Justin J. Salamon

MuSearch Query by Example Search Engine

Computer Science Tripos, Part II

Clare College

14th May 2007

Proforma

Name:	**Justin J. Salamon**
College:	**Clare College**
Project Title:	**MuSearch Query by Example Search Engine**
Examination:	**Computer Science Tripos, 2007**
Word Count:	**11,850**
Project Originator:	Justin J. Salamon
Supervisor:	Dr D. J. Greaves
Supervisor:	Martin Rohrmeier

Original Aims of the Project

To implement a music search engine which allows users to search for a song by entering part of the song's musical content as a query. Suggested extensions include implementing a client-server architecture to make the system accessible from a web browser, supporting a MIDI controller for playing queries and supporting audio input for singing queries.

Work Completed

The project specifications were implemented successfully and all the success criteria were met. The implementation includes an original solution to data indexing, and in addition to the basic specifications I have implemented the following extensions: client-server architecture, support for MIDI controller, the ability to match a full MIDI file against the database and a set of additional GUI features.

Special Difficulties

None.

Declaration

I, Justin J. Salamon of Clare College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed

Date 14th May 2007

Contents

List of Figures

Acknowledgements

I would like to thank Martin Rohrmeier and David Greaves for their useful comments and suggestions, Alan Blackwell for some helpful references, the PhD group at the Centre for Music and Science for their interest and support, and everyone who devoted time to participate in the user tests.

Chapter 1

Introduction

1.1 Overview

My project concerns the creation of a content based music search engine. It is a Query by Example system, allowing users to search for a song by playing part of it without the need to know information such as the song title or composer. I have successfully implemented the system according to the specifications in my proposal and have met all the success criteria. My implementation includes an original solution to data indexing based on an approach used in Bioinformatics, and uses the concepts of pitch and rhythm contour to model melodic similarity and provide flexible query input methods. In addition I have implemented the suggested extensions of a client-server architecture to make the system accessible from a web browser, and MIDI controller support for making queries using MIDI instruments. I have also implemented two further extensions: support for full MIDI file comparison (using a complete MIDI file as a query), and a set of additional GUI features.

1.2 Background

1.2.1 Motivation

Music collections have traditionally been indexed according to textual information describing the pieces (*metadata)* such as title, composer or performer. If once the need to search for music was confined to people in music libraries, the transition to digital media and the prevalence of portable media devices now means searching and organising music is a task performed by millions of people every day. Programs such as Apple's iTunes and Microsoft's Media Player are in

common use, and systems providing more elaborate ways of indexing music have been developed[1].

The problem arises when one wishes to find a piece of music without knowing any of its metadata. This could be a tune stuck in their head or a song they have just heard on the radio. In this situation all of the aforementioned systems are of no use, and a different *content based* solution is required. This led me to the idea of developing a system which allows the user to search for a piece of music by entering part of the piece itself as a query. Systems in which queries are made by providing part of the actual content are known as *Query by Example* (QBE) systems.

1.2.2 Audio Fingerprinting, Query by Example and Query by Humming

The service offered by Shazam[2] is a QBE system which allows you to search for a song by putting your mobile phone up next to a speaker sounding the song. Shazam and similar systems use a technique called *audio fingerprinting* which relies on extracting specific acoustic information from the original audio recording to create a "fingerprint" which is used to query the database. The disadvantage of such systems is that you must have the original audio available to record a query, since any different version will produce a different fingerprint and will not match the one in the database. For this reason I decided to implement a QBE system that uses a symbolic representation of the music as a query (for example playing part of the melody on a graphically visualised keyboard). This way users can search for any song from memory, and queries containing mistakes can still retrieve the correct song. In *Query by Humming* (QBH) systems the query is sung or hummed into a microphone[3]. Such systems have the advantage of requiring less musical experience, but are also less flexible (only one way of inputting a query) and could be problematic for people with poor singing abilities. QBH systems use pitch detection to convert the singing into a symbolic representation, after which the search process is similar to that of QBE systems.

[1] The Music Genome Project uses over 400 attributes to classify songs.

[2] http://www.shazam.com/

[3] In my proposal I use the term QBH for both QBH and QBE as this was done in papers I read during the initial research for the project. Further reading identified the term QBE to accurately describe my system.

1.3 Previous Work

QBE and QBH systems are very much a topic of research. Existing systems include David Huron's Themefinder[4] and the Vocal Search[5] system developed by Pardo et. al. which is still a research system. Papers on QBH by music information retrieval (MIR) researchers such as Bryan Pardo and Roger Dannenberg provide a good insight into the current state of the field, and I build on some of the approaches suggested in papers such as [3] and [14] and combine them with my own ideas to provide a complete searching solution. During the implementation phase of the project a beta version of a commercial QBH system called Midomi[6] was launched. Though the details of the system are not published, its launch indicates the growing popularity and demand for content based solutions for music searching.

[4]http://themefinder.org/

[5]http://music.cs.northwestern.edu/research/vocal_search/

[6]http://www.midomi.com/

Chapter 2

Preparation

In the following sections I will describe the preparatory work I undertook in accordance with software engineering principles. This includes a requirements analysis, research into the current state of music information retrieval and existing QBH and QBE systems, and an overall project design plan. I will also describe the main algorithms and concepts which form the core of the system.

2.1 Requirements Analysis

2.1.1 User requirements

The completed system should:

- Run properly on the three major operating systems – Windows, Linux and MacOS.
- Provide a clear and simple graphical user interface.
- Provide an input method which is intuitive for people with basic musical background. This is defined as the ability to play simple melodies on a piano keyboard.
- Search a corpus of songs and retrieve songs which match the query in a way which is musically meaningful[1].
- Be able to match a query even if it is played in a different key or tempo to that of the song in the database.

[1]The concept of melodic similarity is discussed further in section 2.3.1.

- Be able to match a query even if it contains a small number of inaccuracies such as missing notes, added notes or notes with incorrect pitch or duration.
- Display the the top 10 results in order of relevance.
- Allow the user to listen to the results.

2.1.2 Functional Specification

Following the user requirements, the functional specification is as follows:

- The system will be written in Java (explained in section 2.4.1).
- A graphical user interface will provide two methods of inputting queries, provided in two separate panels:
  - A graphically visualised keyboard on which queries can be played using either the mouse or computer keyboard.
  - A text based method which represents a melody as a sequence of characters, intended for more musically experienced users.
- The database will be built using a corpus of MIDI files.
- The results will be displayed as a list containing the songs' titles and an indication of how relevant each song is to the query. The user can select a result and listen to the corresponding MIDI file.

2.1.3 Success Criteria

A successful implementation of the system will meet the following criteria:

- A *Mean Reciprocal Rank* not lower than 0.5 is measured for the test corpus.
- The system can search a corpus of 10,000 songs in under 5 seconds.
- Users should find the system intuitive to use, determined by usability tests.

A detailed explanation of the *Mean Reciprocal Rank* (MRR) metric and a discussion of other common information retrieval (IR) metrics such as *Precision* and *Recall* and their applicability is presented in the Evaluation chapter. It also contains a detailed account of the experimental setup and the data used for the evaluation.

2.2 Data Representation

2.2.1 Overview

Following research into existing systems such as the ones described in [14] and [12] I decided to use the Standard MIDI File Format to represent songs. The main reasons for this are:

- This format offers a digital representation of music from which the information required for melodic similarity computation can be easily extracted.
- Thousands of songs have already been transcribed into the MIDI file format and are freely available on the internet.

In the following sections I will describe the parts of the MIDI specification relevant to the project, and how further refinement of the data contained in a MIDI file is beneficial for system design and efficiency.

2.2.2 The Standard MIDI File Format

The *Musical Instrument Digital Interface*[2] (MIDI) specification defines a *message format* (the "MIDI Protocol") for transferring musical data between electronic devices. For example, to sound a note on a MIDI device you send a "Note On" message, which specifies a key (pitch) value and a velocity (intensity) value. The protocol includes messages for turning a note on or off, changing instrument etc. The MIDI specification also defines a set of messages not directly related to the production of sound, such as *Meta Messages* for transmitting text strings (e.g. lyrics, copyright notice) and *System Exclusive* messages for transmitting manufacturer specific instructions.

The Standard MIDI File Format is a *storage format* in which every message is combined with a timestamp to form a "'MIDI event", so that messages can be recalled and replayed in the correct order at a later date. A MIDI timestamp is specified in "ticks" which can be converted into an actual time value. The events in a MIDI file are divided into one or more *tracks*. Most polyphonic MIDI files use different tracks to store the parts of different instruments within a song.

The `javax.sound.midi` library provides tools for parsing MIDI files, separating them into tracks and extracting the individual MIDI events containing the MIDI messages and timestamps. The only events I will be interested in are the "Note On" and "Note Off" events, summarised in table 2.1.

[2] http://www.midi.org/

MIDI Event	Code	Data Byte 1	Data Byte 2
Note On	0x9	Pitch value (0-127)	Velocity (0-127)
Note Off	0x8	Pitch value (0-127)	Velocity (0-127)

Table 2.1: Useful MIDI events

2.2.3 The Data Refinement Model

A MIDI file will contain a large amount of data which is not needed, and only the relevant musical information should be extracted and added to the database. This will reduce the amount of memory and storage space required by every song, and save having to extract the musical information every time a song is compared to a query, making the system faster. The complete data refinement model is described below (implementation details in section 3.1):

- **Extraction** – the first step is to extract melodies a user can search for from the MIDI file. To allow the user to search for any part of a song, I extract a separate melody from every track in the MIDI file. I do this by considering only the "Note On" and "Note Off" events[3] in a track and converting them into individual note elements with a pitch value and onset/offset times. A sequence of note elements will be referred to as a **Melody**.

- **Standardisation** – next, I convert every Melody into a standardised format called a **Target**. This format preserves the musical essence of the Melody, but gets rid of performance specific information such as the specific key and tempo. Queries will also get converted into this format so they can be matched against the targets. This means a query can be represented in any way that can be converted into this format (e.g. text string, on-screen keyboard, MIDI file), decoupling the user interface (responsible for query production) from the rest of the system and facilitating a modular system design.

- **Minimisation** – the final step is to convert the Target into a compact format, so that it takes up a minimal amount of memory or disk space when not in use. A minimised Target is called a **MinTarget**, and will be the format actually stored in the database. It can be expanded back into a Target to be compared to a query.

The process is summarised in figure 2.1.

[3] A "Note On" with 0 velocity is treated the same as a "Note Off".

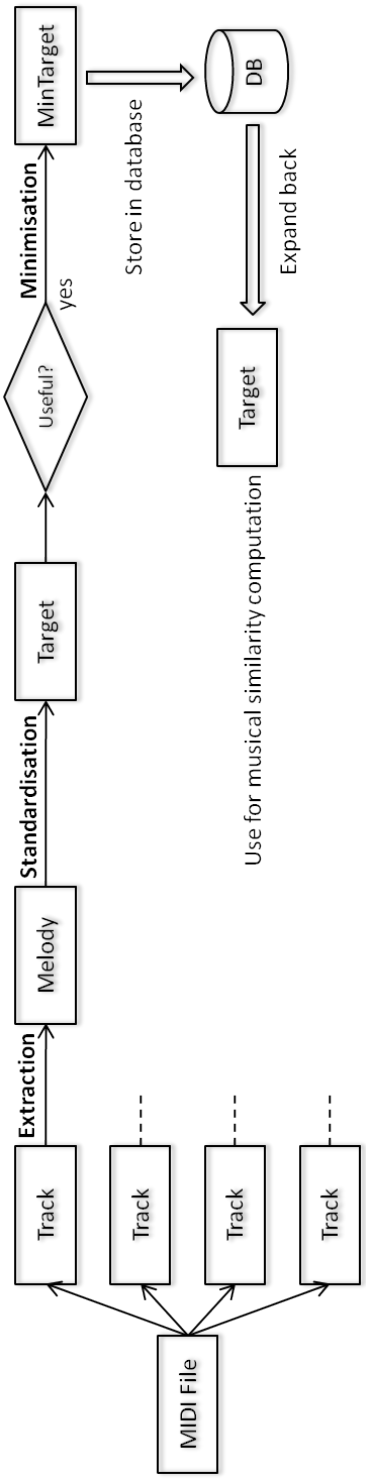


Figure 2.1: The data refinement model

2.3 Key Algorithms

In this section I will present the two main algorithms used in the system.

2.3.1 Computing Melodic Similarity

Modelling an adequate representation of perceptual melodic similarity forms the very core of this project. Given a symbolic representation of two melodies, the aim is to produce a quantitative value indicating the degree of similarity (or dissimilarity) between the two. A good solution will combine Computer Science principles with concepts of music cognition. There are various approaches to this task, and a comparative evaluation of some is given in [3]. It suggests that an approach based on string matching is most promising for practical applications, and in the following section I will described how string matching can be used to compute melodic similarity.

2.3.1.1 Overview

The idea is to represent melodies as strings, and use a string matching algorithm to compare them. In order for the comparison to give a result which is musically meaningful (i.e. melodies should give a good match if they *sound* similar to a human listener), careful thought is given to the format of the strings and the matching algorithm is extended to take into account musical considerations.

2.3.1.2 String Representation

As explained in section 2.2.3, both song and query are converted into in a standardised format for comparison. The format includes a representation of the melody's pitch and rhythm.

Representing Pitch – Relative pitch is the *interval* between two adjacent notes expressed in absolute values. A melody is represented as a sequence of pitch intervals. The advantage of using relative pitch is that it is *transposition invariant*, i.e. the specific key in which a query is represented is abstracted away, meaning it will match a similar melody even if it is played in a different key.

Representing Rhythm – I start by taking every note's *Inter Onset Interval* (IOI), which is the time difference between the onset of the note and the onset of the next note. Note onsets are more perceptually salient than offsets [3], so IOIs represent the user's perception of rhythm more accurately than note durations (the time difference between the note onset and offset). Next I take the ratio between the IOIs of every pair of consecutive notes (the IOIR). IOIRs have the

advantage of being *tempo invariant*, i.e. only the relative duration of notes is important, so the user can play the query slower or faster than the version in the MIDI file without affecting the search. The final step is taking the logarithm of the IOIR and rounding to the nearest integer (LogIOIR [11]), grouping together similar ratio values.

Using this representation a sequence of notes is converted into pairs of $\langle pitch\ interval, LogIOIR \rangle$ values which will be referred to as *note intervals*.

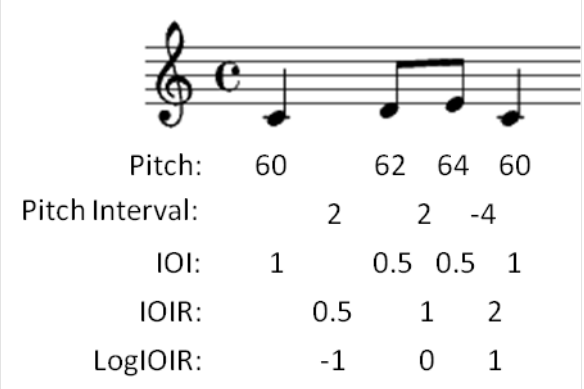


Figure 2.2: From pitch and IOI to pitch intervals and LogIOIRs

2.3.1.3 Local Alignment

Note intervals can be separated into two strings, one containing all the pitch intervals and the other all the LogIOIRs. The similarity between two melodies can be calculated using string matching techniques to compare the pitch strings and the rhythm strings separately. The similarity of two strings is measured by their *edit distance* [10]. Given two strings A and B, the edit distance $d(A, B)$ is the number of edit operations needed to transform A into B, using the following operations:

- *Skipping* a character from A
- *Skipping* a character from B
- *Replacing* a character from A with a character from B

Local Alignment calculates the optimal alignment (and hence the lowest edit distance) of one string with a subsection of another. This successfully models searching for a query (which can be any part of a melody) within a longer target.

Using a dynamic-programming approach, the algorithm calculates the optimal alignment by filling in a *distance matrix*. Given a query string $Q = q_1 q_2 \dots q_m$ and a target string $T = t_1 t_2 \dots t_n$, we fill a matrix D of size $m+1, n+1$ such that every entry $d_{i,j}$ denotes the cost of the optimal alignment of the prefixes $q_1 \dots q_i$ and $t_k \dots t_j$ $(1 \leq k \leq j)$. Starting with $d_{0,0} = 0$, the matrix is filled according to equation 2.1:

$$d_{i,j} = min \begin{cases} d_{i-1,j} + skipCost_q & \text{skip query character } q_i \\ d_{i,j-1} + skipCost_t & \text{skip target character } t_j \\ d_{i-1,j-1} + replaceCost(q_i, t_j) & \text{replace } q_i \text{ with } t_j \end{cases} \tag{2.1}$$

Equation 2.1 is visualised in figure 2.3.

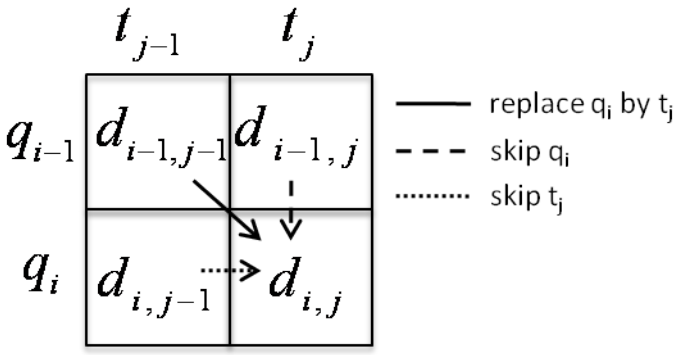


Figure 2.3: Calculation of single matrix cell

When filling in $d_{i,0}$, undefined matrix elements are considered as ∞. Since the query can appear anywhere in the target, every cell in the first row $d_{0,j}$ is set to zero, so that skipping target characters before the alignment section is not penalised. The final edit distance is given by $min_{0 \leq j \leq n}(d_{m,j})$, which is the cost of the best local alignment, without penalising for skipped target characters after the aligned section. As an example consider using the beginning of the theme from the last movement of Beethoven's 9th symphony as a query and matching it against the beginning of "Twinkle Twinkle", assuming $skipCost_q = 1$, $skipCost_t = 1$ and that $replaceCost(q_i, t_j)$ is given by equation 2.2. The calculation is shown in figure 2.4, with bold cells indicating the optimal alignment.

$$replaceCost(q_i, t_j) = \begin{cases} 0 & \text{if } q_i = t_j \\ 1 & \text{otherwise} \end{cases} \tag{2.2}$$

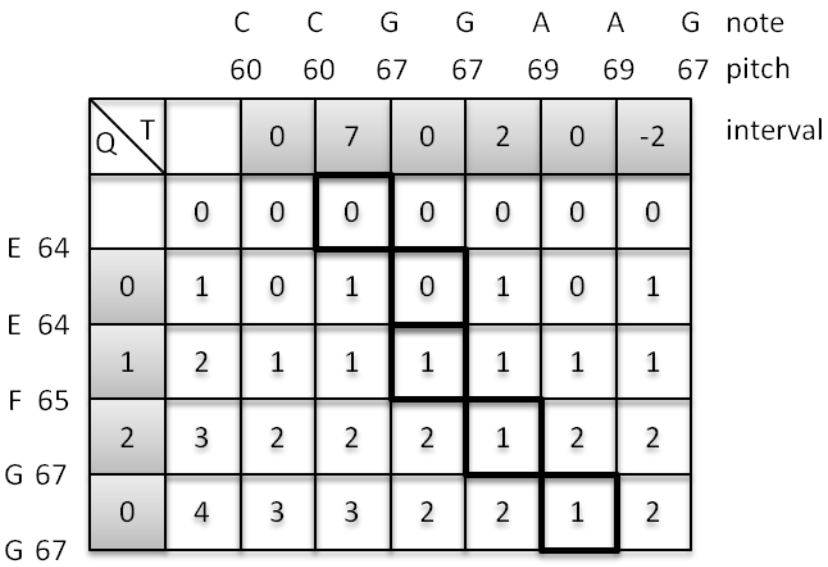


Figure 2.4: Local Alignment computation

The edit distance is given by $min_{0 \leq j \leq n}(d_{m,j}) = 1$, which corresponds to the alignment shown in table 2.2. Matches are aligned vertically and skips are indicated by an asterisk (-2 is considered as one character).

	Alignment
Target:	`0 7 0 * 2 0 -2`
Query:	`* * 0 1 2 0 *`

Table 2.2: Best local alignment

The choice of costs in the previous example is arbitrary and does not take musical considerations into account. This is addressed by replacing $skipCost_q$ and $skipCost_t$ by the functions $skipQueryCost(q_i)$ and $skipTargetCost(t_j)$, and implementing them (and $replaceCost(q_i, t_j)$) in a way which takes into account musical considerations[4]. The improved version is given in Algorithm 1.

[4]Described in the Implementation chapter.

Algorithm 1: Local Alignment with Cost Functions

1. for $0 \leq j \leq n$
2. $\quad d_{0,j} = 0$
3. for $1 \leq i \leq m$
4. $\quad d_{i,0} = d_{i-1,0} + skipQueryCost(q_i)$
5. for $1 \leq i \leq m$
6. $\quad$ for $1 \leq j \leq n$
7. $$d_{i,j} = min \begin{cases} d_{i-1,j} + skipQueryCost(q_i) \\ d_{i,j-1} + skipTargetCost(t_j) \\ d_{i-1,j-1} + replaceCost(q_i, t_j) \end{cases}$$
8. $cost = min_{0 \leq j \leq n}(d_{m,j})$

2.3.2 Indexing

2.3.2.1 The Problem of Indexing Non-Metric Spaces

Indexing concerns organising data in a data structure that facilitates fast searching. Given a set of data elements in a search space, we want to organise the data such that finding the data element which is "nearest" to a query element can be done efficiently without traversing the entire search space. Much research has gone into data structures for indexing Euclidean spaces and general *metric* spaces[5]. A metric space has a distance function $d(x, y)$ which obeys the following conditions:

- $d(x, y) = d(y, x)$ (symmetry)
- $0 < d(x, y) < \infty, x \neq y$
- $d(x, x) = 0$
- $d(x, y) \leq d(x, z) + d(z, y)$ (triangle inequality)

In the case of my system, the search space is non-Euclidean, as only the pairwise distances between elements can be computed. It is also non-metric, since in order to get musically meaningful results, there must be a difference between comparing a query to a target and comparing two targets, and so the distance function does not obey the symmetry condition. To the best of my knowledge there is no well known solution to indexing non-metric spaces, so I had to come up with my own solution to the problem, described in the next section.

[5]For example the mvp-tree [2].

2.3.2.2 Seed Search as a Solution to Indexing

My solution is based on the BLAST algorithm [5] used in Bioinformatics for DNA similarity searching, and adopts the idea of dividing the search into stages, the first of which is a "Seed Search" (Algorithm 2). Given a query string, I extract short subsequences of the string called *seeds*. For example, given the string `12345` and using a seed length of 3, the seeds are `123`, `234` and `345`. Next, I perform an exact search for these seeds in the database. This will retrieve the targets which contain at least one exact match to one of the seeds extracted from the query. An exact search is a lot faster than Local Alignment, and will provide a significant reduction in the number of targets to search through. The retrieved targets are matched against the query using Local Alignment and the songs which produced the top matching targets are returned.

Algorithm 2: Seed Search

1. $S = extractSeeds(Q)$
2. $Targets = \varnothing$
3. while $notEmpty(S)$
4. $\quad seed = getElement(S)$
5. $\quad Targets = Targets \cup exactSearch(seed)$
6. return $Targets$

The exact search can be precomputed when a target is added to the database. Given a target, I extract all the seeds it contains and use them as keys into a hash table. The table takes a seed as a key and returns the set of all targets containing that seed. This reduces the exact search to retrieving an element from a hash table which is $O(\log(n))$ if the number of seeds in the database is n.

2.4 Project Design

In the following sections I will describe the design work and implementation decisions made prior to the implementation phase.

2.4.1 Language and Environment

I decided to implement the project in Java, using the Eclipse development environment. The reasons for this are:

- The portability of Java and Java Applets means the program can easily support multiple platforms, and could easily be extended to support a

client-server architecture which offers a complete graphical user interface online.

- Java has built in libraries for handling MIDI, including reading in MIDI files, generating new MIDI events and synthesizing sound.
- I have experience developing Java applications using the Eclipse IDE.

2.4.2 Data Backup and Documentation

I will use a CVS repository to maintain recent backups of the project code. The entire repository will be backed up on my personal computer, the SRCF server and on my PWF filespace several times a week. A hard copy will also be taken periodically on CD-ROM. The program code will be documented using the JavaDoc facility and continuous progress will be reported in a wiki.

2.4.3 Modular Design

The project is divided into several core modules, which communicate with each other through defined interfaces. This modular design will allow independent development and testing of each module, and provides checkpoints for monitoring the overall progress of the project. The modules are:

- **Database** – Stores the Targets and song metadata in relevant data structures, also facilitating the seed search.
- **Extractor** – Extracts Melodies from MIDI files.
- **Standardiser** – Converts Melodies into standardised format – Targets.
- **Matcher** – Compares a Query to a Target.
- **Searcher** – Performs the complete search process including the seed search.
- **User Interface** – Provides ways of producing queries, issuing search requests and viewing the results.

In the case of a client-server architecture extension, I will also have a **Networking** module for handling communication between the client (which will include the user interface) and the server (comprised of all the other modules).

In addition to the core modules, other code will include:

- A collection of test programs for testing the modules separately, as well as testing module integration.
- A set of programs to assist project evaluation.

The overall structure is depicted in figure 2.5 overleaf.

2.4.4 Evaluation Design

2.4.4.1 Test Corpus

Most evaluation tasks I intend to carry out require a carefully compiled test corpus in which all MIDI files are validated to be uncorrupted, polyphonic and properly named. For this purpose I have compiled a corpus of 1,076 MIDI files. For scalability tests I have also compiled a corpus of approximately 16,000 files.

2.4.4.2 Evaluation

Project evaluation will take into account three factors – search performance, efficiency and usability. The first two require collecting user queries, and the third performing usability tests. The evaluation will involve data collection through a set of user tests combining usability evaluation and production of user queries, followed by a detailed analysis of the collected data, all of which is fully described in the Evaluation chapter.

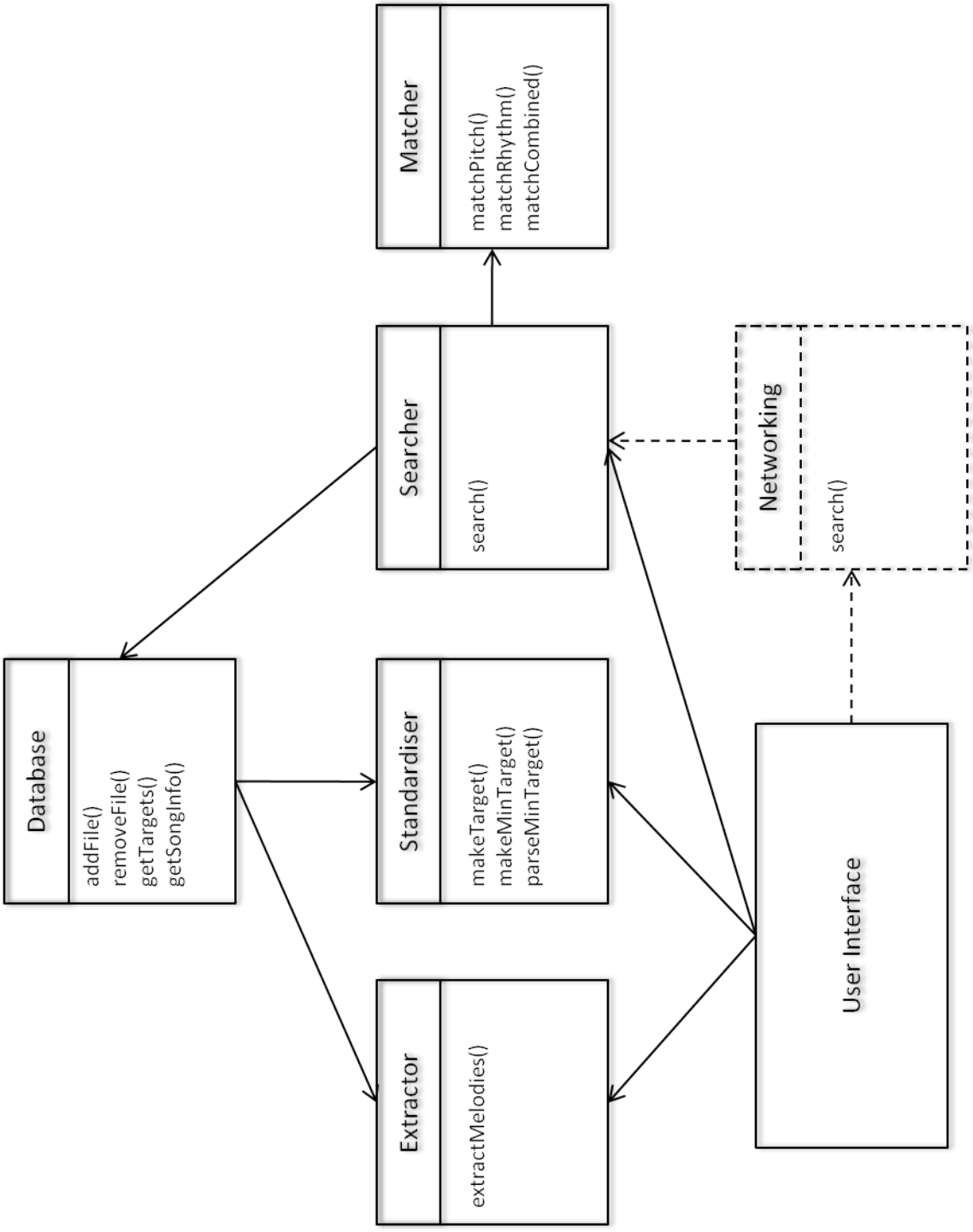


Figure 2.5: The overall project structure

Chapter 3

Implementation

In this chapter I will describe how I implemented the modules and how they combine to form the complete system. I will also describe the extensions I have implemented and any important implementation decisions made.

3.1 Data Representation and Organization

I will start by describing the classes implementing the data refinement model described in section 2.2.3, producing MinTargets for database storage. I will then explain how the database is implemented and organises the data.

3.1.1 Target Representation

3.1.1.1 Melody Extraction

The `javax.sound.midi.MidiSystem.getSequence()` method reads in a MIDI file and returns a `Sequence` object, a data structure containing the MIDI file's tracks. The `Extractor` class converts a `Sequence` into a set of `Melody` objects which are later passed to the `Standardiser`. A `Melody` object stores a vector of `Note` objects, each containing the data of a single note – its *pitch*, *onset tick*, *offset tick* and *IOI* (measured in ticks).

Melody extraction is performed by the method `extractPolyMelodies()`, which extracts a melody from every track. The method scans through a track's MIDI events converting pairs of "Note On" and "Note Off" events with matching pitch values into a series `Note` objects, ignoring any other MIDI events encountered. A track can be polyphonic (contain overlapping notes or chords), in which case a monophonic melody is extracted from it using the following rules:

- If a chord is encountered, only the highest note (likely to be the most perceptually salient [13]) is kept.
- If a note starts before the previous note has ended, the overlap is removed by trimming the duration of the previous note.

Once a series of notes is extracted, a second pass is performed to set the IOI value for every note. The method `setIOIs()` takes a collection of `Melody` objects and sets the IOI value for every note in every melody. The final step is to check that the extracted melody is longer than a certain threshold (some tracks contain a single cymbal hit or a few isolated notes, and should be disregarded). The complete extraction process is described in pseudo code in Algorithm 3.

Algorithm 3: Melody Extraction

```
1.  melodies = new Vector<Melody>()
2.  for every Track in the Sequence
3.      melody = new Melody()
4.      haveStart = false
5.      startTick = 0
6.      currentPitch = 0
7.      for every Event in the Track
8.          message = event.message
9.          if (message == Note On)
10.             /* If the previous note has ended, start a new note */
11.             if (haveStart == false)
12.                 startTick = event.tick
13.                 currentPitch = message.pitch
14.                 haveStart = true
15.             /* If it is a chord keep the highest note */
16.             else if (event.tick == startTick)
17.                 currentPitch = max(message.pitch, currentPitch)
18.             /* If it is an overlapping note truncate the previous
19.             note and start a new note */
20.             else
21.                 Melody.addNote(currentPitch, startTick, event.tick)
22.                 startTick = event.tick
23.                 currentPitch = message.pitch
24.         else if (message == Note Off)
25.             if (haveStart == true && currentPitch = message.Pitch)
26.                 melody.addNote(currentPitch, startTick, eventTick)
27.                 haveStart = false
28.     /* Make sure melody length is above threshold */
29.     if (melody.size > threshold)
30.         melodies.add(melody)
31. /* Set the IOIs for the notes in the extracted melodies */
32. setIOIs(melodies)
33. return melodies
```

3.1.1.2 Standardisation, Minimisation and Target Filtering

`Standardiser` class provides methods for converting a `Melody` into a `Target`, and minimising a `Target` into a `MinTarget`.

As described in section 2.3.1, a target is a sequence of $\langle pitch\ interval, LogIOIR \rangle$ note intervals. To minimise storage requirements pitch and rhythm are stored in two separate `Byte` arrays. In addition a target will store the filename of the MIDI file from which it was derived, and a target ID to differentiate it from other targets with the same filename. The method `makeTarget()` converts a `Melody` (n notes) into a `Target` ($n-1$ note intervals) using equations 3.1 and 3.2.

$$pitch[i] = notePitch(i+1) - notePitch(i) \tag{3.1}$$

$$rhythm[i] = round\left(\log\left(\frac{noteIOI(i+1)}{noteIOI(i)}\right)\right) \tag{3.2}$$

The minimisation phase is performed by `makeMinTarget()` and was designed to separate the format used for matching from the storage format, so that the latter can be optimised or changed without affecting the matching code. In practice the target format described above is both convenient for matching and efficient for storage, so there is no internal difference between a `Target` and a `MinTarget`.

The final step before returning a target is passing it through the filtering method `isUseful()`. In section 2.2.3 I explained that a melody is extracted from every track. There are however two types of tracks which are never used. The first is the drum track, and the second is any track which is too short to be meaningful. Short tracks are filtered out during the extraction phase. Targets created from drum tracks tend to have very long sequences of pitch intervals all with value 0, so they can be detected and disregarded.

3.1.2 Query Representation and Contour

As explained in section 2.2.3, a query made by the user is converted into the same standardised format as a Target. A standardised query is represented by the `Query` class, which is identical to the `Target` class with one important exception. In addition to the pitch interval and LogIOIR every note interval is extended to contain the four values $\langle pitch\ interval, LogIOIR, pitch\ contour, rhythm\ contour \rangle$. The contour values are stored in two additional character arrays – *pitchContour* and *rhythmContour*.

Pitch Contour is a further abstraction of pitch intervals in which only the direction of the melody is considered. I divide pitch contour into five categories – *big jump down* ($interval \leq -5$), *little jump down* ($-4 \leq interval \leq -1$), *same* ($interval = 0$), *little jump up* ($1 \leq interval \leq 4$) and *big jump up* ($interval \geq 5$), represented in the *pitchContour* array by the characters `D`, `d`, `s`, `u` and `U` respectively.

Rhythm Contour abstracts rhythm ratio by only considering whether the next note is *shorter* ($LogIOIR < 0$), *equal* ($LogIOIR = 0$) or *longer* ($LogIOIR > 0$), stored in the *rhythmContour* array as `s`, `e` and `l` respectively.

The four values $\langle pitch\ interval, LogIOIR, pitch\ contour, rhythm\ contour \rangle$ form a *query interval*. The first two are *precise* elements and the last two are *contour* elements. Given a query, the contour elements of each query interval are determined by checking which contour category fits the precise elements. Contour is used to refine the Local Alignment algorithm, as detailed in section 3.2. Including contour also means the user can make a query by specifying the contour values directly. This offers users who do not remember the exact notes or rhythm a simple yet powerful method of searching.

3.1.3 Database and Indexing

3.1.3.1 Data structures

Following the data refinement model detailed in section 2.2.3, the `Database` class consists of two main data structures, one for storing the MinTargets extracted from the MIDI representation of a song, and the other for storing the song's metadata. The relationship between song and MinTargets is shown in figure 3.1.

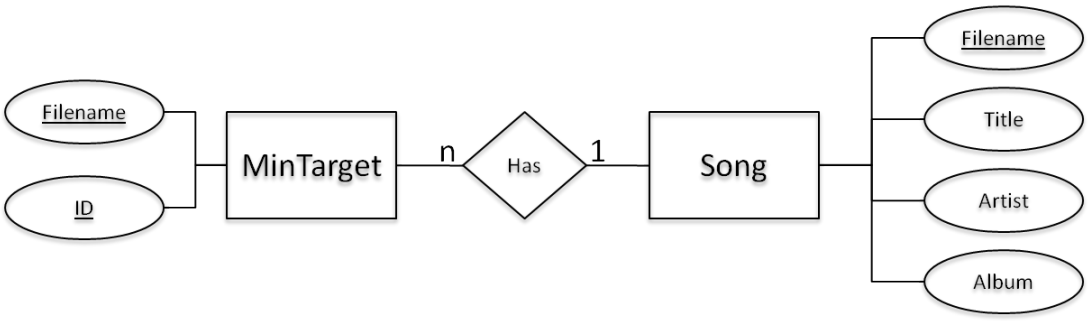


Figure 3.1: Entity-relationship model for database

A song is added to the database using using the `addFile()` method which takes a MIDI file and song metadata. The metadata is stored in a `Song` object, and all `Song` objects are stored in a hash table keyed by filename. The MIDI file is passed through the `Extractor` and the `Standardiser` to get the MinTargets. As explained in section 2.3.2.2, MinTargets are organised in a hash table (the *seed table*) to facilitate a fast exact seed search. Each seed contains m pitch intervals, and is represented as a comma separated string. The seed size m is parameterised so that the effect of different seed sizes on search performance can be evaluated later. The seed table is keyed by seed, and returns a `HashGroup` – a container class holding a reference to every MinTarget containing that seed. To add a MinTarget to the database I first extract the seeds from it. Seeds containing a pitch interval whose absolute value is greater than 24 (two octaves) are disregarded. Such intervals are highly unlikely to appear in a query, and this helps limit the size of the seed table. I then key the seed table using the seeds and add the MinTarget to the relevant HashGroups.

Both the seed table and the MinTargets are always kept in memory, rather than keeping the table in memory and the MinTargets on disk. This is a design choice I made following these observations:

- Keeping the MinTargets in memory will make the search process faster.
- A single MinTarget object is shared internally by all the HashGroups.
- The MinTargets are designed to be very sparse. For the purpose of the project, even the largest database can be kept in memory (a corpus of 25,000 MIDI files, taking over 1GB in disk space and resulting in over 110,000 searchable targets uses less than 240MB of RAM).

3.1.3.2 Functionalty

The database provides methods for adding new files, removing files, retrieving all the MinTargets matching a seed, saving the database and loading it. The database is saved as a single file `database.msdb`, which holds the two main data structures and the information they contain.

3.2 Matching

In this section I will describe the implementation of the Local Alignment algorithm in the `Matcher` class. I will also explain how I use contour to make the matching more musically meaningful.

3.2.1 Implementing Local Alignment

The algorithm (as described in section 2.3.1.3) is implemented twice in two different methods: `matchTargetPitch()` and `matchTargetCombined()`. The former calculates the cost of matching a query to a target based only on the pitch data, whilst the latter factors in the rhythm data as well. The pitch-only version allows users to make pitch-only queries in the text search, or ignore poor rhythm in queries played on the on-screen keyboard.

For the method taking pitch and rhythm into account, the cost stored in every cell of the matrix is the combination of the cost of matching the pitch elements and the cost of matching the rhythm elements. Costs are combined using `combineCosts()`, explained section 3.2.2. The extended algorithm used in the `matchCombined()` method is given in Algorithm 4.

Algorithm 4: Local Alignment Combining Pitch and Rhythm

1. for $0 \leq j \leq n$
2. $\quad d_{0,j} = 0$
3. for $1 \leq i \leq m$
4. $\quad d_{i,0} = d_{i-1,0} + combineCosts \begin{cases} skipQueryPitch(Q,i) \\ skipQueryRhythm(Q,i) \end{cases}$
5. for $1 \leq i \leq m$
6. $\quad$ for $1 \leq j \leq n$
7. $\quad\quad skipQueryCost = d_{i-1,j} + combineCosts \begin{cases} skipQueryPitch(Q,i) \\ skipQueryRhythm(Q,i)) \end{cases}$
8. $\quad\quad skipTargetCost = d_{i,j-1} + combineCosts \begin{cases} skipTargetPitch(T,j) \\ skipTargetRhythm(T,j)) \end{cases}$
9. $\quad\quad replaceCost = d_{i-1,j-1} + combineCosts \begin{cases} replacePitch(Q,T,i,j) \\ replaceRhythm(Q,T,i,j)) \end{cases}$
10. $\quad\quad d_{i,j} = min \begin{cases} skipQueryCost \\ skipTargetCost \\ replaceCost \end{cases}$
11. $cost = min_{0 \leq j \leq n}(d_{m,j})$

Choosing an appropriate string representation for a melody (as detailed in previous sections) is the first step in ensuring the Local Alignment algorithm returns musically meaningful results. The second is ensuring that the cost functions *skipQueryPitch*(), *skipQueryRhythm*(), *skipTargetPitch*(), *skipTargetRhythm*(), *replacePitch*() and *replaceRhythm*() return costs that reflect the degree of musical similarity between the note intervals under consideration. I do this by taking *contour* into account, explained in the next section.

3.2.2 The Cost Functions

The `SearchParameters` class holds all the values returned by the cost functions and other parameters used in the matching and the searching. This will later enable me to easily change costs and search parameters to examine their effect on search performance. The implementation of the functions is based on the following musical assumptions:

- The user might jump an octave or forget to jump an octave while playing.
  - Replacing a pitch interval which differs from the correct interval by a multiple of 12 should have a lower cost.
- The user might repeat the same note fewer or more times than in the original melody.
  - Skipping a pitch interval of 0 should have a lower cost.
- The user might remember the general pitch contour of a melody, but get the precise pitch value wrong.
  - Replacing a query interval with an incorrect pitch interval but correct pitch contour should have a lower cost.
- The user might remember whether the next note is longer or shorter, but not exactly by how much.
  - Replacing a query interval with an incorrect LogIOIR but correct rhythm contour should have a lower cost.
- The pitch and rhythm might be represented with different degrees of accuracy, i.e. the pitch might contain more mistakes than the rhythm and vice versa.
  - The pitch and rhythm costs should be scaled by different factors when combined into a single cost.

Following these assumptions, parameters are grouped into three categories:

- **Full Costs** – costs for skipping or replacing completely incorrect query intervals. Set by default to 2.
- **Reduced Costs** – costs for skipping or replacing query intervals with incorrect precise elements but correct contour element; costs for replacing pitch intervals which are off by an integer number of octave; and the cost for skipping pitch intervals of 0. Set by default to 1.
- **Factors** – the *combineCosts*() function multiplies the pitch cost by a `PITCH_FACTOR` and the rhythm cost by a `RHYTHM_FACTOR` and sums them together. The factors determine the pitch to rhythm ratio, i.e. which of the two has greater influence on the matching. Set by default to 1:1.

A complete list of all search parameters is provided in appendix A. Every cost function compares the query interval to the target interval and determines whether the cost should be 0, reduced or full, and returns the appropriate cost. The algorithm for *replacePitch*() is given as an example (Algorithm 5), and the Java code is provided in appendix B. The other cost functions are implemented similarly.

Algorithm 5: Calculating the cost of replacing a pitch interval

```
1.  Compare the pitch interval from the query to the interval from the target
2.  First check for a match:
3.  If they are equal
4.      Return 0
5.  Then check for a reduced cost:
6.  If they differ by an integer number of octaves
7.      Return REPLACE_PITCH_OCTAVE_COST
8.  If they belong to the same contour category
9.      Return REPLACE_PITCH_CONTOUR_COST
10. Otherwise must return a full cost:
11.     Return REPLACE_PITCH_COST
```

3.3 Searching

In this section I will explain how the data structures and the matching algorithm described so far are combined to perform the task of searching for a query in the entire database.

3.3.1 Concurrent Searching

The complete search process is implemented in the `Searcher` class and carried out by the `searchReturnSongsMultiThreaded()` method. The method takes a `Query` object, and parameters indicating whether the search should be "pitch-only" or include rhythm and how many results should be returned. The entire process is multi-threaded to make the search faster – I compared the final version of the search to the initial non-threaded version by measuring the search time for a set of user generated queries. Multi-threading was found to reduce the average search time by a factor of 3 when running on a server with two dual-core processors[1].

The search starts by calling `getTargetsForSearch()` to retrieve the targets for the Local Alignment stage. The method takes a `Query` object, extracts the seeds, keys the seed table and pools all the resulting MinTargets together. Getting seeds from a query is trivial except in the case when the user includes contour in their query, in which case some query intervals will only contain contour elements and no pitch intervals. In this case the seeds (which are meant to be subsequences of pitch intervals) will contain gaps, which can be filled by any pitch interval in the range of the specified contour category. For example, if a query contains the seed `2,u,-3`, every seed starting with `2`, followed by a value between `1` and `4` and ending with `-3` is relevant to the query. This is solved by scanning through the keys in the seed table (i.e. the seeds) for relevant seeds and including them in the seed search. Next the retrieved targets are split into m equally sized groups (eight by default). Each group is passed to a separate `SearcherThread` which computes the Local Alignment cost for every target and returns the top n matching songs together with their costs[2]. The results from all the threads are pooled together, and the top n results overall are returned. The process is thread safe since the only write operation performed by a thread is adding its search results to the pool of results, done by calling `addSongs()` which is *synchronized* (uses a mutual exclusion lock). The operation of each individual thread is described in the following section.

[1] Full specification of the SRCF server and a detailed evaluation of search performance is given in chapter 4.

[2] n is parameterised. For displaying results in the user interface $n = 10$.

3.3.2 The Search Process

The `SearcherThread` takes a `Query` and a set of MinTargets and returns the top n scoring `Song` objects together with their match cost. I convert every `MinTarget` into a `Target` and compute the match cost against the query using the `Matcher`. I use a scoreboard to keep a Target's match cost together with the `Song` object containing the metadata of the song from which the target was extracted. If another target extracted from the same song gives a lower cost the scoreboard is updated, so overall the lowest cost for a song is kept. Once I have computed the matching cost for all the targets, the top n songs in the scoreboard are returned to the `Searcher`.

3.4 Client-Server Architecture

3.4.1 Motivation

In my project proposal I suggested implementing a client-server architecture as a possible extension. Bearing this in mind I designed the system in a modular manner so that none of the components described so far would require any alteration. A client-server architecture has clear advantages:

- The system is easily accessible to anyone through a web browser.
- The MIDI files and database only need to be stored on a single server machine.
- Any heavy computation is performed by the server.

I realised that these properties would be very useful for the evaluation of the project. I could run user tests simultaneously, or let users do part of the test remotely. With the project on schedule, I decided to implement this architecture as part of the core project.

3.4.2 The Networking Module

The networking module consists of two main classes, `MuSearchClient` and `MuSearchServer`. The server class is an executable java program which runs continuously on a server machine. Once started it loads `database.msdb` from disk and then listens for incoming TCP connections. To handle concurrent search requests, a new `MuSearchServerThread` is run for every incoming request.

Communication between client and server is implemented using the `java.io` and `java.net` libraries which provide methods for creating and handling TCP connections. When a user issues a search request the user interface instantiates a new `MuSearchClient` object and calls `sendQuery()`. The method is passed a `Query`, the `SearchParameters` to be used and a boolean `pitchOnly` indicating whether rhythm should be included in the search[3]. The client sends these objects to the server using the MuSearch Protocol which runs over the TCP connection. The server performs the search and sends the client a `Vector` containing the search results. The client finally invokes a method to display the results in the user interface.

The MuSearch Protocol works as follows. Once a connection is established the client can optionally send a *setSP* message, after which it must send a `SearchParameters` object. It then sends a *search* message, after which it must send the `Query` object followed by the `pitchOnly` boolean. The server will then use a `Searcher` to perform the search and obtain a `Vector` of search results (each result is a `ResultPair<Song>` object containing a `Song` object and a cost) which is sent back to the client. Finally the client must send an *end* message to close the connection (also closed if the client crashes). A typical session is described in figure 3.2.

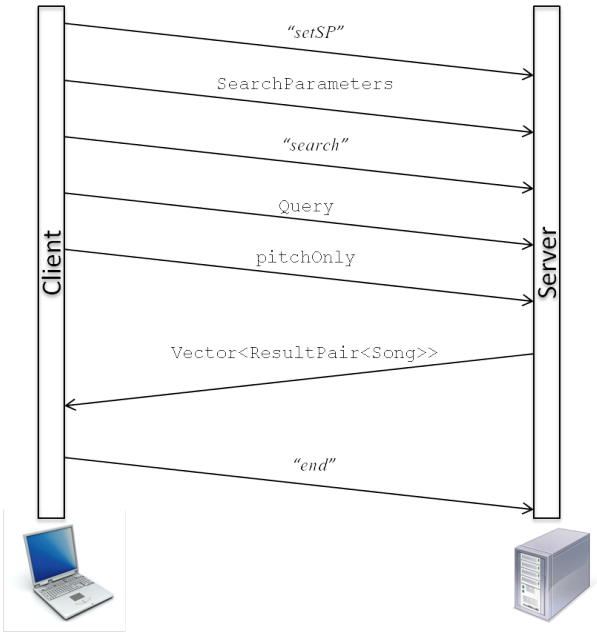


Figure 3.2: The MuSearch Protocol

[3]The objects are validated by the user interface before being passed to the `MuSearchClient`.

3.5 User Interface

Following the project specifications, the MuSearch GUI is divided into three panels. The *Text Search* panel for inputting queries in textual form, the *Keyboard Search* panel for inputting queries by playing on a graphically visualised keyboard and the *Results* panel for displaying the results of the search. In the following sections I will describe the overall design of the user interface and how each panel is implemented.

3.5.1 Design Overview

The GUI is implemented as a `JApplet` embedded in a web page. To make it clear I decided to have all three panels visible at all times, separated by borders. The initial layout design is shown in figure 3.3 and the end result including added extensions is shown in figure 3.4.

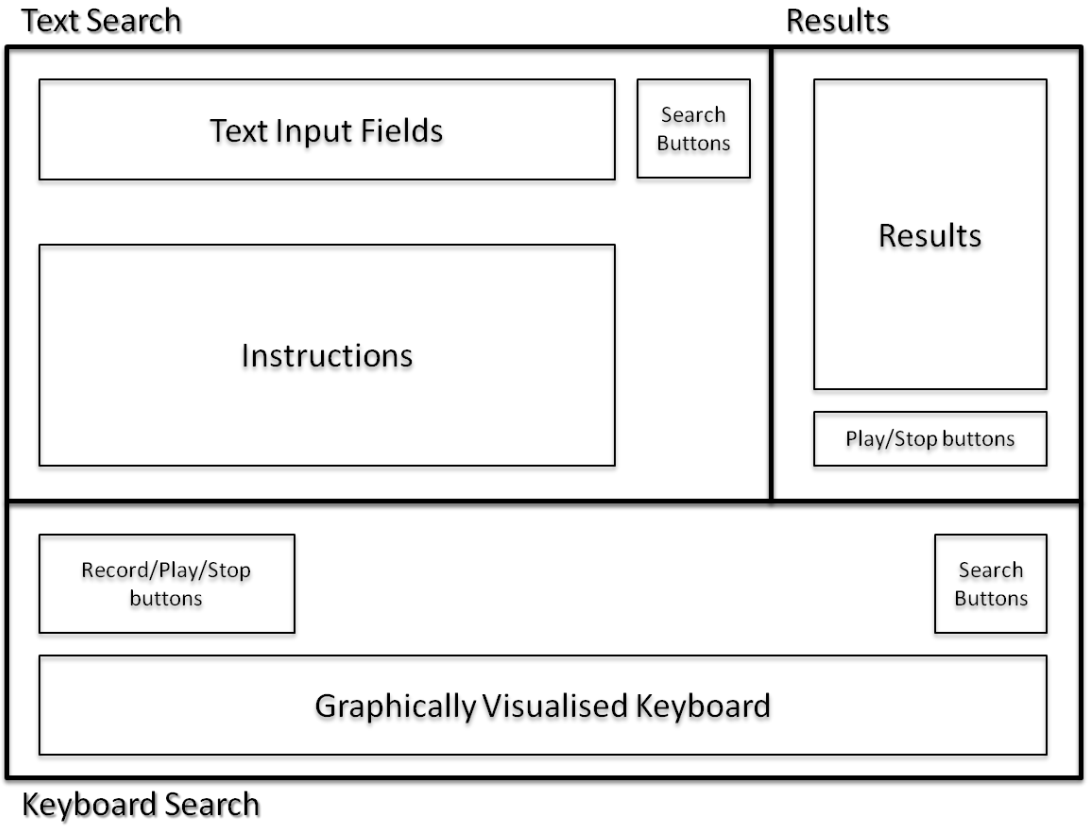


Figure 3.3: The initial MuSearch GUI layout design

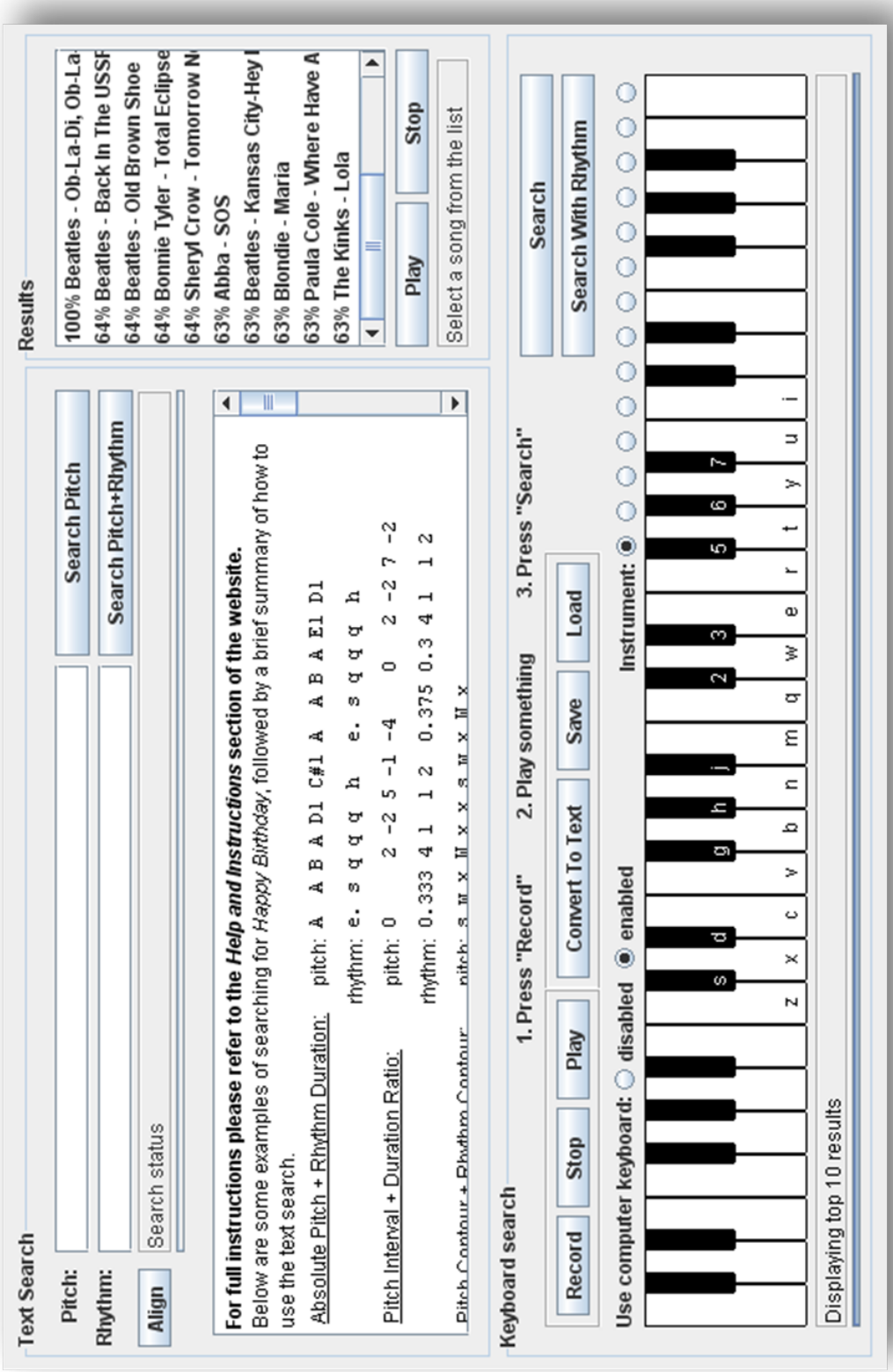


Figure 3.4: The MuSearch user interface

3.5.2 The Keyboard Search

3.5.2.1 Overview

The keyboard search is intended as the primary search method for non-expert users. As explained in [1], a *direct manipulation* approach to query specification often evokes enthusiasm from users, and is in many cases easier to use than other methods.

3.5.2.2 The Keyboard Search GUI

The on-screen keyboard is implemented as a set of `JPanel` objects representing individual keys, which makes detecting mouse events on specific keys easy. The GUI has *Record*, *Play* and *Stop* buttons allowing the user to record a query and play it back. It has two search buttons – *Search* for making pitch-only queries and *Search With Rhythm* for including rhythm in the search[4]. It also has a status panel to display error messages and a progress bar to indicate search progress or that the program is recording.

3.5.2.3 Making Queries

I use the `javax.sound.midi` library to get access to the client machine's default MIDI devices – the `Synthesizer` and the `Sequencer`. MIDI messages can be sent to the `Synthesizer` to produce sound, and to the `Sequencer` for creating new MIDI sequences and playing them back. Using MIDI messages to log played notes has significant advantages over an ad-hoc solution:

- The recorded query can easily be replayed.
- The query can be saved to a MIDI file.
- Query standardisation can be done using the existing methods in the `Extractor` and the `Standardiser`.

When the *Record* button is pressed, a new `Sequence` containing a single `Track` is created, and the `Sequencer` is set to record any MIDI messages it receives as events in this `Track`. When the user presses a key on the on-screen keyboard, the key will repaint itself to indicate that it is pressed and will invoke a `keyPressed()` method from the keyboard panel. This method uses the key information to create a new MIDI message and timestamp. The message is sent to the `Synthesizer`

[4]The pitch-only search button is labeled "Search" since initial tests showed that in most cases a pitch-only search produced better results.

so that a note is sounded, and to the `Sequencer` together with the timestamp so that it is added to the `Track`. A `keyReleased()` method does the equivalent when a key is released. To increase usability keys can be played using either the mouse or the computer keyboard. The user can stop the recording by pressing the *Stop* button or the search buttons. Pressing the `Play` button will start playback of the recorded `Sequence`.

When the search button is pressed, a new `KeyboardMultiSearchThread` is created to handle further execution. This allows the GUI to remain responsive while the search is performed in the background. The thread will use the `Extractor` and the `Standardiser` to generate the precise pitch and rhythm elements of the query, and use them to determine the contour elements. If no query was recorded or if it is too short an error message is displayed (the minimum number of notes must be greater than the seed size by at least one so that seeds can be extracted). Otherwise a new `MuSearchClient` is instantiated and passed the `Query`, a `SearchParameters` object initialised to default values, and depending on which of the two search buttons was pressed a `pitchOnly` boolean. The client performs the search and invokes `displayResults()` from the results panel which is described in section 3.5.4.

3.5.3 The Text Search

3.5.3.1 Overview

The text search allows querying the system by representing melodies as strings of characters. As such, it requires more musical ability than the keyboard search, but it is also more flexible and powerful for advanced users.

3.5.3.2 The Text Search GUI

The text search panel consists of two input fields, one for inputting pitch values and one for rhythm values. It has two search buttons – *Search Pitch* and *Search Pitch+Rhythm*, the former only includes the values from the pitch field in the search whilst the latter includes the values from both fields. Values must be space separated, and an *Align* button is provided to help the user see which pitch value goes with which rhythm value. Below the text fields there is a status panel for displaying error messages and a progress bar to indicate search progress. Finally there is an instructions box which contains example queries and instructions for using the text search.

3.5.3.3 Making Queries

The text search supports three different query formats. The formats are detailed in appendix C, and example queries are given in table 3.1. The *contour* format in particular is very simple – pitch is represented by `W,w,s,x,X` corresponding to *big up*, *little up*, *same*, *little down* and *big down*, and rhythm is represented by `<,|,>` corresponding to *shorter*, *equal* and *longer*. Thus it is expected to be useful even for users with little musical experience. Allowing contour-only searches is possible due to the built in support for contour in the `Query` and the matching code.

Query Format	**Query**	
Absolute	pitch:	`A A B A D1 C#1 A A B A E1 D1`
	rhythm:	`e. s q q q h e. s q q q h`
Relative	pitch:	`0 2 -2 5 -1 -4 0 2 -2 7 -2`
	rhythm:	`0.333 4 1 1 2 0.375 0.3 4 1 1 2`
Contour	pitch:	`s w x W x x s w x W x`
	rhythm:	`< > | | > < < > | | >`

Table 3.1: Different query formats for "Happy Birthday"

The user can choose one of the formats or use a combination of all three in the same query. I convert the queries from these formats into the standardised format using the `Parser` class. The `Parser` is passed the pitch and the rhythm entries (or just the pitch for a pitch-only search) as two strings. The pitch string is converted into pitch tokens and the rhythm into rhythm tokens by looking for spaces in the string. I then perform a second pass on the tokens to ensure that they are valid entries. Finally the pitch tokens are converted into pitch intervals and pitch contours, the rhythm tokens into LogIOIRs and rhythm contours, and returned in a `Query` object.

When the user presses one of the search buttons the `Query` object is passed to a newly instantiated `MuSearchClient` which handles the rest of the search process. Which of the two buttons is pressed determines the value of the `pitchOnly` parameter of the search. As before, this process is executed in a new thread so that the GUI remains responsive while the search is performed.

3.5.4 Results Panel

The results panel displays the search results in a list. Results are ordered by cost (the lowest cost gets the highest rank) and then alphabetically by name. The user can select any of the results from the list and listen to the MIDI file using the *start* and *stop* buttons in the panel. A result entry includes the artist name and song title. Displaying the matching cost would not be meaningful to the user, and instead I display a match percentage indicating the relevance of the result to the query. The percentage is calculated according to equation 3.3.

$$percentage = 100 * \left(1 - \frac{cost}{worst\ possible\ cost}\right) \tag{3.3}$$

3.6 Extensions

In addition to the core project, I have implemented several extensions. As mentioned in section 3.4, the client-server architecture extension was implemented as part of the core project. The other extensions are detailed in the following sections.

3.6.1 Additional GUI Features

3.6.1.1 Convert to Text

Following initial usability tests users mentioned that it would be useful be able to view and edit the query they had recorded in the keyboard search by converting it into text and displaying it in the Text Search in the absolute format. I added this functionality by creating a *Convert to Text* button. I first use the `Extractor` to convert the recorded sequence into a `Melody`. From the `Melody` I can easily extract the pitch values and convert them into the appropriate characters. I then use the `Standardiser` to convert the melody into a target from which I can retrieve the LogIOIRs. I choose a starting value (e.g. `q`)[5] and add the rest according to the LogIOIRs.

3.6.1.2 Instrument Selector

I added a set of radio buttons for switching between instrument sounds. The MIDI specification defines a "program change" message for changing instrument sound. Whenever the user selects a different instrument a "program change" message is

[5]The value is chosen using an algorithm to avoid overflowing the available duration values.

sent to the `Synthesizer` and the `Sequencer` with the new instrument number, so future notes are played with the chosen sound.

3.6.1.3 Save and Load

I added *Save* and *Load* buttons so that the user can save a query as a MIDI file, and later load it again. The `javax.sound.midi` library provides methods for saving a `Sequence` as a MIDI file, and getting a `Sequence` from one. As my implementation uses MIDI sequences and message directly, this feature was easy to add.

3.6.2 Full MIDI File Comparison

I decided to extend the *Load* functionality so that instead of only loading single-track MIDI files generated from queries, the system could load any multitrack MIDI file and use it as a query. I load the MIDI file and separate it into tracks, and then convert the tracks into `Query` objects (similar to converting a sequence recorded with the on-screen keyboard into a `Query`). Searching for every `Query` separately, pooling the results and displaying the top ranking ones would not be meaningful, since different tracks will give high ranks to different songs, and the result would be a list of many different songs all with high ranks. To return songs which match the MIDI file as a whole I search for all the queries and *sum* all costs returned by different queries for the same song. The sum is normalised by the number of tracks used as queries, so high ranking songs are the ones which match the MIDI file in more tracks than others.

As expected, using a file which exists in the database as a query returns the correct song with 100% match, with all other results under 20%. For a different MIDI version of the same song the results depend on the degree of similarity between the arrangements. For very different arrangements (e.g. piano versus orchestration) the search is unsuccessful. For more similar arrangements with roughly the same number of parts the best match score is still low (around 35%), but the similarity between the arrangements as a whole is significant enough that the correct song is listed first, with all other results under 20%.

3.6.3 MIDI Controller Support

As a final extension I added support for playing queries using a real MIDI controller, such as a MIDI keyboard which connects to the computer via a MIDI or USB port. I use the `javax.sound.midi` library to detect MIDI controller devices and use the default controller as a `Transmitter`. I wrote a custom `Receiver`

which routes MIDI messages from the `Transmitter` to the `Synthesizer` and the `Sequencer`. The `Receiver` also checks the messages for note events and instrument changes so that the keys of the on-screen keyboard are highlighted when the corresponding notes are played on the controller, and the selected instrument changes in the selector when the instrument is changed through the controller.

Chapter 4

Evaluation

4.1 Overview

In this chapter I will describe how I evaluated the project. As explained in section 2.4.4.2, I will be evaluating three aspects of the project – usability, search performance and efficiency.

4.2 Test Corpus

In order to get meaningful test results, the corpus of MIDI files used to create the database must be chosen carefully. Firstly, every MIDI file must be checked to ensure that it is not corrupted, and that the music corresponds to the title of the file. Every song must only appear in the database once, and all songs should be polyphonic and of roughly the same genre. For this purpose I have compiled a corpus of 1,076 polyphonic MIDI files of rock and pop songs including 200 songs by The Beatles. Once added to the database, the songs translate into 6,541 searchable targets. For scalability tests I have compiled several more corpora with size increasing up to 16,077 MIDI files resulting in 76,805 searchable targets.

4.3 Data Collection Procedure

The first stage of evaluation was collecting test data. I devised and ran a user test on 13 participants of varying musical experience, ranging from amateur guitar players to people with music degrees. The test was divided into a usability section and a searching section. Due to Java's current implementation of the MIDI libraries, there is a slight delay between sending a MIDI message and a note being sounded, depending on the quality of the computer's sound card. The

majority of the usability tests were run on PWF machines with basic sound cards, resulting in a small yet noticeable delay.

4.3.1 Usability Section

Subjects were given a list of tasks to perform. This included recording queries on the on-screen keyboard, listening to results, using the different query formats in the text search and several other tasks. The complete list of tasks is given in appendix D. Upon completion of the tasks subjects were asked to fill in a questionnaire, given in appendix E. I observed every subject perform the tasks and took notes to complement the questionnaire data.

4.3.2 Searching Section

Next, subjects were asked to record queries using the on-screen keyboard. They were presented with a list of 200 Beatles songs, and were asked to record queries for songs they can remember from the list. This stage simulates the event where a user remembers part of a song they have not heard recently, and resulted in 63 "memory" queries saved as MIDI files. Finally, subjects were asked to listen to 10 audio recordings of Beatles songs and then record a query for each song. This stage simulates the event where a user has recently heard a song (e.g. on the radio or in a live performance), and resulted in 123 "afterplay" queries (some users did not have time to record queries for all 10 songs). This adds up to a total of 186 recorded user queries, which I could then use to evaluate the system.

4.4 Usability Evaluation Results

All subjects managed to use the basic features of the project with ease. That is, they were able (without guidance) to record a query using the on-screen keyboard, play it back, search for it and listen to the results. This confirms that the GUI is intuitive enough for a user to just "pick up and use", as would be expected from an online search facility. As expected, the Text Search proved to be more complicated and not all users managed to complete all the related tasks. Here are the results obtained from analysing the questionnaire data:

- The on-screen keyboard received an average score of 4.8 on a scale of 1 to 6 for comfort. The only comments were about the slight delay which made playing the rhythm accurately harder.

- There was a slight preference to using the computer keyboard for controlling the on-screen keyboard over the mouse, suggesting that providing the two alternative methods is a good idea.
- Most subjects found the instructions for using the text search clear, but prefer the keyboard search.
- As expected, contour was the preferred query format when using the text search, with subjects describing it as easier to use, especially when not entirely sure about the melody. A small number of the more musically experienced subjects preferred the absolute format as it is closest to musical notation.
- Most subjects received at least one error message when using the Text Search, usually due to a format mismatch between the pitch and rhythm entries. All subjects found the error messages helpful and all but one were able to correct their mistakes after referring to the instructions.
- Subjects were asked to compare the system to a QBH system in which queries are sung into a microphone. They were asked to specify whether using my system would be easier or harder by choosing a value on the the scale: 1 = much harder, 2 = harder, 3 = slightly harder, 4 = the same, 5 = slightly easier, 6 = easier, 7 = much easier. The mean result was 3.66 but with a standard deviation of 1.93 and answers ranged from 1 to 7. Subjects' comments indicate that their answer depended very much on their singing abilities. Some mentioned that my system is probably more for the musically inclined, but is also a good alternative for people with poor singing abilities.

4.5 Quantitative Results

4.5.1 Overview

When evaluating search performance of IR systems it is common to use metrics such as *precision* and *recall*. Precision is the proportion of relevant results out of all the retrieved results, and recall is the proportion of relevant results retrieved out of all relevant results in the database. In the case of my system however, there is always only a single relevant result – the specific song the user is searching for. For this reason it makes more sense to use a metric based on the *rank* of the correct song [4], and I use the *Mean Reciprocal Rank* (MRR). This is similar to

calculating the average rank of the correct song for a set of queries, but is less sensitive to poor ranking outliers. The formula for calculating the MRR is given in equation 4.1. n is the number of queries, and $rank_i$ is the rank of the correct song for query i.

$$MRR = \frac{\sum_{i=1}^{n} \frac{1}{rank_i}}{n} \tag{4.1}$$

For a database containing m songs, the MRR will be a value between 1 and $\frac{1}{m}$, with higher values indicating better search performance. The best MRR value achieved for a QBH system in [3] is 0.329. Since my system does not suffer query degradation by the need to perform pitch detection on recorded voice, the MRR value should be higher and in the success criteria I set a minimum value of 0.5 for the MRR. I also had to address the fact that it is not uncommon for different songs to have the same match cost for a given query, and thus the same rank. This means that though the correct song might have a high rank, in practice it will appear lower down the list of results, because there are other songs with the same rank which come first alphabetically. To address this I have introduced another metric which I call the *Ordered Mean Reciprocal Rank* (oMRR), which uses the actual position in the results list of the correct song to compute a MRR value.

System efficiency will be measured by calculating the average search time for a query. The usefulness of my proposed Seed Search as an indexing method will be measured by calculating the average number of targets returned by the Seed Search for the Local Alignment phase (labeled in tables as "#Targets to Search"), in comparison with the total number of targets in the database. All tests were performed on the SRCF server which hosts the MuSearch server – a machine with two Intel® Dual Core Xeon® 5130 @ 2GHz with 4MB Cache and 4GB RAM, running Linux 2.6.17-10 and Java HotSpot™ 64-Bit Server VM.

4.5.2 Initial Results

I started by computing the MRR and oMRR values for a pitch-only search on the test database of 1076 songs using a seed size of 3 and with the search parameters set to their default values (full cost = 2, reduced cost = 1, pitch to rhythm ratio 1:1).

The results (shown in table 4.1) are very pleasing, with all MRR and oMRR values well above 0.5 which is the minimum threshold for success[1]. As expected, there is a significant difference (as determined by a t-test with p-value of 0.00042)

[1]Search efficiency is evaluated in section 4.5.7.

Query Group	#Queries	MRR	oMRR
All queries	186	0.800	0.659
memory	63	0.765	0.627
afterplay	123	0.818	0.675

Table 4.1: Initial results

between the MRR and the oMRR values. Also, I noted that the "afterplay" queries gave slightly higher results than the "memory" queries. This might be because the melody was still fresh in the subject's memory when recording the query. I used a t-test to compare the reciprocal ranks of queries from both groups, and discovered that the observed difference was not statistically significant (p-value of 0.28443), so for the rest of the evaluation I used all the queries combined.

4.5.3 The Effect of Rhythm

Next I measured how including rhythm affects search performance. The results are shown in table 4.2.

Pitch Only	MRR	oMRR
true	0.800	0.659
false	0.667	0.594

Table 4.2: The effect of rhythm on search performance

Including rhythm generally degrades the results of the search. This can be explained by the difficulty of entering very accurate rhythm with the on-screen keyboard due to the slight delay.

4.5.4 Choice of Seed Size

So far I have been using a seed size of 3. The next stage of the evaluation was determining the effect of different seed sizes on both search performance and efficiency. A bigger seed size requires an exact match with a longer segment of the query, and so it is expected to improve efficiency (by reducing the number of targets retrieved by the Seed Search) at the cost of degrading search performance for queries containing mistakes. The results for different seed sizes are displayed in table 4.3.

Seed Size	Pitch Only	MRR	oMRR	Avg. Search Time/s	Avg. #Targets To Search	Fraction of Total #Targets in DB
2	true	0.801	0.659	0.475	5415.226	0.827
3	true	0.800	0.659	0.372	3250.559	0.496
4	true	0.791	0.664	0.198	1509.758	0.230
5	true	0.777	0.665	0.113	647.984	0.099
2	false	0.668	0.590	0.660	5415.226	0.827
3	false	0.667	0.594	0.480	3250.559	0.496
4	false	0.664	0.605	0.253	1509.758	0.230
5	false	0.664	0.620	0.146	647.984	0.099

Table 4.3: The effect of seed size on search performance and efficiency

As expected, the number of targets to search drops considerably as the seed size is increased, which in turn reduces the search time. I noted that the MRR gradually decreases, but the oMRR is surprisingly increased, which means the Seed Search gets rid of more irrelevant targets than it does relevant ones. I also noted that increasing the seed size increases the memory usage and storage requirements of the database. In conclusion, there is a trade-off between efficiency, resource usage and search performance, and for the rest of the evaluation I use a seed size of 4.

4.5.5 Search Parameter Optimisation

Next I turned to optimising the search parameters for searches which include both pitch and rhythm. As there are many search parameters, I decided to focus on two factors – the optimal pitch to rhythm ratio, and the optimal ratio between the full costs and the reduced costs when using the same value for all full costs, and the same value for all reduced costs. I wrote a program `Annealing` which uses the *Simulated Annealing* approach for global optimisation [6]. I divided the queries into two groups so that the optimisation is performed only on group 1, and the results can be validated on group 2. The results for the two groups before optimisation are give in table 4.4

Query Group	**Full Cost**	**Reduced Cost**	**Pitch to Rhythm Ratio**	**MRR**	**oMRR**
1	2	1	1:1	0.704	0.646
2	2	1	1:1	0.634	0.574

Table 4.4: Results for the groups before optimisation

4.5.5.1 Pitch to Rhythm Ratio

The optimal pitch to rhythm ratio was determined by changing the `PITCH_FACTOR` and the `RHYTHM_FACTOR` to get the highest MRR and oMRR values. The results are displayed in figure 4.1.

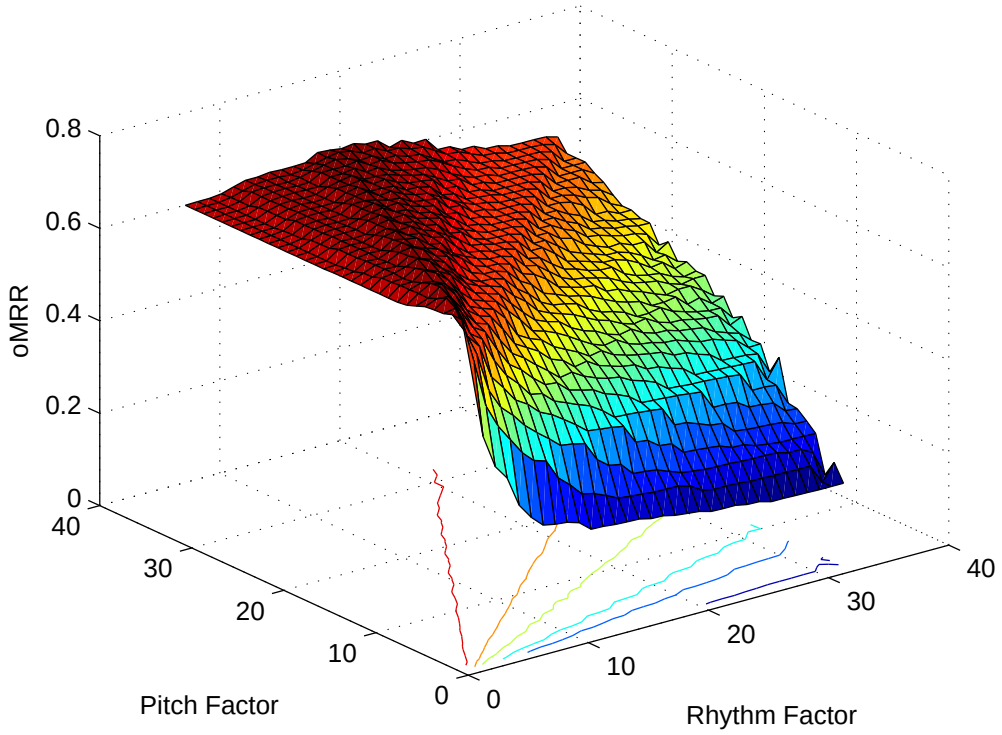


Figure 4.1: oMRR as a function of the pitch and rhythm factors

The optimal pitch to rhythm ratio was found to be 3:1, and is used in all further evaluations.

Ratio	**MRR**	**oMRR**
1:1	0.704	0.646
3:1	0.784	0.745

Table 4.5: Before and after pitch/rhythm ratio optimisation

4.5.5.2 Full Cost to Reduced Cost Ratio

The optimal ratio between full costs and reduced costs was determined by changing two values, one assigned to all *full cost* parameters (e.g. `SKIP_QUERY_PITCH_INTERVAL_COST`), and the other to all *reduced cost* parameters (e.g. `REPLACE_RHYTHM_CONTOUR_COST` and `REPLAC_PITCH_OCTAVE_COST`). The results are displayed in figure 4.2.

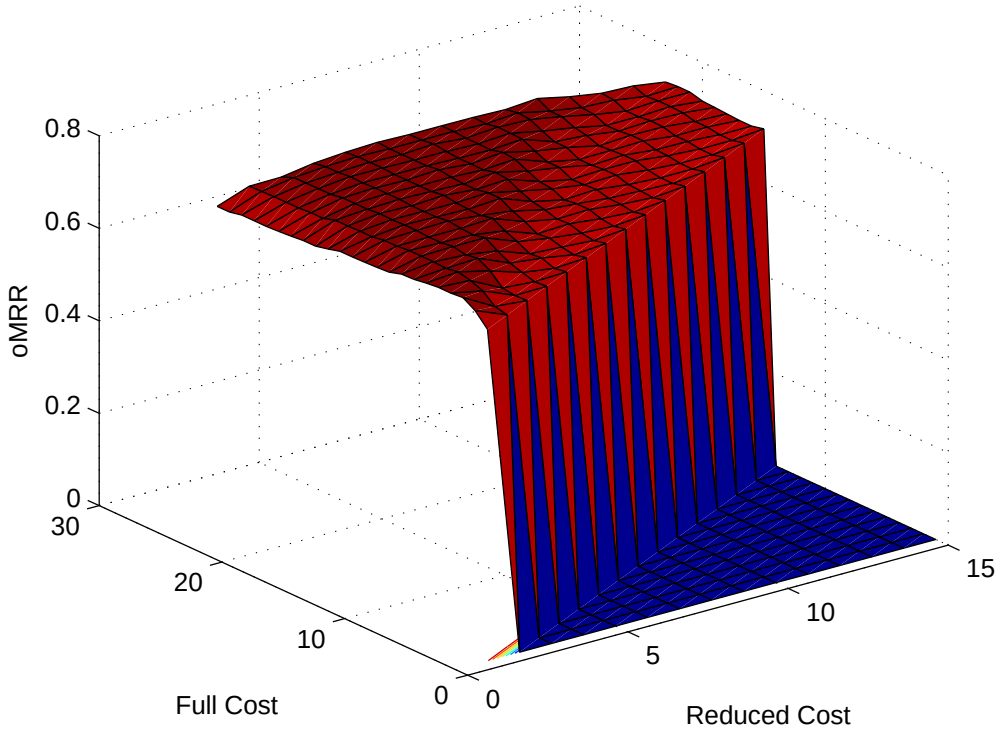


Figure 4.2: oMRR as a function of the full and reduced costs

The optimal values are 3 for full cost and 1 for reduced cost.

Full Cost	**Reduced Cost**	**MRR**	**oMRR**
2	1	0.784	0.745
3	1	0.787	0.761

Table 4.6: Before and after full cost/reduced cost optimisation

Finally I used the optimised parameters on group 2, and validated that they improve the search results, as seen in table 4.7.

Group	Pitch to Rhythm Ratio	Full Cost	Reduced Cost	MRR	oMRR
1	1:1	2	1	0.704	0.646
1	3:1	3	1	0.787	0.761
2	1:1	2	1	0.634	0.574
2	3:1	3	1	0.708	0.683
1+2	1:1	2	1	0.664	0.605
1+2	3:1	3	1	0.742	0.717

Table 4.7: Validation of optimised search parameters

4.5.6 Seed Filtering

I then measured which seeds were the most common in the queries and in the database. Full seed distributions are given in appendix F. The three most common seeds in the queries were {0,0,0,0}, {0,0,0,-2} and {0,0,0,-1}. As these seeds convey little melodic information (all represent the same note being played at least four times), I discovered that they could be ignored during the seed search, resulting in a further reduction in the average number of targets to search, at the cost of only slightly reducing the MRR and oMRR values.

Ignore Seeds	MRR	oMRR	#Targets to Search
no	0.742	0.717	1509.758
yes	0.733	0.712	951.575

Table 4.8: Results before and after seed filtering

4.5.7 Scalability

Finally, I tested how search performance and efficiency are affected as the size of the database is increased. I measured the MRR, oMRR, average search time and number of targets to search for databases of increasing sizes, starting from 6,536 targets (1,076 songs) up to 76,805 targets (16,077 songs). Note that since the search includes rhythm, these results provide a lower bound on performance. The results are plotted in figures 4.3–4.7, with trend-lines added.

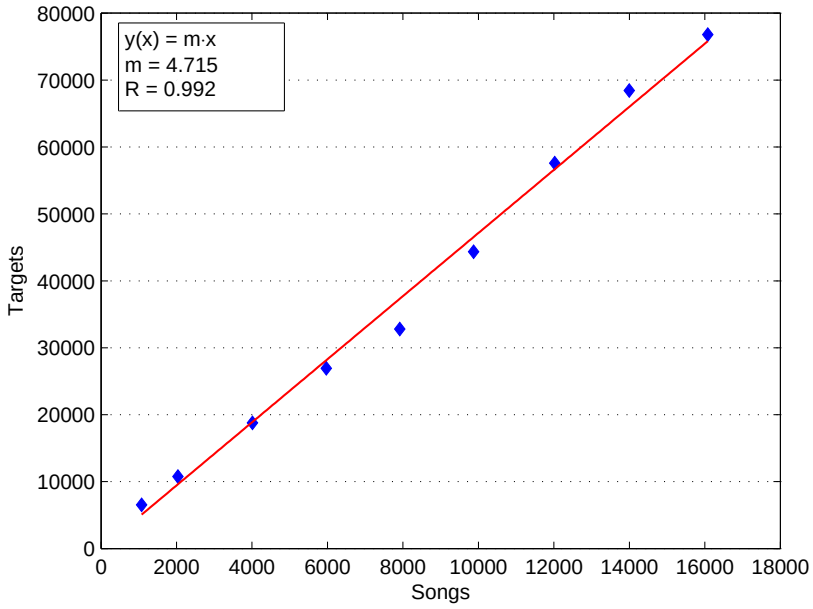


Figure 4.3: Targets vs Songs

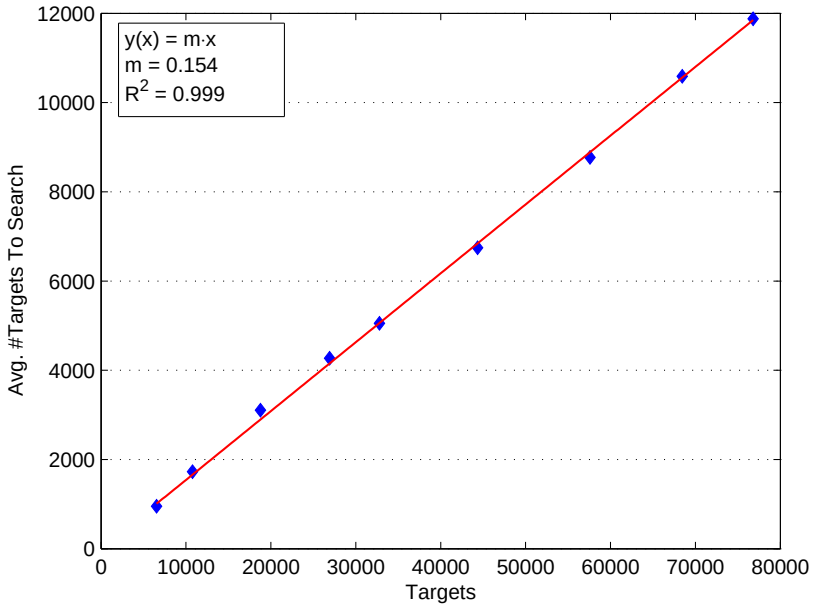


Figure 4.4: Avg. #Targets to Search vs #Targets in DB

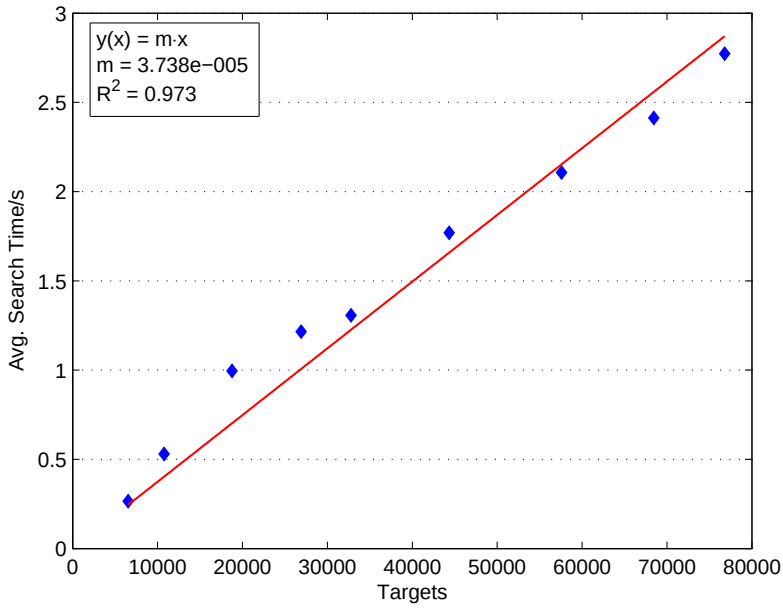


Figure 4.5: Avg. Search Time vs #Targets in DB

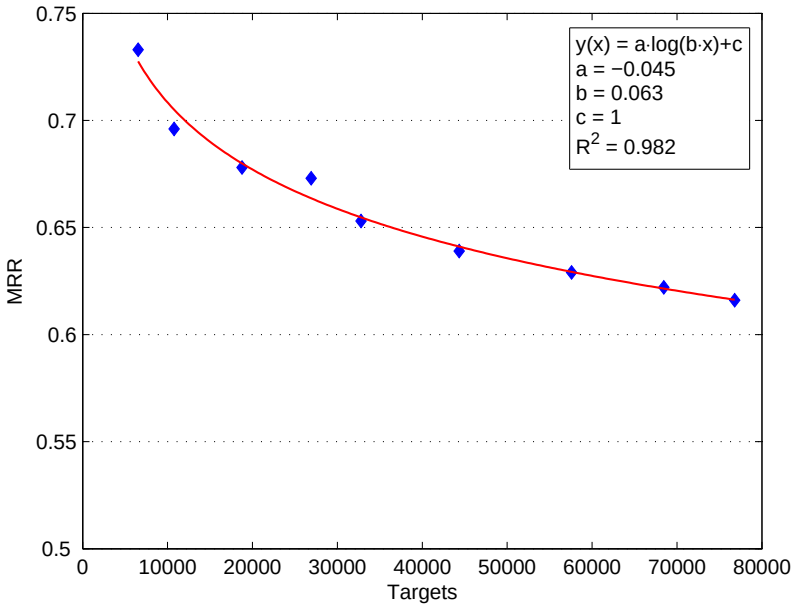


Figure 4.6: MRR vs #Targets in DB

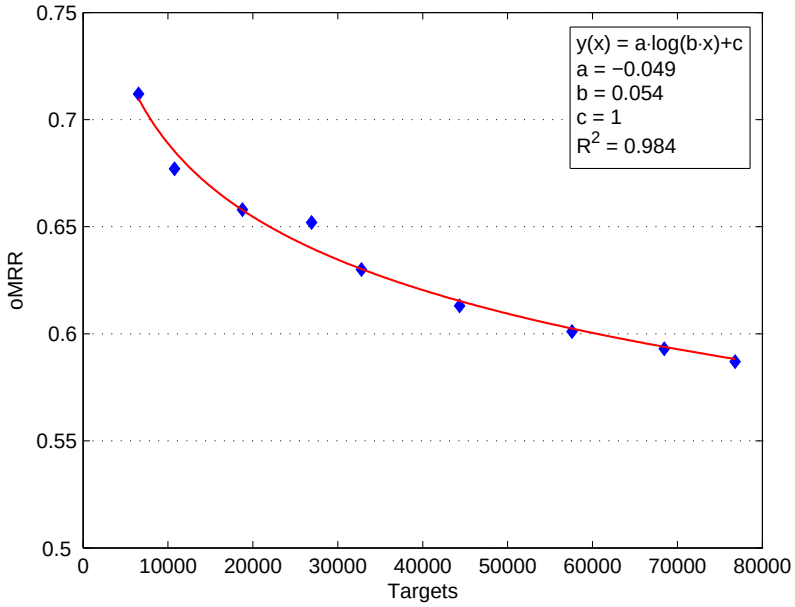


Figure 4.7: oMRR vs #Targets in DB

We see that every song produces on average 4.7 targets. The average search time even for the largest database is well under 5 seconds which meets the success criteria. Next we note that the Seed Search provides a reduction in the number of targets to search which is linearly proportional to the size of the database (reduces to 15% of original size). Though it is not as good as the sub-linear indexing methods available for metric search spaces, as a solution for the non-metric search space of the project it is very successful, even for the largest database. The final two graphs are very encouraging – they show that both the MRR and oMRR only decrease as $O(log(n))$ as we increase the size n of the database, which indicates search performance will remain satisfactory (above 0.5) even for very large databases.

Chapter 5

Conclusion

5.1 Goals Achieved

Overall I am very satisfied with the outcome of the project. I have successfully managed to implement a working system which satisfies all the success criteria:

- The MRR (and oMRR) value is well above 0.5 for the test corpus.
- The system can search a database of over 10,000 songs in under 5 seconds.
- Usability tests show the user interface is clear and intuitive.

I have also implemented the extensions of a client-server architecture, support for a real MIDI controller, full MIDI file comparison and additional GUI features. In addition, I have come up with my own solution to the indexing problem, and it has proved to be highly satisfactory for the purposes of the project.

5.2 Unresolved Issues

Although the Seed Search proved to be a good practical solution, it is still not as good as the sub-linear methods which exist for indexing metric search spaces. This means that in its current state it would not be good enough for very large databases containing millions of songs. I believe however that through further research into seed statistics for musical data and melody extraction techniques my solution could be applicable to very large databases as well.

The slight delay between sending a MIDI message and sounding a note means that in most cases including rhythm in the search produces worse results than when considering only the pitch. Unfortunately this is something which depends on the internal implementation of the Java libraries and on the quality of the

soundcard used, but as these are two things which are constantly being improved, it is definitely something that can be resolved.

5.3 Future Extensions

5.3.1 Melodic Extractor

My project only disregards a small number of the tracks in a MIDI file. Though this has the advantage of allowing the user to search for any part of a song, it increases the number of targets produced from every song thus increasing the search time and memory requirements. A possible extension would be to develop a melodic extractor [9] which identifies the melodic themes which are most perceptually salient in a song and throws away the rest of the data in the MIDI file.

5.3.2 Elaborate Seed Filtering

As mentioned in 5.2, the Seed Search could be improved by devising a more elaborate seed filtering mechanism which is based on the statistical properties of musical data. The seeds themselves could also be enhanced, similar to the spaced seeds used by advanced Bioinformatics algorithms such as Pattern Hunter [7].

5.3.3 Audio input

The project could be extended to support audio input through a microphone making it a combined QBE and QBH system providing the most flexibility in query specification, and making it useful for a very wide range of users.

5.4 Final Comments

I enjoyed working on the project very much, and through my work I have developed a keen interest in music information retrieval. I learned a lot from researching the project and gained new programming skills. I have already found myself using the project to identify songs, and believe it has real potential for practical use. I hope to continue developing the ideas presented in this dissertation and make my own contribution to the development of content based information retrieval systems.

Bibliography

[1] R. Baeza-Yates and B. Riberto-Neto. *Modern Information Retrieval*. Addison Wesley and ACM Press, 1999.

[2] T. Bozkaya and M. Ozsoyoglu. Indexing Large Metric Spaces for Similarity Search Queries. *ACM Transactions on Database Systems*, 24(3):361–404, September 1999.

[3] R. B. Dannenberg, W. P. Birmingham, B. Pardo, N. Hu, C. Meek, and G. Tzanetakis. A Comparative Evaluation of Search Techniques for Query-by-Humming Using the MUSART Testbed. *Journal of the American Society for Information Science and Technology*, February 1, 2007.

[4] R.B. Dannenberg and N. Hu. Understanding Search Performance in Query by Humming Systems. Audiovisual Institute, Universitat Pompeu Fabra Barcelona, Spain, 2004. 5th International Conference on Music Information Retrieval.

[5] W.J. Ewens and G. Grant. *Statistical Methods in Bioinformatics: An Introduction (Statistics for Biology and Health)*. Springer, 2 edition, 2005.

[6] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598), 1983.

[7] M. Li, B. Ma, D. Kisman, and J. Tromp. PatternHunter II: Highly Sensitive and Fast Homology Search. *Journal of Bioinformatics and Computational Biology*, 2004.

[8] D. Mazzoni and R. Dannenberg. Melody Matching Directly From Audio. 2nd Annual International Symposium on Music Information Retrieval, Bloomington, Indiana, USA, 2001.

[9] C. Meek and W.P. Birmingham. Automatic Thematic Extractor. *Journal of Intelligent Information Systems*, 2003.

[10] G. Navarro and M. Raffinot. *Flexible Pattern Matching in Strings*. Cambridge University Press, Cambridge, UK, 2002.

[11] B. Pardo and W.P. Birmingham. Encoding Timing Information for Musical Query Matching. Paris, France, 2002. ISMIR Conference Proceedings.

[12] B. Pardo, J. Shifrin, and W. Birmingham. Name That Tune: A Pilot Study in Finding a Melody From a Sung Query. *Journal of the American Society for Information Science and Technology*, 2004.

[13] Hubbard T.L and D.L Datteri. Recognizing the Component Tones of a Major Chord. *The American Journal of Psychology*, 114(4):569–589, 2001.

[14] A. Uitdenbogred and J. Zobel. Melodic Matching Techniques for Large Music Databases. ACM Multimedia, 1999.

Appendix A

Search Parameters

```
//********** PITCH FIELDS **********

//***** SKIP QUERY VALUES *****
// The default cost of skipping a pitch contour in the query
public int SKIP_QUERY_PITCH_CONTOUR_COST;
// The default cost of skipping a pitch interval in the query
public int SKIP_QUERY_PITCH_INTERVAL_COST;
// The cost of skipping a prima pitch interval in the query
public int SKIP_QUERY_PITCH_PRIMA_COST;
// The cost of skipping an octave pitch interval in the query
public int SKIP_QUERY_PITCH_OCTAVE_COST;

//***** SKIP TARGET VLAUES *****
// The default cost of skipping a pitch intervals in the target
public int SKIP_TARGET_PITCH_INTERVAL_COST;
// The detault cost of skipping a prima pitch interval in the target
public int SKIP_TARGET_PITCH_PRIMA_COST;
// The cost of skipping an octave pitch interval in the target
public int SKIP_TARGET_PITCH_OCTAVE_COST;

//***** REPLACE VALUES *****
// The minimum pitch interval included in the BIG_DOWN contour
public int BIG_DOWN_MIN;
// The maximum pitch interval included in the BIG_DOWN contour
public int BIG_DOWN_MAX;
// The minimum pitch interval included in the LITTLE_DOWN contour
public int LITTLE_DOWN_MIN;
// The maximum pitch interval included in the LITTLE_DOWN contour
public int LITTLE_DOWN_MAX;
// The minimum pitch interval included in the SAME contour
public int SAME_MIN;
// The maximum pitch interval included in the SAME contour
public int SAME_MAX;
// The minimum pitch interval included in the LITTLE_UP contour
public int LITTLE_UP_MIN;
// The maximum pitch interval included in the LITTLE_UP contour
public int LITTLE_UP_MAX;
// The minimum pitch interval included in the BIG_UP contour
public int BIG_UP_MIN;
// The maximum pitch interval included in the BIG_UP contour
public int BIG_UP_MAX;
```

```
// The cost of matching a query pitch contour to a target pitch interval
public int REPLACE_PITCH_CONTOUR_COST;
// The cost of replacing a query pitch
public int REPLACE_PITCH_COST;
// The cost of matching a query pitch to a target pitch, but wrong octave
public int REPLACE_PITCH_OCTAVE_COST;

//********** RHYTHM FIELDS **********

//***** SKIP QUERY VALUES *****
// The cost of skipping a rhythm contour
public int SKIP_QUERY_RHYTHM_CONTOUR_COST;
// The cost of skipping a rhythm ratio
public int SKIP_QUERY_RHYTHM_RATIO_COST;

//***** SKIP TARGET VLAUES *****
// The cost of skipping a rhythm ratio
public int SKIP_TARGET_RHYTHM_RATIO_COST;

//***** REPLACE VALUES *****
// The cost of replacnig a matching contour
public int REPLACE_RHYTHM_CONTOUR_COST;
// The cost of replacing a query rhythm
public int REPLACE_RHYTHM_COST;

//********** COMBINING FIELDS **********
// The factor by which to multiply the pitch cost
public int PITCH_FACTOR;
// The factor by which to multiply the rhythm cost
public int RHYTHM_FACTOR;

//********* SEARCH FIELDS ***********
// The seed size
public int SEED_SIZE = 4;
// The highest cost for any pitch operation
public int MAX_PITCH_COST;
// The highest cost for any rhythm operation
public int MAX_RHYTHM_COST;
```

Appendix B

Java Code for *replacePitch()*

```
private int replacePitch(Query query, Target target, int i, int j)
{
    int pitchCost = 0;

    // Calculate the boundaries for the query contour
    char queryContour = query.pitchContour[i];
    int intervalMax = 0;
    int intervalMin = 0;

    switch((int)queryContour)
    {
        case (int)'D': intervalMin = sp.BIG_DOWN_MIN;
                       intervalMax = sp.BIG_DOWN_MAX; break;
        case (int)'d': intervalMin = sp.LITTLE_DOWN_MIN;
                       intervalMax = sp.LITTLE_DOWN_MAX; break;
        case (int)'s': intervalMin = sp.SAME_MIN;
                       intervalMax = sp.SAME_MAX; break;
        case (int)'u': intervalMin = sp.LITTLE_UP_MIN;
                       intervalMax = sp.LITTLE_UP_MAX; break;
        case (int)'U': intervalMin = sp.BIG_UP_MIN;
                       intervalMax = sp.BIG_UP_MAX; break;
    }

    // Compare to the target interval
    int targetInterval = target.pitch[j];

    // If the query only contains a pitch contour, look for a contour match
    if (query.pitch[i] == Byte.MIN_VALUE)
```

```
    {
        if (targetInterval >= intervalMin && targetInterval<=intervalMax)
            { pitchCost = sp.REPLACE_PITCH_CONTOUR_COST; }
        else
            { pitchCost = sp.REPLACE_PITCH_COST; }
    }
    // Otherwise, look for a pitch interval match
    else
    {
        byte queryPitch = query.pitch[i];
        byte targetPitch = target.pitch[j];

        // Exact pitch match
        if (queryPitch == targetPitch)
            { pitchCost = 0; }
        // Out by n octaves
        else if ((queryPitch % 12) == (targetPitch % 12))
            { pitchCost = sp.REPLACE_PITCH_OCTAVE_COST; }
        // Contour matches
        else if (targetInterval >= intervalMin && targetInterval<=intervalMax)
            { pitchCost = sp.REPLACE_PITCH_CONTOUR_COST; }
        // Total mismatch
        else
            { pitchCost = sp.REPLACE_PITCH_COST; }
    }
    return pitchCost;
}
```

Appendix C

Text Search Formats

The text search supports the following formats:

- **Absolute** – A pitch value is represented by specifying the note letter, whether the note is sharp of flat and how many octaves above or below middle C it should be (e.g. `A#2`). Rhythm is represented by the characters `w,h,q,e,s` corresponding to the durations *whole*, *half*, *quarter*, *eighth* and *sixteenth* (*quarter* = *crochet*). The user can add a `3` or a `.` after the character to make it a triplet or a dotted note respectively.

- **Relative** – Pitch is represented as intervals (e.g. `2 -3`), and rhythm as ratios (e.g. `1 0.3`).

- **Contour** – Pitch contour is represented by `W,w,s,x,X` corresponding to *big up*, *little up*, *same*, *little down* and *big down*. Rhythm contour is represented by `<,|,>` corresponding to *shorter*, *equal* and *longer*.

The user can choose one of the formats or use a combination of all three in the same query. Examples are provided in the table overleaf.

Query Format	Query	
Absolute	pitch:	A A B A D1 C#1 A A B A E1 D1
	rhythm:	e. s q q q h e. s q q q h
Relative	pitch:	0 2 -2 5 -1 -4 0 2 -2 7 -2
	rhythm:	0.333 4 1 1 2 0.375 0.3 4 1 1 2
Contour	pitch:	s w x W x x s w x W x
	rhythm:	< > | | > < < > | | >
Combined	pitch:	A A 2 -2 W -1 C#1 A A B x W x
	rhythm:	e. s 4 1 | 2 h e. s q | | >

Table C.1: Different query formats for "Happy Birthday"

Appendix D

Task List for Usability Test

MuSearch Experiment

You will now take part in a quick experiment. In front of you is MuSearch, a search engine for music. In the same way that Google lets you search for web pages by entering the text those pages might contain, MuSearch lets you search for a song by entering a musical phrase the song might contain. You have a mouse and keyboard, and you may use both as you see fit.

You will now be presented with a short series of tasks to perform. Once you have completed the tasks you will be asked to answer a short questionnaire.

Note: I can not help you during the experiment unless the task explicitly mentions it. The website contains instructions which you should refer to if uncertain about how to complete a task. If you are stuck on a task, or if you have completed it, let me know before going on to the next task.

It would be helpful if whilst performing the tasks you explain your actions and why you chose to take them.

Tasks

1. Play the beginning of the song "Twinkle Twinkle Little Star" (this is not a test of your memory, if you do not remember the song ask me and I'll play it to you).

2. If you used the mouse – can you use something else to play the song?

3. Search for the song "Twinkle Twinkle Little Star"

4. Listen to the results
5. Did your search take rhythm into account? If not, can you do so?
6. Search for the Beatles song "Yesterday" (again, ask me if you do not remember how it goes)
7. Save your query
8. Restore your query and search again
9. Delete the last note from your query by editing it in text mode
10. Search for the text query you have produced, including pitch and rhythm
11. If you have not already: search for "Twinkle Twinkle" using the text search
12. If you only used pitch, include rhythm in your search as well
13. If you did not use contour in your search, try it now

Appendix E

Usability Test Questionnaire

Details

Please fill in your details below. They are guaranteed to be kept confidential without your explicit permission to use them for written work (my dissertation).

Please tick the box at the end of a line if you agree for this information to be included in the dissertation:

Name: ..... ☐
Occupation: ..... ☐
Musical instruments played (please include grade if applicable): ..... ☐
Academic music qualifications: ..... ☐
On a scale of 1 (very unconfident) to 5 (very confident), how confident are you playing the piano: ..... ☐
On a scale of 1 (very unconfident) to 5 (very confident), how confident are you using a computer: ..... ☐
On a scale of 1 to 5, how familiar are you with the Beatles: ..... ☐

Questions

1. On a scale of 1 to 6 (where 1 is very uncomfortable, and 6 is very comfortable), how comfortable was it to play a song on the on-screen keyboard?

2. Whats uncomfortable about it?

3. Did you manage to play on the on-screen keyboard using the computer keyboard instead of the mouse? If yes, is it more/less comfortable?

4. Very briefly, would you change anything in the keyboard search?

5. Are the instructions for using the text search clear? If not, which part was most unclear?

6. Would you ever use the text search in preference to the keyboard search?

7. In the text search, which of the three input methods would you prefer for entering pitch? And rhythm? Please state why.

8. Very briefly, would you change anything in the text search?

9. Did you get any error messages during your search? If yes, were they useful?

10. On the scale: 1- much harder, 2- harder 3- slightly harder, 4- about the same, 5- slightly easier 6- easier 7- much easier, how would you compare using this search to a search facility that would let you sing into a microphone instead?

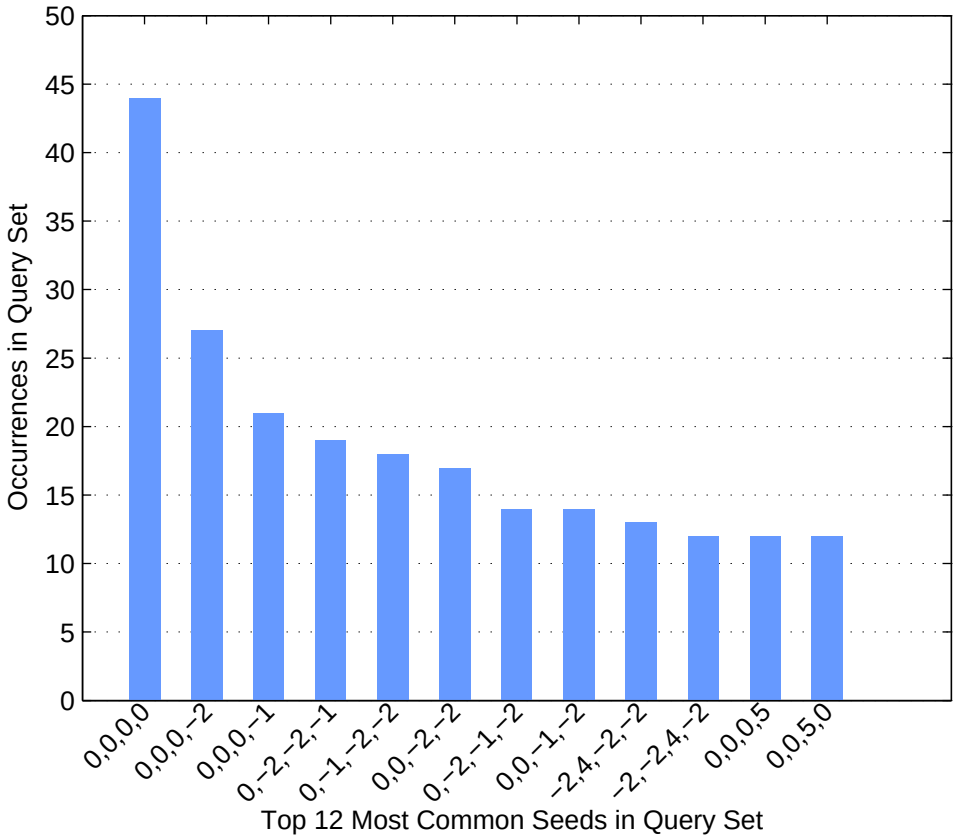
11. Briefly comment on how MuSearch and the system described in question 10 would otherwise compare:

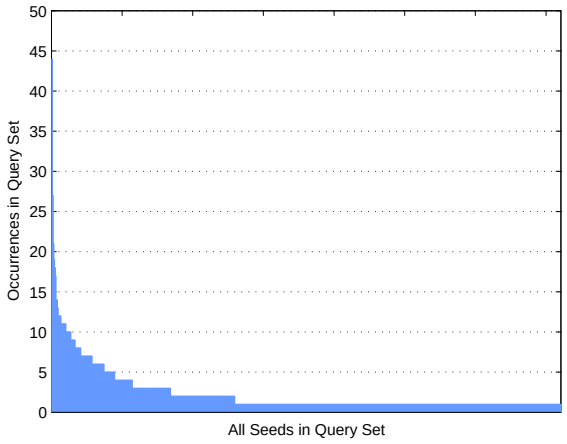
12. Any final comments:

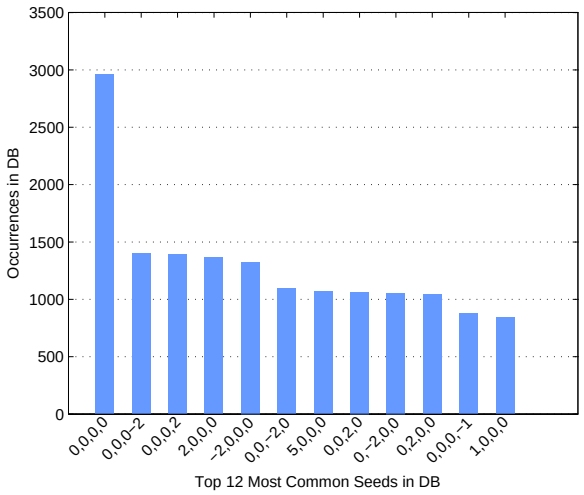
Thank You!

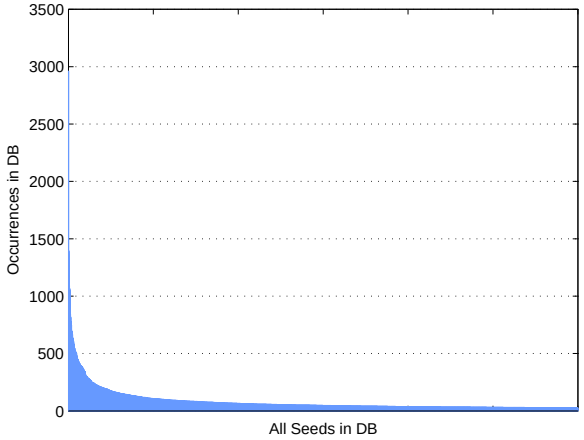
Appendix F

Seed Distributions




Occurrences in Query Set
All Seeds in Query Set


Occurrences in DB
0,0,0,0
0,0,0,−2
0,0,0,2
2,0,0,0
−2,0,0,0
0,0,−2,0
5,0,0,0
0,0,2,0
0,−2,0,0
0,2,0,0
0,0,0,−1
1,0,0,0
Top 12 Most Common Seeds in DB


Occurrences in DB
All Seeds in DB

Appendix G

Project Proposal

Computer Science Part II Project Proposal

Query By Humming Search Engine

Originator: Justin J. Salamon

19th October 2006

Special Resources Required

Permission to carry out tests on human subjects
Filespace on my PWF account
Test data – collection of (MIDI) files
Server – my own PC or SRCF machine
My own PC (2000MHz AMD Athlon64, 512Mb RAM, 80Gb Disk, Windows XP Pro/ Linux Fedora Core 5 dual boot).

Project Supervisor: Dr D. J. Greaves

Project Supervisor: Martin Rohrmeier

Director of Studies: Prof L. C. Paulson

Project Overseers: Dr A. F. Blackwell & Dr I. J. Wassel

Introduction

It is often the case that a user wishes to find information about a musical piece, when they know nothing about it other than remembering part of the music, such as a verse or chorus. This problem is tackled by systems which use a Query By Humming (QBH) approach – the user provides a short input to the system (for example by humming, whistling, playing on an on-screen keyboard etc.), and the system then tries to find a match in a collection of known pieces.

QBH systems do exist, however it is very much an ongoing topic of research, and there is room for improvement. The aim of the project is to produce a QBH system, and evaluate its performance and usability.

Work that has to be done

The project breaks down into the following main sections:

1. Research into current QBH techniques. This will lead to the choice of a specific technique to implement, and the consideration of possible alterations/improvements. The key choices that need to be made during this phase are choice of query format and input method, selection of search algorithm, and a way of managing the data. A programming language and development environment must also be selected, a possible candidate being Java, using the Eclipse IDE.

2. Implementation. The key component of the system is the search unit, and it will require a significant amount of research and programming effort. The query format should be some form of symbolic music representation, and the input method is likely to initially be a simple text query of that format. This should form the basis for a later GUI based input method (e.g. an on-screen keyboard). The searchable collection of music needs to be managed, and the research phase could potentially lead to the design and implementation of a database. Acquiring the actual test data will also require thought and effort, and could potentially take a considerable amount of time (see resources section below). Once the search unit is complete, a user interface for performing the search more easily will be created. It should be kept simple and intuitive, requiring careful thought and design.

3. Evaluation of system. From a performance aspect, the project needs to be evaluated as an information retrieval system. This will be done using information retrieval evaluation methodologies.

The project must also address the issues of user interface and usability. HCI techniques will be used to evaluate the system, including comparison with existing systems, and evaluation of the project on it's own.

4. **Extension:** User input method – support use of a MIDI device (such as a MIDI keyboard) to create the search query.

5. **Extension:** User input method – could be extended further by supporting audio input (such as the user singing into a microphone). This is in itself a very hard task, and one which is unlikely to be achievable in a short amount of time.

6. **Extension:** Client-Server architecture – the program could be extended to provide an online version accessible from a web browser, which would receive the user query and display results. It would communicate with a server containing the searchable corpus, and performing the actual search.

Starting Point

The idea for the project is my own, and does not rely on some existing work base. In other words – the project will be developed from the ground up. The topic of this project is an active area of research, and so I will also have to read the existing literature to get acquainted with the current techniques and approaches to QBH searching, before I can begin work on the project. This also means that though there are research systems in existence, the problem of QBH searching is far from solved, and problems to which I will have to find my own solution are likely to present themselves.

I have a fair amount of musical experience, including playing, composition and arrangement, but I have not done any work on music information retrieval before.

Resources

- I will be working on my own machine. Depending on the language chosen for the project (during the initial research phase) I will use a relevant IDE (such as Eclipse for Java), and possibly some freely available user interface design and creation tools.

- As part of my backup plan, the project will be synced to a cvs/svn repository on my PWF filespace. The project will also be backed up periodically on CDs.
- If the extension of client-server architecture is implemented, I will need a server machine. This can be my personal machine, or the SRCF machine, on which I have an account.
- The project will be evaluated by people who are the intended end users of the system.
- I will need to compile a corpus of music to use as test data, and this will be in MIDI format. There are currently no reliable algorithms to find the perceptually salient melody from a MIDI file – so, the project requires a well-prepared (or hand-prepared) corpus or a MIDI database consisting of one particular style which allows for sufficient assumptions to automatically extract melodic cues. There are collections of MIDI files freely available online, and so if there are collections which are deemed appropriate for the project they will be used, but I will potentially have to prepare a test corpus myself.

Success criteria

The core of this project is the design and implementation of a successful search algorithm. As a result, the key features of the project, which should be completed successfully are:

- Simple and efficient user input – the symbolic music query should be sufficient to perform all relevant queries, and sufficiently robust to overcome basic user inaccuracies such as missing or erroneous notes, key modulation and rhythmic mistakes. It should also be designed so that additional user input methods (such as on-screen keyboard) can be efficiently and successfully translated into a symbolic query.
- Search unit: should successfully return "relevant" results for a given query. The problem at hand is still a topic of research, as are the existing QBH systems. This means that it is not possible to provide numeric values for desired evaluation metrics such as precision and recall before the research phase of the project. Nonetheless, it is clear that these values should indicate that the system is sufficiently good to provide a service users would

consider useful. Note also that corpus size is likely to be limited due to the required specifications of the test data, but it should be large enough to provide a useful account of performance.

- Basic single machine architecture and UI: the end result should have a simple and intuitive user interface, and display the search results clearly. Note that as the input method is likely to involve construction of a symbolic musical query, by playing or by some GUI based method (but probably not by singing), the system is not aimed at users with no musical background at all, and basic playing capabilities are expected from potential users.

Timetable

Michaelmas term

Package 1 (Research & Design): 21st Oct – 3rd Nov

- Research into currently applied QBH techniques
- Decide on user input method and query format, search algorithm and data management
- Decide on programming language and environment, architecture

Package 2 (Research & Design): 4th – 17th Nov

- Continue research
- High level design of architecture and unit interfaces
- High level design of user interfaces

Package 3 (Implementation): 18th Nov – 1st Dec

- Compile music corpus and develop database
- Develop input unit

Lent term

Package 4 (Implementation): 16th – 29th Jan

- Develop search algorithm and unit

Package 5 (Implementation and Progress Report): 30th Jan – 12th Feb

- Continue work on search unit
- Create basic user interface to obtain system in working state
- Run some initial usability tests
- Write progress report

Package 6 (Implementation): 13th – 26th Feb

- Make modifications in light of test results

- Complete user interface
- Implement possible extensions
- Leave time to make up for possible underestimates of previous packages

Package 7 (Implementation & Evaluation): 27th Feb – 12th Mar

- Run performance tests
- Run user evaluation tests
- Finalise code

Package 8 (Dissertation): 13th Mar – Easter vacation

- Plan and start writing dissertation

Easter term

Package 9 (Dissertation): 24th Apr – 7th May

- Write dissertation

Package 10 (Dissertation): 8th – 17th May

- Finish dissertation