FINDING SONG MELODY SIMILARITIES USING A DNA STRING MATCHING ALGORITHM

A thesis submitted
to Kent State University in partial
fulfillment of the requirements for the
degree of Master of Science

by

Jeffrey Daniel Frey

May, 2008

Thesis written by

Jeffrey Daniel Frey
B.S., Kent State University, USA, 1998
M.S., Kent State University, USA, 2008

Approved by

_____
Dr. Johnnie Baker
Advisor

_____
Dr. Robert Walker
Chair, Department of Computer Science

_____
Dr. Jerry Feezel
Dean, College of Arts and Sciences

TABLE OF CONTENTS

Acknowledgements ..... **vii**

CHAPTER 1 Introduction **1**

1.1 The Problem 1

1.2 Solution Concept 1

1.3 Approach 2

1.4 Document Organization 3

CHAPTER 2 The Need for a Solution **4**

2.1 Copyright Infringement 4

2.2 Royalty Distribution 6

2.3 Finding or Suggesting Songs 8

CHAPTER 3 Melody and Theme **11**

3.1 Melody 11

3.2 Documenting Melody 13

3.3 Theme 16

CHAPTER 4 Music Information Retrieval **19**

4.1 Patents and Tools 20

4.2 Query-By-Humming 22

4.3 Issues 23

4.3.1 Pitch 24

4.3.2 Duration 25

4.3.3 Tempo and Rhythm 26

4.3.4 Polyphonic 27

CHAPTER 5 A Sampling of Existing Search Applications 30

5.1 Medieval Music Directory 30

5.2 SEMEX 30

5.3 MELDEX 31

5.4 Themefinder 31

5.5 Real-time in an Off-the-Shelf Computer 32

5.6 MusArt 32

5.7 K-BOX 33

5.8 VocalSearch System 33

5.9 Shazam 33

5.10 Classical Music Database 34

5.11 Tyberis 34

5.12 Gracenote 35

5.13 Digital Nirvana 35

5.14 MusicBrainz 36

CHAPTER 6 The Smith-Waterman Algorithm 37

6.1 Biological Sequences 38

6.2 The Algorithm 39

6.3 Adaptations 40

6.4 Use in MIR Systems 42

6.5 Summary 43

CHAPTER 7 A Modified Implementation 44

7.1 Base Algorithm 45

7.1.1 Smith-Waterman Algorithm Implemented in Java 46

7.1.2 Assumptions 46

7.1.3 Gap Scoring 47

7.1.4 Java Code 48

7.2 Adaptation for Melodies 48

7.2.1 Pitch 49

7.2.2 Duration 50

7.2.3 Tempo and Rhythm 51

7.2.4 Polyphonic 52

7.3 Modified Implementation 53

7.3.1 Initialization 53

7.3.2 Sharp and Flat Conversions 53

7.3.3 Transposing 54

7.3.4 Smith-Waterman 55

7.3.5 Timing 55

7.3.6 Implementation Modification Summary 55

7.4 Command Line and Input File 56

7.4.1 Weighted Distance Matrix 58

7.4.2 Gap Penalty 59

7.4.3 Input Song 59

7.4.4 Song Database 60
7.5 Output 60
7.5.1 Header 62
7.5.2 Strings and Scores 62
7.5.3 Timing 63
CHAPTER 8 Results 66
CHAPTER 9 Conclusion 72
CHAPTER 10 Contribution 75
CHAPTER 11 Future work 76
GLOSSARY 77
APPENDIX A Base Code 81
APPENDIX B Sheet Music 87
APPENDIX C Modified Code 88
APPENDIX D Melodies Used for Matching 96
REFERENCES 98

Acknowledgements

I would like to acknowledge and thank Dr. Johnnie Baker for his guidance and patience during this process. My thanks also go to Dr. Robert Walker and Dr. Arvind Bansal for serving on my thesis committee. I also would like to acknowledge the others at Kent State University who supported me in this effort, especially Marcy Curtiss. I am grateful to both Peter Angelov Petrov for the use of his code and to Omer A. Piperdi for help with modifications. I greatly appreciate the support that faculty in the school of Computer Science, staff in the Information Technology department, and the students at Rice University have given me in this effort. In addition, I would like to thank my colleagues at other universities around the country who have had input, and great impact, on this paper. Lastly, I would like to thank my wife and family for the encouragement, without which none of this would have been possible.

Jeffrey Daniel Frey

January 17, 2008, Kent, Ohio

CHAPTER 1

Introduction

Music is woven into the fabric of everyday life. If you do not have a favorite song, band, or genre, chances are you still listen to music. Sometimes, we are subjected to music whether we want to listen or not; music is present in the background of movies, television shows, restaurants, car rides, and elevators. With over 50 radio stations in every metropolitan city, over 100 stations on satellite radio, and over 1,000 stations on the Internet, millions of songs are heard every day.

1.1 The Problem

With so many songs played all over the world, there is growing demand for music and services surrounding the music industry. Writers are pressured by artists to write new songs, but face the challenge of ensuring they are not writing a melody that has already been published. Licensing agencies are asked by artists to track song usage, but most methods of doing so are either not accurate or very slow. Businesses are asked by consumers to provide new ways of searching for songs; simply suggesting another song within a genre no longer meets the needs of increasingly sophisticated consumers.

1.2 Solution Concept

Based on the issues above, there must be an easier way to search through songs that approximately match a song under consideration. Given the proper tools, writers could submit their songs and see how closely they match songs already developed. Licensing

agencies could have a system "listen" to radio stations and record song usage based on matches. Businesses could provide consumers with an interface to submit songs they are interested in, and then return to them songs that come closest to their desires. Some research and algorithm development has been done in this area using pattern matching, case-based reasoning, and auditory song recognition based on sound waves, but a fresh perspective can be gained by considering other fields of research where similar problems have been very well defined and solutions executed. Take deoxyribonucleic acid (DNA) research, for example. Pattern matching using precise algorithms has been refined and many articles and papers written on the subject. Could the algorithms currently used for the matching of DNA strings be repurposed for the type of close approximation melody recognition needed to solve the problems mentioned above? Though modifications explained herein could be applied to a number of bioinformatic DNA string matching algorithms, an implementation of the widely used Smith-Waterman algorithm will be considered to prove how this is possible.

1.3 Approach

Though prior research has been conducted on the possibility of using DNA string matching algorithms for music similarity applications, the idea has not been extensively explored because of the time and effort it takes to implement. The typical method for most is to write a program from scratch. The approach here is to explore the use of a DNA string matching algorithm further by focusing in on one example, the Smith-Waterman algorithm, and making slight modifications to the input and output of a pre-existing implementation. Leaving the specific code in the program that performs the

algorithm as used for DNA matching, the solution then becomes an example of how implementations of other algorithms can be more easily used for the purpose of melodic song recognition and matching. Regardless of the algorithm selected, the methods and processes defined could be used to map most bioinformatic string matching programs to song melodies. Since much is known specifically about the Smith-Waterman algorithm's origins and modifications, if it can be acutely proven as a good base to use with straightforward implementation, there are many other possibilities for expansion.

1.4 Document Organization

This document will first take a proper look into the need for a solution to the melodic song recognition problem. It will then look at music in general along with definitions of melody and theme, explain the concept of music information retrieval, and describe what aspects of music are important to song recognition specifically. It will then go into a sampling of existing search applications created using different techniques and algorithms. Next, the Smith-Waterman algorithm will be discussed, first in general and then as specifically related to melody recognition. A modified algorithm based on Smith-Waterman then will be shown and explained, including results when used for song melody recognition. Finally, since proving this initial step is simply the beginning, future work on the modified implementation of the algorithm will be discussed.

CHAPTER 2

The Need for a Solution

The issues of song and melody recognition are not just an exercise in mathematics and computer science. There are very timely, real world applications for which precise solutions are needed. Songwriters, rights organizations, and consumers all have reasons to spur on the creation of systems that accurately retrieve songs based on melodic information given. Copyright infringement, royalty distribution, and locating or suggesting songs are just three of many reasons.

2.1 Copyright Infringement

Copyright is a very important term in the music industry. It refers to the rights to reproduce, or authorize others to reproduce, musical works [37]. A copyright must be applied for and is granted for a specific number of years [14] by the government to the individual or group who authored the work. Copyright infringement is a serious criminal offense that, for the purposes of this paper, refers to the author of a musical work reproducing some aspect of a prior copyrighted work. Obviously, a song is a musical work, and though in many cases copyright infringement happens knowingly, in some cases an author of a song may not be aware that he or she is infringing on an existing copyright. In the United States, the plaintiff or prosecutor in a copyright infringement case must only prove that the act of copying was committed by the defendant and does not have to prove intent. Good faith is not a defense [106], and an author not knowing

that he or she has infringed does not negate the fact that it was done. The best way to avoid unintended copyright infringement is to be certain there is no infringement ahead of time. At this time, the best tool available to be certain an author has not infringed on a copyright is a copyright catalog search from the U.S. Copyright Office. There are three ways of searching the catalog: in person at the Library of Congress, online, or by asking the Copyright Office to conduct a search of the records for a fee [101]. The catalog contains five key elements in establishing the history of the original copyright registration: the author name, the title of the song, the claimant (i.e. owner(s) of the copyright who may be different from the author), the date of publication, and the registration number [98, 99, 101]. This method of searching works well if you are aware of a title or author on which you may be infringing, but falls far short for finding parts of songs. Copyright law states that anyone wishing to recreate an original song in some way, even a few notes of an existing copyrighted song, must be granted permission by the author [14, 99]. There is no special number of notes that can be used without permission; the copyright owner is able to use discretion on what he or she feels is infringement before bringing a case against a new song. The U.S. Supreme Court has stated that the infringement will be considered against the test of fair use [20], such as if the new song had a different expression, meaning, or message than the copyrighted song. Musical melodies, though the song may have a different expression, meaning, or message, may be considered by a court to be copied if there is a certain percentage of alignment [99]. With that in mind, how is an author to know if the song that he or she has created may be seen as having infringed on an existing copyright? A search engine solution for published

melodies is needed to ensure there are no exact matches, nor very close matches [3], to a new melody.

2.2 Royalty Distribution

Once a copyright is given to an author, the author is entitled to collect royalties for that copyrighted song. Royalties are payments to the owners of a song, usually governed by an agreement from a performance rights organization (PRO). For this discussion, the term royalty will be used to describe the payments by publishers to authors with rights to a song, but it could also be used to describe the payments of recording companies to composers that hold rights to a song [46]. There are several types of rights given to an author; they are sound recording rights, mechanical rights, and performance rights. Performance rights appear to be the most important [23], because most ongoing royalties are tied to them. When a song is played on radio, television, in a hotel, club, restaurant, bar, elevator, doctor's office or store, it is tracked by a PRO, and then royalties are paid to the owners of the performance rights to that song [22, 23]. An author, typically the owner of the performance right, has a choice of PROs with which to make their rights agreement. There are three main PROs in the United States: the American Society of Composers, Authors and Publishers (ASCAP), Broadcast Music Incorporated (BMI), and The Society of European Stage Authors and Composers (SESAC) [70]. Each PRO has a list of authors with whom it has made rights agreements and notes what songs are under the agreement. These lists can be searched online through the PROs' websites [12, 19, 90].

Tracking song play has become an even more complicated task due to Internet radio play, downloads, and the ability to play one digital copy of a song multiple times [22], but the PRO has the responsibility to track the number of times a certain song is played in order to accurately distribute royalties quarterly to rights owners [100]. No matter how good the technology is today, it is not possible to track every play of every song. How, then, do the PROs track and estimate the number of songs played? There are some flat rate agreements with restaurants and bars along with licensed theme parks, circuses and so forth, but for television, PROs depend on a cue sheet given to them by the television stations or entered by the television station into the PROs' database. For live performances, they typically review set lists provided by the performing artist or printed programs obtained from the event. For radio, there are play reports completed by the stations, sample surveys done by the PRO, and monitoring systems [52]. These monitoring systems are the focus of much attention, because if an application could be written to "listen" to every radio station, television station, and Internet channel, a more accurate distribution of royalties could be accomplished.

ASCAP stated in a recent press release that their success was built on seven strategic thrusts. The first thrust was focused on capturing licensing revenues from new and growing media channels, such as cable television, the Internet, mobile devices, and satellite radio. The fourth of the seven thrusts was to more broadly monitor music usage and more effectively curb infringement activity [13]. They intended to do this by deploying "state-of-the-art technology," but did not say what that technology may be. The way royalty distributions are calculated now is not an exact science.

The method employed by BMI is a quarterly survey of a diverse sample of radio stations to collect a list of every song played during a two or three-day period. Using these results, BMI says they are able to determine what songs are played and the frequency of their airplay on all stations [18]. They go on to say that it would be enormously expensive, if not "impossible," to determine exactly what songs are played in every business or organization across America, so they use these radio survey results combined with television and commercial music service lists to "determine approximately how frequently" a song is played [18]. Some artists and songwriters feel this approximation of how many times a song has been played is unacceptable. Considering this aspect of rights and royalty distribution, there is an obvious need for a song-listening solution that matches to a database of known songs, accurately identifies entire songs, and matches small sections of songs (for example, partial songs played during television commercials).

2.3 Finding or Suggesting Songs

After a song is written by an artist, copyrighted, and licensed by a PRO, it is then released to the public. Radio, television, and stores are all ways for a consumer to hear and buy a song that has been released, but the fastest growing medium is the Internet. As consumers have more and more access to a growing list of songs (over 3,000,000 songs in Apple iTunes alone [11]), how can the consumer find the song that he or she is looking for? Better yet, how does a consumer find a song that he or she does not know about, but would like if hearing it? Current search technology in almost every Internet song repository allows for the searching of metadata such as song title, artist name, album, and

genre. A consumer might be able to catch the name of the song on the radio, or search through a popular list of songs to find a specific one he or she is searching for, and sometimes the consumer may even be able to listen to a preview of the song found. In some cases, the consumer may remember a few words of the song to help with the search. However, what happens when the consumer can only remember the melody [62] of the song? Some consumers may even think they remember the melody, but in fact only have a partial section of it, or may have mistaken one or two of the notes. This calls for a melody retrieval system that can accurately take a suggested series of notes and try matching them, both exactly and to a certain percentage, against existing melodies.

The question of a song that the consumer would like, but has not heard yet, still remains. Like metadata searching, song suggestions have been incorporated into almost every Internet song repository. A manual process at Amazon.com, consumers are asked to set up lists in Listmania [10]. The compilation of songs and albums on an individual's list is used to suggest what other similar songs or albums might be of interest. The information is displayed when the individual views a certain item with a statement such as "If you like this, then you also might like: _____" [10]. Some song suggestions are based on songs that you already have. Apple's iTunes, seeing the artists of your existing songs, shows you other albums from those artists along with artists in the same genre [11]. MusicMatch is a product of Yahoo! that also relies on input from its users. Knowing that a user likes artist A, and owns and listens to music from artist B, allows similarities to be drawn across all users. When entering any artist name, it will display the most closely related artists based on popularity among users [71]. They even display

a daily recommendation that is supposed to propose a song that would suit the user. Similarly, Google recently entered the song suggestion realm by offering to track the music a consumer listens to on his or her computer, and then publish that information anonymously for other consumers to view [24]. The new offering, called Google Music Trends, compiles the information to show what consumers are listening to. It will then give the user an idea of what they might like, based on what they have.

All of the song suggestion technologies above rely on active or passive consumer input. Obviously, the passive techniques are more desirable, but they also then rely on information or suggestions that other consumers have made. From this review, it is evident that the need for song suggestion software, based only on the songs themselves, is required. For example, a request could be made based on the characteristics of song A to find a song B that is similar. Song B may have the same type of tempo, rhythm, or instrumentation, but may not be related in any way to song A. For the purposes of this discussion, the focus will be on melody as the primary means by which a song can be categorized and searched.

CHAPTER 3

Melody and Theme

There are many questions to consider when creating a music matching software application. How is the song input into the computer? What technology will be used to listen to the music? What features of the song should be stored, and how to store them? What matching algorithms should be used? How to return the results to the users? To address these questions, the following expounds on what is meant by the term "melody," how a melody is extracted from a song, and how a melody can be encoded or stored in a computer or database for matching.

3.1 Melody

A melody is a succession of notes, varying in pitch, taking an organized and recognizable shape [62]. Each note making up the melody is a specific sound, like when a key is pressed on a piano. The melody of a song usually refers to the most recognizable succession of notes for the entire song. Theme refers to a shorter amount of notes (a subset of the melody) that may be repeated and identified by individuals as a more easily recognized part of a song, for example the first four notes of Beethoven's "5th Symphony" [63].

Next, the pitch of a note actually has a frequency; for example, the note A to which an orchestra tunes has a frequency of 440 Hertz (Hz), or cycles per second [3]. Each note has a different frequency, also called pitch, and the distance between those pitches is

called an interval [3]. Rhythm also is such an important element that when notes of a popular melody are left intact, and rhythm is drastically altered, it becomes difficult to recognize the melody [62]. Rhythm refers to the time distance between each note in a melody, whereas tempo refers to the overall speed by which the notes are being played across the entire melody [25].

Finally, an important part of the definition of a melody or theme is that it is recognizable. Human perception of music is a key element to consider when thinking about computer systems that will try to do the same. Research done on the human ability to recognize different things by specific senses alone found that auditory recognition is closely related to visual recognition [1]. Perception channels coming from the eyes, fingers, nose, and ears all lead to a part of the brain that organizes the information coming in and matches it against the history of information stored there. Music is stored in the human brain as relative information of pitch (frequency) and duration (length) of consecutive notes; it is believed that the brain automatically performs processing on the pitch and duration, turning them into contours and structures [40]. A melody contour refers to the shape of the melody, indicating whether the next note goes up, down, or stays at the same pitch [3]. With regard to music specifically, research found the way the melody is presented to the perceiver has a lot to do with recognition. Identification of melodies for people is most strongly associated with notes located at the beginning or end, such as a theme that starts or ends with a recognizable few notes. Most people, if they have not recognized a theme in the first or last 5 to 7 notes, will not recognize it when the entire melody or theme is played [60]. Over time, this popular topic inspired

radio and television programs dedicated to the human ability to discover a tune [29]. Another issue of perception centers on whether there is other music going on with the melody or theme. Harder to discover, the term polyphonic is used to describe music where there are two or more notes being played at the same time. A singular melody, with only one note played at a time, is called monophonic [3].

3.2 Documenting Melody

There are software programs available that will allow a user to play a series of notes on a keyboard, guitar, or other instrument interfaced with a computer, and will then print out (or transcribe) musical notation representing what was just played. The key words in that sentence are "connected to a computer." If the computer can take the input electronically, there is not much problem with transcription, which is defined as the process of recovering the musical score of a piece of music describing the individual notes played in a recording [34]. How though can a computer listen to a piano, guitar, other instrument, or human voice, and achieve the same musical notation? The first step in melody transcription is to capture the analog input and convert it to digital form [85], which can take on many representations. Descriptors are then added to this digital form to describe it. Sometimes referred to as taxonomy, this group of descriptors or labels is an important part of the process. Although no one taxonomy standard has been adopted across all melody documentation systems, there is a general common language used by developers of these systems and another for users of the systems [65]. Rhythm, pitch, and tempo, some of the vocabulary introduced earlier, help describe the higher levels of description used for songs and melodies. Lower level descriptors include frequency,

duration, and volume, which are used to describe specific notes or sections of sound [65]. These sections of sound in digital form are the pieces that some approaches use to begin identifying a song. A waveform representing the sound [3] can be used like this when it is seen as a sequence of acoustical events. Not listening to the melody itself, programs using this approach would identify the song or track, but not necessarily find the exact melody and transcribe it. The idea is to extract characteristic information not only from the whole piece of music but also from smaller parts, building a unique AudioDNA [44] of each song.

One popular method of documenting melody is to try to instantly identify the frequency of the sound(s) being heard. From that point, in European/Western music using a typical musical scale, a note can be identified [85]. C, D, E, F, G, A, and B are the notes in a major C scale. This scale can then be duplicated higher or lower, starting with D or B respectively. A repetition of a scale, from C, D, E, and so forth up to B and back to C again, is called an octave. Frequencies are typically categorized by their note and octave; for example, the middle C note on a piano is referred to as C4: C note, fourth octave [85] with a frequency of 261.6 Hz [83]. Octaves are divided into 12 notes that are a semitone (a half step or just slightly apart) from one another, but most simple songs only use a subset of those [3]. The absolute pitch labels C, D, E, F, G, A, B, and C, which are more commonly known as do, re, me, fa, so, la, ti, and do, are not sufficient labels for notes in some approaches to documenting. Since user input can start in many different locations, sometimes a relative scale is used [40]. Using the first note recognized as a base, say 0, the next note may be two semitones or half steps above that

note, written as +2. Then, the next note may be just one above that one (+1), or one below (-1). Using these two techniques, the melody for "London Bridge Is Falling Down" may look like this: D E D C B C D or 0 +2 -2 -2 -1 +1 +2 [40]. This popular way of describing melody is often also tied to duration. Though two melody's pitches or frequencies could match, the duration or length of each note held could vary greatly. When duration is matched with pitch, a decision has to be made on whether to weight one more important than the other or equally [57]. Some researchers say using only pitch and duration is not good enough because it is possible for different songs to share such similar characteristics. The challenge is to represent the music content including melody, vocal, and song structure information holistically [74]. A multi-layer pyramid structure has been used to describe music information conceptually. It shows not only the mid-level concepts of a song, such as the melody, but also the lower level constructs of instrumentation and higher level organization structures of verses and chorus. Hidden Markov Models (HMMs) are an excellent tool for modeling music melodies [102]. HMMs are constructed based on the pitches and durations noted in a score using features based on amplitude, activity, and frequency content of a raw musical file [7]. After training, the HMMs are used to align audio data with the model and score that represents it. Once the HMM model has been created for an audio file, it can be decomposed into a symbolic representation [76]. The Markov property holds that the probability of transitioning from a given state (or note in this case) to another state (note) is assumed to depend only on the current state [15]. The current state in a song may include the pitch of the note that is now being played, its duration, and if it is higher or lower than the note

before it. Using these probabilities to encode a song melody helps when trying to match the song against another melody that may be close but not exact.

3.3 Theme

Some approaches try only to extract the most memorable part of a song. Most songs include an introduction, a verse (or multiple verses), a chorus, an interlude of some sort, a bridge, and an ending. Though song structure is not important to this approach of song identification, the chorus is usually the section of a song that is repeated the most. And though it may not be the case for all listeners, the chorus is usually the section of melody that is the most identifiable [61]. The most repeated part of a chorus, or theme, is usually found by a computer in a piece of music by lining the patterns of a song on top of the other very quickly and in many different ways while listening to it. A interval matrix also can be created to help document and determine all repeating patterns in a song, which now essentially becomes a search for self-similarity in a piece of music [56]. The most significant part of the theme is typically repeated at least once, and smaller sections of a theme are likely to repeat multiple times. Not storing the entire melody of a song, and holding only the theme, keeps the size of the data for each song small [96], which is helpful given the number of songs that exist. Identifying relevant parts of a song that need to be held (such as the theme) and discarding the others presents a smaller section of data for each song to be looked at and matched against when searching for a song. Using this technique, an inventory of songs could be created and held separate from the actual song list including only the thematic information needed for each song [53]. In classical music, such as sonatas and symphonies, a theme is a melody used by a computer as a

starting point for developing the different melodic elements of the song [56]. A piece of music may have several themes, but each of them will repeat and be a slightly changed variation on the main. In modern music, the theme is typically called the hook [83]. It is what a songwriter uses to hook a listener and bring them into the song, or what a listener finds hooked (or stuck) in his or her head. It could be a series of words or an interesting rhythm, but most often, it is that short musical melody used repeatedly. There is a big difference, however, between finding a hook or theme in a produced piece of music and finding one in a non-musical human representation (played or sung). Humans organize songs in their brain a certain way for recall, as stated earlier, and studies show the theme is indeed the most memorable part of a song [83]. In addition, when thinking about acquiring a theme from a produced piece of music, it is not only necessary to look at how a person might remember a song, but also how they will communicate that to a person or computer. When asked to reproduce a song from memory, humans are most likely to sing the song and start at the hook, which often occurs at the beginning of the chorus, rather than at the beginning of a song [83]. For example, most people know the song "Take Me Out to the Ballgame," written by Jack Norworth and Albert Von Tilzer in 1927, by the hook starting with the chorus "Take me out to the ballgame…", but the song actually starts with the line "Nelly Kelly loved baseball games…" [105]. Another way researchers found to document themes is to pick an arbitrary number of notes and collect them starting with the first note of the song [108]. Since, as stated, the theme is not necessarily at the beginning part of a song, it is also necessary to collect that number of successive notes from the second note on, third note on, and so on until the end of the

song. Those note collections can then be compared against each other for matches within the song. The highest numbers of note collections that match each other become the most repeated theme of the song. However, these matching series of notes are problematic when discussing theme. There are highly redundant elements that may appear to be themes, but should be ignored (i.e. scales, redundant melodic introductions, and drone notes). They may be added as filler to a song, but they are not necessarily part of the hook or theme itself [102].

We shift focus now from this short discussion on melody, melody documentation, and acquiring themes, to look at the technology portion of song retrieval.

CHAPTER 4

Music Information Retrieval

It has been shown that there is a great need for a solution to the difficulty of retrieving a piece of music, and music can be digitally represented and documented in many different ways depending on what is to be done with it. Addressing the problems and using the documentation methods mentioned, there is a growing body of research dedicated to what is called music information retrieval (MIR). The goal of MIR is to process musical information and search music databases by content [72]. Though scientists and researchers involved in MIR traditionally have focused on melody retrieval, they have expanded to searchable databases containing musical content [16]. This content could be what instruments were playing in the music, separating verses from chorus, identifying genre, and recognizing the lyrics of a song. A good resource for learning about the technologies used in MIR is the online proceedings of the annual International Symposium on Music Information Retrieval (ISMIR), located at http://www.ismir.net/allpapers.html. Comparative evaluations of algorithms for efforts such as identifying songs and melody extraction are done at the Music Information Retrieval Evaluations eXchange (MIREX), which takes place in conjunction with ISMIR. Descriptions of algorithms, evaluation procedures, and collections of songs to use for testing are available on the MIREX Wiki at http://www.musicir.org/mirexwiki [16]. Each year MIREX has a formal evaluation of MIR systems, ranking them in their ability to identify the melody in popular music recordings [34]. The ISMIR series of

conferences started in 2000 and have a growing body of knowledge and participants around the world [30].

4.1 Patents and Tools

The following five patents are related to the research being done in the field of music information retrieval. They were selected from the many existing patents for inclusion here due to their unique matching algorithm or interesting application.

- Music search by melody input, 2001 [81] – This includes the idea of taking a front-end interface for a query, a database of existing melodies, and the use of a differential matching algorithm. The method is meant to enable a user to search for a song title when only its melody is known.
- Music annotation system for performance and composition of musical scores, 2002 [67] – A system for annotating music, this patent describes retrieving a musical score from an Internet source based on a query from the computer. Once the computer has the score, it opens it in a viewer that will allow the user to annotate it.
- Music search engine, 2004 [69] – This search method involves a user inputting their query by vocalizing it for a computer to record. This is similar to the query-by-humming approach that will be discussed later. The matching technique uses a calculation to find the relative difference between adjacent notes in a song and in the query. The song is considered found if the difference sequence for the query matches a portion of the difference sequence in one of the songs.
- Song-matching system and method, 2005 [33] – This is an interesting idea envisioned for use in karaoke. It starts with a user beginning to sing a song and, as that happens,

a computer records the audio and proceeds to create a melody for the song by deciphering the pitch from the audio of the user. The system then attempts to match the melody against the melodies (stored as pitch templates) of the songs in the database. If a match is found, the system then downloads the song and begins to play the accompaniment tracks of the song at the point where the user is singing it.

- System and method for music identification, 2006 [86] – A song matching technique for finding an entire song from a brief microphone recorded sample of the song, it computes a feature vector and processed time signal for each song in the database and from the recorded sample. Then the songs are sorted by the feature space distance with respect to the feature vector, and the time signals are compared. The most likely match is returned to the user.

The tools used by MIR researchers and system designers/developers, ranging from microphones to supercomputers, touch both the music world and computer science discipline. Since most research focuses on electronic music documentation, storage, and retrieval, top on the list of subjects covering MIR tools are audio/signal processing, machine learning, algorithm prototyping, and song visualization [78]. Most tools are publicly or commercially available, and MIR researches have found how to use these tools in very interesting ways. Some tools were discovered at the International Music Information Retrieval Systems Evaluation Laboratory (IMIRSEL). It provides a place to evaluate MIR algorithms as well as Music Digital Library techniques. IMIRSEL has supercomputing power and a large database of rights managed music for researchers who would not necessarily have the resources or money for the technical and musical data

required to test their MIR systems [49]. Experiments conducted and explained at the end of this paper made use of typical desktop computers as well as the supercomputer named Ada, a Cray XD1 Cluster located at Rice University, accessed with the cooperation of the Information Technology and Computer Science departments.

To explain how MIR data is created, the next topic illustrates a popular method by which users can input queries into a computer.

4.2 Query-By-Humming

Query-by-humming (QBH) is a technology allowing users to find a song in a database by humming the melody into a microphone connected to a computer. The sound is then transformed into a melodic representation and is used for comparison with items in a melody database. Pattern recognition is used to find the matching song on a CD, hard drive, or web site [54]. Since QBH relies on a person to input a melodic signal into a computer, it is necessary to understand how this signal is produced. Humming is considered the easiest way for an ordinary non-musician to express a music query [72] and occurs with the vibrations of vocal chords when voicing a sound. Actually caused as a consequence of other forces [2], muscles in the neck can be manipulated against the laryngeal walls as air flows through the gap between the vocal chords (known as the glottis). The glottis repeatedly opens and closes pushing bursts of air through the vocal tract causing sound. The issue of an individual who does not have melodic singing ability is obviously a weakness of a system like this, and one study showed a 22% decrease in melody recognition between musically trained and untrained individuals humming tune queries [40]. Since input signal is a concern, most QBH systems are

designed for singing without lyrics. Systems that include lyrics in their search query are called query-by-singing (QBS) systems [35]. Most QBH systems put restrictions on the type of syllables that can be used (mostly "da") [55]. It has been shown that QBH systems should be created to handle people starting up to two seconds after the start of the query recording and reproducing a few seconds of the tune up to thirty seconds [65]. Research also shows that the average vocal query for a system like this is fourteen seconds. The main melodic representations of a QBH system include pitch contour and rhythm information [104]. Some systems have been built to model the human auditory system using a physiological model of the ear-channel, cochlea, and neuronal processing [54].

4.3 Issues

It is important to note that with all the different aspects of music, the fundamental challenge addressed in this paper is the comparison of songs with single melody lines. The issues below related to specific problems facing MIR researchers are listed to show the complexity of the computational elements in MIR. Mostly defining the issues, each explanation below also will briefly describe how MIR researchers are tackling the challenge. It has been found over the years that pitch, duration, tempo, and rhythm are crucial for recognizing melodies [59], but the largest issue facing melody recognition is polyphonic music interpretation.

4.3.1 Pitch

Pitch is a subjective attribute of auditory sensation in terms of which sounds may be ordered on a scale extending from high to low [43]. Pitch perception in humans is an amazing coordination between muscles, fibers, and neurons. Pitch in terms of music comes from repetition in a sound waveform [34]. As a note on a keyboard is sustained, the fundamental frequency can be found by spectral analysis. To a human ear, it can be described as how high or low the sound is (perceptual analog frequency) [31], and to a computer it is the calculated digital frequency. Another issue that goes along with pitch is tuning. A song is usually played in one key where a C note is a C note throughout the song. In audible song queries like QBH systems, users seldom have perfect pitch and do not stay in tune [32]; meaning they start out on a C note, but by the end of the query, what they think is a C is actually closer to a B (below, called going flat) or a D (above, called going sharp). Another issue, even if a singer does stay in tune, is where they start [61, 58]. A song may be stored in the database as starting on a C note, and the singer may start on a D note. Each one of these issues must be addressed with algorithms to normalize the tune or move the entire sung melody up and down to match with a song in database. It is interesting to note that some research has shown there is not much difference between a sung query and a whistled query. Sung and whistled note perception and going sharp or flat is a matter of the skill of the individual singing or whistling instead of the delivery method [6]. The fundamental problem with pitch, however, is interference [34]. A more thorough discussion of this interference is in the polyphonic section below; in short, when other sounds or multiple notes are happening

around the melody, it is easy to understand how difficult it is to retrieve the specific searched-for pitch. Probably the most popular way some researchers choose to deal with pitch is by not specifically targeting the pitch itself, but instead by addressing melody contour, sometimes called slopes [109]. The contour is the shape of the melody. That shape can then be matched with other contours in the database. This method helps solve the problem of the singer starting on a different note than the song is written to start in, and in most cases the method solves going out of tune (sharp or flat). For example, a song melody could be annotated as a series of notes with "S" for same, "U" for up, and "D" for down in relation to the last note that was played [45]. The first note is usually not represented. Therefore, a melody like "London Bridge Is Falling Down" would be "UDDDUUDUUDUUSUDDDUUDUDD." If a singer started on a different note, it would still be the same exact annotation. If the user went sharp while singing the query, the note would still most likely be above the last note sung if it was supposed to be, even if it was too far above or not enough above.

4.3.2 Duration

Melodies have recurring pitch periods [83, 84]. A pitch, discussed above, is usually caused by a note being played or syllable of a word being sung. That note or syllable has a start and an end time. The duration of that note is how long the note lasts [31]. For sung melodies, it is easier if a user sings a single syllable like "ta" or "da" [85]. The "t" or "d" sound in the front of the note creates a pulse in the musical wave that is generally recognizable and can be seen as the start of a new note [84]. Slides, or glissandos as they are called in musical terms, are harder to detect. Those would most

likely be shown as a series of ascending or descending notes. As with user input for pitch, there are many problems to user input concerning duration. Most singers would not hold on to a note exactly as a recorded song, instead stopping too soon or holding on too long [104]. As in the pitch solution above, some researchers choose to annotate the duration of the notes by using "S", "s", "R", "l", and "L" for "much shorter", "a little shorter", "same", a "little longer", and "much longer" respectively [45]. Keeping with the "London Bridge Is Falling Down" example, it would produce the following: "RRRRRlsRlsRlsRRRRRlRRSL." Taking this idea one step further, these strings of letters can be converted into an integer number using a bijective function [97] that can be stored in a computer database and searched more efficiently. The use of these integer numbers also increases the ability to take existing computer algorithms designed for numeric evaluation and use them for MIR purposes.

4.3.3 Tempo and Rhythm

Tempo is a basic property of a musical fragment [34] and pertains to the consistent pace of a musical piece. Rhythm is the word most used to describe the times at which notes are played or sung in a song. While the tempo of a piece of music usually stays consistent for the length of the song, the rhythm of the notes within that tempo can be erratic. For example, some fragments of melodies are made of only one note or frequency. When comparing two songs with this type of melody, they may have the same tempo, but the rhythm information is usually important [31]. For example, the song "Jingle Bells," composed by James Lord Pierpont in 1857, starts out with seven notes of the same pitch: "Jin-gle bells, jin-gle bells, jin…" The rhythm of those notes could be

envisioned as very short, very short, long, very short, very short, long, short. "Winter Wonderland," written by Felix Bernard and Richard B. Smith in 1934, starts out with five notes, all of the same pitch: "Sleigh bells ring, are you…" Looking at pitch alone, the songs match for the first five notes, but the rhythm of "Winter Wonderland" is different in that the third note is held three times as long as in "Jingle Bells:" very short, very short, very long, very short, very short. The rhythm is needed to distinguish the songs. Some researchers solve this issue by giving the user a click track to play or sing along with, and others allow the users to tap something while singing or playing so that information will be stored with the audio [104]. The first approach, using a click track, appears to be very effective and uses technology with which the musical community is already familiar. A click track is often used during song production to synchronize the various instruments in a song, and it is used by drummers so they can avoid speeding up or slowing down a song. The second approach has the user model their own rhythm, which is input into a computer and normalized; meaning the rhythm is mapped to a steady beat even though the user may have sped up or slowed down. However the issue is solved, tempo and rhythm cannot be ignored when looking to solve song search issues.

4.3.4 Polyphonic

Polyphonic music is defined as having more than one sound happening in the piece of music at any given time [31]. Most music is this way, and seldom is there a single note sung or played at one time, a style called monophonic, without any background instruments, harmony, or percussion. Gregorian chants are good examples of monophonic music. From what has already been covered, a clip of audio with one piano

playing a single note at a time or one voice singing would be easier for a computer to discover pitches, and ultimately melody, than if there were other noises in the audio. This also brings up the discussion of interference and noise, in both recorded songs and audio queries. Recordings can often have purposeful noises included with the audio track to model distortion and age (i.e. record fuzz), but the issue of noise is most seen in live recordings. Audience screams, coughs, applause, and singing along are all considered noise around the true music to be analyzed. Audio queries, if not performed in a controlled environment, may have any number of background noises included in the audio recording as well [8]. Some recent automatic transcription methods based on multiple-F0 estimates and note detection have been effective in dealing with polyphonic music [68]. More so, the combination of those with musical modeling makes the transcriptions very accurate. The models use key estimates and note bigrams to determine the transition probabilities between notes. Take an instrument that, because of background noise, appears to be playing a note in-between two notes. Based on a musical model of where melodies usually go from this point (considering the past), the computer can determine whether to use the note above or below the appeared note. Finding this note, or as some call it the fundamental frequency of the song, is one of the most important sources in recognizing a song [95]. The human voice in a sung polyphonic song usually carries the melody (the fundamental frequency). This usually can be found in simple songs by finding which frequency carries the most energy. For complicated songs, the melody or singer may get lost in the noise of the band, so a harmonic sum that calculates the average energy of the harmonics of each fundamental

frequency is used [95]. For example, you may not be able to pick out the exact note, like C, as the fundamental frequency, but it may be possible to narrow the search by looking across all the frequencies. From this, we may be able to tell that we are probably in the key of C or G, and from there, use a model to understand which note is most likely carrying the melody.

CHAPTER 5

A Sampling of Existing Search Applications

Since MIR has been around for some time, there are a growing number of song search applications. These applications were written for many reasons, including proving an academic hypothesis, for personal use, commercial sales, and the belief that a music search engine is a needed tool and should be free to all who would use it. Some of the systems were designed on ideas and theories that have significantly furthered the technologies of MIR.

5.1 Medieval Music Directory

The La Trobe University Medieval Music Database makes use of the SCRIBE computer program, which enables the encoding, storing, searching, and retrieval of medieval music in its original notation [93]. It has over 6,000 songs in its database, was created mostly for students and scholars of medieval music, and receives over 40,000 hits a month. Learn more at http://www.lib.latrobe.edu.au/MMDB/scribe.htm.

5.2 SEMEX

SEMEX uses what the authors call bit-parallel algorithms for locating matching of melodies [50]. Bit-parallel algorithms are very fast in practice, and SEMEX was the first of its kind to use almost all bit-parallel algorithms for both matching and retrieval. It has a limitation of only working on monophonic databases, but has achieved a significant increase in speed over dynamic programming techniques of on-the-fly melodic queries.

5.3 MELDEX

The New Zealand Digital Library's MELody inDEX (MELDEX) capitalizes on advances in digital signal processing, music representation techniques, and computer hardware technology to transcribe melodies automatically from microphone input [82]. It was created for librarians who are often asked to find a piece of music based on a few sung or hummed notes. It has recently been re-designed to fit within the greenstone digital library software application. An analysis of the usage logs showed that most users are entering text queries on a vague topic (for example, "fire") to see which tunes result from the search [48]. A user can also search the database by playing a query example on a virtual keyboard on a web page [16]. Designed, created, and used for a very specific purpose, the database is filled with 9,000 international folk tunes and can be searched at http://www.nzdl.org/musiclib.

5.4 Themefinder

Sponsored by the Stanford University Center for Computer Assisted Research in the Humanities, Themefinder consists only of a text interface used to help musicians, composers, and conductors search for themes in 20,000 pieces of music. The themes are queried by, and saved as, melodic contours of the music. The musical pieces from which the themes were derived come from the Classical, Baroque, and Renaissance periods, along with some European folk songs [16]. Learn more at http://www.themefinder.org.

5.5 Real-time in an Off-the-Shelf Computer

There is no real name for this system, but the Music Technology group at Fabra University created a system for use in explaining an idea in statistics. The purpose was to show the statistical significance in their matching technique, which yielded no false positives. They built a system using Hidden Markov Models and that ran the algorithm FASTA on a Pentium III 5000Mhz desktop, achieving near real-time song matching while listening to radio stations. Ten radio stations were monitored over twenty-four hours, and a 91% match rate was achieved [76]. While this match rate is good, it would not be acceptable for some of the scenarios discussed earlier.

5.6 MusArt

MusArt is a research project developing and studying new techniques for MIR. MusArt stands for Music Analysis and Retrieval Technology [105]. It makes use of the theme approach, mentioned earlier, by automatically building a thematic index of the pieces of music in its database. Indexing by theme is not an exact matching method, but it can greatly improve both the precision and speed of the retrieval system using the method. Since the average person generally remembers and requests the theme of a piece of music, systems using the method are able to search for just those smaller sections [105]. This system originally found 2653 themes in the 277 pieces of public domain music collected from a wide variety of genres [102]. Learn more at http://www.cs.cmu.edu/~music/search.

5.7 K-BOX

Along with other algorithm and system architecture changes to help speed up queries in a query-by-singing music retrieval system, the authors of K-BOX introduce a database-clustering scheme. The database is clustered to place songs of similar properties together in the same cluster, and the input query is identified as a member of a particular cluster [35]. Efficiency is enhanced further by using two-stage clustering: by segment length and by K-means of features. When a query is input, it is reviewed for length of segment and then compared to all the centers of each cluster; the closest is located, and then the query is compared to all others in that cluster. K-BOX, as with other systems mentioned, trades accuracy for speed in some cases.

5.8 VocalSearch System

VocalSearch uses a probabilistic string-alignment algorithm to measure similarity between targets and queries [103]. A sung example is transcribed and encoded with possible pitch and time constraints. A ranked list is presented to the user based on a similarity ranker that looks at the database of encoded melodies. As systems go, this appears to be the most advanced academic system available. The system is under active research and development; more information on the VocalSearch System can be found at http://jose.cs.northwestern.edu/mpWiki/index.php/Main_Page.

5.9 Shazam

Shazam's service is built around a patented and proprietary pattern recognition technology that can identify recorded audio under noisy conditions. This is used to

enable users of the service to capture audio samples by calling the service and sampling the audio they are hearing into their phone. The service records a ten second clip of the song, makes an audio-fingerprint of the song [16], looks for a match, and then returns the identification results (title, artist, album) to the user's phone [8]. The Shazam service currently holds over 3,200,000 music tracks [91], and the company recently launched new services online at http://www.shazam.com.

5.10 Classical Music Database

This database contains over 1,500 classical music melodies and covers the most famous ones. It makes use of MIDI files and simplifies the melody of a song in some cases to help with the user search function [80]. Users enter a melody using an online representation of a keyboard, and a visual representation of the melody can be seen. The melody entered can be manipulated and played until the user feels that the melody is right, after which the user clicks the search button, and the title of the closest match from the database is returned. Along with the title, the composer and a percentage-matched figure (based on the number of notes entered and the number of notes matched) are returned. More information is available at http://iwamura.home.znet.com.

5.11 Tyberis

TyMusicDB is a stand-alone freeware program able to recognize musical pieces or other audio data in real-time. The main purpose of the program is to monitor a radio station, television channel or other (streaming) audio source for specific songs, commercials or jingles. A log file is created with a detailed description of which songs

were played when, and how long. The recognition algorithm is designed to identify songs based on their acoustical properties and appears to be very robust against noise and other distortion. The program does not come with music included in the database to match against; rather, the software relies on the user loading songs into it from his or her collection of music and then fingerprinting each song [87]. The software is available at http://music.tyberis.com.

5.12 Gracenote

Gracenote is a technology company that provides what they call the Gracenote Media Recognition Service. It is an Internet-based service licensed to developers of software, CD players, CD burners, MP3 players and encoders, catalogers, and other applications. Gracenote charges a fee to companies and commercial software developers who make revenue-generating software applications featuring the service. The database has over 65 million songs and stores the song title, artist, album, track list, and other music-related information. iTunes, the Apple online music store, is a customer [42]. Learn more at http://gracenote.com.

5.13 Digital Nirvana

Another company making use of fingerprinting is Digital Nirvana. It includes a copyrighted content tracking and reporting service that markets to the music industry. The suite of applications was written to impose checks on music piracy and perform song tracking for the radio. The software can also monitor television stations, podcasts, landlines, and cellular telephone activity to automatically track any type of audio and the

time that the audio is being played, though it needs twenty seconds of audio to do so [36]. The website at http://www.digital-nirvana.com is available for more information.

5.14 MusicBrainz

Why not "name that tune for free" [92]? MusicBrainz is a music database that anyone can access to comment and label songs. It attempts to create a comprehensive music information site by enabling users of the site to add data around each song [9]. A non-profit organization, MusicBrainz released a free open-source service that recognizes file tags associated with digital music. It has been public since 1998, but AgentArts is the company that provides the technology used for the data mining of the information [89]. It is a public database of file-tagged digital songs [92], so it does not make use of any algorithms to encode melodic representations or song themes. You can use the MusicBrainz data either by browsing the web site (http://musicbrainz.org) or accessing the data from a client program.

CHAPTER 6

The Smith-Waterman Algorithm

In 1981, T. F. Smith and M. S. Waterman published a paper titled "Identification of Common Molecular Subsequences." In it, they described an algorithm that improved on previous research from each author. It measured the minimum number of events required to convert one sequence string of characters into another. The paper starts by stating:

> "The identification of maximally homologous subsequences among sets of long sequences is an important problem in molecular sequence analysis. The problem is straightforward only if one restricts consideration to contiguous subsequences (segments) containing no internal deletions or insertions" [94].

It is a trivial matter to find a string inside another. Simply match the first string A of length n to the second string B of length n+m by aligning their first characters together a0 and b0. Slide A along B by aligning a0 with b1, b2, b3, and so on, and register if there is a match. The algorithm defined in the paper above, which would later become commonly known as the Smith-Waterman algorithm, does just what was described and also suggests a method for expanding and contracting string A with substitutions and deletions in order to make it match string B. Each modification to the string has a numerical value attached to it so that each comparison (a0 with b0, b1, b2, etc.) can be

ranked and returned in order, typically from the least amount of modifications to the greatest. This algorithm is now used to do many things, but its first intent and largest use today is in the field of bioscience.

6.1 Biological Sequences

From simple paternity cases to complicated issues such as the "investigations of mosquito serpins affecting invasion by the malaria parasite" [28], biological sciences are heavily using sequence and genome analysis algorithms. As a bit of background information, the most basic data type in biology is a base such as A, C, T, U, and G, which in combination link together to form sequences called deoxyribonucleic acid (DNA) [47]. People, plants, and animals use DNA to create amino acids that join to form protein sequences. The more scientists know about DNA and protein sequences, the more they can understand how the human body works and make predictions on things like the behavior of organs, cures for diseases, and, more recently, cloning and regeneration.

The most basic sequence analysis task is to ask if two sequences are related [21, 41]. Coffee in the U.S. might be extremely similar to tea in the U.K. In genomics, a more likely scenario is a scientist performing a search of the human genome to see if humans carry similar genes (i.e. to identify sequences in the human genome that resemble another human, or something like the mouse gene, for example, based on similarity of sequence [88]). The main idea is to find a pair of segments, one from each of two long sequences, such that there is no other pair of segments with greater similarity [94], measured by match, mismatch, and gap scores [66]. When the Smith-Waterman

algorithm is involved, the similarity, or homology as it is called in biological sciences, takes into account any number of arbitrary length deletions and insertions to make the match. The homology detection that takes place by using the Smith-Waterman algorithm also enables biologists to do things such as characterize unknown DNA sequences and proteins by how closely they match other substances that have already been categorized [5, 17].

Another interesting use of the algorithm is the matching of records across health information systems. There are many data sources within the healthcare network of systems, and sometimes there are duplicate records or, more problematic, disjoint records where it is hard to know that a record X belongs to the same individual as record Y. Researchers are now looking into the possibility of tagging each record with a DNA sequence of the patient. It would then take a system using an algorithm like Smith-Waterman to match those sequences with a certain amount of assurance so that they could be linked [38].

6.2 The Algorithm

As stated, the most basic sequence analysis task in bioinformatics is to ask if two sequences are related. The most fundamental algorithm to address that issue is the Smith-Waterman algorithm [41]. Since Smith-Waterman is a dynamic programming algorithm, it is guaranteed to find the optimal local alignment with respect to the scoring system being used. Substitution scoring comes from a substitution matrix, and gap scoring comes from what is commonly known as a gap scoring scheme, both of which will be discussed later when reviewing actual implementation. Internally, the algorithm

creates a scoring matrix between the two sequences being compared. Backtracing starts at the highest scoring matrix cell and proceeds until a cell with score zero is encountered. The path, taken from highest score to zero, is the area of highest scoring local alignment. Using their original example from the 1981 paper, the two sequences to be compared are:

-A-A-U-G-C-C-A-U-U-G-A-C-G-G-
-C-A-G-C-C-U-C-G-C-U-U-A-G-

Given this simple example, the alignment that was returned was:

-G-C-C-A-U-U-G-
-G-C-C----U-C-G-

This example contains a mismatch (U and C), but also an internal deletion (A and --). Before the Smith-Waterman paper was published, there was no way to find this information in this form. The paper proved there was a mathematical way to search for pairs of maximally similar segments. Today, some regard the algorithm as too slow; many are trying to devise faster solutions. A parallel implementation of the Smith-Waterman algorithm would be ideal. True parallelization should not change the results produced, but just produce results faster. Research has been done in the areas of bit parallelism [51], heuristics (discussed later), and other techniques for increasing the algorithm's speed, but usually at the cost of accuracy, as they can only provide approximations of what Smith-Waterman would yield.

6.3 Adaptations

The major complaint about the Smith-Waterman algorithm is speed of execution. Said to be too slow for use with large databases, most adaptations of Smith-Waterman use heuristic approximation algorithms to gain speed. While gaining speed, the heuristics

can only approach the accuracy, sometimes called sensitivity, of the Smith-Waterman algorithm [21].

- BLAST stands for the Basic Local Alignment Search Tool and is very valuable in the field of bioinformatics. Like Smith-Waterman, it is primarily used for comparing biological sequence information, such as the amino-acid sequences of different proteins or nucleotides of DNA sequences [88]. Since the input and output for BLAST are very similar to the Smith-Waterman algorithm, the techniques used later to modify Smith-Waterman and perform music melody matching would apply to BLAST as well. It is, however, a heuristic search technique, as stated earlier, which means that users accept less than accurate results in exchange for getting them quickly [21].
- eBLAST improves on the sensitivity of BLAST by incorporating Smith-Waterman alignments. The algorithm involves taking the results after the BLAST algorithm is run and submitting them as input to the Smith-Waterman algorithm [5]. By doing this, the benefits of BLAST can be used to narrow the field of potential matches that Smith-Waterman takes time to process (10% of all returned results are given to the Smith-Waterman algorithm).
- The Diagonal Aggregating Search Heuristic (DASH) algorithm improves even on the speed of BLAST [77]. It is a three-stage algorithm requiring indexed versions of the database being searched. In addition, since the algorithm is highly parallel, there is a distributed version of the algorithm.

- Paracel BLAST is the only commercially supported BLAST solution for parallel computing [79]. It takes advantage of a high performance parallel computer to split large complex tasks across multiple processors automatically.

Though the algorithms listed above have progressively resulted in speed improvements over the Smith-Waterman algorithm, it must be stated again that each successive speed improvement has been at the expense of sensitivity [77]. This has serious consequences for those that use them. The fact is, the best related sequences actually might not be found in a search using the aforementioned tools [41]. It should be noted that effort was made to locate a publicly available parallelized implementation of the Smith-Waterman algorithm for a Cray supercomputer to use in the experiments for this paper, but no suitable one was found.

6.4 Use in MIR Systems

Some researchers have compared the bioinformatics methods as they pertain to music matching and ranking, finding that local alignment was more effective than almost any other method derived from information retrieval [4, 64]. When there is not an exact match, most MIR systems based on local alignment return their ranking of what they consider to be mathematically the closest ranked melody in the database to the submitted melody to be matched. How can their accuracy be proven, though? It is interesting to note that some researchers have checked their work by asking trained musicians to listen to the findings of their MIR systems and judge accuracy. When it comes to using the Smith-Waterman algorithm in these systems, most results show there is a strong agreement between the human subjects or musicians used and the rankings given by the

computer systems [57]. The reason it is so accurate is that melodies are easily represented as character strings. Once that is complete, the melody comparisons can benefit from the more mature research area of string matching. One step further, dynamic programming is the most popular technique of approximate string matching, so it is only natural to expect that it also can be broadly used in the area of melodic search [73]. In fact, dynamic programming-based techniques comparable to the Smith-Waterman algorithm are already being used for local alignment as well as longest common subsequence problems in several melody-matching experiments [4].

6.5 Summary

The Smith-Waterman algorithm is more accurate than any other method of comparing strings in the field of bioscience. Because melodies can be transformed into strings similar to those found in bioscience, Smith-Waterman is easily used for song melody data comparisons. In order to do so, the Smith-Waterman algorithm must be implemented in a programming language for this purpose, or an implementation of the algorithm must be acquired and modified. An implementation will be selected and modified as a model to show how other implementations, and even other algorithms, may be modified for this purpose as well.

CHAPTER 7

A Modified Implementation

As stated earlier, there are very evident needs for:

- a search engine solution for published melodies that ensures there are no exact matches, nor very close matches, to a new melody;
- a song-listening solution that matches to a database of known songs, that can accurately identify entire songs, and can also match small sections of songs (for example, partial songs played during television commercials);
- and song suggestion software, based only on the songs themselves.

It has also been stated that the bioinformatics algorithm known as Smith-Waterman is used in areas where the need exists to ensure there are exact matches, or to reveal close matches, and that the algorithm performs well where there are gaps in a biological sequence (much similar to breaks or interference in audio). Given the needs, and the mentioned benefits, it is probable that the Smith-Waterman algorithm can be applied to functions of music information retrieval (MIR). Also, though some systems are theme based and use things such as feature vectors and extraction [39] that are computed from the first n notes in a tune, then from the second note and successive n notes and so on, there are those that believe that it is important to maintain the integrity of the melodies searched [108]. Even those that have updated the Smith-Waterman algorithm with faster techniques know that a more mathematically rigorous alignment, such as Smith-

Waterman, would be desirable [5]. In short, the Smith-Waterman algorithm will be used here because it is the only option to find like melodies when comparing songs and achieve 100% accuracy [66]. The adaptations made to the algorithm can then be used as a model for those who would like to use other implementations or algorithms.

7.1 Base Algorithm

To create a program for melodic string matching, a suitable base algorithm implementation must be found and modified. The implementation must concisely execute the selected Smith-Waterman algorithm with no flaws. Starting from scratch is not needed, since much research has been done with the Smith-Waterman algorithm as discussed earlier. Many algorithms used to measure melodic similarity are actually text-based string-matching algorithms that have either been adopted directly or somewhat altered to suit this new role. One of the most commonly used of these is the edit-distance family of algorithms (along with variations), which essentially calculates the cost of taking one string/melody and converting it to another [57]. The Smith-Waterman algorithm has been written in many programming languages, including Delphi, Perl, Java, TCL/TK, Matlab, AWK [78], C, C++ [107], and more. For the purposes of this paper, it was believed that the best implementation would be one written in Java. When a Java coded program is compiled into Java bytecode, it allows the script file to run in any environment that is currently running Java Virtual Machine (JVM) [47]. No compatibility issues are known to exist from environment to environment.

7.1.1 Smith-Waterman Algorithm Implemented in Java

With the decision made to select an algorithm written in Java, a search was conducted that included talking to authors for some explanation of implementations, deciding on agreement of use, and questioning them on basic research. The algorithm implementation selected, "Smith-Waterman Algorithm Implemented in Java" [75], requires Sun Java Development Kit (JDK) version greater than or equal to 1.4 in order to compile the Java file, and Sun Java Runtime Environment (JRE) version greater than or equal to 1.4 to run the .class file. As stated earlier, the implementation selected is a single processor, single threaded version that will run on average desktop computers along with on the single processor of a Cray supercomputer.

7.1.2 Assumptions

For the purposes of this experiment, it is important to simply implement the Smith-Waterman algorithm using dynamic programming. It is being run in a controlled environment against only a few songs. It also must be noted that the code written does not take into consideration error-checking, improper input, nor does it represent an optimized implementation of the algorithm [75]. In addition, the program has a configurable variable that was set at the time of these experiments to accept input songs of 1024 characters of less. Some of the optimization limitations of the implementation could be solved with time. The future work section of this document describes some improvements that could be made.

7.1.3 Gap Scoring

When using an algorithm such as Smith-Waterman, substitutions and deletions can be made. Weights are given to each substitution and deletion, but the highest penalty occurs when there is a gap. In bioinformatics, a gap scoring scheme defines what the penalty is if there are consecutive gaps, for example:

AAAAAAAAA - - - - TCA
AAAAAAAAAGGGGTCA

When looking at this sample alignment, there is a question of how to penalize the existence of the four consecutive gaps (i.e. the existence of the substring " - - - - " in the alignment.) The three popular approaches, called gap scoring schemes, are one to one, fixed penalty, and exponential penalty. The one to one penalty method assigns the same penalty value for each gap. If there is a gap of length K, where K=4 in the example above, then the penalty is defined as P(K) = K*(-Q), where -Q is the gap penalty for the unit gap (a gap of length one). If -Q = -10, the penalty for the example above is P(K) = -40. The second approach, fixed, penalizes the gaps of length K in a more liberal way: P(K) = -Q, where -Q is the penalty for a unit gap of any length. If such a gap scoring scheme is adopted, the gap length is irrelevant. The penalty for a gap having length of K=4 would be the same as the penalty for a gap having length of K=6. The last gap scoring scheme is exponential in that it combines two penalties (a) -D is the penalty for one gap opening (b) -E is the penalty for gap length. A gap of length K would then have a total penalty of P(K) = -D + (K-1)*(-E) which simplifies to W(K) = -(D + (K-1)*E) [75].

The implementation selected for this experiment employs the first approach listed above. It also uses a variable for the penalty of one gap that can be modified in the input file. It was set to -8 (one over the largest distance penalty given to a rest) at the time of the experiments. Using the formula P(K) = K*(-Q), where Q now equals 8, a gap of 4 would receive a penalty of P(4) = 4*-8, or P(4) = -32.

7.1.4 Java Code

Taken from http://sedefcho.icnhost.net/web/algorithms/smith_waterman.html, the full and original code of the implementation can be viewed in APPENDIX A. Later in the paper, the modified version of the implementation will be viewed in granular specifics and explained in detail.

7.2 Adaptation for Melodies

The implementation selected had to be modified, along with its input file, for melody retrieval. In addition, decisions and assumptions had to be made on how to address the issues mentioned previously in the music information retrieval section of this paper. With melody matching, there are properties and practical problems that do not exist in regular string matching [73]. First, pitch was addressed by the transposition of melodies. A system to combat the issue of duration of notes had to be created, along with tempo and rhythm. Based on previous research not included in this paper, the decision was made to only contend with monophonic melodies. The input file also had to be drastically changed.

7.2.1 Pitch

A known method for dealing with pitch, described earlier, is only recording intervals between notes. This form of melody standardization allows melodies to be matched regardless of the key of the query or pieces in the database [4]. In the case of the implementation selected, the sequence matching works more efficiently when using actual letters, gaps between them, and weights associated with each gap distance. So that the algorithm could work as intended, instead of modifying the algorithm, all simple melodies will be transposed into the key of C for the purposes of this experiment. Though various other methods were attempted and appeared to work as efficiently, the idea of transposing was simple to implement and will ensure accuracy across all songs. The method will also preserve the melodies in their original state, merely moving them up or down the melodic note scale. Effectively implementing the transposition involves knowing the song's original key, and then using a formula to calculate what existing notes should be transposed to. The code written and added to the implementation transposes the input melody before any attempt to match it to a melody in the database, and the database of songs is run through the same program before being saved. Errors in transcription and variations in pitch can cause substantial mismatches between input melodies and the database [73], so the most exact method was implemented. Octaves were not taken into account due to the proof of concept nature of this experiment, but could be added into the implementation with modifications to the code and input file.

7.2.2 Duration

To handle duration of notes, the algorithm was modified to handle the concept of a rest by making changes to the input matrix and songs. The duration of notes and rests will work as if a metronome was set for the song, clicking multiple times per measure. The melody will be played, and each note will be recorded on the downbeat of the metronome. For example, "Twinkle, Twinkle, Little Star" was written in 4/4 time. Since it is a simple melody, there are literally four notes in the first measure and three notes in the second; the third note in the second measure is held for two beats. See APPENDIX B for a simple sheet notation of the song. To explain further, dividing the words of the song in 4/4 with four beats per measure would look like this: Twin-kle-Twin-kle-Lit-tle-Star-(hold)-How-I-Won-der… etc. The (hold) notation is used to repeat the note being played immediately before. Some songs use more than four notes in a measure, so to allow for more complicated songs to be used in this implementation and to normalize the data, it is best to use sixteen beats per measure (each beat mentioned earlier would receive four opportunities for notes or rests.) Using this approach, all songs can be reduced to strings of single notes. If a note is held for more than one of the sixteen metronome clicks in the measure, such as the "star" note mentioned earlier, the letter for that note will be repeated as long as it was held. This allows for drastic changes in tempos to be addressed with only single insertions or deletions, and a minor penalty in terms of edit distance [73]. This is beneficial because of the agreement this method has with the Smith-Waterman algorithm. For example, two songs that use the same notes, but different note durations, are still very much alike. One song may be twice as long as

the second but, in looking at the notes, exactly the same. Using this method of representing duration in the string itself, as well as rests and holds, allows the Smith-Waterman algorithm to find the matches between the two songs as it should. Using four measures of "Twinkle, Twinkle, Little Star" as an example to demonstrate how the melodies were modified to handle duration, four beats per measure yields:

C C G G A A G G F F E E D D C C
twin kle, twin kle, lit tle, star, (hold) how I won der what you are. (hold)

Taking the first two measures of the same melody and normalizing it to eight beats per measure yields:

C C C C G G G G A A A A G G G G
Twi n k le, twi n k le, li t t le sta r (hold)

Finally, taking those same two measures and applying one more division of notes to get sixteen beats per measure yields:

C C C C C C C C G G G G G G G G A A A A A A A A G G G G G G G G
Twi n k le, twi n k le, li t t le sta r (hold)

As stated earlier, rests will be treated the same as notes but symbolized by an R.

7.2.3 Tempo and Rhythm

Time signatures, such as 4/4 and 3/4, were addressed in the implementation modifications by allowing for different time signatures based on the earlier modification for duration. Time signatures can be defined as the base rhythm for a song. As mentioned before, 4/4 time means there are four beats to a measure. 3/4 time would mean there are only three beats per measure. The duration modification to melodies above mentioned sixteen beats per measure for 4/4 time. That allows for four notes per

beat. Applying the same approach for 3/4 time yields twelve notes per measure. The song “Three Blind Mice” is written in 3/4 time, and with the twelve beat per measure modification would look like this:

E E E D D D C C C C C C E E E D D D C C C C C C
three blind mice (hold) three blind mice (hold)

Given this approach, if a melody consists of the same notes and relatively similar durations, the tempo will not be a large factor in the issue of overall similarity. The implementation will be allowed to remain sensitive to the melody itself without having to deal with differences in tempo. This is important because the Smith-Waterman algorithm will be able to work as intended. Most research done with similar melody matching techniques did not address tempo at all. Some went as far as using only songs with the same time signature [27]. Granted, rhythm is not a necessary consideration for the way the melodies are implemented in this experiment. Already in string format, actual audio recordings will not be used. Tempo and time signatures, however, are an important aspect of music that should not be overlooked.

7.2.4 Polyphonic

Extracting a melody from a recorded piece of music is a complicated process. Most agree that it is necessary to reduce the set of notes against which a query should be matched to just the melody of each piece of music, as there would be far too many irrelevant matches otherwise [4]. So, following the prevalent approach in literature to convert both target and query into monophonic sequences of notes [26], for the purposes of this experiment only monophonic melodies of songs will be used.

7.3 Modified Implementation

The non-commented full code of the modified implementation can be found in APPENDIX C. Important segments of the code will be shown and discussed here as they pertain to modifications made. The modifications made to the implementation are a direct result of taking an algorithm used for one thing (bioinformatics) and mapping it to another discipline (melody recognition). As others who work with the Smith-Waterman algorithm have done, ignoring efficiency issues at this time allows for the ultimate accuracy and flexibility of the system [73].

7.3.1 Initialization

The first part of any program like this starts with instantiating variables and initializing them with default data. Again, see APPENDIX C for details. In this program, one of the main variable constructs is the cost matrix (costMatrix), which is read in from the input file and filled in by the parseInput function. A weight matrix (weightMatrix) is also initialized and will be used to perform the Smith-Waterman algorithm. Other smaller functions such as InputStreamReader, stringReplace, and getNextToken are all used for that purpose as well; their work is critical, but not worthy of detail here.

7.3.2 Sharp and Flat Conversions

The function replaceDoubleLetterNote is used to convert the input song values to single letters so they can be more easily matched to other songs. As explained earlier,

notes like C# and Db are the same, so they both get the new value of V. D# and Eb are translated to W, and so on. Here is a complete listing of the mappings:

```
inStr = stringReplace(inStr,"C#", "V");
inStr = stringReplace(inStr,"Db", "V");
inStr = stringReplace(inStr,"D#", "W");
inStr = stringReplace(inStr,"Eb", "W");
inStr = stringReplace(inStr,"F#", "X");
inStr = stringReplace(inStr,"Gb", "X");
inStr = stringReplace(inStr,"G#", "Y");
inStr = stringReplace(inStr,"Ab", "Y");
inStr = stringReplace(inStr,"A#", "Z");
inStr = stringReplace(inStr,"Bb", "Z");
```

7.3.3 Transposing

After the sharps and flats are converted, the input string is transposed into the key of C. To do this, a transposition matrix was coded into the program. The function transposeNote shown below, takes the key of the input song entered in the input file (called charKey), and the input song string of notes itself and transposes the song.

```
int keyIndex = -1;
for (int i=0; i<13; i++) {
  if (key == transposeTable[0][i])
     keyIndex = i;
}
returnString = inStr;
if (keyIndex > 0) {
   int position = -1;
   for (int i=0; i<returnString.length(); i++) {
      for (int j=0; j<13; j++) {
            if (returnString.charAt(i) == transposeTable[keyIndex][j])
               position = j;
      }
      if (returnString.charAt(i) != 'R')
         returnString = replaceCharAt(returnString,i,transposeTable[0][position]);
   }
}
return returnString;
```

Each character in a key other than C is shifted down the number of letters, or more accurately notes, required to translate the song to C.

7.3.4 Smith-Waterman

Once the input song has been dealt with, the Smith-Waterman matching can be done. As mentioned earlier, it makes use of a weightMatrix variable. The function called to start the process is the fillMatrix function. The Smith-Waterman implementation in this Java program is fairly straightforward.

7.3.5 Timing

Another modification made to the original implementation is the addition of this line at the beginning of the program:

```
long t = System.currentTimeMillis();
```

It logs the time at the start of the program in milliseconds and assigns that value to t. At the end of the program, the following line appears:

```
System.out.println("This program took " + (System.currentTimeMillis() - t)
+ " millisecond.");
```

This takes the current time and subtracts the time stored in t from the beginning of the program execution. That leaves the total time that the program took to process, and the println command prints the resulting value to the screen with the rest of the results.

7.3.6 Implementation Modification Summary

By the successful modification explanations above, it is evident that the Smith-Waterman algorithm implementation selected can be used as a model for modifying

implementations that are more complicated or algorithms that do the same thing. Though the implementation had to be modified drastically to accommodate the melodic input, transposition, matching, and output, the specific code used to perform the Smith-Waterman algorithm was not touched. It is believed that this would hold true for any implementation of this or another bioinformatic algorithm, which would allow more robust solutions to be used.

To match with the new code, the input file of the implementation had to be modified as well.

7.4 Command Line and Input File

The Java program implemented needs, as stated earlier, the Sun JRE version 1.4 installed to run, but it also needs a text file as input. To execute, the input of the program must be redirected to use the file by the following command line argument:

```
java -jar SW.jar < input1.in
```

Where "java" is the execute command for JDK or JRE, "-jar" indicates the type of file to be executed, "SW.jar" is a zip file containing all of the compiled code, "<" is the character used to say where the input is coming from, and "input1.in" is the name of the input file. An original input file for the implementation looked like this:

```
    A   G   C   T
A   10  -1  -3  -4
G   -1  7   -5  -3
C   -3  -5  9   0
T   -4  -3  0   8

GAP PENALTY: -5

AATGCCATTGACGG
```

CAGCCTCGCTTAG

AATGCCATTGACGGGCTCTCTGGAAAGAGAGAGAGAGATTTATATGCC
GTCCTCTCAGAGAGGGAAATTGGGAAATGGGCGCGCGC

ATCAGAGTC
GTCAGTCA

AAAAAAAAATCA
AAAAAAAAAGGGGTCA

After modifications were made to the program to handle melodies, a complete, and non-annotated sample input file looks like this:

	C	V	D	W	E	F	X	G	Y	A	Z	B	R
C	10	-1	-2	-3	-4	-5	-6	-5	-4	-3	-2	-1	-7
V	-1	10	-1	-2	-3	-4	-5	-6	-5	-4	-3	-2	-7
D	-2	-1	10	-1	-2	-3	-4	-5	-6	-5	-4	-3	-7
W	-3	-2	-1	10	-1	-2	-3	-4	-5	-6	-5	-4	-7
E	-4	-3	-2	-1	10	-1	-2	-3	-4	-5	-6	-5	-7
F	-5	-4	-3	-2	-1	10	-1	-2	-3	-4	-5	-6	-7
X	-6	-5	-4	-3	-2	-1	10	-1	-2	-3	-4	-5	-7
G	-5	-6	-5	-4	-3	-2	-1	10	-1	-2	-3	-4	-7
Y	-4	-5	-6	-5	-4	-3	-2	-1	10	-1	-2	-3	-7
A	-3	-4	-5	-6	-5	-4	-3	-2	-1	10	-1	-2	-7
Z	-2	-3	-4	-5	-6	-5	-4	-3	-2	-1	10	-1	-7
B	-1	-2	-3	-4	-5	-6	-5	-4	-3	-2	-1	10	-7
R	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	-7	10

GAP PENALTY: -8

Input Song - C :
CCCCDDDDEEEECCCCCCCCCDDDDEEEECCCCEEEEFFFFGGGGGGGGE
EEEFFFFGGGGGGGGGGAAGGFFEEEECCCCGGAAGGFFEEEECCCCCCC
CGGGGCCCCCCCCCCCCCGGGGCCCCCCCC

Three Blind Mice -
EEEEEEDDDDDDCCCCCCCCCCCCCEEEEEEDDDDDDCCCCCCCCCCCCG
GGGGGFFFFFFEEEEEEEEEEEEGGGGGGFFFFFFEEEEEEEEEEGGCCCCC
CBBAABBCCCCGGGGGGGGCCCCCCBBAABBCCCCGGGGGGGGCCCC
CCBBAABBCCGGGGGGGGFFEEEEEEDDDDDDCCCCCCCCCCCCC
Twinkle Twinkle Little Star –
CCCCCCCCGGGGGGGGAAAAAAAAGGGGGGGGFFFFFFFFEEEEEEEED

DDDDDDDCCCCCCCCGGGGGGGGFFFFFFFFEEEEEEEEDDDDDDDDGG
GGGGGGFFFFFFFFEEEEEEEEDDDDDDDDCCCCCCCCGGGGGGGGAAA
AAAAAGGGGGGGGFFFFFFFFEEEEEEEEDDDDDDDDCCCCCCCC

Taking the same sample input file, breaking it into parts, and explaining each for understanding, it is easy to see what was changed and why.

7.4.1 Weighted Distance Matrix

Much like the four letter alphabet used in DNA (A, C, G, and T), music has its own alphabet (A, B, C, D, E, F, G). As detailed in a previous section, there are also half steps between some notes. Between F and G, for example, is a note that can be labeled F# (pronounced F-sharp, a half step above F) or Gb (pronounced G-flat, a half step below G). Though the accommodation of these notes in the code is explained above, the first line of the input grid is as follows:

C V D W E F X G Y A Z B R

Starting with C because the most common musical scale also starts with C, the next character (V) has been used to represent C# and Db, D = D, W = D# and Eb, and so on. The letter R, at the end of the line, is used for a rest in the music. The next line, and all those subsequent, shows the weighting between matched letters.

	C	V	D	W	E	F	X	G	Y	A	Z	B	R
C	10	-1	-2	-3	-4	-5	-6	-5	-4	-3	-2	-1	-7

If the letter at the beginning of line two was compared to each letter in line one, a perfect match would be given the score of 10 (C matches C). The letter V, noting a C# or Db, is one half step above C, so the penalty given is just -1. Likewise, since the scale is circular, B is one half step below C and is given a penalty of -1. The further away from

the note being compared, the larger the penalty given. F# or Gb is the furthest away from C and has the highest penalty (-6). Rests are given a penalty of -7 when compared to any note, and have a score of 10 on a match much like other notes. Using this method of weighting with scoring and penalties, it is easy to mathematically equate a score for a comparison between string sequences. While this score does not mean much alone, when evaluated with other scores of comparisons it can be ranked to determine if one comparison is a closer match than another.

7.4.2 Gap Penalty

The next line in the input file is a variable that can be changed:

GAP PENALTY: -8

Chosen here to be incremented two more than the furthest note away from a match and one more than a rest, the gap penalty is enforced when a gap is added inside a string for optimum alignment. Gap scheme approaches were explained in a previous section, but for the purposes of this experiment, each gap was given the same penalty (i.e. 1 gap = -8, 2 gaps = -16, 3 gaps = -24, etc.).

7.4.3 Input Song

The input song is the song that is being compared to the other songs in the music database located at the bottom of the input file. In the original implementation (before modification) only two strings were compared. The implementation was modified to compare the input song with multiple songs and print out the results of each. The input song in the input file has two parts:

Input Song - C :
CCDDEECCCCDDEECCEEFFGGGGEEFFGGGGGAGFEECCGAGFEECCC
CGGCCCCCCGGCCCC

The letter between the dash (-) and the colon (:), designates in what key the song is written. As previously explained, the songs in the comparison database were all entered in the key of C, and the code was modified to transpose the input song to C for matching. In order to do so, the program needs to be told the original key. The input song above is "Frère Jacques," already notated in the key of C. After the colon (:), there is a string of the actual notes of the song listed in sixteen notation symbols per measure.

7.4.4 Song Database

At the bottom of the input file are songs that the modified implementation will use to compare to the input song explained above. Simple children's songs were selected for this experiment because of the simplicity of the melodies, notes, and time signatures. In addition to "Twinkle, Twinkle, Little Star" and "Three Blind Mice" above, APPENDIX D lists melodies added to the input file for matching (some were originally conceived and some taken from other sources [7]). As stated earlier, all melodies have been transposed to the key of C and converted to four notes per beat (sixteen notes per 4/4 time measure and twelve notes per 3/4 time measure).

7.5 Output

Original output for the program, specifically, for the original sample input file shown earlier, looked like this:

ATGCCATTGAC
A-GCC-TCG-C

Score: 44

GGCTCTCTGGAAAGAGAGAGAGAGATTT--ATATG--C-C
GTC-CTCT--CA-GAGAGGGA-A-ATTGGGAAATGGGCGC
Score: 138

TCAGAGTC
TCAG--TC
Score: 41

AAAAAAAAA----TCA
AAAAAAAAAGGGGTCA
Score: 97

Each string pair was matched against the other, and the modifications needed to make each string look like the other were taken by the program. The matching weights (positive numbers) and penalty scores for gaps and substitutions (negative numbers) were added and a number is produced labeled "Score." An output file after the program was modified looks like this:

Key : C
Input Song :
CCCCDDDDEEEECCCCCCCCCDDDDEEEECCCCEEEEFFFFGGGGGGGGE
EEEFFFFGGGGGGGGGGAAGGFFEEEECCCCGGAAGGFFEEEECCCCCCC
CGGGGCCCCCCCCCCCCCGGGGCCCCCCCC
Input Song after removing double char. notes :
CCCCDDDDEEEECCCCCCCCCDDDDEEEECCCCEEEEFFFFGGGGGGGGE
EEEFFFFGGGGGGGGGGAAGGFFEEEECCCCGGAAGGFFEEEECCCCCCC
CGGGGCCCCCCCCCCCCCGGGGCCCCCCCC
Input Song after fixed to C key :
CCCCDDDDEEEECCCCCCCCCDDDDEEEECCCCEEEEFFFFGGGGGGGGE
EEEFFFFGGGGGGGGGGAAGGFFEEEECCCCGGAAGGFFEEEECCCCCCC
CGGGGCCCCCCCCCCCCCGGGGCCCCCCCC

DDDDEEEECCCCCCCCDDDDEEEE------CCCCEEEEFFFFGGGGGGGG----
EEEEFFFFGGGGGGGGGGAAGGFFEEEECCCC--
GGAAGGFFEEEECCCCCCCCCGGGG--CCCCCCCC----CCCCGGGGCC--
CCCCCC

DDDDDDCCCCCCCCCCCCCEEEEEEDDDDDDCCCCCCCCCCCCCGGGGGG
FFFFFFEEEEEEEEEEEEGGGGGGFFFFFFEEEEEEEEEEGGCCCCCCBBAA
BBCCCCGGGGGGGGCCCCCCBBAABBCCCCGGGGGGGGCCCCCC
Three Blind Mice - Score: 386

EEEECCCCCCCCDDDDEEEECCCCEEEE----FFFFGGGGGGGGEEEEFFFF----
GGGGGGGGGGAAGGFFEEEECCCCGGAAGGFFEEEECCCCCCCCGGGG
EEEEEEEEDDDDDDDDCCCCCCCCGGGGGGGGFFFFFFFFEEEEEEEEDD
DDDDDDGGGGGGGGFFFFFFFFEEEEEE--EEDD--
DDDDDDCCCCCCCCGGGG
Twinkle Twinkle Little Star - Score: 226

This program took 23 milliseconds.

There are three parts to the output file: the header information, the closest aligned substrings with their scores, and the milliseconds.

7.5.1 Header

The first few lines of the output repeat the key and the original input song for verification. As shown in the code review, the next step is to remove double character notes from the input song. With this first stage conversion complete, the input song is once again printed out to show what was done, labeled “Input Song after removing double char. notes.” Next, the input song is then transposed into the key of C, labeled “Input Song after fixed to C key.” It is also printed out to show that the program is doing a proper conversion. This converted and transposed song is used to match against the rest of the input file.

7.5.2 Strings and Scores

When the input song is matched against the other songs in the database, the program performs a number of tasks using the Smith-Waterman algorithm. Ultimately,

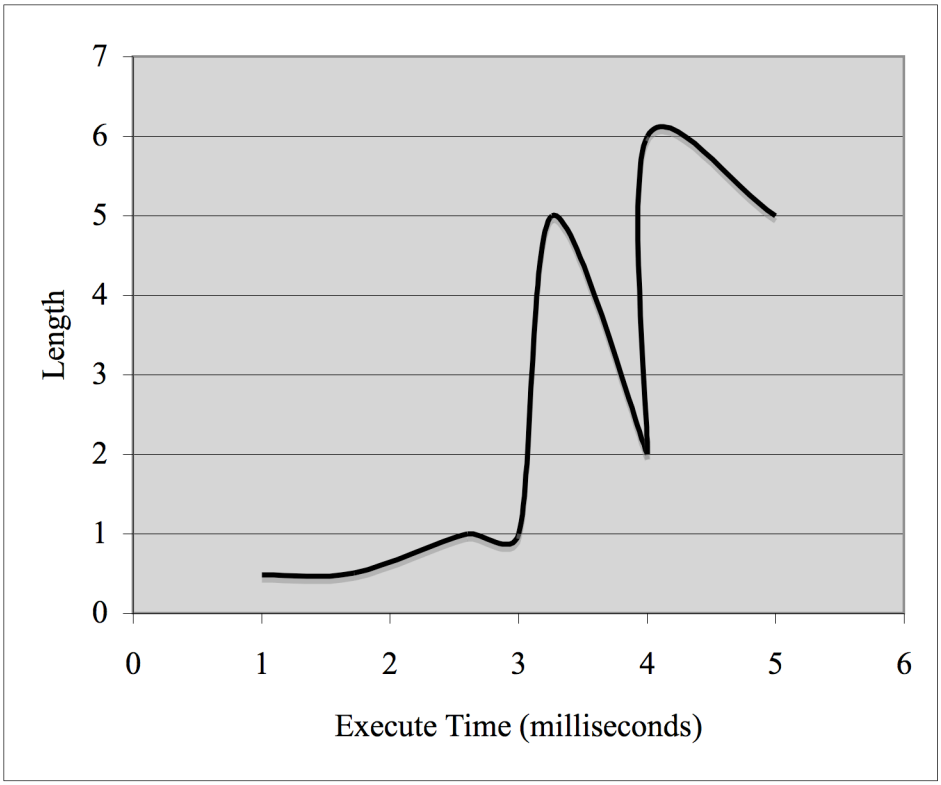
substitutions are made, gaps are inserted, and a sequence of notes found to be the best match between the two is found. The sequence for each match is then printed on screen for human observation and verification. Both of the strings and their modifications are printed. A dash (-) is used to represent a gap. After the pair of sequences is printed for each song, the title and score of the song are shown. The title is straightforward, but the score represents the culmination of the matches, insertions, deletions, and gap penalties enforced by the program while performing the Smith-Waterman algorithm. Because of how the weighted input matrix was configured, the higher the number, the closer the match between the input song and the song to which it was compared. Therefore, the song title in the output file with the highest score behind it is the song that most closely resembles the input song.

7.5.3 Timing

The last part of the output is the timing. The application was written in a language specifically designed to make it easy to run on multiple platforms, and the main purpose for including the timing in the output is to help with the identification of an increase or decrease in speed when the program is ported. It is expected that the algorithm will take less time on newer hardware, but it is interesting to understand how the program execute time will increase as the song database grows. The amount of increase is not entirely dependant on the number of songs added, but instead on the length of those songs and their variance from the input song to be matched. During the examples given in the last section, it was found that the input song, or song to be matched, has the greatest impact on execute time. The larger the difference of the input

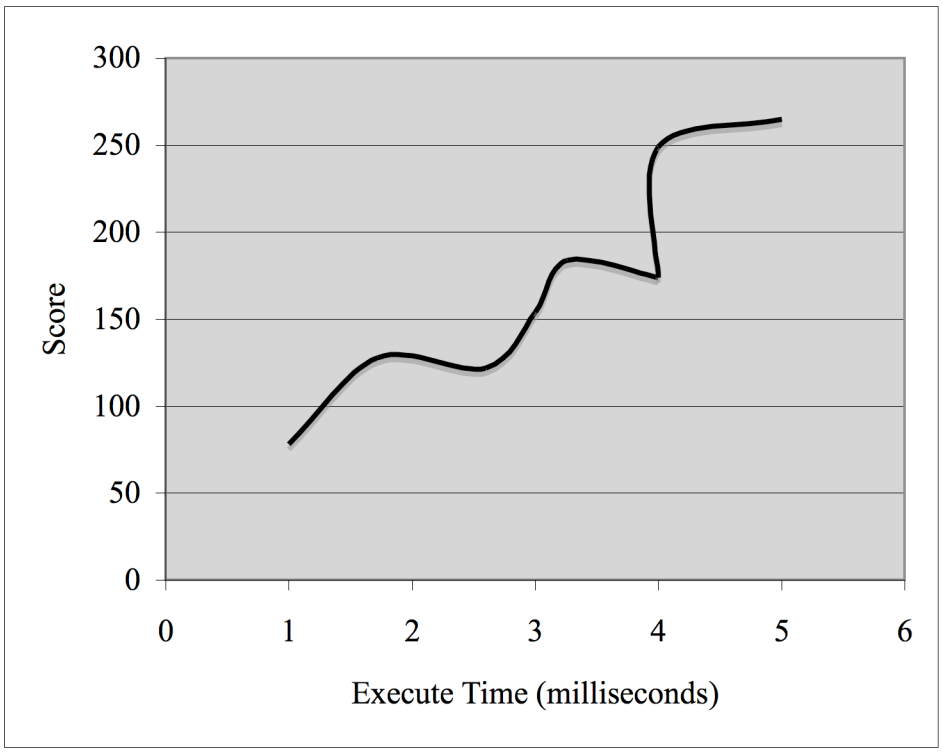
song from the song being matched against in the database, the longer the execute time due to the number of failed attempts for insertions, deletions, and substitutions to find maximally similar substrings. To demonstrate, the program was executed against each song in the database individually. The execute time and length of the melody were recorded and are straightforward, but the score of the string was noted and found to be an indication of how hard the program had to work in the matching. The lower the score, the less success in using insertions, deletions, and substitutions, and requiring more time in the attempt. A higher score would indicate a closer match which and indicates needing less time. Figure 7.5.3.1 shows the comparison of length with execute time.

Figure 7.5.3.1. Length of compared song in database by execute time.



There appears to be a steady progression upward in an almost linear fashion, but if a melody is very short or very long the amount of time the program spends comparing that song will vary too greatly to predict. On the other hand, Figure 7.5.3.2 shows the comparison of score with execute time. It shows a more normalized ascent.

Figure 7.5.3.2. Score of comparison song in database by execute time.



From Figure 7.5.3.2, we can conclude that the input song has much to do with the length of program execution time. If the song is very similar to a song in the database, the matching of those songs will not take long. In addition, we can also conclude that if the songs in the database are very similar to each other, and a similar song is used for input, the program will complete its entire run through the database faster.

CHAPTER 8

Results

Now that the code, input, and output have been explained, it is interesting to analyze the results of running the program. First, the program was run using the songs in the input file as a control. For example, when the song "BINGO" was used as the input file, the results were as follows:

```
CFFFFCCCCDDDDCCCCFFFFGGGGAAAAFFFFAAAAAAAAZZZZ
ZZZZGGGGGGGGAAAAAAAAFFFFFFFFGGGGGGFFEECCDDEEF
FFFFFFF
CFFFFCCCCDDDDCCCCFFFFGGGGAAAAFFFFAAAAAAAAZZZZ
ZZZZGGGGGGGGAAAAAAAAFFFFFFFFGGGGGGFFEECCDDEEF
FFFFFFF
BINGO  - Score: 970
```

Obviously, "BINGO" is a one to one match and, as a control, should have the highest score. The score is calculated as 970 because there are 97 characters in the song, and each match in the weighted distance matrix entered in the input file is scored with a numerical value of 10. There are no substitutions, deletions, or gaps here to minimize the score at all. The following is the output when "BINGO" was run against the second song in the input file:

```
CCCCDDDD----CCCC----
FFFFGGGGAAAAFFFFAAAAAAAAZZZZZZZZGGGGGGGGAAAA
AAAAFFFFFFFF--GGGGGGFFEE--CC
CCCCDDDDEEEECCCCEEEEFFFFGGGGGGGGEEEEFFFFGG--------
--GGGGGGGGAAGG----FFEEEECCCCGGAAGGFFEEEECC
Frere Jacques  - Score: 158
```

The dashes (-) indicate gaps that were inserted to achieve the optimum alignment of the two sequences. In the case of the first four dashes, Es are in "Frère Jacques," but they are not in "BINGO." Note the score of 158 is very low compared to the exact match above of 970.

GGGGAAAAAAAAFFFF----FFFFGGGGGGFFEECCDDEEFFFFFFFF
GGGGAAAAGGGGFFFFEEEEFFFFGGGGGGGGDDDDEEEEFFFFF
FFF
London Bridge - Score: 264

Having the second highest score after the exact match, "London Bridge Is Falling Down" is more like "BINGO" than any other song compared. Nine characters into the strings, the G and A mismatches are only a -2 penalty, there is a gap where Es are replaced, and a few other mismatches throughout. Overall, these are two close sequences. The rest of the "BINGO" comparison strings and scores are as follows:

ZZZZZZZZ
ZZZZZZZZ
Mary Had a Little Lamb - Score: 80

FFFFCCCCDDDD----
CCCCFFFFGGGGAAAAFFFFAAAAAAAAZZZZZZZZGGGGGGGG
AAAAAAAAFFFFFFFFGGGGGGFFEECCDDEEFFFFFFFF
FFFFCCCCDDDDDDDDCCCC--------CCCC----AAAAAAAA--------
GGGGGGGG--------FFFFFFFFRRRRRR----CCFFFFFFFFFFFF
Old MacDonald - Score: 150

DDDDCCCCFFFFGGGGAAAAFFFFAAAAAAAAZZZZZZ--------
ZZGGGGGGGGAAAAAA
DDDDCCCCDDEEFFGGAAAA----
AAAAAABBCCCCCCCCCCCCCCBBAAGGXXGGAAAAAA
Take Me Out to the Ballgame - Score: 168

CCCCDD--
DDCCCCFFFFGGGGAAAAFFFFAAAAAAAAZZZZZZZZGGGGGG
GGAAAA--AAAAFFFFFFFFGGGGGGFFEE
CCCCBBAABBCCCCGGGGGGGGCCCC----CCBBAA--BBCCCC--
GGGGGGGGCCCCCCBBAABBCCGG--GGGGGGFFEE
Three Blind Mice - Score: 134

GGGGGGGGAAAAAAAAFFFFFFFFGGGGGGFFEECCDDEE
GGGGGGGGAAAAAAAAGG------GGGGGGFFFFFFFFEE
Twinkle Twinkle Little Star - Score: 190

FFFFCCCCDDDDCCCCFFFFGGGGAAAA
FFFFEEEEDDDDCCCCBBBBGGGGAAAA
Yankee Doodle - Score: 160

Each song was then used as the input song and compared to every other song; the results are shown in Table 8.1.

Table 8.1. Output scores from matching sample songs.

	BINGO	Frère Jacques	London Bridge	Mary Had a Little Lamb	Old MacDonald	Take Me Out to the Ballgame	Three Blind Mice	Twinkle Twinkle Little Star	Yankee Doodle
BINGO	970	158	264	80	150	168	134	190	160
Frère Jacques	158	1280	272	148	188	182	386	226	300
London Bridge	264	272	1280	0	128	202	188	284	160
Mary Had a Little Lamb	80	148	0	1200	80	140	120	80	172
Old MacDonald	150	188	128	80	1200	214	294	538	152
Take Me Out to the Ballgame	168	182	202	140	214	1970	266	226	284
Three Blind Mice	134	386	188	120	294	266	1920	680	262
Twinkle Twinkle Little Star	190	226	284	80	538	226	680	1920	152
Yankee Doodle	160	300	160	172	152	284	262	152	1280

The shaded cells of Table 8.1 show the highest match, excepting exact matches of the same song, of the input song in the left column to the songs in the header row. The scores fluctuate because the songs vary in size, but it is easy to see which songs are most like others in melody. "Twinkle, Twinkle, Little Star" matches three of the input songs the best, and "Three Blind Mice" and "Yankee Doodle" both match best with two of the input songs. A few other songs were matched against the existing songs; Table 8.2 shows the results.

Table 8.2. Output scores for three more songs.

	BINGO	Frère Jacques	London Bridge	Mary Had a Little Lamb	Old MacDonald	Take Me Out to the Ballgame	Three Blind Mice	Twinkle Twinkle Little Star	Yankee Doodle
It's a Small World	128	250	250	160	120	184	322	200	290
Row, Row, Row, Your Boat	92	450	276	160	100	154	276	206	264
Ring Around the Rosies	116	184	88	174	150	243	268	132	172

Of the three new songs introduced, two of them most closely matched the song "Three Blind Mice," a pattern continued from the results discussed earlier. It is no surprise that

“Row, Row, Row Your Boat” and “Frère Jacques” are closely matched as they were derived, or have origins such that they appear to be from, the same melody.

Each of the experiments listed above were conducted on multiple machines to see if there were differences in the time needed to run the program. Though the implementation was run on a supercomputer, it was not able to take advantage of its multiple processor feature. In fact, for single processor programs, the supercomputer is slower than typical average desktop computers available. Specifically, the same input file was run on an Intel Xeon X5355 running at 2.66 GHz in a Dell desktop and on the Ada Intel Opteron 275 at 2.2 GHz in the Cray supercomputer. The program took 151 milliseconds to run on the desktop and 156 milliseconds to run on the Cray.

CHAPTER 9

Conclusion

Music is a part of everyday life, mostly in the form of songs that are written, recorded, published, and finally enjoyed, but the music industry faces challenges of copyright infringement, royalty distribution, and effective music suggestion. A superior method of song searching could advance the industry, support musicians, and introduce the listening public to previously unknown works.

A more advanced song searching technology is a tall order, though. To search for a song, a computer must be able to identify a unique characteristic of that song to match. Humans usually find the melody the most recognizable part of a song, and though computers could be programmed to do the same, the melodies must first be documented. Melodies can be documented in different ways, including pulling out certain portions called themes, but the universal technique of documenting a melody is simply identifying a series of notes with letters in a string. A number of companies and systems do melody and song matching now, but none seem to be an industry standard. Research has also been done in the area of song matching, resulting in patents and tools for finding songs, but many of these are based on sound waves and other auditory methods that use techniques that cannot approximate matches while ensuring accuracy.

Other fields of research, such as DNA matching in bioscience, make heavy use of sequence matching algorithms. The Smith-Waterman algorithm is a very powerful tool used in this manner and has a mature body of research around it, as well as research

around improvements and adaptations. It is the purpose of this paper to show that bioinformatic algorithms, such as Smith-Waterman, are uniquely suited for song matching and that existing implementations can be modified for this purpose. For experiments, an specific implementation of the Smith-Waterman algorithm was found in Java. Assumptions were made on how to implement the program that ultimately resulted in modifications to the code, the input file, and output. The main issues when trying to match songs are pitch, duration, tempo, and rhythm; the modified implementation overcomes these by converting sharps and flats and transposing the input song before running the Smith-Waterman algorithm and writing the output. It was tested with multiple input files that included a weighted distance matrix and a gap penalty variable. The output consisted of two strings showing the substitutions, gaps, and scores to see which song was a closer match. In the end, the exercise revealed that algorithms like Smith-Waterman are a perfect fit for song melody matching.

The issues of copyright infringement, royalty distribution, and music suggestion can all be addressed by applying the deoxyribonucleic acid (DNA) string matching algorithms to the problem of song melody recognition. Specifically, the Smith-Waterman algorithm's properties of approximation, accuracy, and the inherent ranking of scores to show degrees of matching are essential components to solve the issues facing existing search programs. Using a system which includes a Smith-Waterman implementation, songwriters could find a close match, rights organizations could be certain of findings, and businesses could offer consumers a list of ranked songs based on preferences. It will be exciting to see what happens as more researchers look into this

finding and create robust and efficient solutions for music matching including modified algorithms typically used in the area of bioinformatics and DNA string matching.

CHAPTER 10

Contribution

Striving to solve song identification concerns surrounding the music industry, this paper shows how existing deoxyribonucleic acid (DNA) string matching algorithms, with modifications to handle issues of pitch, note duration, and song tempo, can be made suitable to compare song melodies. An explanation of the Smith-Waterman algorithm and its application in the biosciences demonstrate how it is uniquely suited for matching a given song to a database of songs. The modifications made to the Smith-Waterman algorithm have been explained in detail so that others will understand how to adapt similar implementations or other DNA string matching algorithms.

No suitable synopsis of music information retrieval (MIR), nor the issues surrounding the research, could be found when researching this topic. There was also no singular place to view a sampling of the systems currently being used in the field. The definition of MIR, current research issues, and example list of existing song search applications given in this paper can be used as a guide for those interested in MIR research and it's present state.

This paper also presents a clear discussion of the issues and problems surrounding the music industry (i.e. copyright infringement, royalty distribution, and the introduction of new songs to consumers), along with an explanation of music, melodies, themes, and how melodies of songs are documented for those who are unfamiliar with music and musical terms.

CHAPTER 11

Future work

Though the modified implementation works as intended in its existing state, it would be beneficial to explore a more efficient implementation of the algorithm. Moreover, a parallel implementation of the Smith-Waterman algorithm would provide an increase in speed over the current implementation, while maintaining the same level of sensitivity [41]. The input file is cumbersome and should be split into pieces. The program would need to be modified to take an actual played melody, say from a midi keyboard, and to then search a database structure consisting of thousands of songs. Along those same lines, error handling was not done during this experiment but should be included in any future code. Lastly, the modifications made to the existing algorithm do not handle octaves and time signatures fully. To handle octaves, a numeric symbol would be placed after each note to indicate which octave it was in (typically noted by using a standard eighty-eight key piano). For time signatures, much like octaves, values would be assigned to the note durations so that they could be matched [108].

GLOSSARY

Ada: the Cray XD1 Supercomputer Cluster located at Rice University.

Algorithm: a step-by-step procedure for carrying out a mathematical computation to solve a problem.

American Society of Composers, Authors and Publishers (ASCAP): one of the top three performance rights organizations in the United States along with BMI and SESAC.

Broadcast Music, Inc. (BMI): one of the top three performance rights organizations in the United States along with ASCAP and SESAC.

Copyright Infringement: reproducing some aspect of a prior copyrighted work.

Copyright: the right to reproduce, or authorize others to reproduce, musical works.

Deoxyribonucleic Acid (DNA): a nucleic acid that contains the genetic instructions used in the development and functioning of all known living organisms.

Duration (of a note): length of a note, or how long the note is held.

Flat: a note that is lower in pitch by a semitone (half step), for example, a G flat (noted Gb) is a half step below G.

Frequency: can describe a music note by the number of cycles per unit time of a sound wave; for example, the note A to which an orchestra tunes has a frequency of 440 Hertz (Hz), or cycles per second, also see Pitch.

Heuristic: a trial-and-error method of problem solving used when an algorithmic approach is impractical.

Hidden Markov Model (HMM): a statistical model in which the system being modeled is assumed to be a Markov process with unknown parameters, and the challenge is to determine the hidden parameters from the observable parameters.

Hook: see Theme.

International Symposium on Music Information Retrieval (ISMIR): a forum for those involved in work on accessing digital musical materials.

Java Developers Kit (JDK): contains the minimal set of tools needed to develop and compile Java programs.

Java Runtime Environment (JRE): a software bundle that allows a computer system to run a Java application.

Java Virtual Machine (JVM): a piece of software that runs Java programs.

Melody: a succession of notes in a song, varying in pitch, taking an organized and recognizable shape.

Metronome: an instrument that is used to indicate or regulate the tempo of a song.

Monophonic: a singular melody, with only one note played at a time.

Music Information Retrieval (MIR): a growing body of research to process information and search music databases by content.

Octave: the interval between two notes that have the same name, the second note will be eight steps from the first and will either be double (higher) or half (lower) the frequency.

Performance Rights Organizations (PRO): collects royalties on behalf of its members and then distributes these royalties minus administration costs.

Pitch: the highness or lowness of a sound / note, also see Frequency.

Polyphonic: music that has more than one sound happening at any given time.

Query-By-Humming (QBH): a technology allowing users to find a song in a database by humming the melody into a microphone connected to a computer.

Rhythm: the time distance between each note in a melody.

Royalty: payment to the owner(s) of a song, usually governed by an agreement from a performance rights organization (PRO).

Sharp: a note that is higher in pitch by a semitone (half step), for example, a G sharp (noted G#) is a half step above G.

Smith-Waterman: short for Smith-Waterman algorithm, the last names of two individuals who wrote the paper defining the algorithm.

Tempo: the overall speed by which notes are being played across an entire piece of music.

The Music Information Retrieval Evaluation eXchange (MIREX): an event where MIR researchers conduct scientific evaluations of MIR-related tasks and algorithms.

The Society of European Stage Authors and Composers (SESAC): one of the top three performance rights organizations in the United States along with BMI and ASCAP.

Theme: a subset of a melody that may be repeated and identified by individuals as an easily recognized part of a song.

Time Signature: the base rhythm for a song, the symbol used in this paper consists of two numbers, one above the other (for example 3/4): the top number refers to the

number of beats per measure and the bottom number refers to which note value gets the beat.

Transpose: the changing of a pitch or set of pitches (as in a song melody) by raising or lowering it/them by a consistent interval.

APPENDIX A

Base Code

Taken directly from http://sedefcho.icnhost.net/web/algorithms/smith_waterman.html:

```
package main;

import java.util.StringTokenizer;

/**
 * SmithWaterman.java
 *
 * Created on 19 February 2007
 *
 * @author Peter Petrov
 *
 * http://sedefcho.icnhost.net
 */
public class SmithWaterman {

	private static int[][] costMatrix = new int[4][4];
	private static int[][] weightMatrix = null;

	private static final int MAX_LINE_LENGTH = 1024;

	private static int gapPenalty = -1;

	public static void main(String[] args){

	   parseInput();

	   while (true){

		String s1 = getNextToken();
		if (s1==null) break;

		String s2 = getNextToken();
		if (s2==null) break;

		int[] score = new int[]{0};
```

```
        String alignment[] = fillMatrix(s1, s2, score);

        System.out.println(alignment[0]);
        System.out.println(alignment[1]);
        System.out.println("Score: " + score[0]);
        System.out.println();
    }

}

private static String[] fillMatrix(String s1, String s2, int[] score){
    int lenS1 = s1.length();
    int lenS2 = s2.length();

    int scoreZero = 0;

    weightMatrix = new int[lenS1 + 1][lenS2 + 1];
    weightMatrix[0][0] = 0;
    for (int i=1; i<=lenS1; i++){
        weightMatrix[i][0] = scoreZero;
    }
    for (int i=1; i<=lenS2; i++){
        weightMatrix[0][i] = scoreZero;
    }

    int bestI = 0;
    int bestJ = 0;

    int bestScoreMatrix = scoreZero;

    int scoreIfGoingDown = 0;
    int scoreIfGoingRight = 0;
    int scoreIfGoingDownRight = 0;

    int bestScore = 0;
    int ind1 = 0;
    int ind2 = 0;
    for (int i=1; i<=lenS1; i++){
        for (int j=1; j<=lenS2; j++){
            // Next 5 lines can be separated into a helper method (see also below)!
            ind1 = getLetterIndex(s1.charAt(i-1));
            ind2 = getLetterIndex(s2.charAt(j-1));
            scoreIfGoingDown = weightMatrix[i-1][j] + gapPenalty;
```

```
            scoreIfGoingRight = weightMatrix[i][j-1] + gapPenalty;
            scoreIfGoingDownRight = weightMatrix[i-1][j-1] +
            costMatrix[ind1][ind2];

            bestScore = Math.max( Math.max(scoreIfGoingDown,
            scoreIfGoingRight), Math.max(scoreIfGoingDownRight, scoreZero) );
            weightMatrix[i][j] = bestScore;
            if (bestScore > bestScoreMatrix){
                  bestScoreMatrix = bestScore;
                  bestI = i;
                  bestJ = j;
            }
      }
}

StringBuffer res1 = new StringBuffer();
StringBuffer res2 = new StringBuffer();

int i = bestI;
int j = bestJ;

// This way we return the score to the calling method!
score[0] = bestScoreMatrix;

while (i>0 && j>0){

      if (weightMatrix[i][j]==scoreZero) {
         break;
      }

      // Next 5 lines can be separated into a helper method (same method as the
         one above)!
      ind1 = getLetterIndex(s1.charAt(i-1));
      ind2 = getLetterIndex(s2.charAt(j-1));
      scoreIfGoingDown = weightMatrix[i-1][j] + gapPenalty;
      scoreIfGoingRight = weightMatrix[i][j-1] + gapPenalty;
      scoreIfGoingDownRight = weightMatrix[i-1][j-1] + costMatrix[ind1][ind2];

      if (weightMatrix[i][j] == scoreIfGoingDown){
         // Go up !
         res1.insert(0, s1.charAt(i-1));
         res2.insert(0, '-');
         i--;
      }else if (weightMatrix[i][j] == scoreIfGoingRight){
```

```
            // Go left !
            res1.insert(0, '-');
            res2.insert(0, s2.charAt(j-1));
            j--;
        }else if (weightMatrix[i][j] == scoreIfGoingDownRight){
            // Go up and left !
            res1.insert(0, s1.charAt(i-1));
            res2.insert(0, s2.charAt(j-1));
            i--;
            j--;
        }
    }
    while (i>0){
        if (weightMatrix[i][j]==scoreZero) {
            break;
        }

        // Go up !
        res1.insert(0, s1.charAt(i-1));
        res2.insert(0, '-');
        i--;
    }
    while (j>0){
        if (weightMatrix[i][j]==scoreZero) {
            break;
        }

        // Go left !
        res1.insert(0, '-');
        res2.insert(0, s2.charAt(j-1));
        j--;
    }

    weightMatrix = null;

    return new String[] { res1.toString(), res2.toString() };
}

private static int getLetterIndex(char letter){
    int ind = -1;
    switch (letter){
        case 'A': ind = 0; break;
        case 'G': ind = 1; break;
        case 'C': ind = 2; break;
```

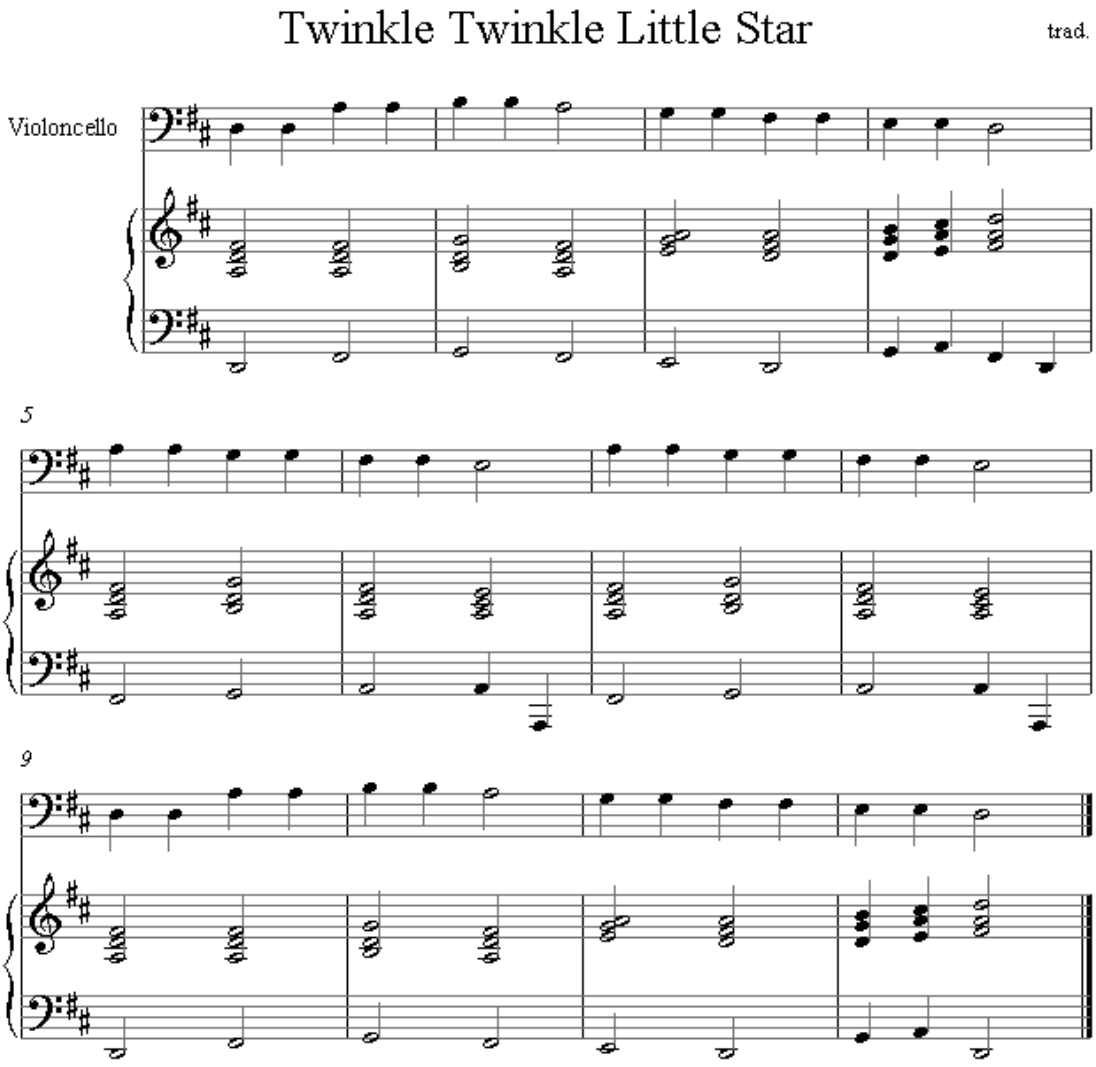
```
            case 'T': ind = 3; break;
        }
        return ind ;
    }

    /* IO Related Methods */

private static String line = "";

private static StringTokenizer st =
new StringTokenizer("", "\t :\r\n");

    private static void parseInput(){
        String[][] matrix = new String[5][5];
        matrix[0][0] = "";
        for (int k=1; k<=24; k++){
            matrix[k/5][k%5] = getNextToken();
        }

        // Store the label of the i-th row in rowLabels[i-1]
        char[] rowLabels = new char[4];
        for (int i=1; i<=4; i++){
            rowLabels[i-1] = matrix[i][0].charAt(0);
        }

        // Store the label of the j-th column in colLabels[j-1]
        char[] colLabels = new char[4];
        for (int j=1; j<=4; j++){
           colLabels[j-1] = matrix[0][j].charAt(0);
        }

        for (int i=1; i<=4; i++){
            for (int j=1; j<=4; j++){
               int w = Integer.parseInt(matrix[i][j]);
               int indI = getLetterIndex(rowLabels[i-1]);
               int indJ = getLetterIndex(colLabels[j-1]);
               costMatrix[indI][indJ] = w;
            }
        }

        getNextToken(); // Skip word "GAP"
        getNextToken(); // Skip word "PENALTY:"

        // Read the gap penalty value!
```

```
            gapPenalty = Integer.parseInt(getNextToken());

        }

    private static String getNextToken(){
    while (!st.hasMoreTokens()){
        line = readLn(MAX_LINE_LENGTH + 10);
        if (line==null) return null;
        st = new StringTokenizer(line);
    }
    return st.nextToken().trim();
    }

    /**
     * Utility method to read lines
     * of input from System.in ( STDIN )
     * using only basic java IO classes.
     */
    private static String readLn (int maxLg)
    {
    byte lin[] = new byte [maxLg];
    int lg = 0, car = -1;
    try {
        while (lg < maxLg){
        car = System.in.read();
        if ((car < 0) || (car == '\n')) break;
        lin [lg++] += car;
        }
    }catch (java.io.IOException e){
        return (null);
    }

    if ((car < 0) && (lg == 0)) {
        // EOF
        return (null);
    }
    return (new String (lin, 0, lg));
    }

}
```

APPENDIX B

Sheet Music

Taken from http://www.8notes.com, September 12, 2007.



APPENDIX C

Modified Code

```
package main;

import java.util.StringTokenizer;
import java.io.*;

/*
Main.java
Peter Petrov
Omer Piperdi
Jeff Frey
11/15/2007
 */

public class Main {

	private static int[][] costMatrix = new int[15][15];
	private static int[][] weightMatrix = null;

	private static final int MAX_LINE_LENGTH = 1024;

	private static int gapPenalty = -1;

	public static void main(String[] args){
	   parseInput();
	   long t = System.currentTimeMillis();
	   InputStreamReader sr = new InputStreamReader(System.in);
	   BufferedReader br    = new BufferedReader(sr);
	   String line = "";
	   String inputSong = "";
	   String key = "";
	   while (line != null) {
	      try {
	         line = br.readLine();
	      } catch (IOException e) {
	         break;
	      }
	      if (line != null) {
```

```
            if (!line.equals("")) {
                String[] splitLine = line.split("-");
                if (splitLine.length == 2) {
                    if (splitLine[0].trim().equals("Input Song")) {
                        String[] splitInputSong = splitLine[1].split(":");
                        key = splitInputSong[0].trim();
                        inputSong = splitInputSong[1].trim();
                        inputSong = replaceDoubleLetterNote(inputSong);
                        System.out.println("------------------------------");
                        char charKey = key.charAt(0);
                        System.out.println("Key : " + charKey);
                        System.out.println(splitLine[0].trim() + " : " +
                            splitInputSong[1].trim());
                        System.out.println(splitLine[0].trim() + " after removing double char.
                            notes : " + inputSong );
                        inputSong = transposeNote(charKey,inputSong);
                        System.out.println(splitLine[0].trim() + " after fixed to C key : " +
                            inputSong );
                        System.out.println("------------------------------");
                        System.out.println();
                    }
                    else {
                        if (!inputSong.equals("")) {
                            int[] score = new int[]{0};
                            String alignment[] = fillMatrix(inputSong,
                                replaceDoubleLetterNote(splitLine[1].trim()), score);
                            System.out.print(splitLine[0].trim());
                            System.out.println("  - Score: " + score[0]);
                            System.out.println();
                        }
                    }
                }
                else
                    System.out.println("Wrong Input Format!!!");
            }
        }
    }
    System.out.println("This program took " + (System.currentTimeMillis() - t) + "
        millisecond.");
}

static String replaceDoubleLetterNote(String inStr) {
    inStr = stringReplace(inStr,"C#", "S");
    inStr = stringReplace(inStr,"Db", "S");
```

```
    inStr = stringReplace(inStr,"D#", "U");
    inStr = stringReplace(inStr,"Eb", "U");
    inStr = stringReplace(inStr,"F#", "W");
    inStr = stringReplace(inStr,"Gb", "W");
    inStr = stringReplace(inStr,"G#", "X");
    inStr = stringReplace(inStr,"Ab", "X");
    inStr = stringReplace(inStr,"A#", "Y");
    inStr = stringReplace(inStr,"Bb", "Y");
    return inStr;
}

static String transposeNote(char key, String inStr) {
    String returnString = "";
    char[][] transposeTable = { {'C','S','D','U','E','F','W','G','X','A','Y','B','C'},
                                {'S','D','U','E','F','W','G','X','A','Y','B','C','S'},
                                {'D','U','E','F','W','G','X','A','Y','B','C','S','D'},
                                {'U','E','F','W','G','X','A','Y','B','C','S','D','U'},
                                {'E','F','W','G','X','A','Y','B','C','S','D','U','E'},
                                {'F','W','G','X','A','Y','B','C','S','D','U','E','F'},
                                {'W','G','X','A','Y','B','C','S','D','U','E','F','W'},
                                {'G','X','A','Y','B','C','S','D','U','E','F','W','G'},
                                {'X','A','Y','B','C','S','D','U','E','F','W','G','X'},
                                {'A','Y','B','C','S','D','U','E','F','W','G','X','A'},
                                {'Y','B','C','S','D','U','E','F','W','G','X','A','Y'},
                                {'B','C','S','D','U','E','F','W','G','X','A','Y','B'},
                                {'C','S','D','U','E','F','W','G','X','A','Y','B','C'},
                              };
    int keyIndex = -1;
    for (int i=0; i<13; i++) {
      if (key == transposeTable[0][i])
         keyIndex = i;
    }
    returnString = inStr;
    if (keyIndex > 0) {
       int position = -1;
       for (int i=0; i<returnString.length(); i++) {
          for (int j=0; j<13; j++) {
                if (returnString.charAt(i) == transposeTable[keyIndex][j])
                   position = j;
          }
          if (returnString.charAt(i) != 'R')
             returnString = replaceCharAt(returnString,i,transposeTable[0][position]);
       }
    }
```

```
    return returnString;
}

public static String replaceCharAt(String s, int pos, char c) {
    return s.substring(0,pos) + c + s.substring(pos+1);
}

static String stringReplace(String str, String pattern, String replace) {
    int s = 0;
    int e = 0;
    StringBuffer result = new StringBuffer();
    while ((e = str.indexOf(pattern, s)) >= 0) {
        result.append(str.substring(s, e));
        result.append(replace);
        s = e+pattern.length();
    }
    result.append(str.substring(s));
    return result.toString();
}

private static String[] fillMatrix(String s1, String s2, int[] score){
    int lenS1 = s1.length();
    int lenS2 = s2.length();

    int scoreZero = 0;

    weightMatrix = new int[lenS1 + 1][lenS2 + 1];
    weightMatrix[0][0] = 0;
    for (int i=1; i<=lenS1; i++){
            weightMatrix[i][0] = scoreZero;
    }
    for (int i=1; i<=lenS2; i++){
            weightMatrix[0][i] = scoreZero;
    }

    int bestI = 0;
    int bestJ = 0;

    int bestScoreMatrix = scoreZero;

    int scoreIfGoingDown = 0;
    int scoreIfGoingRight = 0;
    int scoreIfGoingDownRight = 0;
```

```
int bestScore = 0;
int ind1 = 0;
int ind2 = 0;
for (int i=1; i<=lenS1; i++){
      for (int j=1; j<=lenS2; j++){
         ind1 = getLetterIndex(s1.charAt(i-1));
         ind2 = getLetterIndex(s2.charAt(j-1));
         scoreIfGoingDown = weightMatrix[i-1][j] + gapPenalty;
         scoreIfGoingRight = weightMatrix[i][j-1] + gapPenalty;
         scoreIfGoingDownRight = weightMatrix[i-1][j-1] +
             costMatrix[ind1][ind2];

         bestScore = Math.max( Math.max(scoreIfGoingDown,
             scoreIfGoingRight), Math.max(scoreIfGoingDownRight, scoreZero) );
         weightMatrix[i][j] = bestScore;
         if (bestScore > bestScoreMatrix){
               bestScoreMatrix = bestScore;
               bestI = i;
               bestJ = j;
         }
      }
}

StringBuffer res1 = new StringBuffer();
StringBuffer res2 = new StringBuffer();

int i = bestI;
int j = bestJ;

score[0] = bestScoreMatrix;

while (i>0 && j>0){

      if (weightMatrix[i][j]==scoreZero) {
         break;
      }

      ind1 = getLetterIndex(s1.charAt(i-1));
      ind2 = getLetterIndex(s2.charAt(j-1));
      scoreIfGoingDown = weightMatrix[i-1][j] + gapPenalty;
      scoreIfGoingRight = weightMatrix[i][j-1] + gapPenalty;
      scoreIfGoingDownRight = weightMatrix[i-1][j-1] + costMatrix[ind1][ind2];

      if (weightMatrix[i][j] == scoreIfGoingDown){
```

```
            res1.insert(0, s1.charAt(i-1));
            res2.insert(0, '-');
            i--;
        }else if (weightMatrix[i][j] == scoreIfGoingRight){
            res1.insert(0, '-');
            res2.insert(0, s2.charAt(j-1));
            j--;
        }else if (weightMatrix[i][j] == scoreIfGoingDownRight){
            res1.insert(0, s1.charAt(i-1));
            res2.insert(0, s2.charAt(j-1));
            i--;
            j--;
        }
    }
    while (i>0){
        if (weightMatrix[i][j]==scoreZero) {
            break;
        }

        res1.insert(0, s1.charAt(i-1));
        res2.insert(0, '-');
        i--;
    }
    while (j>0){
        if (weightMatrix[i][j]==scoreZero) {
            break;
        }

        res1.insert(0, '-');
        res2.insert(0, s2.charAt(j-1));
        j--;
    }

    weightMatrix = null;

    return new String[] { res1.toString(), res2.toString() };
}

private static int getLetterIndex(char letter){
    int ind = -1;
    switch (letter){
        case 'C': ind = 0; break;
        case 'S': ind = 1; break;
        case 'T': ind = 2; break;
```

```
            case 'D': ind = 3; break;
            case 'U': ind = 4; break;
            case 'V': ind = 5; break;
            case 'E': ind = 6; break;
            case 'F': ind = 7; break;
            case 'W': ind = 8; break;
            case 'X': ind = 9; break;
            case 'A': ind = 10; break;
            case 'Y': ind = 11; break;
            case 'Z': ind = 12; break;
            case 'B': ind = 13; break;
            case 'R': ind = 14; break;
        }
        return ind ;
    }

private static String line = "";

private static StringTokenizer st =
new StringTokenizer("", "\t :\r\n");

    private static void parseInput(){
        String[][] matrix = new String[16][16];
        matrix[0][0] = "";
        for (int k=1; k<=255; k++){
            matrix[k/16][k%16] = getNextToken();
        }

        char[] rowLabels = new char[15];
        for (int i=1; i<=15; i++){
            rowLabels[i-1] = matrix[i][0].charAt(0);
        }

        char[] colLabels = new char[15];
        for (int j=1; j<=15; j++){
           colLabels[j-1] = matrix[0][j].charAt(0);
        }

        for (int i=1; i<=15; i++){
            for (int j=1; j<=15; j++){
               int w = Integer.parseInt(matrix[i][j]);
               int indI = getLetterIndex(rowLabels[i-1]);
               int indJ = getLetterIndex(colLabels[j-1]);
               costMatrix[indI][indJ] = w;
```

```
            }
        }

        getNextToken();
        getNextToken();

        gapPenalty = Integer.parseInt(getNextToken());

    }

    private static String getNextToken(){
    while (!st.hasMoreTokens()){
        line = readLn(MAX_LINE_LENGTH + 10);
        if (line==null) return null;
        st = new StringTokenizer(line);
    }
    return st.nextToken().trim();
    }

    private static String readLn (int maxLg)
    {
    byte lin[] = new byte [maxLg];
    int lg = 0, car = -1;
    try {
        while (lg < maxLg){
        car = System.in.read();
        if ((car < 0) || (car == '\n')) break;
        lin [lg++] += car;
        }
    }catch (java.io.IOException e){
        return (null);
    }

    if ((car < 0) && (lg == 0)) {
        return (null);
    }
    return (new String (lin, 0, lg));
    }

}
```

APPENDIX D

Melodies Used for Matching

BINGO –
CFFFFCCCCDDDDCCCCCFFFFGGGGAAAAFFFFAAAAAAAABbBbBbBbBbBbBbB
bGGGGGGGGAAAAAAAAFFFFFFFFGGGGGGFFEECCDDEEFFFFFFFF

Frère Jacques –
CCCCDDDDEEEECCCCCCCCCDDDDEEEECCCCEEEEFFFFGGGGGGGGEEEEFFF
FGGGGGGGGGGAAGGFFEEEECCCCGGAAGGFFEEEECCCCCCCCCGGGGCCCC
CCCCCCCCGGGGCCCCCCCC

London Bridge is Falling Down –
GGGGAAAAGGGGFFFFEEEEFFFFGGGGGGGGDDDDEEEEFFFFFFFFEEEEFFFF
GGGGGGGGGGGGAAAAGGGGFFFFEEEEFFFFGGGGGGGGDDDDDDDDGGGG
GGGGEEEEGGGGRRRRRRRR

Mary Had a Little Lamb –
CCCCBbBbBbBbG#G#G#G#BbBbBbBbCCCCCCCCCCCCCCCCBbBbBbBbBbBbBb
BbBbBbBbBbCCCCD#D#D#D#D#D#D#D#CCCCBbBbBbBbG#G#G#G#BbBbBbBbC
CCCCCCCCCCCCCCCCBbBbBbBbBbBbBbBbCCCCBbBbBbBbG#G#G#G#G#G#G#G
#G#G#G#G#G#G#G#G#

Old MacDonald had a Farm–
FFFFFFFFFFFFCCCCDDDDDDDDCCCCCCCCCAAAAAAAAGGGGGGGGFFFFFFFF
FRRRRRRCCFFFFFFFFFFFFCCCCDDDDDDDDCCCCCCCCCAAAAAAAAGGGGG
GGGFFFFFFFF

Take Me Out to the Ballgame –
CCCCCCAAGGEEGGGGGGDDDDDDCCCCCCCAAGGEEGGGGGGGGGGRRAAG
GAAEEFFGGAAAAFFDDDDDDDDDDDDDDAAAAAAAAABBCCDDBBAAGGEEDD
CCCCCCAAGGEEGGGGGGDDDDDDCCCCDDEEFFGGAAAAAAAAAABBCCCC
CCCCCCCCCCBBAAGGF#F#GGAAAAAABBBBBBCCCCCCCCCCCC

Three Blind Mice –
EEEEEEDDDDDDCCCCCCCCCCCCCEEEEEEDDDDDDCCCCCCCCCCCCCGGGGG
GFFFFFFEEEEEEEEEEEEGGGGGGFFFFFFEEEEEEEEEEGGCCCCCCBBAABBC
CCCGGGGGGGGGCCCCCCBBAABBCCCCGGGGGGGGCCCCCCBBAABBCCGG
GGGGGGFFEEEEEEDDDDDDCCCCCCCCCCCC

Twinkle, Twinkle, Little Star –
CCCCCCCCGGGGGGGGAAAAAAAAGGGGGGGGFFFFFFFFEEEEEEEEDDDDD
DDDCCCCCCCCCGGGGGGGGFFFFFFFFEEEEEEEEDDDDDDDDGGGGGGGGFFF
FFFFFEEEEEEEEDDDDDDDDCCCCCCCCGGGGGGGGAAAAAAAAGGGGGGG
GFFFFFFFFEEEEEEEEDDDDDDDDCCCCCCCC

Yankee Doodle –
CCCCCCCCDDDDEEEECCCCEEEEDDDDGGGGCCCCCCCCDDDDEEEECCCCC
CCCBBBBBBBBCCCCCCCCDDDDEEEEFFFFEEEEDDDDCCCCBBBBGGGGAAA
ABBBBCCCCCCCCCCCCCCC

REFERENCES

[1] A. Csinger. The Psychology of Visualization. Technical Report TR-92-28, Dept. of Comp. Sci., U. of British Columbia, November 1992.

[2] A. Ghias, J. Logan, D. Chamberlin, B. C. Smith. Query By Humming --Musical Information Retrieval in an Audio Database. ACM Multimedia 95 -Electronic Proceedings. November 1995

[3] A. L. Uitdenbogerd, J. Zobel. Manipulation of Music for Melody Matching. Proc. ACM Multimedia '98. 1998, 235-240.

[4] A. L. Uitdenbogerd, J. Zobel. Music ranking techniques evaluated. International Symposium on Music Information Retrieval. Plymouth, Massachussetts, USA, October 2000.

[5] A. Pearson, H. Peri, O. Jabado, O. Wood. eBLAST: Building a Better BLAST. http://thraxil.org/w4761/proj2.html. Accessed September 1, 2006.

[6] A. Raju, P. Rao. Building a Melody Retrieval System. Presented at National Conference on Communications. NCC 2002, January 2002.

[7] A. S. Durey. Melody Spotting Using Hidden Markov Models. Int. Symposium on Music Information Retrieval (ISMIR), 2001.

[8] A. Wang. The Shazam music recognition service. Communications of the ACM. Vol. 49, Issue 8, August 2006, 44-48.

[9] Agent Arts. Welcome to MusicBrainz. http://musicbrainz.org. Accessed December 11, 2005.

[10] Amazon.com. Listmania Lists. Accessed May 14, 2006.

[11] Apple. In Tune With Your Taste. iTunes. http://www.apple.com/itunes/playlists. Accessed August 1, 2006.

[12] ASCAP. ACE Title Search. http://www.ascap.com/ace/search.cfm. Accessed March 15, 2006.

[13] ASCAP. ACSAP Adapts to Rapidly Changing Music Marketplace. ASCAP News. March 13, 2006. http://www.ascap.com/press/2006/031306_financial.html. Accessed March 15, 2006.

[14] ASCAP. Career Development. http://www.ascap.com/musicbiz/money-copyright.html. Accessed March 15, 2006.

[15] B. Pardo, S. Bryan, W. Birmingham. Name That Tune: A Pilot Study in Finding Melody From a Sung Query. Journal of the American Society for Information Science & Technology. February 2004, Vol. 55, Issue 4, 283-300.

[16] B. Pardo. Music Information Retrieval. Communications of the ACM. Vol. 49, Issue 8, August 2006, 29-31.

[17] B. Pearson. FASTA Sequence Comparison at the University of Virginia. FASTA Programs. http://fasta.bioch.virginia.edu. Accessed February 6, 2006.

[18] BMI. More on Doing Business with BMI Music. http://www.bmi.com/licensing/business/generalfaq.asp. Accessed March 16, 2006.

[19] BMI. Search BMI's Repertoire. http://www.bmi.com/search. Accessed March 16, 2006.

[20] BMI. Songwriters and Copyright. http://www.bmi.com/songwriter/resources/pubs/copyright.asp. Accessed March 16, 2006.

[21] C. Dwan. Speedup at What Cost? O'Reilly Media Newsletter. November 30, 2001.

[22] C. Florio. Digital Royalties. PerformerMag.com. http://www.performermag.com/digitalroyalties.php. Accessed August 30, 2006.

[23] C. Knab. Copyright and Songwriting Basics. Music Biz Acadamy.com. April 2002. http://www.musicbizacademy.com/knab/articles/copyright.htm.

[24] C. Lombardi. Google offers to track your music. CNET News.com August 17, 2006. http://news.com.com/2061-10812_3-6106697.html. Accessed August 20, 2006.

[25] C. Meek and W. Birhmingham. Johnny can't sing: A comprehensive error model for sung music queries. Proceedings of the Third International Conference in Music Informatics Retrieval, 2002.

[26] C. Parker. Applications of Binary Classification and Adaptive Boosting to the Query-by-Humming Problem. Proceedings of ISMIR 2005. http://ismir2005.ismir.net/proceedings/1104.pdf.

[27] C. Raphael. A Graphical Model for Recognizing Sung Melodies. Proceedings of ISMIR 2005. http://xavier.informatics.indiana.edu/~craphael/papers/ismir05.pdf

[28] Copugen. Bioccelerator FAQ. July 27, 1997. http://eta.embl-heidelberg.de:8000/Bic/Bicbioc-faq.html. Accessed January 30, 2006.

[29] D. Bainbridge, M. Dewsnip, I. H. Witten. Searching digital music libraries. Information Processing and Management: an International Journal archive. Vol. 41, Issue 1, January 2005, 41-56.

[30] D. Byrd, M. Fingerhut. The History of ISMIR – A Short Happy Tale. D-Lib Magazine, Vol. 8, Issue 11.

[31] D. Byrd, T. Crawford. Problems of music information retrieval in the real world. Information Processing and Management. Vol. 38, 2002, 249-272.

[32] D. Mazzoni, R. B. Dannenberg. Melody matching directly from audio. In 2nd Annual International Symposium on Music Information Retrieval. Bloomington, Indiana, USA, 2001.

[33] D. Ozick. Song-matching system and method. United States Patent 6,967,275, November 22, 2005.

[34] D. P. W. Ellis. Extracting Information from Music Audio. Communications of the ACM. Vol. 49, Issue 8, August 2006, 32-37.

[35] D. Tao, H. Liu, X. Tang. K-BOX: a query-by-singing based music retrieval system. Proceedings of the 12th annual ACM international conference on Multimedia. Technical poster session 3: multimedia tools, end-systems, and applications, 2004, 464-467.

[36] DigitalNirvana. Song Tracking. http://www.digital-nirvana.com/products/song-tracking. Accessed August 11, 2006.

[37] E. A. Martin. copyright. A Dictionary of Law. Oxford University Press, 2002.

[38] E. A. Sauleau, J. Paumier, A. Buemi. Medical record linkage in health information systems by approximate string matching and clustering. BMC Medical Informatics and Decision Making 2005. Vol. 5, Issue 32.

[39] E. Batlle, J. Masip, E. Guaus, Automatic song identification in noisy broadcast audio. Proc. of the SIP, Aug. 2002.

[40] E. Unal, S. S. Narayanan, E. Chew. A Statistical Approach to Retrieval under User-dependant Uncertainty in Query-by-Humming Systems. MIR '04, October, 2004, 113-118.

[41] F. Zhang, X. Qiao, Z. Liu. Parallel divide and conquer bio-sequence comparison based on Smith Waterman algorithm. Science in China. Series F, Information Sciences April 2004. Vol. 47, Issue 2, 221-231.

[42] Gracenote Company Profile. http://www.gracenote.com/music/corporate/index.html. Accessed August 30, 2006.

[43] H. Chen. Restoring pitch in cochlear implant users. Dissertation. University of California. Irvine, 2005.

[44] H. Neuschmied, H. Mayer, E. Battle. Identification of Audio Titles on the Internet. Proceedings of International Conference on Web Delivering of Music 2001.

[45] I.S.H. Suyoto, A.L. Uitdenbogerd. Effectiveness of note duration information for music retrieval. Proc. International Conf. on Database Systems for Advanced Applications 2005. Beijing, China, 265-275.

[46] J. Black. royalties. A Dictionary of Economics. Oxford University Press, 2002.

[47] J. Eng. Biological Scripting Language. http://www1.cs.columbia.edu/~sedwards/classes/2003/w4115/BSL.final.pdf. Accessed May 19, 2006.

[48] J. R. McPherson, D. Bainbridge. Usage of the MELDEX Digital Music Library. Proceedings of ISMIR 2001.

[49] J. S. Downie. M2K (Music-to-Knowledge): A tool set for MIR/MDL development and evaluation. http://music-ir.org/evaluation/m2k. Last modified October 27, 2005. Date accessed: February 16, 2006.

[50] K. Lemstrom, S. Perttu. SEMEX -an efficient music retrieval prototype. Proceedings of the International Symposium on Music Information Retrieval (ISMIR'2000), 2000.

[51] K. Lemstrom. String Matching Techniques for Music Retrieval. PhD Thesis, University of Helsinki, Department of Computer Science, April, 2004.

[52] L. A. Obringer. How Music Royalties Work. Stuffo. http://www.stuffo.come/music-royalties7.htm. Accessed March 13, 2006.

[53] L. A. Smith, R. J. McNab, I. H. Witten. Music Information Retrieval Using Audio Input. Proceedings of ISMIR 2003.

[54] L. D. Paulson. New Search Engines Keep Humming Along. Computer. April 2003. Vol. 36, Issue 4, 2-3.

[55] L. P. Clarisse, J. P. Martens, M. Lesare, B. DeBaets, H. DeMeyer, M. Leman. An Auditory Model Based Transcriber of Singing Sequences. Proc. of 3rd International Conference on Music Information Retrieval, ISMIR '02, 2002.

[56] L. Smith, R. Medina. Discovering Themes by Exact Pattern Matching. 2nd International Symposium on Music Information Retrieval, ISMIR2001.

[57] M. Cahill, D. Maidnin. Melodic Similarity Algorithms -Using Similarity Ratings for Development and Early Evaluation. Proceedings of ISMIR 2005. 450-453.

[58] M. Carre, P. Philippe, C. Apelian. New Query-by-Humming Music Retrieval System Conception and Evaluation Based on a Query Nature Study. Proceedings of the COST G-6 Conference on Digital Audio Effects. Limerick, Ireland, December 2001.

[59] M. Clausen, R. Engelbrecht, D. Meyer, J. Schmitz. Proms: A web-based tool for searching in polyphonic music. Proceedings of the 1st International Symposium on Music Information Retrieval. ISMIR 2000.

[60] M. D. Schulkind, R. J. Posner, D. C. Rubin. Musical features that facilitate melody identification. Music Perception. Winter 2003, Vol. 21, Issue 2, 217-249.

[61] M. K. Shan, F. Kuo. Music Style Mining and Classification by Melody. IEICE TRANSACTIONS on Information and Systems. Vol.E86-D, Issues 3, 655-659.

[62] M. Kennedy. melody. The Concise Oxford Dictionary of Music. Oxford University Press, 1996.

[63] M. Kennedy. motif. The Concise Oxford Dictionary of Music. Oxford University Press, 1996.

[64] M. Kschischo, M. Lassig, Y. Yu. Toward an accurate statistics of gapped alignments. Bulletin of Mathematical Biology. Vol. 67, Issue 1, January 2005, 169-191.

[65] M. Lesaffre. User-Dependant Taxonomy of Musical Features as a Conceptual Framework for Musical Audio-Mining Technology. Proceedings of the Stockholm Music Acoustics Conference, August 2003, Stockholm, Sweden.

[66] M. Li, B. Ma, D. Kisman, J. Tromp. Patternhunter II:: highly sensitive and fast homology search. Journal of Bioinformatics & Computational Biology. September 2004, Vol. 2 Issue 3, 417-439.

[67] M. M. Hamilton. Music annotation system for performance and composition of musical scores. Unites States Patent 6,483,019, November 19, 2002.

[68] M. Ryynanen. Automatic transcription of the singing melody in polyphonic music. Updated April 2006. http://www.cs.tut.fi/sgn/arg/matti/demos/melofrompoly. Accessed 8/30/2006.

[69] M. Woo. Music search engine. United States Patent 6,678,680, January 13, 2004.

[70] Music Publishers' Association of the United States. Copyright Resource Center. http://www.mpa.org/copyright_resource_center/copyright_search. Accessed May 18, 2006.

[71] MusicMatch. The MusicMatch Discovery Engine. http://www.musicmatch.com/download/ music_discovery_intro.htm. Accessed March 20, 2006.

[72] N. Hu, R. B. Dannenberg, A. L. Lewis. A Probabilistic Model of Melodic Similarity. Proceedings of the International Computer Music Conference. Gotheborg, Sweden, September 2002.

[73] N. Hu, R. B. Dannenberg. A Comparison of Melodic Database Retrieval Techniques Using Sung Queries. Joint Conference on Digital Libraries. Association for Computing Machinery. 2002, 301-307.

[74] N. Maddage, H. Li, M. Kankanhalli. Music structure based vector space retrieval. Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval. Session: Speech and Music, 2006, 67-74.

[75] P. A. Petrov. Smith-Waterman Algorithm Implemented in Java (JDK 1.4). http://sedefcho.icnhost.net/web/algorithms/smith_waterman.html. Date accessed: September 26, 2007.

[76] P. Cano, M. Kaltenbrunner, O. Mayor, and E. Batlle. Statistical Significance in Song-Spotting in Audio. Proceedings of the International Symposium on Music Information Retrieval. Bloommington, IN, Oct. 2001.

[77] P. Gardner-Stephen, G. Knowles. DASH: A New High Speed Genomic Sequence Search and Alignment System. WSEAS Transactions on Biology and Biomedicine. January, 2004. Vol. 1, Issue 1, 59-64.

[78] P. Lamere. The Tools We Use. http://www.music-ir.org/evaluation/tools.html. Version 1.5, May 2005. Date accessed: February 13, 2006.

[79] PARACEL. Algorithm Primer. http://www.paracel.com/faq/faq_algorithm_primer.html. Accessed September 3, 2002.

[80] R. Iwamura. Classical Music Search. http://iwamura.home.zdnet.com/page2.html. Accessed April 16, 2006.

[81] R. Iwamura. Music search by melody input. United States Patent 6,188,010, February 13, 2001.

[82] R. J. McNab, L. A. Smith, D. Bainbridge, I. H. Witten. The New Zealand Digital Library MELody index. DLib Magazine, 1997, 11-18.

[83] R. J. McNab, L. A. Smith, I. H. Witten, C. L. Henderson, S. J. Cunningham. Toward the digital music library: tune retrieval from acoustic input. Proc. ACM Digital Libraries. 1996, 11-18.

[84] R. J. McNab, L. A. Smith, I. H. Witten, C. L. Henderson. Tune retrieval in the multimedia library. Multimedia Tools and Applications 10, 2000, 113-132.

[85] R. J. McNab, L. A. Smith, I. H. Witten. Signal processing for melody transcription. Proc. 19th Australasian Computer Science Conf. 1996, 301-307.

[86] R. Samadani, A. Said. System and method for music identification. United States Patent 6,995,309, February 7, 2006.

[87] S. Bucheli. Tyberis Music Database. http://music.tyberis.com. Accessed June 2, 2006.

[88] S. F. Altschul. BLAST. Wikipedia. http://en.wikipedia.org/wiki/BLAST. Accessed September 1, 2006.

[89] S. Rupley. I Know that Song! PC Magazine Online Edition. February 12, 2003.

[90] SESAC. Repertory Search. http://www.sesac.com/repertory/sRepertorySQL.asp. Accessed August 17, 2006.

[91] Shazam Company Profile. http://shazam.com. Accessed August 30, 2006.

[92] "SR." I Know That Song! PC Magazine. Vol. 22, Issue 6, 4/8/2003, 26.

[93] T. Chrisfield, R. Cosgrove, J. Stinson. Building scholarly online multimedia collections and services. The Electronic Library. Vol. 18, Issue 5, 2000, 328-335.

[94] T. F. Smith, M. S. Waterman. Identification of common molecular subsequences. Journal of Molecular Biology. Vol. 147, 1981, 195-197.

[95] T. Leung, C. Ngo. Index and Matching of Polyphonic Songs for Query-by-Singing System. MIR '04, October, 2004, 308-311.

[96] T. Sorsa, J. Huopaniemi. Melodic Resolution in Music Retrieval. Proceedings of International Symposium on Music Information Retrieval. Bloomington, IN, USA, 2001.

[97] T. Weyde, C. Datzko. Efficient Melody Retrieval With Motif Contour Classes. Proceedings of ISMIR 2005. http://ismir2005.ismir.net/proceedings/2118.pdf.

[98] U.S. Copyright Office. Copyright Catalog. http://www.copyright.gov/records/cohm.html. Accessed June 27, 2006.

[99] U.S. Copyright Office. Copyright Registration for Musical Compositions. Circular 50.

[100] U.S. Copyright Office. Royalty Payments. Digital Audio Recording Devices and Media. Supchapter C. 2006.

[101] U.S. Copyright Office. The Copyright Card Catalog and the Online Files of the Copyright Office. Circular 23.

[102] W. Birmingham, B. Pardo, C. Meek, J. Shifrin. The MusArt Music-Retrieval System. D-Lib Magazine. February 2002. Vol. 8, Issue 2.

[103] W. Birmingham, R. Dannenberg, B. Pardo. Query by humming with the VocalSearch system. Communications of the ACM. Vol. 49, Issue 8, August 2006, 49-52.

[104] W. Chai. Melody Retrieval On The Web. Master's Thesis. Massachusetts Institute of Technology. Fall 2000.

[105] W. P. Birmingham, R. B. Dannenberg, G. H. Wakefield, M. Bartsch, D. Bykowski, D. Mazzoni, C. Meek, M. Mellody, W. Rand. MUSART: Music retrieval via aural queries. Int. Symposium on Music Information Retrieval (ISMIR), 2001.

[106] Wikipedia. Copyright infringement. http://en.wikipedia.org/wiki/Copyright_violation. Accessed May 17, 2006.

[107] X. Yang. Generic C++ implementations of pairwise sequence alignment: Instantiation for local alignment. Dissertation. Concordia University, Canada. December 2004.

[108] Y. Nakagawa, Y. Nakamura. Tune Retrieval Based on The Similarity of Melody. Proceedings of the Second International Conference on Web Delivering Music. 2002.

[109] Y. Zhu, M. Kankanhalli. Similarity Matching of Continuous Melody Contours for Humming Querying of Melody Databases. LIT Technical Report, February 2002.