that particular time. After the polyphonic piece is modified, the Shift-And algorithm is used for exact distributed matching.

Iliopoulos and Kurokawa [14] also considered the octave displacement problem (i.e. music played in a higher or lower octave than the original score). To do so, the pitch

Table 2.8: Matching query pattern in modified score

j	0	1	2	3	4	5	6	7
Part1	72	72	70	69	69	69	69	69
Part2	-1	67	67	67	-1	67	65	64
Part3	52	52	48	48	53	53	48	48
Pattern	**72**	**52**	**70**	**67**	**69**	**53**	**65**	**64**

values are modified to be the original pitch values modulo 12, (the number of semitones in an octave) and then add 1.(In western music, single octave is essentially made of 12

notes.) Rest values are not modified. So, a C note with a pitch value of 60 becomes 1 (i.e. 60 *mod* 12+1). The search pattern also is modified in a similar way. Table 2.9 shows the octave-modified pattern and the polyphonic music parts, and the shaded cells represent a match of the pattern. The complexity of the matching algorithm i.e. Shift-And is O (*mn*/*w*) where *m* is the length of the pattern and *n* is the length of the single part.

Table 2.9: Searching of query pattern in modified score

j	0	1	2	3	4	5	6	7
Part1	1	1	11	10	10	10	10	10
Part2	-1	8	8	8	-1	8	6	5
Part3	5	5	1	1	6	6	1	1
Pattern	**1**	**5**	**11**	**8**	**10**	**6**	**6**	**5**

Iliopoulos and Kurokawa also considered key transposed matching by doing the following modifications:

- Created 11 transposed patterns, by adding 1 to the pitch values of the original pattern, and then add 2 to the original pattern, and then add 3 to the original pattern and so on until 11. The rest value in the pattern is not changed. If pitch value exceeds 12, then subtract 12 from it.
- Then the matching process is performed with all the 12 patterns formed.

2.5 Myers' Algorithm

Gene Myers described an algorithm that can be used to find approximate occurrences of a pattern in text. The classic approach to this problem is to compute the dynamic programming matrix $D_{0\ldots m,\, 0\ldots n}$ that contains the edit distances between the text

and the pattern. (Edit distances are discussed in section 2.2.2.) The matrix is calculated using the following equations.

$$D_{0,j} = 0 \text{ and } D_{i,0} = i$$

$$D_{i,j} = \textbf{min}\{D_{i-1,j-1} + (\textbf{if } (P_i = T_j \textbf{ then } 0 \textbf{ else } 1), D_{i-1,j}+1, D_{i,j-1}+1\}$$

$$\text{for } 0 \leq i \leq m \text{ and } 0 \leq j \leq n$$

The solutions to the approximate string matching problem are all locations *j*-1in the text such that D [m, j] ≤ k (where *k* is allowed number of errors).The time complexity of this approach is O (*mn*) where *m* and *n* are the lengths of the pattern and the string, respectively.

For example, let us consider a text string "CAGAT" and a pattern "GATA"; the dynamic programming matrix of the two strings is shown in Table 2.10.

Table 2.10: Dynamic programming matrix

		0	*1*	*2*	*3*	*4*	*5*
			C	**A**	**G**	**A**	**T**
0		0	0	0	0	0	0
1	**G**	1	1	1	0	1	1
2	**A**	2	2	1	1	0	1
3	**T**	3	3	2	2	1	0
4	**A**	4	4	3	3	2	1

Ukkonen [22] showed that computing column *j* in a DP-matrix only requires knowing the values of the previous column *j-1*. This makes it possible to save space by storing only two columns at a time during DP-matrix calculation. Myers used the same concept in his algorithm, which computes the edit distances without first calculating a DP-matrix.

The DP-matrix has a property that the difference between adjacent entries in any row or any column is limited to 1, 0, or -1. Using this property, the following delta values are derived, for all (i, j) ϵ [1, m] × [1, n]:

vertical adjacency property $\Delta \mathbf{v_{i,j}} = \boldsymbol{D_{i,j} - D_{i-1,j}} \in \{-1, 0, 1\}$

horizontal adjacency property $\Delta \mathbf{h_{i,j}} = \boldsymbol{D_{i,j} - D_{i,j-1}} \in \{-1, 0, 1\}$

diagonal property $\Delta \mathbf{d_{i,j}} = \boldsymbol{D_{i,j} - D_{i-1,j-1}} \in \{0, 1\}$

Table 2.11, shows the Δv matrix (i.e. the matrix of $\Delta v_{i,j}$ values) for the dynamic programming matrix in Table 2.10. The values of D [m, j] (i.e. the edit distances) can be calculated using the summation $D[m, j] = \Sigma_{i=1\ldots m} (\Delta v_{i,j})$.

Table 2.11: $\Delta \mathbf{v_{i,j}}$ matrix

		C	A	G	A	T
	0	0	0	0	0	0
G	1	1	1	0	1	1
A	1	1	0	1	-1	0
T	1	1	1	1	1	-1
A	1	1	1	1	1	1

Myers' algorithm computes successive Δv_j's (i.e. the columns in the Δv matrix) using the horizontal, vertical and diagonal adjacency properties (i.e. the delta values). The Δv, Δh and Δd matrices are represented as boolean matrices, so that the computation of matrix values can be done using bit operations. The Δv matrix is represented using two boolean vectors VP and VN, Δh is represented using two boolean vectors HP and HN, and Δd matrix is represented using the boolean vector D0.The values of the VP,VN,HP,HN and D0 vectors are calculated using the following conditions:

VP[i] = 1 at text position j iff $\Delta v_{i,j} = 1$ (a.k.a. the vertical positive delta vector)

VN[i] =1 at text position j iff $\Delta v_{i,j} = -1$ (a.k.a. the vertical negative delta vector)

HP[i] =1 at text position j iff $\Delta h_{i,j} = 1$ (a.k.a. the horizontal positive delta vector)

HN[i] =1 at text position j iff $\Delta h_{i,j} = -1$ (a.k.a. the horizontal negative delta vector)

D0[i] =1 at text position j iff $\Delta d_{i,j} = 0$ (a.k.a. the diagonal zero delta vector)

In all other cases the values of the delta vectors are set to 0. Myers' proved that the delta vectors above have the following dependencies, and they are used in the algorithm to find the edit distances. (Proof of the dependencies is discussed in [14].)

$$D0_{i,j} \leftrightarrow P_i = T_j \mid VN_{i,j-1} \mid HN_{i-1,j}$$

$$HN_{i,j} \leftrightarrow VP_{i,j-1} \;\&\; D0_{i,j}$$

$$VN_{i,j} \leftrightarrow HP_{i-1,j} \;\&\; D0_{i,j}$$

$$HP_{i,j} \leftrightarrow VN_{i,j-1} \mid \sim (VP_{i,j-1} \mid D0_{i,j})$$

$$VP_{i,j} \leftrightarrow HN_{i-1,j} \mid \sim (HP_{i-1,j} \mid D0_{i,j})$$

Figure 2.11 shows the complete algorithm. There are two steps in the algorithm. The first step is to preprocess the pattern and the next step is to search the text for the pattern. In the preprocessing step, a bit-vector B[c] of length *m* is created for each different character *c* in the query pattern. The bit $B[c]_i$ (i.e. i-th bit in the bit vector) is set to 1 iff $P_{m-i+1} = c$; all other bits of B[c] are set to 0.

In the search phase, the bit-vectors D0, HN, HP, VN and VP are calculated using the dependencies among them for each text position *j* ($0 \leq j \leq n-1$). Instead of computing the dynamic programming matrix, the five bit-vectors described above are computed for each text position. The edit distance is calculated at each text position using the bit-vectors.

```
///Preprocessing of the pattern
        1   for c ϵ Σ do B[c] = 0^m
        2   for j ϵ 1….m do B[P_j]= B[P_j] | 0^(m-j)10^(j-1)
        3   VP = 1^m      VN=0^m
        4   Score = m
///Searching the pattern
        5    for pos ϵ 1….n do
        6     X = B [T_pos] | VN
        7     D0 = VP+(X&VP) ^ VP) | X;
        8     HN = VP & D0
        9     HP = VN | ~ (VP | D0)
        10   X = HP << 1
        11   VN = X & D0
        12   VP = (HN<<1 ) | ~ (X | D0)
//output
        13   if HP & 10^(m-1) ≠ 0^m
        14   then Score + = 1
        15   else if HN & 10^(m-1) ≠ 0^m
        16   then score -=1
        17              if score ≤ k report occurrence of the pattern
```

Figure 2.11: Myers' algorithm [14].

For example, consider the text T = 'CAGAT' and the pattern P= 'GATA'. In the preprocessing step, the bit-vectors for each character of the pattern are calculated. The bit values of the bit-vectors of all other characters (represented as *) are set to 0.

B[G] = 0001 B[A] = 1010 B[T] = 0100 B[*] = 0000.

Initially VP = 1^m and VN = 0^m. Table 2.12 shows the values of the vectors D0, HN, HP, VN and VP for each text position of the previous example. The row labeled *score* represents the edit distance.

Table 2.12: Search process of the Myers algorithm

	'C'	**'A'**	**'G'**	**'A'**	**'T'**
D0	0000	1110	0011	1110	1100
HN	0000	1110	0001	1110	1100
HP	0000	0000	0000	0001	0010
VN	0000	0000	0000	0010	0100
VP	1111	1101	1110	1101	1001
Score	4	3	3	2	1

The running time of the algorithm is O (mn/w). Distributed string matching (discussed in section 2.2.1.) is done using Myers' algorithm by modifying line number 6 in Figure 2.8 in the following way [14]:

$$X= \{ B[T_j^1] \mid B[T_j^2] \mid B[T_j^3] \mid \ldots\ldots B[T_j^h] \} \mid VN$$

where $T_j^1, T_j^2, \ldots T_j^h$ represent the text characters at position j in each of the text strings T^1 through T^h.

2.6 LCTS Algorithm

String P is a *subsequence* of string T if P can be derived from T by deleting zero or more characters from T. For example, the string '**abc'** has eight subsequences: **abc**, **ab**, **bc**, **ac**, **a**, **b**, **c**, and the empty string. A *common subsequence* of two strings is a subsequence that appears in both strings.

Given two strings $P_{1\ldots m}$ and $T_{1\ldots n,}$ the *longest common subsequence* (LCS) is the longest string that is a subsequence of both P and T. The Dynamic programming method generally is used to calculate the length of longest common subsequence denoted as LCS (P, T). The dynamic programming method computes a (m+1) × (n+1) matrix using the following recurrence relation, where $0 \leq i \leq m$ and $0 \leq j \leq n$:

$$LCS_{i,j} = 0 \qquad \textit{if } i=0 \textit{ or } j=0$$

$$LCS_{i,j} = 1 + LCS_{i-1,j-1} \qquad \textit{if } i, j > 0 \textit{ and } \boldsymbol{P_i = T_j}$$

$$LCS_{i,j} = \boldsymbol{max}\ \{LCS_{i,j-1}, LCS_{i-1,j}\} \qquad \textit{if } i, j > 0 \textit{ and } \boldsymbol{P_i \neq T_j}$$

$LCS_{m,n}$ evaluates to the length of the longest common subsequence. The complexity of the algorithm is O (mn). For example, consider the strings 'ABAB' and

'BABA' as shown in Table 2.13; the LCS value is calculated using the above recurrence relation, and the value of the LCS is 3.

Table 2.13: LCS Matrix

		A	**B**	**A**	**B**
	0	0	0	0	0
B	0	0	1	1	1
A	0	1	1	2	2
B	0	1	2	2	3
A	0	1	2	3	**3**

The length of the longest common subsequence (LCS) can be used to find the similarity between two melodies that are represented as string of pitch values (integers). The greater the LCS value is, then the better the similarity is between melodies. The LCS value will always be less than or equal to length of the pattern ($m \leq n$). Western people tend to listen to music by observing the intervals between the consecutive pitch values more than the actual pitch values themselves. For example, melody performed in two distinct pitch levels is perceived the same regardless if it's performed in a lower or higher level of pitches. This leads to the concept of *transposition invariance*. When LCS is used to find the similarity between two melodies, which are represented as string of pitch values (integers), transposition invariance is not considered. LCS calculation can be modified, so that transposition is considered while matching the melodies.

Let P and T be pattern and text strings with lengths *m* and *n* respectively, over a finite integer alphabet. The length of the longest common subsequence between P and T at transposition *c,* denoted as LCS^c (P, T) is to find the LCS value between T and P+c. (I.e. allowing P to be 'shifted' by adding some fixed integer *c* to the values of all

characters. Here characters are integers). The dynamic programming method generally is used to calculate LCS^c (P, T) at transposition *c*. The dynamic programming method computes a $(m+1) \times (n+1)$ matrix using the following recurrence relation, where $0 \le i \le m$ and $0 \le j \le n$:

$$LCS^c_{i,j} = 0 \qquad \textit{if } i=0 \textit{ or } j=0$$

$$LCS^c_{i,j} = 1 + LCS^c_{i-1,j-1} \qquad \textit{if } i,j > 0 \textit{ and } \boldsymbol{P_i + c = T_j}$$

$$LCS^c_{i,j} = \max \{ LCS^c_{i,j-1}, LCS^c_{i-1,j} \} \qquad \textit{if } i,j > 0 \textit{ and } \boldsymbol{P_i + c \neq T_j}$$

For example, let $\Sigma = \{0, 1, 2, 3, 4\}$, pattern P = '012' of length 3, and the text string T = '234'of length 3, the LCS value when no transposition is considered will be 1 but when a transposition of 2 is considered the LCS value is 3.

LCS^c (P, T) will calculate the LCS value at only one transposition value *c*. When a set of transpositions are considered instead of one value, then the maximum LCS value among all transpositions will give the *longest common transposition invariant subsequence* (LCTS). It is defined as follows [23]:

> Given a text string $T_{1\ldots n}$ of length *n* and the pattern $P_{1\ldots m}$ of length *m* ($m \le n$) over a finite integer alphabet $\Sigma = \{0\ldots \sigma\}$, a string $W_{1\ldots p}$ ($p \le m$) is a longest common transposition invariant subsequence of P and T, iff W is subsequence of P, W+c is subsequence of T (for some constant c, $-\sigma \le c \le \sigma$) and its length is maximal. W+c denote a constant adding to every character of string W. I.e. $W+c = W_1+c\ W_2+c \ldots W_m+c$. $[-\sigma, \sigma]$ is the set of transpositions.

The simplest technique to compute LCTS (P, T) is to compute LCS^c (P, T) for all *c* ($-\sigma \le c \le \sigma$) from the set $\{-\sigma\ldots\sigma\}$, and then choose the maximum among the calculated

LCS^c (P, T) values. This requires a triple iteration to compute $LCS^c_{i,j}$ for every $i \in \{0 \ldots m\}, j \in \{0 \ldots n\}$, and, $c \in \{-\sigma \ldots \sigma\}$, which takes O ($\sigma mn$) time.

Lemstrom and Navarro [23] described an algorithm to find the LCTS value in a better way than above using bit operations. In their algorithm, LCS^c (P, T) is computed for several *c* values simultaneously. The main concept in the algorithm is to compare individually on every (P_i, T_j) pair, but solve several transpositions simultaneously. The values of $LCS^c_{i,j}$ will be in the range $\{0...m\}$ where *m* is the length of the pattern ($m \leq n$). Therefore, to store each value $LCS^c_{i,j}$, $\lceil log_2(m) \rceil$ bits are required. In a single machine word of length *w*, *w*/ $\lceil log_2 (m) \rceil$ (represented as *k*) different values can be stored. This means that *k* values of *c* (transpositions) can be computed simultaneously. So, the process of computing LCS^c [P, T] for every $c \in \{-\sigma \ldots \sigma\}$ is divided into ($2\sigma+1$ / *k*) separate bit-parallel computations. Figure 2.12 shows the complete algorithm. The Time complexity of the algorithm is O (σmn *log(m)*/*w*).

Distributed string matching (discussed in section 2.2.1.) is done using LCTS algorithm by modifying the *for* loop in the line 3 of Figure 2.12. So the modified code (from line 3-7) will be:

For h=0; h<max_distribution; h++

If i=0 | j=0 then $Lcs_{i,j} = 0^{A(length+1)}$

Else

int val = $\boldsymbol{T_j^h - P_i}$

If C ≤ ***val*** *< C+A Then*

$\boldsymbol{B \leftarrow B \mid (0^{(A+C-1-(val)\ (length+1)}\ 1^{(length+1)}\ 0^{(val\ -c)(length+1)})}$

Else $B \leftarrow B \mid 0^{A(length+1)}$

End for

where $T_j^1, T_j^2, \ldots T_j^h$ represent the text characters at position *j* in each of the text strings T^1 through T^h. The complexity is O (hσmn *log(m)/w*).

```
// Bit parallel computation of LCS

   LCTS (P,T,C,A,length)
 1                  For i ϵ 0 ….. |P| Do
 2                       For j ϵ 0 …. |T| Do
 3                              If i=0 | j=0 then Lcs i,j  = 0^A(length+1)
 4                              Else
 5                              If C ≤ Tj-Pi < C+A Then
 6                               B ← 0^(A+C-1-(Tj-Pi) (length+1)) 1^(length+1) 0^(Tj-Pi -c)(length+1)
 7                              Else   B ← 0^A(length+1)
 8                  Lcs i,j ← (B& (Lcs i-1,j-1 + (0^length 1)^A ) | (~B & Max(Lcs i-1,j ,Lcs i,j-
1)
 9       Return Lcs |P|, |T|.


// dividing into different set of transpositions.


     RangeLcts (P, T, σ)
10            Length ← Ceil(log2 (min(|P,||T|)+1))
11            A ← floor( w/(length+1)
12           C ← - σ
13           Max ← 0
14           While c ≤ σ
15                     V ← RangeLcts(P,T,c,A,length)
16                     For t ϵ c…c+A-1 Do
17                Max ← max (Max, (V >> (t-c) (length+1) & 0^((A-1) (length+1)) 01^length
18                 c← c+ A
         Return Max
```

Figure 2.12: Algorithm for calculating Longest Common Transposition invariant subsequence [22].

CHAPTER 3

SYSTEM ARCHITECTURE

The process of retrieving music by humming starts with the query formulation in the front-end through microphone and it leads to the presentation of the best match of music in the database [24]. Figure 3.1 shows how this process is modeled.

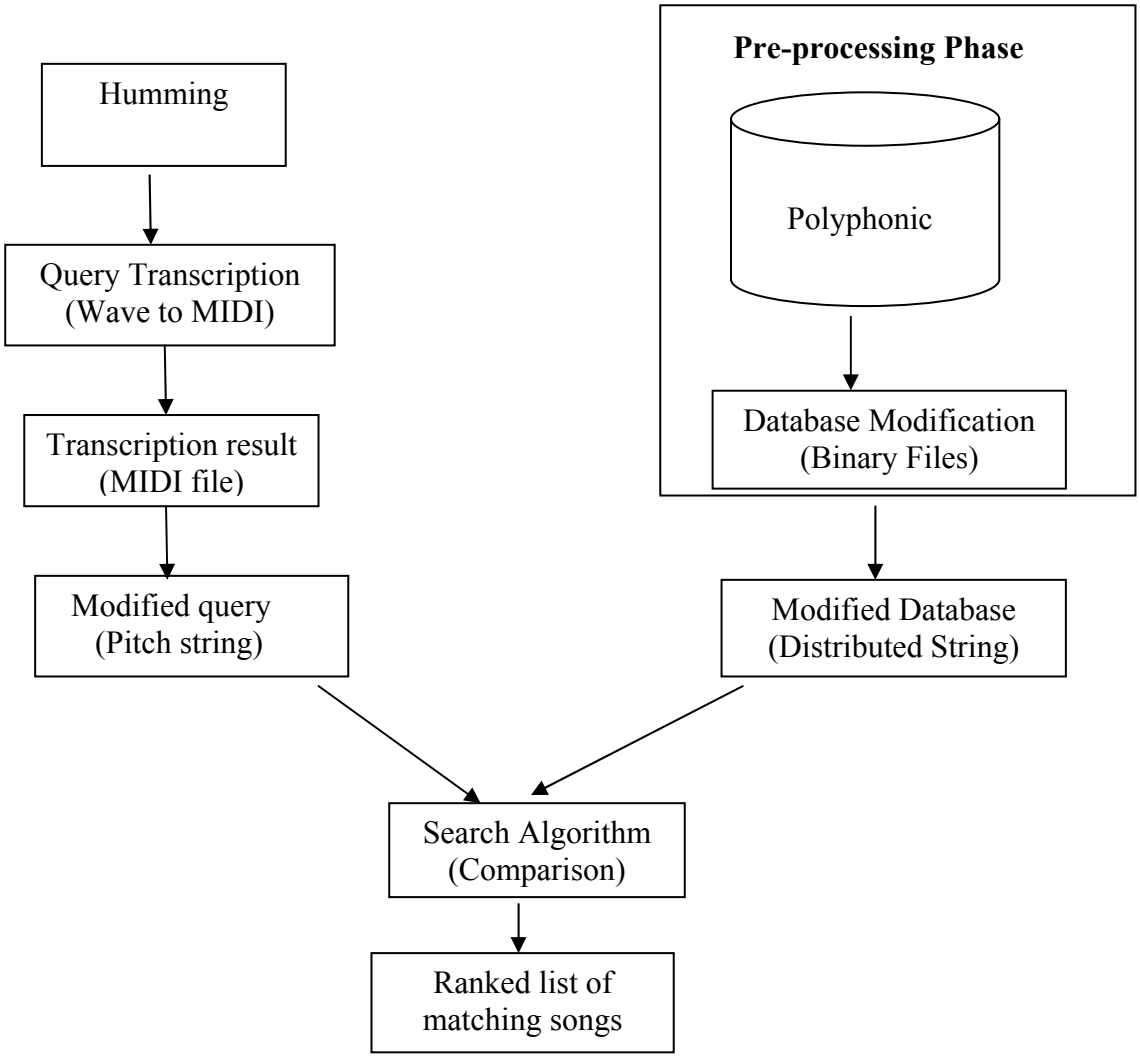


Figure 3.1: Architecture of humming song retrieval in music database.

The humming song retrieval from the database mainly is divided into two stages; preprocessing stage and the matching stage. In the preprocessing stage, the MIDI database files are converted into the binary files. These binary files only contain the information about the notes (discussed in section 3.2). In the matching stage, the following steps are performed:

- Humming is captured as wave file and then converted to MIDI.
- Both the humming tune (MIDI) and database (binary files) are converted to strings.
- Finally the similarity between the hummed tune and the database (both in string format) is calculated.

3.1 Humming and Query Transcription

When the users are searching for musical data, either they know what they want, e.g., song title, artist or genre, where a traditional text-based search is enough, or they want to find musical data based on content similarity. In the case of searching by content similarity, it must be possible for the users to build their queries in an intuitive way. One of the most intuitive ways to find a song in music database is to hum a part of the song (query-by-humming).

The accuracy (i.e. correct match for the tune) of the music retrieval process depends on the quality of the hummed tune. Recording the hummed tune in a relatively noise free environment will reduce the amount of the noise in the humming. In this research 'Sound Recorder' in the Microsoft Windows XP operating system is used for recording the hummed tune, and the hummed tune is captured using a microphone in a

relatively noise free environment, and stored as a wave file. Similarity between the query tune and music stored in the database is calculated using approximate string matching techniques. Therefore, a hummed tune in wave format is to be transformed into a representation that is appropriate for similarity measurement. For this purpose the hummed query saved in wave format needs to be converted to MIDI format [25].

Converting from wave format to MIDI format is done by capturing the discrete pitch information from the wave file [26][27]. The process of capturing pitch information accurately from humming is difficult, even if the user manages to hum perfectly. The performance of currently existing software for converting raw audio data into discrete pitch information is mediocre at best and often will introduce a great deal of noise when extracting the pitches from a user's hum [26]. Many commercial software's are available that convert wave format to MIDI but they fail to convert humming tune which is in wave format to MIDI accurately. This is because humming tune generally is not so clear.

In this research *Solo Explorer* software is used for converting humming stored in wave format to MIDI format [26]. The accuracy of the *Solo Explorer* software is better in the case of humming tune as input [26]. Once the hummed MIDI file is generated as a result of transcription, required information for the comparison purpose is extracted from the file and all other information is discarded. MIDI file is comprised of many events and among them Note-on, Note-off, and tempo events are extracted which contains all information regarding notes and duration of these notes. All events in the MIDI file are followed by timestamps, represented in time units. These time units will give the duration of that particular event. Using this information, MIDI file is converted to a set of notes

where each note is comprised of its start-time, pitch value and duration. The set of notes are sorted based on the Note-on times.

These set of notes are converted into string of pitch values, since approximate string matching is used to find the similarity between the hummed tune and MIDI database. Conversion into strings is done using the following steps.

- First shortest duration is extracted from the set of notes.
- Using this shortest duration, time periods are formed. Time periods are the intervals of times, which are multiples of shortest duration. The pitch (integer value) that is played during that time period is used to form the string.

For example, consider first few notes after sorting formed from the hummed query for the "Happy Birthday" song in the tabular form as shown in Table 3.1. The shortest duration is 240. Using this shortest duration the time periods formed, which are multiples of shortest duration, will be 0-240,240-480,480-720,720-960 and so on. During these time periods, the pitch values of note played are 47,51,49,49 and so on.

Table 3.1: (note-on, pitch, duration) pairs

Note-on	Pitch	Duration
0	47	240
240	51	240
480	49	480
960	47	480
1440	54	960
2400	52	480

Table 3.2: String representation

Duration periods	Pitch (String)
0-240	47
240-480	51
480-720	49
720-960	49
960-1200	47
1200-1440	47
1440-1680	54
1680-1920	54
1920-2160	54
2160-2400	54
2400-2640	52
2640-2880	52

Table 3.2 contains the total time periods and pitch values of the set of notes. Pitch column in Table 3.2 represent the string formed. The string formed from the set of notes is '*47 51 49 49 47 47 54 54 54 54 52 52'.*

In the above example, the duration of notes is exact multiple of shortest duration but this may not be the case always. In that case, if the note is played for more than half of the duration, of that particular period, the pitch value is considered otherwise it is discarded. For example consider the set of notes of the form (Note-on, pitch, duration) be { *(0,52,240) (240, 50, 420) (660,54,300) , (960,52,240)* }. The shortest duration among the set of notes is 240. The duration periods will be [0-240], [240-480], [480-720], [720-960], [960-1200]. In the first duration period [0-240], pitch 52 is considered. In the second duration period [240-480], pitch 50 is considered. In the third duration period [480-720], two pitches should be considered, pitch 50 for duration of 180, and pitch 54 for duration of 60. Since the duration of pitch 50 is greater than half of the duration period it is considered and pitch 54 is discarded. In the fourth duration period [720-960] pitch 54 is considered and in the fifth duration period [960-1200] pitch 52 is considered. The entire string formed will be '52 50 50 54 52'.

3.2 Database files modification

Database used in this research is comprised of both monophonic and polyphonic MIDI files. Database modification is done in two steps. The first step is performed during preprocessing phase and the second step is performed during the matching phase. In the first step, each MIDI file in the database is modified similar to that of the hummed query MIDI file (i.e. conversion from MIDI file to set of notes.). Each set of notes is comprised

of its start-time, pitch value and duration, and these notes are sorted based on the Note-on times. These sorted notes are written into a binary file. In this way all the database MIDI files are converted to binary files. This conversion from MIDI file to binary file does not depend on the hummed query; hence it can be done before the actual matching phase.

In the second step (matching phase), from each of the binary file the set of notes are extracted. These set of notes are converted into sequences of pitch values, similar to that of hummed query modification. The shortest duration of a note among the set of notes of the query is used for calculating the time periods. The database contains polyphonic music files, so the number of notes played during a particular duration period can be more than one.

For example, consider a sorted set of notes base on Note-on, for a database binary file, in (start time, pitch, duration) format be:

{(0, 50, 2) (0, 63, 2.2) (1, 48, 2.6) (2.1, 60, 3) (2, 65, 2.1) (4, 62, 2) (5, 72, 1.4) (5, 71, 2)}

Let the hummed query set of notes be:

{(0, 50, 1.2) (1.2, 63, 1) (2.2, 65, 1.8) (4, 62, 2) (6, 71, 1)}

To modify the set of notes in the database, first the shortest duration of a note in the query is required; from the above example the shortest duration in the query is 1. Therefore the time periods will be [0-1], [1-2], [2-3], [3-4], [4-5], [5-6], [6-7]. During the first time period [0-1], both the pitches 50 and 63 are considered; in the second time period [1-2] pitch values 50, 63, 48 are considered. In the third time period, since the pitch value 63 is only for 0.2 which is less than the half of duration, the period is discarded. In this way all the sequences of strings are calculated. Table 3.3 shows the database modification into set of strings and how the query matches it.

Table 3.3: Database string representation

Time / Polyphony	[0-1]	[1-2]	[2-3]	[3-4]	[4-5]	[5-6]	[6-7]
1	50	50	48	48	60	62	71
2	63	63	60	60	62	72	--
3	--	48	65	65	--	71	--
Query	**50**	**63**	**65**	**65**	**62**	**62**	**71**

In the above example the *degree of polyphony* i.e. maximum number of notes played at the same time is 3.

3.3 Problems in Music Retrieval

After the hummed query and the database files are transformed to the pitch value strings, similarity between them is calculated. In this research Myers algorithm and the LCTS algorithm are used to find the similarity. Myers algorithm output will be the edit distance, and the LCTS algorithm output will be the length of longest common subsequence with transposition invariance. There are many problems to be considered in the music retrieval process; distributed matching, transposition invariance and octave equivalence are some of them. The algorithms are modified accordingly to consider all this problems.

3.3.1 Transposition Invariance and Octave Equivalence

Transposition in music means playing or writing music in a different key i.e. to change the pitch of each note without changing the relationships between the notes. Transposing a melody up or down by one octave will not change the key. Transposing is

a useful skill for people who play an instrument, especially the piano or organ. If a pianist is accompanying a singer and the song is a little too high for the singer's voice it is very useful if he is able to transpose it down so that the music sounds in a lower key. Two musical objects are transposition ally equivalent if one can be transformed into another by transposition. For example, Figure 3.2 shows the first few notes of "Jingle Bells" played in the key of F# and then played in key C i.e. 5 semitones away

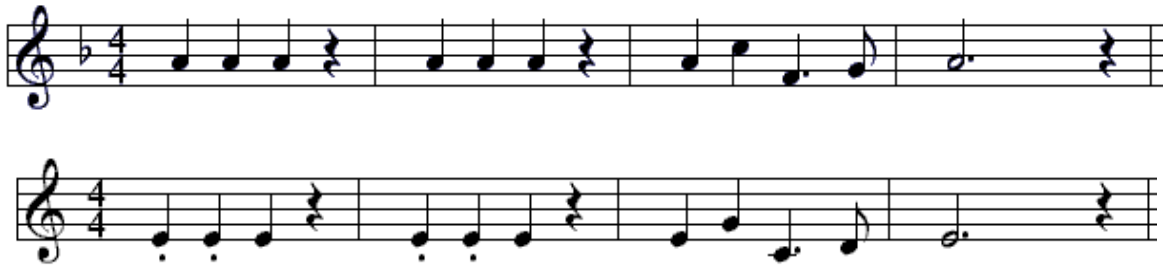


Figure 3.2: Jingle bells song played in F# and C key

The hummed tune may be in a different key (i.e. it may start on different note or a few notes off-pitch throughout the course of the hummed tune.) to than that of stored version in the database. In this case during the calculation of similarity this should be considered. During the similarity calculation, both the hummed query and database MIDI files are converted into strings of pitch values. When there is a transposition in the query, pitch values will be in a different key, i.e. all pitch values are transposed by certain amount.

Let P be a pattern string formed from hummed tune and let T be a set of text strings formed from one of the database MIDI files. Both pattern and text strings are formed from an alphabet, which consists of different pitch values. In MIDI different pitch values are 0-127. So the alphabet size σ is 128. When the pattern is transposed by amount *c* (c is integer), means each character in P is added with an integer *c*. Possible

transpositions in a pattern, i.e. *c* values are {-127 …. 127}.LCTS algorithm calculates the length of common subsequence between P and T with different transpositions in parallel using the bit operations (Discussed in section 2.6.). The complexity of this algorithm is O (*σmn log(m) /w*), where *σ* is the alphabet size. In this case, it is 128.

During the transcription from wave to MIDI of humming tune, sometimes the pitch values extracted from wave file will have a little error [23]. For example, instead of pitch value 65 it will be pitch value 64 or 63, i.e. an error of 1. In music matching, this error allowance should be considered. To consider this case, δ-matching is used while calculating the similarity. Consider two integer strings $A_{1...m}$ and $B_{1...m,}$ then these strings are said to be δ-matched if $A_i \in [B_i-\delta, B_i+\delta]$ for all $1 \leq i \leq m$. For example, string '1234' matches '3456' exactly when the error allowed is 2. Here the characters in the strings are assumed to be {1, 2, 3, 4, 5, 6}.

Longest common subsequence between two strings with δ-matching can be calculated using the dynamic programming method. The dynamic programming method computes a $(m+1) \times (n+1)$ matrix using the following recurrence relation, where $0 \leq i \leq m$ and $0 \leq j \leq n$:

$$LCS^c_{i,j} = 0 \qquad \textbf{if } i=0 \text{ or } j=0$$

$$LCS^c_{i,j} = 1 + LCS^c_{i-1,j-1} \qquad \textbf{if } i, j > 0 \text{ and } P_i \in [T_j-\delta, T_j+\delta]$$

$$\textbf{else } LCS^c_{i,j} = \max \{ LCS^c_{i,j-1}, LCS^c_{i-1,j} \} .$$

LCTS algorithm can be modified so that the algorithm considers both the δ-matching and transposition invariance [23].

Myers algorithm doesn't consider transposition during the calculation of the edit distance. But, transposition can be achieved by modifying the pattern for every different

possible transposition, and then running the actual algorithm for different patterns formed. Therefore by considering transposition, Myers algorithm complexity will be increased by σ times more than the original complexity, which will be O ($\sigma mn/w$).

In music, an octave is the interval between one musical note and another with half or double its frequency and is the point where the most aesthetically related pitches harmonize most closely. For example, if one note has a frequency of 200 Hz, the note an octave above it is at 400 Hz, and the note, an octave below it is at 100 Hz. The ratio of frequencies of two notes an octave apart is therefore 2:1. Further octaves of a note occur at 2^n times the frequency of that note (where n is an integer), such as 2, 4, 8, 16, etc. and the reciprocal of that series. Octave equivalence partitions the notes into twelve equivalence classes (Table 2.1). The user may have less memory about the melody, so the humming can be one octave higher or lower to that of the original melody and also the transcription of the query from wave to MIDI format may cause the same problem. Since the comparison between the query and database is done based on the actual pitch values, when the query is an octave apart, the similarity values will be affected if it is not considered.

During the similarity calculation using the algorithms, octave equivalence is nothing but the transposing of a pattern string with values that are multiples of 12. That is the transposition c can have the value {…-48,-36,-24,-12, 0, 12, 24, 36, 48 …}. Similarity is calculated considering all the possible pitch values i.e. {-127…127}, octave equivalence transpositions will only be the subset of the total transposition values. Therefore no modification is required to the algorithms to consider octave equivalence.

3.3.2 Distributed Matching

Music in the database can be both monophonic and polyphonic. In polyphonic music, more than one note may be sounded simultaneously. So the pitch string formed after modifying the database polyphonic MIDI file will be set of equal length strings. The query is monophonic; it is converted into single string of pitch values. This pitch string is matched against the collection of strings. This matching is called distributed matching (Discussed in section 2.2.1). Gene Myers' algorithm and LCTS algorithm both deal with distributed matching. Using both the algorithms the similarity values are calculated.

CHAPTER 4

EXPERIMENTAL EVALUATION

4.1 Testing Interface

For testing purposes, the interface as shown in Figure 4.1 is implemented.

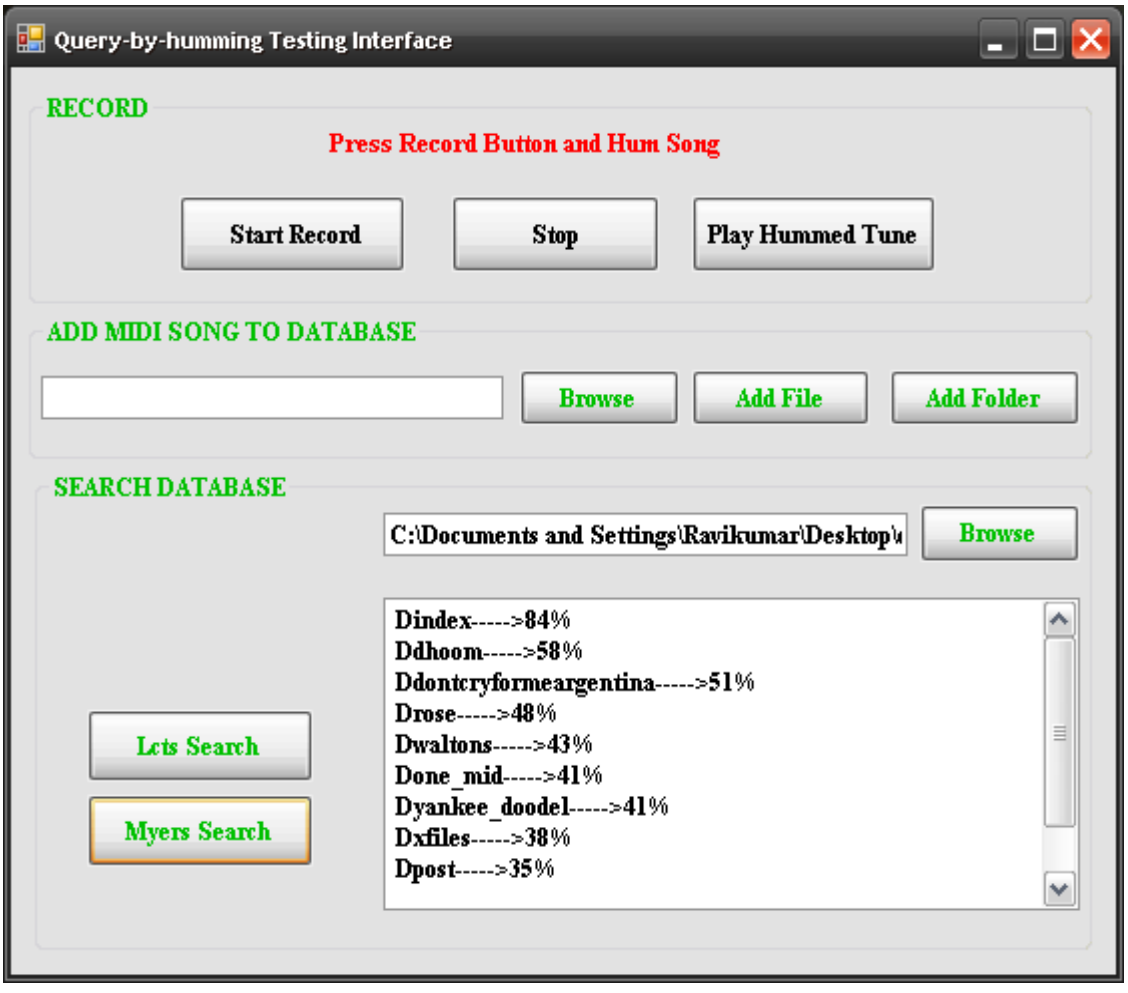


Figure 4.1: Testing Interface

The testing interface, MIDI file processing, and the algorithms are implemented in Visual C++.The testing interface mainly performs three steps.

- Ability to record the humming
- Adding MIDI files to the database
- Searching the database to find similar musical files to that of a hummed tune

The testing interface provides flexibility to hum the tune using a microphone. 'Start Record' and 'Stop' buttons are used to start and stop the recording of the humming respectively. By pressing the 'Play Hummed Tune' button, the hummed tune that is recorded can be played. If the humming is not good (i.e. hummed tune has noise, etc), humming can be recorded again using the 'Start' and 'Stop' buttons. Recording and playback of the humming are implemented using Windows API function mciSendString. This function sends the appropriate command string ('play', 'record' etc) to an MCI device (multimedia control interface). The hummed tune is saved in PCM wave format with sampling rate of 44.1 khz, single channel (mono), bit resolution of 16 bits. The wave format is converted to MIDI using the Solo Explorer software.

Adding MIDI files to the database can be done using the interface. The MIDI file that is to be added to the database can be selected using the 'browse' button or by typing the path of the file in the respective textbox. Once the file is selected, by clicking the 'Add file' button, the MIDI file is converted to a binary file (discussed in section 3.2.) by extracting the note information. Using the 'Add Folder' button all MIDI files in a particular folder can be added to the database.

Searching the database for the original MIDI file of the hummed tune is done by finding the similarity between the hummed tune and each MIDI file in the database. Both LCTS and Myers algorithms can be used to find the similarity. As shown in the interface, using 'LCTS' and 'Myers' buttons, respective algorithms are used to calculate similarity. Output of the algorithm contains the sorted similarity values and among them, the top 10 MIDI files (Figure 4.1) are displayed with file names and percentage of similarity.

All the experiments in this research were run on a computer with Intel Pentium IV processor 1.86 GHz, 1GB RAM under the Windows XP operating system. The length of the machine word is 64 bits. The alphabet used in the algorithms is of size 128, and is comprised of MIDI pitch values {0…127}. Parameters to be considered in all experiments are:

- Average length of the database MIDI file after modification in string format (**n**)
- Average degree of polyphony of database files. (**h**)
- Length of the hummed query MIDI file in string format (**m**)
- Total transpositions considered. (**σ**)

4.2 Retrieval accuracy analysis

In this experiment, retrieval accuracies of both the algorithms are compared. The database in this experiment contains 250 polyphonic musical pieces in MIDI format. These musical pieces are comprised of folksongs, rhymes, classical, and Beatles [29][30][31]. The average degree of polyphony (i.e. number of notes playing at a particular time period.) of musical pieces in the database is four. A total of 150 humming samples are used in this experiment. Among the 150 samples, 50 samples are collected

from three different users, using the testing interface. The remaining 100 humming samples are collected from Erdem Unal and S.S. Narayanan [28]. The average length of the humming samples (m) is 45. All possible transpositions (σ) i.e. 256 are considered during similarity calculation. LCTS algorithm also considers the δ-matching with (δ=1) during similarity calculation. Figure 4.2 shows the percentage of humming samples that produces the correct target MIDI file in the database using both the algorithms. The LCTS algorithm returns correct result within the top 5 with an accuracy of 61%, and the Myers algorithm with an accuracy of 56%. LCTS algorithm performs better in terms of accuracy; one of the reasons for it is δ-matching, which allows slight distortion in pitch value during the similarity calculation. In Figure 4.2 the top 1, top 3, top 5 and top 10 results are shown.

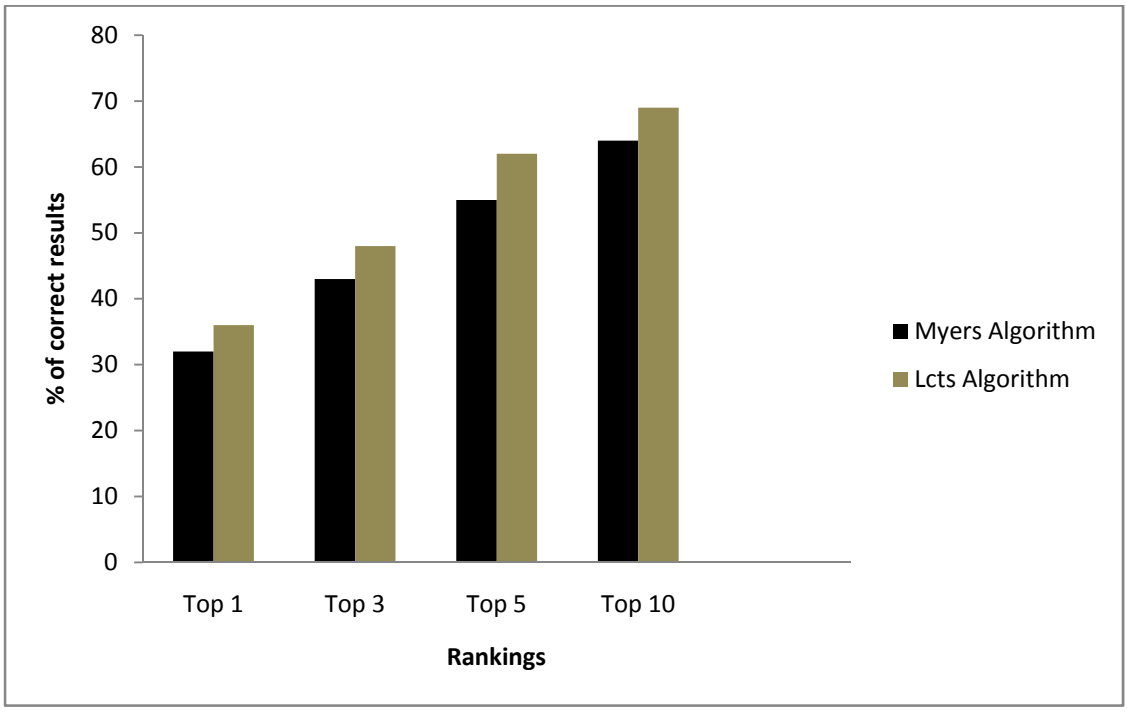


Figure 4.2: Retrieval rankings of the humming samples.

The response time (i.e. the time taken for the similarity calculation) is calculated for each of the humming sample for both the algorithms. Using these response times, the average response time is calculated for this experiment. The average response time for Myers algorithm is 250 milliseconds, and LCTS algorithm is 340 milliseconds.

The same experiment is repeated by considering only the transpositions of two octaves higher, and two octaves lower. In this case, 48 different possible transpositions should be considered. The retrieval accuracy changes < 2% in both algorithms. Figure 4.3 shows the retrieval accuracy of both algorithms. The average response time of the Myers algorithm reduces to 75 from 250 milliseconds. The average response time for LCTS algorithm reduces to 145 from 340 milliseconds.

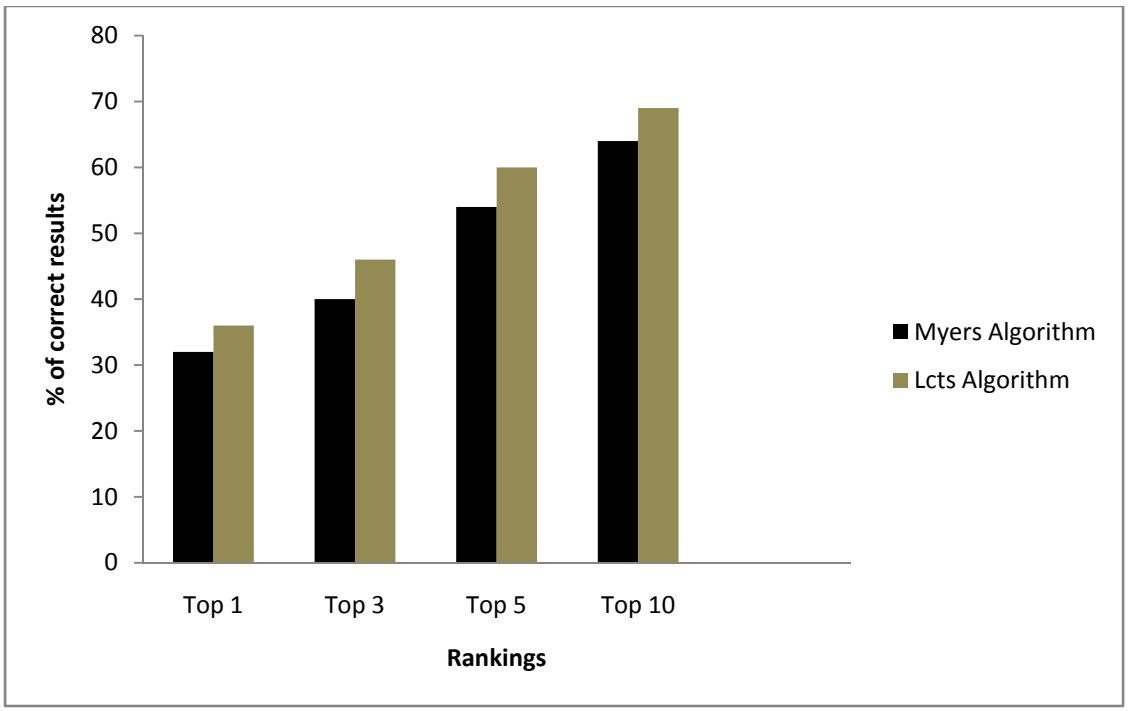


Figure 4.3: Retrieval of humming samples considering fewer transpositions

4.3 Response Time

Response time is the time taken for calculating the similarity values between the query MIDI file and the files in the database. In this section the response time of both the algorithms are compared by altering the values of some of the interesting parameters. The parameters considered are the database size, query length (m) and average degree of polyphony of database files. Response times are measured by varying one parameter at a time, and using a fixed value for other parameters. The results for each experiment were averaged over 20 repetitions to smooth out any small variations that occur. For the purpose of this experiment, 500 MIDI files are collected from internet which includes folksongs, rock songs, and ringtones.

In this experiment, variation in the response time is observed when the size of the database changes. Different size datasets {100, 200, 300 …} are created from the MIDI files so that the average length of the database MIDI file in string format is about 250 and average degree of polyphony is 3. The response times for different datasets are recorded for a query of length 45. Figure 4.4 shows the response time variation of the two algorithms with different database sizes. The x-axis of the graph indicates the size of the database and the y-axis indicates the response time in milliseconds. As shown in the Figure 4.4 the response time increases with the database size; however, the increase in Myer's algorithm is less when compared to LCTS Algorithm.

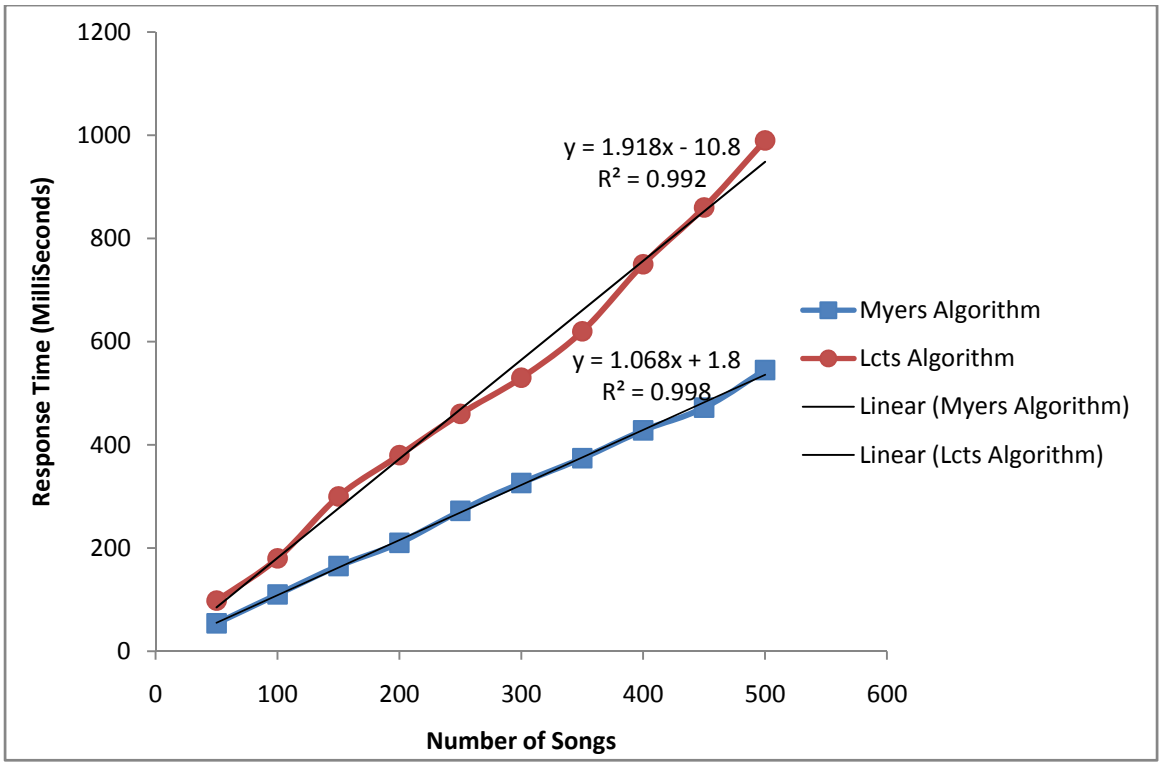


Fig 4.4: Response time comparison with database size

The response time is often affected by the length of the query. In this section response time is studied by varying the length of the query on a fixed database. The database in this experiment consists of 250 MIDI files. The average length of the database files in string format is 250, and the average degree of polyphony is 3. Figure 4.5 shows the variation of response time with query length for both the algorithms. The x-axis of the graph indicates the length of the query and the y-axis indicates the response time in milliseconds. For smaller query length, LCTS algorithm has less response time than the Myers algorithm. But as the query length increases, the Myers algorithm has a better response time.

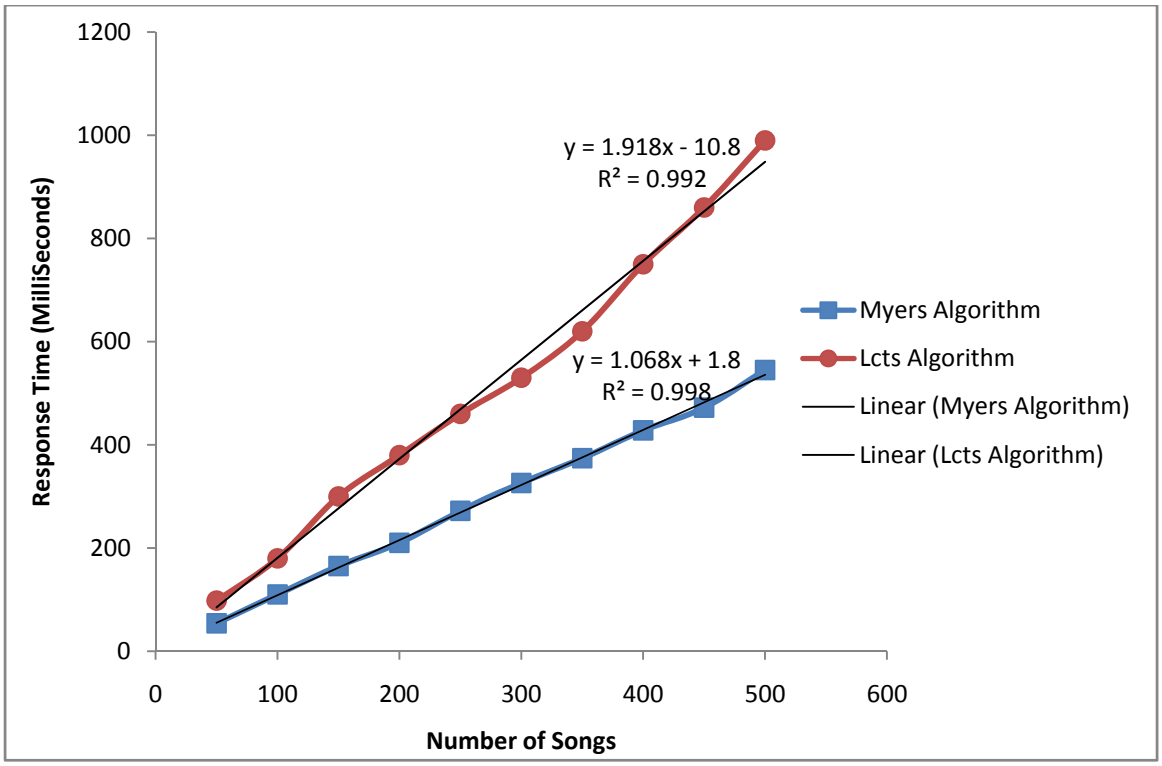


Fig 4.5: Response time comparison with query length

The variation of the response time with average degree of polyphony is considered here. If the database contains only monophonic MIDI files then the average degree of polyphony will be 1. To observe the variation datasets are created from the MIDI files each containing approximately about 50 MIDI files. Each dataset vary in the average degree of polyphony. Response times of both the algorithms are recorded on different datasets. Figure 4.5 shows how both algorithms vary with the degree of polyphony. Myers algorithm is independent on the degree of polyphony, so the response time doesn't change to great extent as degree of polyphony increases. LCTS algorithm approximately varies linearly with the increase in degree of polyphony.

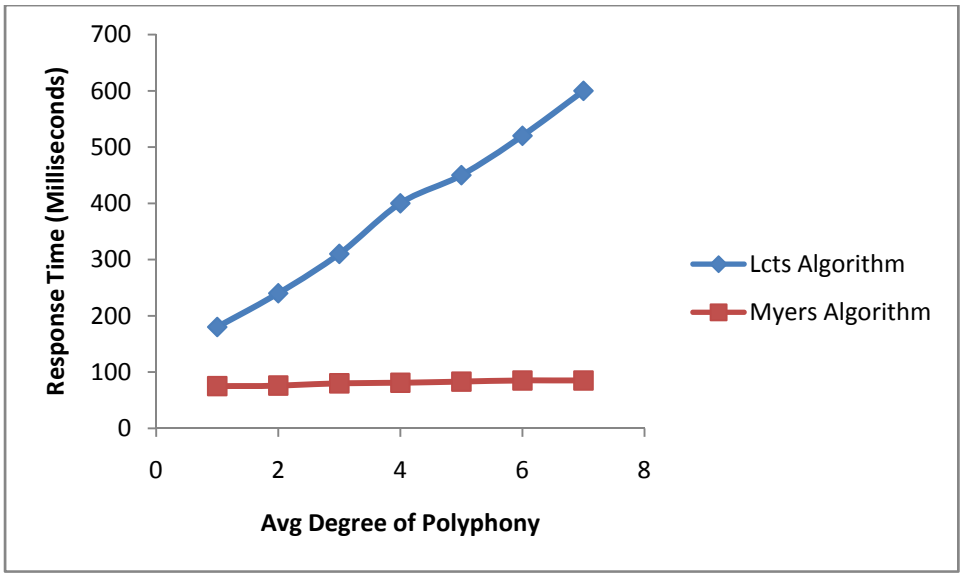


Fig 4.6: Response time comparison with Average degree of polyphony

CHAPTER 5

CONCLUSION

5.1 Summary

In spite of the growth of digital music libraries, or generically speaking, any multimedia database, it is really useful only if users can find what they are seeking in an efficient manner. Query-by-humming is one of the efficient ways of retrieving musical information. Finding the best matching database target to a melodic query has been of great interest in the music information retrieval world**.** Most of the research in music retrieval is focused on monophonic music. However, most music is polyphonic with multiple notes playing the same time. This research is focused on content-based music information retrieval from a polyphonic MIDI music database using a monophonic query. Pitch and duration features of a note are used for music retrieval purpose.

Approximate string matching techniques are used for calculating the similarity between the hummed tune and the database with polyphonic MIDI music files. Two bit-parallel approximate string matching algorithms Myers and LCTS are adapted to the context of music information retrieval in polyphonic database. Modifications are done to the algorithms in order to consider the distributed matching, transposition invariance and octave displacement. Retrieval accuracies of both the algorithms are compared on database of 250 polyphonic MIDI files, using 150 humming samples.

LCTS algorithm returns the correct result within the top 5, 61% of the time, and Myers algorithm 54% of the time. LCTS algorithm is better in terms of accuracy; one of the reasons is δ-matching used in the algorithm which allows slight distortion in pitch value during the query transcription from wave to MIDI format. Response times are calculated by varying different parameters like the query length, degree of polyphony and size of the database. Even though Myers algorithm performs better than LCTS, the response time of both algorithms is less than 1 sec on a database of 250 songs.

5.2 Future Extension

In this research, the conversion of humming tune (wave format) to MIDI is performed using third party software. As a future extension, this can be built into the current framework the conversion using the pitch tracking algorithms. Moreover, it would be interesting to convert the framework to be accessible remotely. This allows the user to hum the query remotely (client-server architecture) and retrieve the similar songs from music a database.

REFERENCES

1. A. Ghias, J. Logan, and D. Chamberlin. Query by Humming. In Proceedings of ACM *Multimedia 95,* pages 231-236, November 1995.
2. MIDI Manufacturers Association. The complete MIDI 1.0 detailed specification, 1996.
3. Boyer R.S., Moore J.S**.** A fast String Searching Algorithm. Communications of the *ACM*. 20:762-772, 1977.
4. Richard M. Karp, Michael O. Rabin. Efficient randomized pattern-matching algorithms. IBM Journal of Research and Development 31 (2), 249-260, 1987.
5. R. A. Baeza-Yates, G. H. Gonnet. A New Approach to Text Searching. Communications of the ACM 35(10), 74-82 , 1992.
6. K. Lemstrom, J. Tarhio. Detecting monophonic patterns within polyphonic sources. In Proceedings of Content-Based Multimedia Information Access Conference Proceedings, 2000.
7. Tetsuya Kageyama, Kazuhiro Mochizuki, and Yosuke Takashima. Melody Retrieval with Humming. In Proceedings of the International Computer Music Conference, pp.349-351, September 1993.
8. Smith, L., & Medina, R. Discovering themes by exact pattern matching. In Proceedings of the Second Annual International Symposium on Music Information Retrieval 2001. pp. 31-32 2001.

9. Needleman, S. B. and C. D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino Acids Sequences of Two Proteins, Journal of Molecular Biology 48:443-453, 1970.
10. M. Mongeau and D. Sankoff. Comparison of musical sequences. Computers and the Humanities, 24:161–175, 1990.
11. R. J. McNab, L. A. Smith, I. H.Witten, C. L. Henderson, and S. J. Cunningham. Towards the digital music library: Tune retrieval from acoustic input. In Proceedings First ACM Conf. on Digital Libraries, pages 11–18, Bethesda, MD, USA, 1996.
12. Lemstrom, K. and S. Perttu. SEMEX - An efficient Music Retrieval Prototype. In Proceedings of International Symposium on Music Information Retrieval 2000.
13. S. Wu and U. Manber. Fast text searching allowing errors. Communications of the ACM v.35 n.10, pages 83-91, Oct. 1992.
14. E. Myers. A fast bit-vector algorithm for approximate string matching based on dynamic programming, J.Assoc Comput.Mach 46, 3, 395-415(1999).
15. C. S. Iliopoulos and M. Kurokawa. Exact and Approximate Distributed matching for musical Melodic recognition 2001.
16. L. Prechelt and R. Typke. An interface for melody input. ACM Transactions on Computer-Human Interaction, 8(2):133–149, 2001.
17. B. Pardo and M. Sanghi. Polyphonic Music Sequence Alignment for Database Search. International Conference on Music Information Retrieval. 2005.

18. Roger Dannenberg, Ning Hu. Understanding search performance in Query-by-humming Systems. In Proceedings of the International Symposium on Music Information Retrieval 2004.
19. A. Uitdenbogerd and J. Zobel. Melodic Matching Techniques for Large Music Databases, in Proceedings of the ACM Multimedia Conference 1999.
20. J.Pickens. A survey of feature selection techniques for Music Information Retrieval. In Proceedings of 2nd International Symposium on Music Information Retrieval 2001.
21. A.L. Uitdenbogerd**,** and J. Zobel**.** An architecture for effective music information retrieval, Journal of the American Society for Information Science and Technology, 55(12), 1053-1057, 2004.
22. Ukkonen, E. Finding approximate patterns in strings. *J. Algorithms 6*, 132–137, 1985.
23. K. Lemstrom and G. Navarro. Flexible and efficient bit-parallel techniques for transposition invariant approximate matching in music retrieval. In Proceedings 10th International Symposium on String Processing and Information Retrieval 2003.
24. C. Yang. Peer-to-peer architecture for content-based music retrieval on acoustic data. In Proc. World Wide Web Conference, pages 376–383, 2003.
25. Sangbo Park, Suckchul Kim, Eenjun Hwang and Kwanjung Byeon**.** Automatic Voice Query Transformation for Query-by-Humming Systems, Proceedings of the Ninth International Conference on Internet and Multimedia Systems and Applications, pages 197-202, 2005.

26. Valeriy Lobaryev, Gene sokolov, Alexandr Gordyeve. Sloud Query-by-Humming Search Music Engine 2006.
27. W.Chai and B. Vercoe. Melody retrieval on the web. In Proceedings of Multimedia Computing and Networking 2002.
28. Erdem Unal, Shrikanth Narayanan, Maverick H.-H. Shih, Elaine Chew, and C.-C. Jay Kuo. Creating Data Resources for Designing User-centric Front-ends for Query by Humming Systems. ACM Multimedia Systems Journal, Special Issue on Music Information Retrieval, 2005.
29. Free polyphonic ringtones. http://www.free-nokia-ringtones.uk.com/nokia-ringtones-midi-A.html. Last accessed February 20 2008.
30. Free MIDI File database. http://www.mididb.com/. Last accessed February 20, 2008.
31. Free MIDI files. http://www.8notes.com/midi/. Last accessed February 20,2008

VITA

Ravikumar Nidadavolu

Candidate for the Degree of

Master of Science

Thesis: CONTENT-BASED RETREIVAL OF MUSIC USING MONOPHONIC QUERIES ON A DATABASE OF POLYPHONIC, MIDI INFORMATION.

Major Field: COMPUTER SCIENCE

Biographical:

Personal Data: Born in Tanuku, Andhra Pradesh, India on August 8,1983

Education:
Received B.Tech in Information Technology from the Andhra University, Vizag, Andhra Pradesh, India in 2004.
Completed the requirements for the Master of Science with a major in Computer Science at Oklahoma State University, Stillwater, Oklahoma in May, 2008.

Experience:

Graduate Assistant in Scholarship and Financial Aid department, OSU, Stillwater Oklahoma.

Name: Ravikumar Nidadavolu Date of Degree: May, 2008

Institution: Oklahoma State University Location: Stillwater, Oklahoma

Title of Study: CONTENT-BASED RETREIVAL OF MUSIC USING MONOPHONIC QUERIES ON A DATABASE OF POLYPHONIC, MIDI INFORMATION

Pages in Study: 62 Candidate for the Degree of Master of Science

Major Field: Computer Science

Due to the large amount of musical data available on the internet in recent years, efficient and intuitive methods are required for searching the musical data. Musical search services, such as the iTunes provides, support querying capabilities on the basis of metadata tags (title, artist, etc) associated with the musical data. The natural way of searching musical data is to search by its content rather than secondary features like title, genre etc, because the content is usually more memorable. In this research, content-based music retrieval is performed on a polyphonic MIDI music database where the query is a hummed tune. Two approximate string matching algorithms, LCTS and Myers algorithms are modified, applied to the problem, and retrieval performance is calculated. Response times of the algorithms are calculated by altering the values of some of the interesting parameters such as the query length, degree of polyphony and size of the database.

ADVISER'S APPROVAL: Dr. Blayne E. Mayfield