Biko Agozino

Inferring sequence specification from Drum Rhythms

Computer Science Tripos - Part II

St John's College

May 12, 2016

Proforma

Name:	**Biko Agozino**
College:	**St John's College**
Project Title:	
Examination:	**Computer Science Tripos - Part II, June 2016**
Word Count:	**TODO**
Project Originator:	Dr A. Blackwell & Dr S. Aaron
Supervisor:	Mr I. Herman

Original Aims of the Project

To build a system capable of inferring the intended sequence of beats on a drum kit from those captured. The system then aims at matching the inferred sequence against popular rock songs, originally performed before the year 2002, regardless of the variations in individual stroke pressure and time and the performance as a whole. The project aims to detail the possibility of using distance metrics on short drum performances as a basis for imitation-based querying of drum notation.

Work Completed

The system infers and extracts a candidate sample pattern from MIDI performance data by analysing for repeated sequences in a repeated bar performance. This sample pattern is then compared with the database and the system ranks all the patterns in the database. Multiple distance metrics have been implemented to assess the advantages and disadvantages of each metric within this domain.

A harvester has been built to parse informal ASCII Drum Tablature. The parser has been successful in parsing over 12,000 individual patterns of the notation into a format appropriate for database queries.

Special Difficulties

None

Declaration

I, Biko Agozino of St John's 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

Contents

Chapter 1

Introduction

This dissertation describes the implementation and evaluation of an query-by-example method for searching collaborative databases of drum rhythms, transcribed by users, found in pre-2002 contemporary music.

1.1 Motivation

The relationship between computers and music has been apparent since the 1950s when Trevor Pearcey and Maston Beard pioneered what was likely the first computer capable of performing music, CSIRAC[1], which was able to broadcast music programmed on punched-paper data tape in a similar notation to standard musical notation[14]. At the time it took specialised, multimillion-dollar, computers hours or even days to generate a few minutes of simple tunes, according to Mathews in his early (1963) discussion on the possibilities of using computers as a musical instrument[6].

Since the early days of Computer Music research, computers have become both more accessible and more powerful. This has allowed computers to be more frequently used in the composition of music by both amateur and professional musicians.This was coupled with the development of the MIDI protocol[2] which connected samplers and sythesisers, and provided a useful framework for comunicating between two electrical musical device. Hobbyists now had the opportunity to have reletively cheap musical studios in their homes to express their creativity which is the environment that many of the popular genres we hear today were spawned from, as a result of this musical revolution.

More recently a significant amount of research has been done into QBE[3] systems as they pertain to music. Most notably Shazam[1], which is a commercial service that aims at

[1]Council for Scientific and Industrial Research Automatic Computer
[2]Musical Instrument Digital Interface: Developed in the early 1980s[7]
[3]Query-by-Example

music recognition by using audio samples, that may be distorted, in order to query a database of over 3 million tracks.

Due to this relation between computers and music, along with the commercial use and success of many musical platforms, it is clear that further research into computer music is a worthwhile pursuit. Further to this, the prevalence of using technology to produce music has shown that continued development of platforms that make it easier for artists to make music is important.

In contemporary musical ensembles the drummer usually serves the vital role of providing the tempo, pace, and rhythm of the performance. Due to this the drummer can often define the entire song as all other musicians typically use the drummer as the context in which to frame their timing. Therefore, many genres, can be defined by their use of drums.

When composing or practising a new song it is useful, from a drummers perspective, to know how similar rhythms were used in the wider context of the song. This has been the primary motivation behind developing a QBE system that relies only on the drum rhythms present in a song.

The main use case that has driven design is as follows. A drummer has a drum rhythm in mind, perhaps they have heard it in a song in the past and want to either see where else it has been used, or perhaps they have designed it themselves and want to see where it, or similar rhythms, have been performed. They play the rhythm on a drumkit as they would usually, this performance is recorded and then used to query the database and returns all of the similar rhythms.

My goal in developing the system was to build a natural drum rhythm recognition system that:

- Returns both the inferred rhythm and all rhythms that are similar found in the database;
- Does not require an extensive digital user interface;
- Be trivially extended to work in real time

1.2 Challenges

With any information retrieval system it is important to consider the database available. At the start of this project there was no easily computer-readable database of drum rhythms. Therefore a significant proportion of the project was spent on how to best compile this resource.

Furthermore, as with any imitation-based QBE system it can be difficult to extract relevant features when the user is unable to provide a good example. In this case, each drummer

has their own nuances to their performance that manifest in the form of a slower/faster tempo or softer/harder beats. This can make it difficult for users when attempting to reproduce the performance, as the intended patterns have been subject to their own personal transformations.

1.3 Summary of related work

1.3.1 Music information retrieval

Shazam[1], provides musical information retrieval by audio-fingerprinting[4] a database of music. Queries are then provided by example which then get fingerprinted and matched against the database.

Many applications(Google Music, Spotify, iTunes) have now implemented a form of audio-fringerprinting to provide users with suggestions. However, it is not so clear whether these techniques would work as well without the large volume of user statistics available to these companies.

Pandora has developed a solution that does not rely heavily on prior user date; the so called Music Genome Project[32] is a collection of audio-fingerprints that were manually allocated by music theorists. It took 30 experts over 5 years to manually assess a large enough database for it to be of any use though so it is clearly a challenging task.

1.3.2 Machine learning approaches to drum inference

There have been attempts at using machine learning techniques to train autonomous drummer. Tidermann-Demiris[26] claim to have developed a system that is capable of mimicking the style of performance of a human trainer. Fox et al.[25] use Variational Bayesian techniques for on-line beat tracking[5] and pattern recogniton to provide automated accomponment. Cicconet et al.[27] produced a Human-Robot percussion ensemble in which the robot learns to drum with the human based on visual stimulae.

While all of these approaches are interesting and some are quite succesful, none provide the level of look-up needed to power a QBE system alone.

[4]The process of extracting unique feature vectors from digital audio that varies from between systems but a general overview was described by Downie[28]

[5]see 2.2.3

1.3.3 Sequence matching

A suffix tree represents all of the suffixes in a trie-like structure. They are often used for general sequence matching and common sub-sequence searching, which often plays a central role in bioinformatics (Delcher et al. [29]) and information retrieval as a whole (Baeza-Yates and Ribeiro-Neto [30]. Suffix Tree's provide us with an efficient method for finding repeated structures[16]. For a more detailed description of suffix trees refer to 3.3.1.

There are many algorithms for building suffix trees (Ukkonen[19], Weiner[15]). For a more detailed comparison of them please refer to 2.3.1.

Other algorithms have been suggested for performing exact matching on 1-dimensional strings (Knuth-Morris-Pratt[31]) and similarly on 2-dimensional strings (Bird[20]). 2.3.2 Distance metrics (Hamming[17] and Levenshtein[18]). Both techniques are considered and discussed in 2.3.2.

Chapter 2

Preparation

With any project of a non-trivial size it is important to spend a good amount of time carefully planning and organising tasks. In the case of this project it proved particularly useful when it became clear that the implementation of the system had to significantly change, from the one that was originally proposed, when the available resources and preferred system features were taken into account.

This chapter outlines the investigations that were undertaken prior to the projects implementation, along with detailed arguments for design decisions.

2.1 System Goal

From an abstract perspective the goal of this project is to pass a MIDI performance sequence into the system and be returned with an inferred Sequence Specification, described in 3.1.1, and ordered similarity rankings for the Sequence Specifications in the database.

2.1.1 Changes from proposal

Originally, the system had the task of generating an inferred pattern from an input MIDI sequence of drum beats (see 2.4.1 for details on MIDI), using machine learning techniques trained on sequences specification harvested from a database of transcriptions. The aim of this was to assist the performance of novice drummers by repairing errors in timing and false/missed beats. However, following the collation of the database it became clear that the data was too sparsly spread to adequetly train a Hidden Markov Model, or any other machine learning module, and so the project moved into the direction of information retrieval.

2.1.2 Success Criteria

Due to the change in project direction, the success criteria were ammended to Implementation and evaluation of a system that:

1. reads MIDI and is able to translate it into Sequence Specifications;
2. compiles a database of at least 1000 unique Sequence Specifications;
3. infers the intended Sequence Specification from an input MIDI sequence.
4. the intended Sequence Specification falls in the first 25% of results in the database

2.2 Database collection

The success of the project heavily relies upon whether an adequate database can be built. Therefore, it is a task that I decided to tackle first so as to provide the base for the future work. Here I will discuss the possible data sources that were considered along with a comparison and argument for the final choice.

2.2.1 Beat tracking techniques

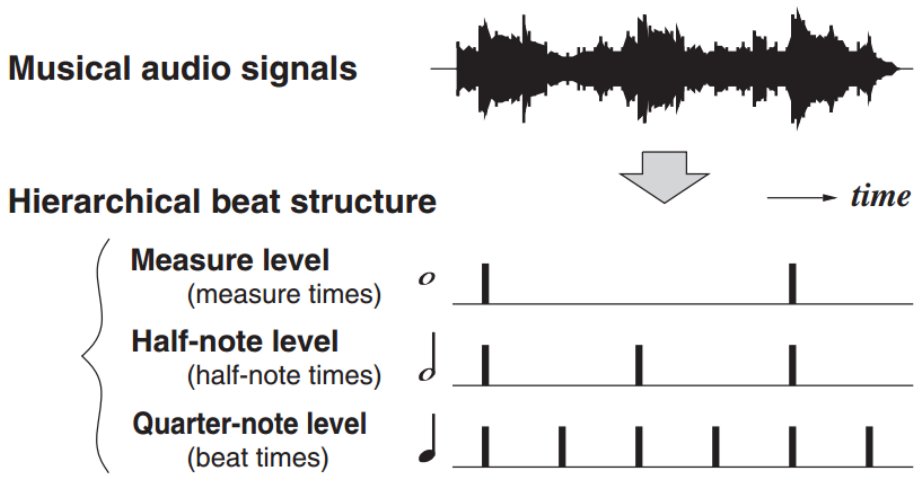

Figure 2.1: Shown here is the aim of beat tracking is to transform musical audio signals into a representation of the beat structure[2]

Beat tracking, as defined by Goto[2], is the process of inferring the hierarchical beat structure from musical audio signals. The idea here, depicted in figure 2.1, is that real-time audio from popular music can be used as a source of data in order to extract features about the song.

The advantage of this is that the amount of data available to extract features from is the number of songs that have been released in an acceptable format. In Goto's case this was compact disks though the approach could be extended to mp3, or any other file encoding. Furthermore, if perfect beat tracking were achievable, the rhythm extracted will be a perfect representation of the underlying rhythm. Additionally this approach may allow further features such as lyrics, melody, tempo, or the Mel-frequency cepstral, which is a measures musical timbre. These additional features would contribute to the database - allowing for a richer searching experience.

Perfect beat tracking, however, is a very challenging field. Recent attempts at solving the problem ([3] [4] [5]) have reported efficient algorithms, but with only an accuracy of around 60%.

2.2.2 Feature extraction from notation

There are two distinct techniques that can be implemented in order to extract features of drum rhythms from rhythm notation that I will outline here. I will discuss each in turn before returning to the comparison between methods in 2.2.3

Optical Music Recognition

Musical notation has been a practice in many cultures since as long as we have record, as Scelta states[10] "Systems of signs and symbols for writing music developed alongside written language as a need to pass along consistent information presented iteself". Much like written language, musical notation has been refined over thousands of years, and what we now think of as 'standard notation' has been firmly established since the 18th century[10]. This standard is of the format of 5 horisontal lines, called a stave, with symbols at varying positions along the staves to represent the pitch and relative time of each note.

Percussion notation, a specific musical notation that pertains to percussion instruments, is much less standardised and only (relatively) recently has there been a push for a standardised practice, Weinberg produced an extensive set of guidlines to the practice[11], an example of this notation can be seen in figure 2.2.

There has been a significant research([12][8]) into Music OCR[1], which involves using computer vision techniques in order to allow computers to read sheet music. Applying these

[1]Music Optical Character Recognition, sometimes referred to as Optical Music Recognition

techniques to Weinberg's percussion notation can enable extraction of feature sets that can be queried.

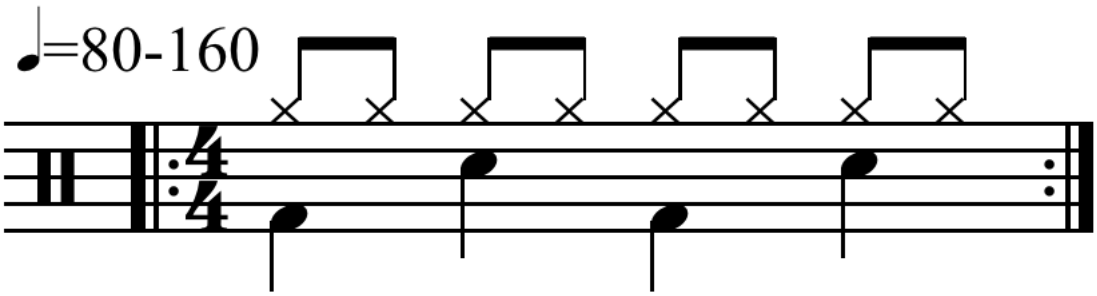

Figure 2.2: Shown here is the standard percussion notation for a popular rhythm often found in rock music

ASCII Drum Tablature Parsing

```
Hi-hat    |x-x-x-x-x-x-x-x-|
Snare drum|----o-------o---|
Bass drum |o-------o-------|
            1 + 2 + 3 + 4 +
```

Figure 2.3: Shown here is a the ASCII drum tablature the same rhythm found in figure 2.2

In drum tablature, each line corresponds to a single component of the drum kit, as shown in figure 2.3. This contrasts with standard notation, where the height of each note refers to the pitch of the note. Enthusiast transcribers have taken to the ASCII character-encoding scheme in order to share their work over collaborative databases[2].

This ASCII format, while intended to be read by people, may prove to be a great candidate for a general tablature parser. In fact, this problem has been attempted to be solved in a similar case of guitar tablature[13], where a parser was written to quite successfully extract

[2]http://drumbum.com/drumtabs

the specifications of a strictly defined guitar notation. However, this solution does not solve it in the general case where the tablature may contain comments to help the human reader. My solution accounts for this as the collaborative database is not perfectly void of errors.

2.2.3 Choice of data source

Both beat tracking and music OCR are very challenging fields, each large enough to justify their independent research. ASCII Drum Tablature parsing is very feasible in the time frame and given the large amount of data available in this format it should be feasible to collate an adequate database.

It is important, however, to keep in mind the disadvantages of selecting this data source:

1. As it is a collaborative database, we can not easily verify the accuracy of the transcriptions
2. As there is no standardised format a parser can only be tailored towards a common practice

Problem 1 would be solved by both of the alternatives discussed, however problem 2 can only be solved by beat tracking as percussion notation is also not standardised. Though beat tracking hasn't been shown to be very accurate, and when the project relies heavily on the timing information in order to match performances to songs it may be better to choose a data source that can directly map the source to the features extracted.

The datasource will therefore be ASCII Drum Tablature using a collaborative database that I found to have thousands of transcribed songs, each containing about 100 patterns, meaning there should is no problem writing a parser that can extract the rhythms of a subset of these songs. The design of the parser is described in 3.2 and it's evaluation is in 4.2.

2.3 Querying

In order to discuss the querying method I will start by outlining methods for inferring the bar from a sequence of beats on the drum. I will then detail the possible ways of using this sequence to query the database.

2.3.1 Suffix trees

Suffix tree's will be used to extract and infer the intended pattern to be returned to the user and query the database.

A suffix tree[15] is a tree in which each path, from root to leaf of the tree, correspond to exactly one suffix of the string used to build it. For example, with the string "BANANA$" (see figure 2.4) there is a path corresponding to the suffix "A$", a path for the suffix "NA$", "ANA$", etc. The reason it is useful is that repeated substrings, e.g. "NA", will share an edge along the tree. This means that it is easy to find repeated structures, and a linear algorithm[**?**] for doing so will be presented in 3.3.4.

Once a suffix tree has been created, the repeated structures can be extracted and the sequence can then be transformed into a similar form to the population of the database in order to perform matching techniques.

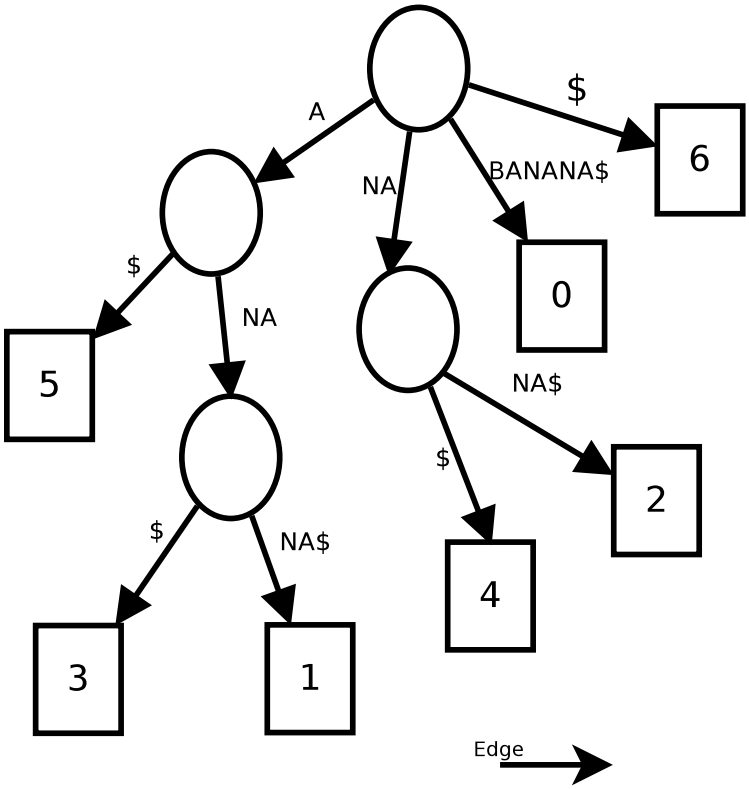

Figure 2.4: Suffix tree of the string "BANANA$"

Exploring algorithms for Suffix tree construction

Weiner's algorithm[15] for building suffix tree's in linear time works by adding one suffix into the tree at a time. For example, if we want to build a tree for BANANA$ we start by adding $ into the tree, and then A$ into the tree, etc. Each of these steps is of constant time and so for a string of length n the suffix tree is built in $O(n)$ time.

An alternative approach, Ukkonen's algorithm[19], is also $O(n)$ in time. Where it differs is that while Weiner's steps consist of iterating over the input string from right to left,

Ukkonen's work in reverse - from left to right. Taking the same example, Ukkonen's algorithm first inserts *prefix* B, then BA, etc.

Weiner's algorithm works well in situations where you need to analyse the tree before it has finished executing, as each stage of the algorithm is a valid suffix tree. Ukkonen's, on the other hand, only guarentees a valid suffix tree when it is sees the terminal symbol, $. However, this is not a real disadvantage as to return a Ukkonen suffix tree early one just adds the terminal symbol prematurely.

More interestingly, as it operates from left to right, Ukkonen's offers the on-line property meaning the suffix tree can be built while the system is live recording.

For this reason I have chosen to implement Ukkonen's algorithm.

2.3.2 Matching techniques

If we assume both the web-based database parser and MIDI sequence inference are able to extract features correctly, and then represent them in our database as two dimensional strings of data, we must now come up with a way of comparing the two of them in order to find the closest matches. Here I will discuss two distinct methods, string metrics and exact matching, and explain my choice towards one of them

String metrics

String metrics measure the reverse similarity, or distance, between two strings. There are many metrics to choose from, such as the Hamming distance[17], and the Levenshtein distance[18].

The Hamming distance of two strings of equal length is a simple pairwise comparison between each element at the same position. Therefore, the Hamming distance is defined as the number positions through the strings that symbols differ. Algorithms for computing the Hamming distance are relatively simple, and cheap in time ($O(n)$ where n is the length of the two strings) and constant in space.

Levenshtein distance between two strings is defined as the minimum number of single-character edits to transform one string into the other. These edits are typically defined as insertions, deletions, and substituitons. Dynamic programming solutions for computing the Levenshtein distance, such as the Wagner-Fischer algorithm[24] that works by populating a table of values, run in time and space of $O(nm)$ for the two strings of length n and m respectively.

The advantage to this technique is that a similarity ranking can be derived where patterns most similar to the inferred pattern are towards the top of the ranking.

Exact Matching

An alternative technique is to perform exact matching on all of the patterns in the database to find a pattern that is the same as the inferred pattern. R.S. Bird presented an algorithm[20], which is an extension of the Knuth-Morris-Pratt algorithm[31], for matching two-dimensional strings.

Choice of matching technique

I chose to use String metrics as the matching technique to perform the database search as it will account for any deficiencies that the pattern inference has. Exact matching, as the name suggests, only returns patterns that are exact copies of the query sequence, and as it is likely the pattern inference will not be exactly correct, and therfore the relevent patterns may not be represented in the search results. Furthermore, all patterns that are close to the query pattern should be returned as they may be of interest to the user, not just the single pattern that is exactly the same.

I will implement both the Hamming and Levenshtein distance calculations to compare whether the increased complexity of the Levenshtein distance is advantageous in this domain.

2.4 Tools Used

Here I will outline the key tools and frameworks I had to familiarise myself in order to implement the project.

2.4.1 Investigating MIDI

MIDI is a protocol that allows for communication between a large majority of electronic musical instruments. The MIDI specification[21], describes MIDI messages as individual packets that are sent over one, or all, of 16 channels. Each message is one byte of data, circumfixed by a start and stop bit, transmitted serially.

MIDI uses the concept of channels in order to group messages of the same origin. Messages can either be "channel messages" - in which the message is broadcast to across a single channel, or "system messages" - in which all the channels are notified of the message.

MIDI also has no requirement to broadcast timing information in absolute terms, though leaves it up to choice of manufacturers. One solution, called MIDI Time Stamping[22], involves marking MIDI events with the time they are played and storing them in a buffer in the MIDI interface ahead of time. This means that the messages timing information will not be disrupted by USB or software latency. However, this solution requires strong

coupling between both the hardware and software of the system and so most MIDI device manufacturers elect to leave the job of timing information to the sequencers

Sequencers are software or hardware devices that can record, edit, and playback music. For this project I require a software sequencer as I need to then manipulate the MIDI recordings. I also require a lightweight solution that will not cause timing delays due to excessive processing. For these reasons, and the extra customisability options stemming from an open-source solution, I have chosen to use the MIDI software sequencer, Midish[3], which is one of the most popular sequencers for Unix-like operating systems.

Investigating Roland HD-1 drum kit

In order to explore the available options for the project I spent some time familiarising myself with the available interface for MIDI input into the system, namely a Roland HD-1 drum kit owned by the Rainbow Group at the computer lab. While I attempted to design the system to make sure it is transferable to any MIDI drum kit, and extendible to any MIDI device for that matter, there may be some slight difference as the implementation was tailored towards this model.

The drum kit does not have any special timing features, like MIDI time stamping, and so I have to rely on the timing information provided by the software sequencer, Midish.

A standard practice with drum kits is to filter out all but the dedicated channel for percussion instruments, channel 10, in order to reduce any noise, and latency, that could occur from messages being sent on other channels. However, looking into the specification, along with some experimentation, I found that while messages were primarily transmitted over channel 10, in the case of polyphonic messages[4] some messages leaked into channel 1. For this reason it is more appropriate to allow all messages on all channels through for recording and filter out the superfluous information at a later stage.

In MIDI systems, the activation and release of a particular note are considered as two separate events - Note On, and Note Off respectively. While this makes sense for some instruments, take an electric keyboard for example, it is less logical to apply this notion to percussion instruments. In fact, most drum machine software ignore note off messages. As such we are only interested in Note On messages to represent timing information associated with the impulse of beats. Both types of messages are followed by two data bytes, which specify key number (in this case these key numbers are mapped to specific drums as outlined in the owners manual[23]), and velocity of impact. As this project deals primarily with the relative timing of patterns the velocity of impact can also be ignored.

I found it logical from here to split the recording into separate note channels, each representing a component of the drum kit, in order to analyse the repeated sequences for each component individually, and then trying to infer the overall pattern from the combination

[3] http://www.midish.com

[4] Messages that occur simultaneously - i.e. more than one drum being hit at the same time

of these patterns. In order to apply the string analytical techniques discussed in 2.3.1 these Note On messages must be converted into some alphabet that encodes their timing information. The logical solution is to use the time delay between each neighbouring pairs of Note On messages.

The components in the drumkit are as follows:

- Bass Drum
- Crash Cymbal
- Floor Tom
- Low Tom
- High Tom
- Hi-Hat
- Ride Cymbal
- Snare Drum

The hi-hat[5] is unique as there are four note numbers, rather than the usual one, that can be transmitted indicating propogation of the component. One for each an open and closed hi-hat being struck directly, one for the foot plate being pressed down to close the hi-hat, and a final one that corresponds to a change in the depth of the foot pedal which serves as a control variable for the sound that the synthesiser produces.

2.4.2 Development Evironment

Java: The majourity of this project is implemented using the programming language Java. I chose Java as I am very familiar with it having completed Part IA and IB of Tripos. Additionally, Java offers an extensive sound library for manipulating MIDI, javax.sound.midi, which will prove useful when inputting the MIDI recordings into the system.

Bash: The scripting language used to develop the user study environment.

Matlab: A statistical language used to produce the analysis of the system.

Git: A version control system that allowed splitting of development between multiple desktops over the course of the year. Allows for checkpointing significant updates and branching in order to implement a process that might introduce bugs into the system.

[5]A component made up of two cymbals that can be propogated direction by hitting it with a drumstick, or indirectly by applying pressure to a foot-pedal that crashes the cymbals together. The foot-pedal can be held down to keep the hi-hat "closed", it is otherwise "open".

2.5 Summary

This chapter discusses the initial research I did into the relevant areas of Computer Science and Computer Music. Then outlines the advantages and disadvantages of each possible method, ending with a conclusion as to why I decided to make certain design decisions in each section. Finally this chapter detailes the tools used for the implementation of the project

Chapter 3

Implementation

3.1 Overview of system

The design of the system revolved around two main modules, the Harvester and the Inference module, that interact only in the database query stage, as shown in figure 3.1.

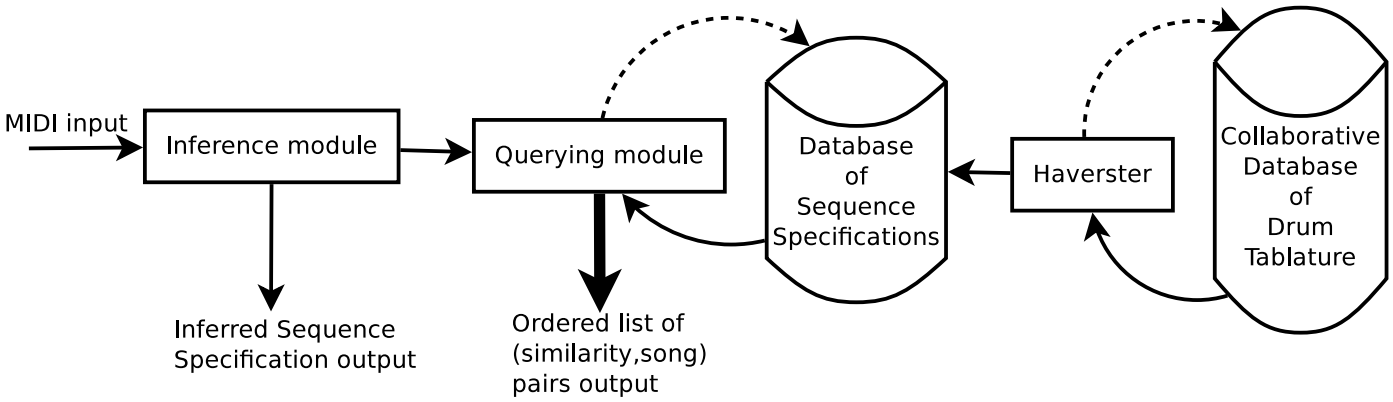

Figure 3.1: Here is a modular view of the system, solid lines represent flow of information, while the dotted lines indicate a look-up request

3.1.1 Specifications

I introduce the concept of sequence specifications. These specifications represent a simplified view of a drum rhythm.

Each specification has a constant 8 rows, each mapping to one of the 8 drum components discussed in 2.4.1. The number of columns is determined by the resolution of the system, for this project I have chosen it to be 16 meaning that we are only looking for patterns, in both the Harverster and Inference Module, that have 16 time divisions.

In each cell of this matrix there are two possible values:

- **1** - Representing a Beat impulse
- **0** - Representing a Rest event (lack of a beat impulse)

3.2 Harvester

As outlined in the 2.2.3 the source for the database is in the format of ASCII drum tablature, which is intended to be human-readable, and so parsing it is not a trivial problem.

The goal of the harvester is to collect a database sufficient enough to act as the search space for the Query-by-Example system.

As shown in figure 3.2 the parser takes an ASCII file containing specifications of the rhythms found in the song and delivers each bar-long rhythm, saved seperately, for use later in the querying stage of the system.

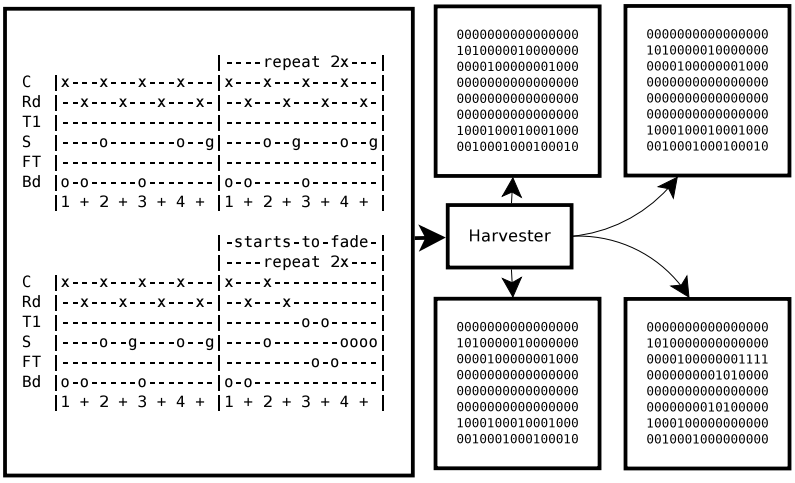

Figure 3.2: Here is an example of how the system will translate and format the drum tablature - creating Sequence Specifications that are saved in the database

The Harvester incorporates a pipeline structure in order to chain the processing elements of the following stages:

1. Lexical Analysis

2. Pre-parser

3. Parser

In this section, each stage will be discussed in detail.

3.2.1 Lexical Analysis

The lexical analysis involves converting a sequence of ASCII characters, which make up the drum tablature being parsed, to a sequence of tokens. These tokens are considered individual units of information that, when viewed in isolation, can convey some entropy about the pattern.

For example, if we take the single line of "HH|x-x-|" which describes a simple bar of 4 quanta where the hi-hat component of the drum kit is struck at the 1st and 3rd quanta, and there are rests in the 2nd and 4th quanta. This string will produce the sequence of Tokens as follows: Instrument HH, TrackDivider |, Beat x, Rest -, Beat x, Rest -, TrackDivider |. The *Instrument* token is used to simplify parsing at this stage and the HH, representing a hi-hat, is matched at a later stage.

In order to lex the drum tablature files without complicated processing I developed a context-free grammar with the following tokens:

- **Instrument -** representing the components of the drumkit for which that line maps to
- **TrackDivider -** representing not only the starts and ends of bars, but also the divisions between them
- **Beat -** representing an impulse on a drum component at that time division
- **Rest -** representing a lack of impulse at that time division
- **NewLine -** representing a new line in the ASCII file

Both TrackDivider and NewLine are used to provide context at a later stage.

Through use of the java.util.regex package, which is available in the standard library, the tokens can be matched to through Regular Expressions. Due to the grammar being context-free, this means that lexing[1] a string occurs on-line with the complexity of $O(n)$.

[1]Meaning providing lexical analysis for

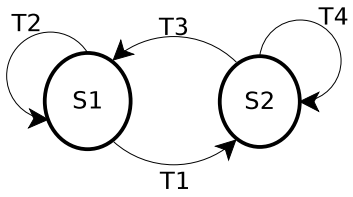

Figure 3.3: Finite State Machine for comment removal, see **Comment removal** for an explanation of the states and transition functions.

3.2.2 Pre-parser

In the pre-parsing stage, the harvester sets up the sequence of tokens for the parser to process. The reason for this conceptual difference is that the pre-parser has no complicated processing tasks and is merely given the job of organising the data into a parse-friendly format.

Comment removal

One important job for the pre-parser is to remove any of the tokens that were incorrectly matched - usually due to their appearance in the comments that are written to inform the reader of any subtlties to the performance.

In order to achieve this context has to be introduced into the system, so that we know what is definitly not a comment. I implemented this through a simple finite state machine shown in figure 3.3. Iterating through the sequence of tokens while in state S1 the pre-parser removes all *TrackDivider*, *Beat*, *Rest*, and solo *Instrument* tokens. The transition function T1 occurs when there is an *Instrument* token followed immediately by a *TrackDivider* token. This transition occurs preemptively, meaning before the pre-parser has the chance to remove the occurance of the *Instrument* token.

While in state S2 the pre-parser does nothing, as it is assumed that the line is now a specification of the beats on a drum component. It transitions on T2 back to the initial state when a *NewLine* token is found.

This solution does not remove all comments but handles most of them so that the parser can find the patterns.

Spliting into sub-sequences

Another job of the pre-parser is to analyse the sequence of tokens, now with most of the comments removed, and split the sequence in to sub-sequences iteratively exactly three times. This is to allow for easy parsing which will be detailed later

In order to describe these splits I must reiterate the structure of the tablature files. Each

file may have multiple "blocks" that may refer to multiple "patterns" which can be be split into individual "lines" with each line pertaining to one drum component, as outlined in figure 3.4.

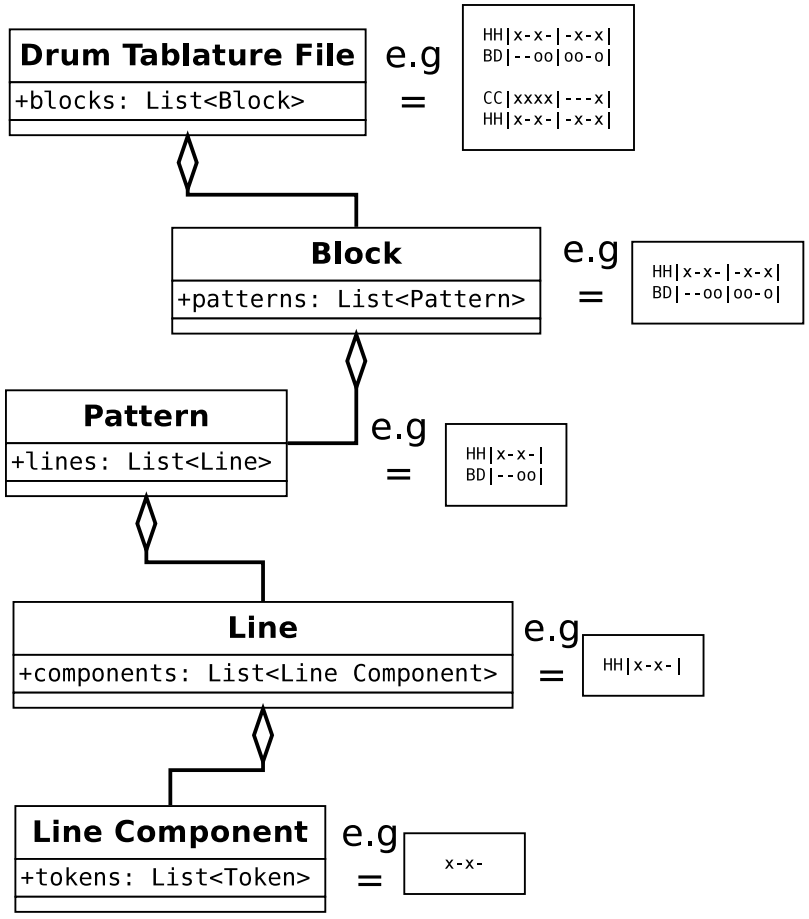

Figure 3.4: Hierarchical structure of a Drum Tablature file

The first split, therefore, deals with splitting the file into these "blocks". This is done quite simply by looking for cases where *NewLine* tokens occur repeatedly after populating a block, i.e. if the number of *Newline* tokens excedes one (the one used to seperate lines within a block) before another token is encountered then we have reached the end of the block we are currently building.

The next split is splitting each block into lines, this is done by splitting each block sequence on *NewLine* tokens. This is counter-intuitive to the concept of the how the tablature is grouped but due to a need to verify the patterns this must be done first. This is done by veryfying that each line is of the same length as the rest. If not then the whole block is discarded as we assume it is corrupt.

The final split is that of the individual patterns. Each line is now split on the *TrackDivider* tokens. The first segment of each line is expected to map to the drum component, with

the successive segments representing the beats and rests of the pattern.

This "Drum Tablature File" object is then passed on to the 3rd and final stage of the pipeline; the parser.

3.2.3 Parsing

Due to all this pre-processing the job of converting the 4 level sequences representing each drum tablature file, which was built in the previous step, to a datastructure that we can match against is very simple.

Parsing to sequence specifications

The job of the parser is now to create Sequence Specifications[2] based on the Drum Tablature File objects that are passed to it by the pre-parser. The method for doing so is outlined in algorithm 1.

Algorithm 1 Parsing the pre-processed tokens

```
1: procedure BuildSuffixTree(GroupedTokens gt)
2:     sequences = empty indexed list of Sequence Specifications
3:     for each Track t in gt do
4:         for each Line l in t do
5:             currentInstrument = l.get(0).get(0)
6:             for i from 1 to l.size() do          ▷ For every sequence segment in the line
7:                 sequenceSegment = l.get(i)
8:                 Add sequenceSegment to the corresponding Sequence Specification in
   sequences (or create a new one) by looking at the position in the line (i) and looking
   at currentInstrument to know which row to of the Specification to add it to
```

One step that is masked in this algorithm in the actual implementation is that only sequenceSegments that fit the resolution of the system are added. This is to simplify the pattern matching process later. This step will not result in incorrect parses of patterns, as if one sequence segment is of an incompatable resolution then all other segments will be of the same incompatable resolution. Leading to whole sequences being ignored, which means we get a smaller database but still one with integrity.

[2] recall from 3.1.1

3.3 Inference Module

In order to extract and infer the pattern the user intended to play the Inference module has the following pipeline.

1. Split the recording into eight note channels representing each drum component
2. Calculate the time deltas between each drum beat on each channel
3. Form a suffix tree, for each note channel, using these time deltas (rounded to some increment) as the alphabet
4. Find the repeated sequence that fits best
5. Extract the inferred pattern
6. Pass inferred pattern on to the Querying Module and the User

The first two steps are justified and explained in 2.4.1. The successive steps are explained in detail here.

3.3.1 Suffix Trees

A suffix tree, as discussed in 2.3.1, is a data structure that will help in discovering repeated structure in each note channel. Gusfield's[16] description of the data structure formed the basis for the implementation, along with the basis for the repeated structure extraction algorithm.

Definition of Suffix Trees

Given a string S of length n a suffix tree is said to be defined as:

1. A tree with exactly n leaves labelled 0 to $n-1$
2. Internal nodes[3] have at least two children
3. Each edge is labelled with a non empty sub-string of S
4. The starting character of the label of each edge coming out of a node is unique
5. Concatenating each path from root to leaf, which is labelled with i, gives the sub-string from i to n, i.e. the suffix starting at index i

[3]Internal nodes are non-leaf, non-root nodes

To ensure that no suffix is a prefix of another, a terminal symbol that is not expected to be seen in the string is used to pad S. In the literature this is usually denoted as '$' though in practice it can be anything that is known to not occur. For this purpose I reserved the time delta '-1' for use as the terminal symbol.

Because of this, all tree's in my implementation will have $n + 1$ leaf nodes. Furthermore, as all *internal nodes* have at least two children there can be at most n such nodes. So with 1 root node there can be at most $(n + 1) + n + 1 = 2n + 2$ nodes in total.

However, this definition requires $O(n^2)$ space due to the labelling of edges with the substrings (point 3). In order to remedy this I introduce the concept of *Index pointers*. These are tuples of the form *(start, end)* representing the start and end point of the substring labelling the edge. This compacts the space requirement to $\Theta(1)$ for each edge.

This leads to the conclusion that it is possible to represent the Suffix tree in $\Theta(n)$ space, and therefore, for large values of n, the overhead for representing the the sequence as a tree is negligible.

3.3.2 Suffix Tree Construction: Ukonnen's Algorithm

Representing the tree in linear space is not enough for the performance of the system. It is also important for the system to be able to construct the trees in linear time. Furthermore, it will be ideal if the system can update the tree on-line[4] to agree with the aim of developing a system that can be trivially extended to work in real-time. Ukkonnen[19] developed an algorithm that does all these things, which will be the basis for my implementation of suffix tree construction.

Implicit suffix tree

The first concept that I will introduce is that of an *implicit suffix tree*. If we have a suffix tree T for the sequence S$, which is the sequence S terminated by the terminal character $, the *implicit* suffix tree for sequence S is formed by removing every copy of $ from the edge labels of the tree, then removing any edge that has no label and removing any node that does not have at least two children. This is depicted in figure 3.5

Further to this, the implicit suffix tree for any sub-sequence S[0..i] of S can be formed by taking the suffix tree for S[0..i]$ and deleting the same required labels, edges, and nodes. This implicit suffix tree is notated as T_i.

The name for this structure stems from the fact that, while all the information for a sequence is encoded in an implicit suffix tree, it may be just that - *implicit* - which occurs when one suffix is prefixed by another, leading to only one path where there would otherwise be two. Each phase, i, the algorithm constructs the implicit suffix tree T_i.

[4]Meaning at most one step per character

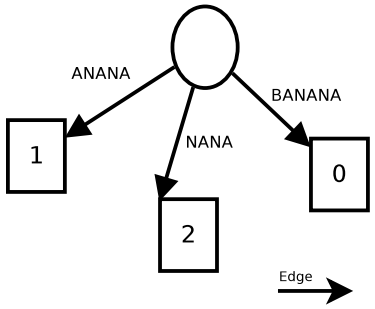

Figure 3.5: Here is the implicit suffix tree for the string BANANA. For a complete suffix tree refer to 2.4

Overview of algorithm

The overview of the algorithm is very simple, as seen in algorithm 2, though there are many concepts that are hidden by this higher view that looks like, at first glance, it would run in $O(n^2)$ time. For much of this discussion we borrow Gusfields[16] definitions of the concepts.

Algorithm 2 High-level view of Ukkonen's algorithm provided by Gusfield[**?**]

```
1: procedure BuildSuffixTree(Sequence S)
2:     Construct T_i
3:     n = length of S
4:     for i from 1 to n-1 do
5:         begin phase i+1
6:         for j from 1 to i+1 do
7:             Begin extension j
8:             Find the end of the path from the root labeled S[j..i] in the current tree. If
   needed, extend the path by adding character S(i+1), thus assuring that string S[j..i+1]
   is in the tree (possibly implicitly).
```

Suffix extension rules

In order to define the important details of the extension stages of algorithm 2 three rules for extension must be discussed.

Consider the example of being in phase i and extension j, so having to add string S[j..i]::S(i+1) to the tree. The algorithm first locates the end of S[j..i] in the tree and then extends that path according to one of the following rules:

1. If in the current tree, the path S[j..i] starting from the root ends at a leaf then to extend the tree character S(i+1) is added to the end of the label on the edge leading to the leaf. See a simple example in figure 3.6

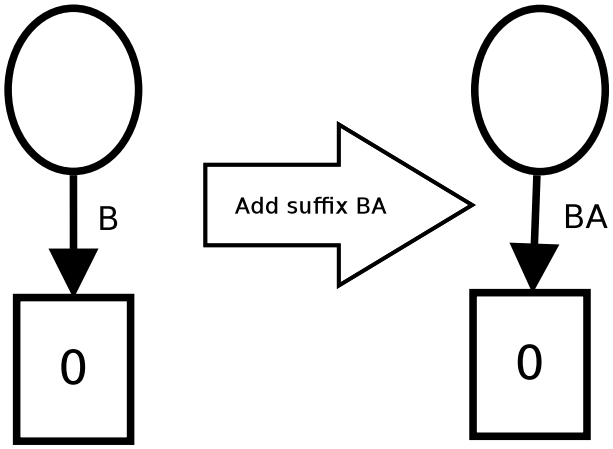

Figure 3.6: Here is the application of rule 1 on Suffix tree for string B when adding Suffix BA. Note that this is not the end of the phase as the suffix 'A' still must be added to the tree in order for it to be implicit suffix tree T_2

2. If in the current tree, the path S[j..i] starting from the root ends at a point where no next character is S(i+1) but there is at least one path to follow then
   - A new edge leading to a leaf from the end of S[j..i] has to be created labelled with S(i+1).
   - If S[j..i] ends in the middle of an edge that edge must also be split at this point, adding another internal node, to accomodate for the new edge
3. If in the current tree, the path S[j..i] starting from the root ends at a point where there is some path starting with character S(i+1), meaning S[j..i+1] is already in the current tree, we do nothing explicitly to the tree.

3.3.3 Speeding up Ukkonen's algorithm

Looking at algorithm 2, in conjunction with the three rules discussed earlier, it is still clear that the solution does not run in $O(n)$ time. Infact, a naive implementation could be as bad as $O(n^3)$ time by finding all of the ends of the suffixes by walking through the tree. Therefore it is important to implement several speedups that were originally detailed by Ukkonen [19].

Suffix links

The most important technique used to speed up the building of suffix trees are the concept of *suffix links*. Suffix links are defined by the following scenario taken from Gusfield [**?**]: Let xA denote an arbitrary string, where x denotes a single character and A denotes a

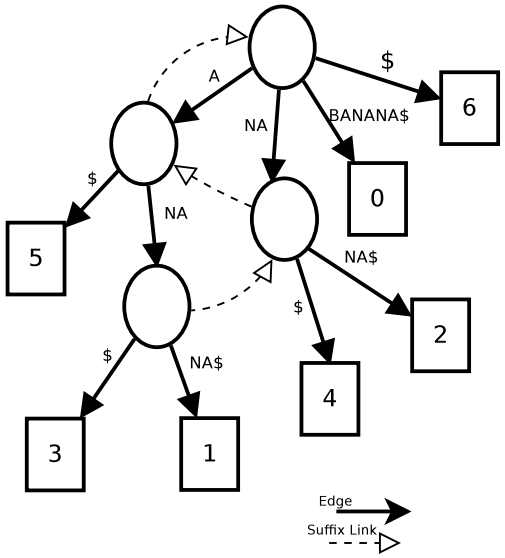

Figure 3.7: Example of the familiar BANANA$ string, shown first at 2.4, now with the suffix links explicitly shown with the dotted lines

(possibly empty) substring. For an internal node v with path-label xa, if there is another node $s(v)$ with path-label A, then a pointer from v to $s(v)$ is called a *(*suffix link). Further to this if a is empty the suffix link will point to the root node, though none wil be pointing out from the root as it is not an internal node.

In extension j for some phase i, consider the case that a new internal node, v, is created with the path-label of xA, then in extension j+1 of the same phase i either the path A either ends at an internal node (or the root in the case A is empty), or a new internal node at the end of the string A will be created. In either case a new suffix link is created leading from node v to the node discovered in extension j+1. It therefore follows that all internal nodes present in extension j for some phase will have a suffix link leading out of it by the end of extension j+1 of the same phase.

The reason suffix links are useful is because they provide a short cut when traversing a tree implementing extensions for some phase of the algorithm. Recall the naive method of walking through the tree to find the substring prefixing the character needed to be added, in extension 1 of phase i you find the string S[1..i], which is always the longest path from the root and so can be pointed to constantly. You then extend it with Rule 1, and instead of now starting from the root in extension 2 to find S[2..i], you may follow the suffix link to the next node, thereby cutting out steps of the costly walkthrough.

Skip/Count Trick

In order to further decrease the cost of walking down the tree to find the end of a suffix A, we may implement the following trick:

- When at a node find the next arc to follow
- If the length of that arc is shorter than the length of A, jump to the next arc

This trick allowes the algorithm to condense the number of comparisons along a path to be proportional to the number of nodes in the path, rather than the number of characters in it. Additionally, with the earlier optimisation of *index pointers* discussed in 3.3.1, the length of each edge can be found in constant time.

Implicit extensions

Recall in 3.3.2 I outlined the rules that are followed when performing suffix extension. Through analysing the cases the rules are followed we can find ways of speeding up the algorithm even more.

For the first optimisation recall rule 3 is followed, leading to no explicit operations to the tree, when the suffix being added $S[j..i+1]$ already appears in the tree. However, as we are adding this suffix to an implicit suffic tree that means that $S[j+1..i+1]$ is also in the tree, and all suffixes $S[k..i+1]$, for $j <= k <= i+1$ where k is an integer, are in the tree. Therefore for the remaining extensions in that phase no explicit operations to the tree need to occur. Additionally, if we consider that rule 1 does not create any new nodes (be it leaves or internal nodes), we can see that a new sUffix link needs to be added only after extensions that involve rule 2. These two observations lead to the following observation trick:

- End any phase $i+1$ the first time that extension rule 3 applies as there is no need to explicitly extend the tree any more in this phase.

This is a good way of reducing work, but it is not clear how this affects the worst-case time bound. However if we combine it with the next observation and optimisation it becomes more clear.

If you again consider the extension rules, it is clear that there is no rule that turns a leaf into an internal node or root, i.e. if a node is a leaf it is always a leaf (and if a node is an internal node it is always an internal node, but that doesn't matter for this optimisation). In phase 1 there is exactly 1 leaf created (creating the implicit suffix tree for just the first character). Therefore in every subsequent phase i there is an initial sequence of consecutive extensions where extension rule 1 or 2 apply before the phase can be terminated by application of rule 3 as per the previous optimisation. This initial sequence can clearly not shrink in size as the phases are iterated through as the number of leaves can only increase due to applications of rule 2.

This observation leads to an optimisation that takes advantage of *index pointers* introduced in 3.3.1:

- When a leaf is initialised, instead of explicitly labelling the edge pointing to it with the index pointer (start,end), instead have some *endGlobal* that coincides with the phase of the algorithm and populate the index pointer with (start,pointer to *endGlobal*)[5]

This allows the *endGlobal* to be set once each phase, i+1, allowing for constant-time application of rule 1 for the entire phase. Now the newest characted S(i+1) is added to each leaf of the tree in one step.

Complexity of algorithm

As discussed in 2.3.1 the complexity of the algorithm after all these enhancements is $O(n)$ in time. This is becuase in each phase we either do nothing explicitly (apart from updating *endGlobal* under rule 1), or we apply the n explicit extensions, under rule 2, we skipped by following rule 3 in earlier phases.

3.3.4 Finding repeated Structures

Once the suffix tree has been built it is a simple problem to find the repeated structures from it as each internal path[6] corresponds to a repeated sequence. For this I implemented a simple DFS[7] algorithm 3.

Remember that, although so far in our discussion on suffix trees I have abstracted suffix trees to be formed of strings of characters, the underlying structure of the sequences are pairwise time offsets, for which characters are extracted by rounding these offsets. Therefore, when computing the sumOfEdge() procedure in algorithm 3 we merely add these characters up, which can be computed in $O(n)$ time and constant space.

A RepeatedStructure is defined as a tuple of natural numbers, ($endTickIndex$ * $length$ * $numberOfRepeats$), that represent the timing information of repeats in the suffix tree, as well as the number of times the subsequence repeats. $endTickIndex$ is found by taking the end value of the lowest edge in the path that corresponds to the repeat, which maps to the last character of the first instance of the repeat. $length$ is determined by summing the path from the root to the end of the repeat, i.e. currentLength in algorithm 3. $numberOfRepeats$ is determined by counting the number of children of the repeats deepest node.

The value for $maxLength$ is defined as the maximum length that we recognise as a bar of a performance, when the bar is repeated multiple times. Therefore ideally it would be exactly $idealLength = (lengthOfPerformance)/(expectedNumberOfRepeats)$. This clearly relies on the user playing exactly the number of repeats they were planning to, and each re-

[5] In Java this pointer is implemented by shared references to an object that acts as an *int* wrapper
[6] Internal paths are paths starting at the root and ending at an internal node
[7] Depth-First Search

Algorithm 3 Algorithm for finding the repeated structures in a suffix tree

```
1: maxLength = the maximum length we allow for a repeated pattern
2: procedure findRepeat(SuffixTreeNode node,int currentLength,RepeatedStructure
   rs)
3:     for all non-leaf children, n, of node do
4:         sumOfEdge = addTimeDeltas(n)
5:         if sumOfEdge+currentLength > maxLength then
6:             The algorithm has overshot the expected bar length so end recursive call
7:         else
8:             currentLength += sumOfEdge
9:             if rs.length ¡ currentLength then
10:                We have found a better match for the repeated structure
11:                So set rs to point to this repeat instead
12:                findRepeat(n,currentLength,rs)
13:            else if rs.length == currentLength then
14:                both repeats are of equal length so must find best candidate
15:                set rs to the one with the most children
16:                findRepeat(n,currentLength,rs)
17:            else
18:                Dont change anything, previous repeat is still the best
19:                findRepeat(n,currentLength,rs)
20: procedure addTimeDeltas(SuffixTreeNode node)
21:     indexPointer = index pointer of the edge leading to node
22:     sum = sum of the timeDeltas represented by indexPointer
23:     return sum
```

peat to end at the perfect time. Therefore in order to allow for some margin of error in both cases allow $maxLength = 3/2 * (lengthOfRerformance)/(expectedNumberOfRepeats)$. This means that it is at max looking for 1.5 bars of the performance, which isn't mathematically sound but has been shown to work decently well in practice when combined with the bar inference step seen next next.

3.3.5 Bar inferrence

Recall that we split the MIDI recording into 8 sequences (3.3) each representing one of the 8 componets to the drumkit. These 8 sequences then each were processed into suffix trees (3.3.1) and analysed to find their repeated structures (3.3.4).

We now consider these repeated structures to be candidates for finding the overall best repeated structure by finding the structure that minimises the distance from the perfect length[8]. We then use the timing information of this repeat as the basis for the inferred bar.

Quantisation

Quantisation, as it pertains to drum rhythms, is the process of taking continious information regarding timing offset of impulses on the drumkit and grouping the information into discrete time slices, or quanta.

The number of quanta corresponds directly with the resolution of the system. Once we have the timing information of the best repeat, we can work out the quanta boundaries of the resulting pattern using the formula $quantumStart_i = startTime + i * (endTime - startTime)/(resolution)$ for all i from 0 to $resolution$, where $endTime$ can be calculated from $endTickIndex$ of the repeated structure, and $startTime$ can be calculated by $startTime = endTime - (repeatedStructure.length)$.

It is then a simple matter of populating the familiar Sequence Specipication datastructure defined in 3.1.1 by mapping the quanta to the divisions between the cells. Multiple beats occuring in the same quanta is fine as they are attributed to noise or a rudiment that intends a double note, e.g. a flam. As the Harvester would consider a 'f' character indicating a flam to map to a single 1 in the Sequence Specification it is appropriate that the Inference module does too.

The Inference module now returns this Sequence Specification to the user as the inferred pattern as well as passing it on to the Querying module to find similar patterns.

[8] defined as $(lengthOfTheRecording)/(expectedNumberOfRepeats)$

3.4 Querying module

The aim of the querying system is to inform the user of the system of all patterns that are similar to the pattern they used to query the database with, which provides a way of fixing any minor mistakes in the inference module, as well as giving the user an intuitive tool for drum rhythm look-up. Further to this, the user might also be just as interested in the most dissimilar rhythms, and therefore I find it appropiate to return an ordered list of all the patterns in the database, for this I used a List<String>[] of size 128 (the largest possible distance) where each index i of the array holds a list of strings which map to the song that is that distance, i, away.

This module assumes the availablility of a database of Sequence Specification objects (3.2.3) as well as a query Sequence object that is of the same resolution. In order to apply these algorithms to this domain it is appropriate to consider each row of the strings as seperate to the others. Therefore the overall distance between two Sequence Specifications is the sum of the individual edit distances across all eight rows.

3.4.1 Hamming Distance

The Hamming distance[17] between two strings, of equal length, measures the minimum number of substitutions in order to change one string into the other. The algorithm, shown in Algorithm 4, computes the Hamming distance in $O(n)$ time where n is the length of one of the sequences. It is important to note here that the traditional Hamming distance is not defined for strings of unequal length, though given the domain we will never deal with strings of unequal length so there is no need to extend the algorithm.

Algorithm 4 Algorithm for computing the Hamming distance between two strings

```
1: procedure HAMMINGDISTANCE(Sequence A, Sequence B)
2:     distance = 0
3:     for i from 0 to lengthOf(A) do
4:         if A.get(i) does not equal B.get(i) then
5:             distance = distance+1
6:     return distance
```

3.4.2 Edit distance

The edit distance between two strings is informally defined as the number of edits to get from one of the strings to another. The so called edits in this case are defined by Levenshtein[18] as insertions, deletions, and substitutions of individual characters.

A dynamic programming algorithm for computing the edit distance of two strings one dimensional of lengths m and n was devised by Wagner-Fischer in 1974[24] which computes the edit distance in $O(nm)$ time and space as seen in Algorithm 5.

The advantage the Edit distance has over the Hamming distance is that it does not over penalise for transposed data. For example if we take two strings '1010' and '0101' the edit distance is 2 while the Hamming distance is 4. Furthermore, for two strings of similar pattern to the previous example '1010...0' and '0101...1' each of length $O(n)$ the edit distance is still 2 while the Hamming distance is n.

Algorithm 5 Wagner-Fischer algorithm for computing the Edit distance of two strings

```
1: procedure editDistance(Sequence A,Sequence B)
2:     m = lengthOf(A)
3:     n = lengthOf(B)
4:     distances = new 2D array of dimensions (m,n)
5:
6: initiate the edges of the array
7:     for i from 0 to m do
8:         distances[i][0] = i
9:     for i from 0 to n do
10:        distances[0][i] = i
11:
12: Now populate the remainder of the array
13:    for j from 1 to n do
14:        for i from 1 to m do
15:            if A.get(i) equals B.get(j) then          ▷ No need edits needed
16:                distances[i][j] = distances[i-1][j-1]
17:            else                                     ▷ Take minimum of operations
18:                distances[i][j] = MIN(distances,i,j)
19:    return distances[m-1][n-1]
20: procedure min(int[][] array,int i, int j)
21:    deletion = array[i-1][j]
22:    insertion = array[i][j-1]
23:    substitution = array[i-1][j-1]
24:    return smallestOf(deletion,insertion,substitution)+1
```

3.4.3 Cyclic extensions

When we consider the two patterns shown in figure 3.8 we can see standard implementations of these edit distance algorithms may not return the best result. If one were to play one of the rhythms continiously it would be unintelligible from the other, due to that the

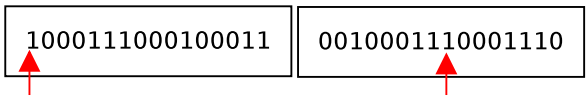

Figure 3.8: Example of two patterns that have a cyclic distance of 0 are distant when seen in isolation. The red arrow notates the start point of the cycle.

only difference is the phase offset. Therefore, if the querying module aims at returning all relevent information to the user then surely the pattern should be returned with a distance of 0, meaning an exact match, rather than than an edit distance of 8 and a Hamming distance of 10.

In order to extend the distance metrics to find the best match regardless of whether the bar is inferred (or played) with a phase offset we must rotate one of the sequences and compute the distance metric at every point of rotation. We then take the lowest distance as the *cyclic distance* between the two Sequences.

Rotating a two dimensional list is ofcourse very expensive in time, and so I use an implementation trick of passing a third parameter, the offset, when calling the distance metric which is an integer that refers to the position (column) we consider the begining of the Sequence. Then through simple modular arithmitic we can work out where we are in the list at every iteration of the algorithms.

3.5 Summary

In this chapter I discussed the design decisions and implementation I undertook to achieve the aims of each module.

I started with a thorough discussion on how the Harvester module was implemented in 3.2 that aims at being a general solution for extracting drum rhythms from ASCII drum tablature, and populating the database for later comparisons.

I followed this with an explanation of the Inference module in 3.3, which uses the comparison of multiple suffix trees in order to extract a repeated structure and infer a quantised pattern from a MIDI recording of performance data.

I ended with a detailed depiction of the algorithms used to compare the distance between two sequences in 3.4. They aim to return the user with an ordered list of the most similar patterns.

Chapter 4

Evaluation

Due to the modular design of the system each module works in isolation to the others and can be seen as seperate deliverables. In order to best evaluate the effectiveness of the system as a whole it is apt to think of the system as a sum of its parts. This is because there is room for optimisations and improvement in each module, and so improving one will have an effect on the rest. In this chapter I will discuss the level of success of each module, along with possible ways of improving the module.

4.1 Dataset collection

In order to collect a dataset of MIDI performance data to use as the basis for any inference techniques, as well as a method of evaluation, I designed a user study in which skilled participants played 10 selected rhythms (transcribed into ASCII drum tablature) on a drum kit.

For each rhythm there were 4 sets of data collected:

1. 1 recording of the participant learning how to play the rhythm
2. 10 recordings of the participant playing the rhythm once in isolation
3. 1 recording of the participant playing the rhythm continiously 10 times
4. 1 recording of the participant playing the rhythm with any augmentation that the participant feels would be added during live performance

Throughout the analysis I will refer back to these datasets, though dataset 1 and 4 were not used in the end as the information gleamed was too noisy. Dataset 3 is used to evaluate the inference module while dataset 2 is used for the baseline for its evaluation.

When thinking about the design of the system from an end-user stand point I originally wanted the user to play the drum rhythm they wish to look up just once, which is then used as the example to query the database, which is why dataset 2 is significantly larger than any others. The problem with this is that the latency of the system caused by start up and shut down lead to the example being distorted. In order to tackle this problem I decided it was wise to design a system that could extract repeated structures from dataset 3, with the hope that it would be at least as good, and less susceptible to timing errors, than dataset 2. However, as a result of doing the data collection first dataset 3 is not large, in comparison to dataset 2. Additionally the study only had 2 participants due to the need of them being skilled drummers who could reliably play the required rhythms, as a result the results may be skewed to their performance. Both issues could be addressed by an expanded study.

4.2 Evaluating the Havesting module

4.2.1 Information lost

As discussed in 2.2.3 ASCII Drum Tablature has a lot of superfluous information that cannot be retrieved without significant natural language processing. Furthermore due to the lack of a standardised practice in terms of formatting, and symbol use, a large subset of the drum tablature cannot be harvested without a significantly more complicated parser. As this was not the focus of the project I decided to create a solution that would serve to create a decent database rather than one that solves the general case.

Parsing to Sequence Specifications has the disadvantage of simplifying the pattern from one that detailed at least the type of beat (such as an Accent, Ghost note, flam, drag, soft one-handed roll, etc), and typically much more information such as beats-per-minute and transitions between rhythms, to one that merely specifies binary on-set timing information of a rhythm in isolation from the song that it is part of.

4.2.2 Level of success

The original aim of the harvester (see 2.1.2) was to extract a database containing at least 1000 rhythms. The harvester implemented in fact goes above and beyond this as shown in figure 4.1, with over 10 times the number of patterns present in the database than was initially required.

However, nearly half of the patterns present in the database are duplicates, and so the actual level of success corressponds with the number of acyclic unique patterns (calculated using the hamming distance), or the number of cyclic unique patterns (calculated using the cyclic hamming distance described in 3.4.3) depending on the definition of sequence

Original Aim	Actual Result	Acyclic Unique	Cyclic Unique
1,000	12,016	6,129	6,057

Figure 4.1: Here is a table depicting the level of success of the Harvester. Acyclic Unique, and Cyclic Unique, show how many items in the database are unique based on the hamming and cyclic hamming distances respectively.

specification similarity. In either case, a sizeable database was still developed.

I have estimated the total number of patterns in the database used as the data source to be of the order of 100,000, of the order of 1,000 songs each containing of the order of 100 patterns on average, and so the harvester does not perform at a good rate of success, as only an estimated 12% of what is in the database is parsed correctly.

4.3 Evaluating the Inference module

4.3.1 Information Lost

In a similar way to the Harvester, the Inference module also loses information due to converting the complex types of beats, that are conveyed over MIDI as varying in impulse velocity and number of messages, to one that, again, just specifies binary on-set timing information.

A lot of the information transmitted regarding the hi-hat, discussed in the final paragraph of 2.4.1, was also lost. Three of the four note numbers map to the same row in the Sequence Specification, and the fourth - the control variable - is ignored entirely. The reason for this abstraction was due to drum tablature having no clear way (other than comments) to refer to an direct open, direct closed, or indirect propogation of the hi-hat. So without a significantly more intelligent parser with the ability to infer meaning from comments in drum tablature, Sequence Specifications are unable to specify the difference between the different note types, which manifests in the Inference module as lost information.

4.3.2 Level of success

In order to evaluate the level of success of the inference model we need to ask the simple question of "how often is the pattern inferred correctly?", or rather "on average how far off was the inferred pattern from the true pattern?". The answer to the latter came in the form of figure 4.2. However, with no context to frame these values it is hard to know whether this is a success or not.

The more basic approach of using the length of the recording in dataset 2 as the basis for

Distance Type	Mean	Std. Deviation
Hamming	20.0	7.27
Edit	14.8	5.39
Cyclic Hamming	13.6	6.23
Cyclic Edit	10.2	4.58

Figure 4.2: Here is a table representing the mean and standard deviation of the distance metrics for the 20 samples in dataset 3.

Distance Type	Mean	Std. Deviation
Hamming	17.3	3.96
Edit	12.0	2.96
Cyclic Hamming	12.11	3.27
Cyclic Edit	8.96	2.56

Figure 4.3: Here is a table representing the mean and standard deviation of the distance metrics for the 200 samples in dataset 2.

quantisation, and hence skipping the string analysis via suffix trees steps, a baseline for evaluation was produced, and its results are seen in figure 4.3. In order for the Inference model to be considered a success it has to perform at least as well as this baseline method.

Both of these distributions are represented in figure 4.4.

The inference module is expected to perform worse for the acyclic distance metrics as the inferred repeats are not framed with the correct start and end points. However, we expect the cyclic distances to not either be better, due to more precise timing of the underlying sequence, or not significantly worse.

If we run a Student's t-test for a confidence level of 95% on each metric to assess whether the two distributions have the same underlying distribution we get what we expect. Both acyclic distance's distributions are rejected, meaning the inference module is significantly worse than this baseline, and both cyclic distance's distributions are not, meaning there is no significant difference between the two methods at the cyclic level.

While these results are not promising at first glance it is important to understand the usability improvements that come with using the inference module over the the baseline method. It is not common that drummers play a single bar in isolation, and controlling the start and stop times of the recording on top of performing a single bar may be detrimental to the performance. For the study a third party controlled the timing information and so the participant just had to play the rhythm. Playing continious rhythms is far more intuitive, and any mistakes in individual bars are less critical for the performance.

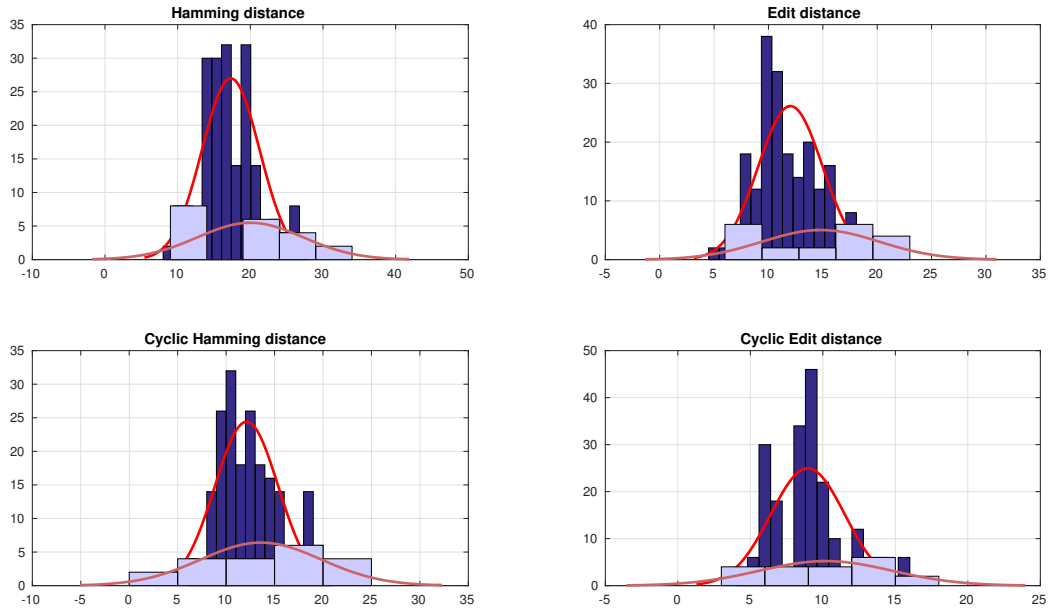

Figure 4.4: The navy-blue bars and red fit corresponds with the single rhythm performances, while the light-blue bars and pastel-purple fit corresponds with repeated rhythm performances. On the x-axis we have distance, and on the y-axis we have frequency at that distance

4.4 Evaluating the Querying module

4.4.1 Level of success

When evaluating the Querying module we shall look at the percentage ranking of the intended Sequence Specification that each distance metric returns. For example, if the distance metric returns the percentage ranking of 30%, it means that 30% of the list was thought to be a better match. We consider a percentage ranking of 25% to be an acceptable result. Again, we will use dataset 2, as it is a large dataset, and add dataset 3 into the discussion to assess if there is any significant difference.

Hamming Distances

The Hamming distance rankings performed worse than expected, as seen in figure A.1, with a mean rank of 46.4% and a standard deviation of 20.13%. This is surprising as dataset 2 has the quality of already being timing framed, and so bypassing the inference module. The Cyclic Hamming distance performance was also dissapointing, as seen in figure A.3, with a mean of 30.0% and a standard deviation of 19.05%, but if one were to

Distance Type	Mean	Std. Deviation
Hamming	36.7	29.2
Edit	3.9	5.9
Cyclic Hamming	11.6	12.7
Cyclic Edit	5.4	5.5

Figure 4.5: Here is a table representing the mean and standard deviation of the percentage ranking for the 20 samples in dataset 2.

perform poorly on this dataset you would expect the other to as well. This is because any timing uncertainty caused by human error stretches the pattern rather than rotating it. Therefore the Hamming distance will not considerably improve by checking for cycles.

Edit Distances

The edit distance rankings performed much better than the Hamming distance, which was expected due to that it does not penalise as much for mistakes in timing. As seen in figure A.2 the mean and standard deviation for the Edit percentage rank are 14.0% and 12.26% respectively, while the corresponding values for the cyclic edit distance, figure A.4, are 22.8% and 17.62% . The edit distance is expected to be the best performing for this dataset as the cyclic distances positively skew the data, while the hamming distance over penalises for timing errors.

Percentage ranking from dataset 3

Most interestingly, all of the distance metrics out performed the percentage rankings generated from dataset 2 when ran on dataset 3. In fact, only one of the metrics (Hamming) did not reject the null hypothesis of the same underlying distribution as it's counter part for dataset 2. This is somewhat a shock as, while analysis of the inference module seemed to suggest no significant improvement from the baseline method, the same baseline method produces a worse results in the querying method.

Note that the edit distance is expected to perform better than the cyclic edit distance for this test as cyclic edit distance is a looser metric, ranks are more positively skewed and with being returned first the best we can hope for it limits the comparison at this level of accuracy.

In any case, this data still supports the conclusion that the edit distance is a better metric than the hamming distance when querying the database.

4.5 Summary

In this section I evaluated the three majour modules using datasets collected from user performance data, the collection process detailed in 4.1.

I showed the showed that the Harvester is able to extract about 12,000 Sequence Specifications in section 4.2, 6,129 of which are acyclicly unique and 6,057 cyclicly unique.

I then analysised the Inference module in 4.3, detailing the information that is lost in this module and going on to a discussion on the level of success of the system where I showed it to be minimally sufficient.

Finally I compared the suitability of the different distance metrics for querying the database in 4.4. This lead to an interesting insight into the performance of the inference module may be better than what was previously thought when you take into account the system as a whole.

Chapter 5

Conclusion

In this project I have described the implementation of a pattern inference and matching system for drum rhythms. The system is shown to be made up of 3 individual modules, and two databases, 2 of which can work in isolation and the 3rd links execution.

5.1 Comparison with original aims

Recall the success criteria from 2.1.2.

Criterion 1, regarding whether the system can translate MIDI performance data to Sequence Specifications, is shown to be acheived in 3.3.5 as the Sequence specification was populated, but in general the system is able to map any MIDI sequence to a Sequence Specification if the start and end times of the sequence are known.

Criterion 2, regarding whether the system can collate a database with at least 1000 unique Sequence Specifications, is shown to be acheived in 4.2. The implementation acheived over 6x the target

Criterion 3, regarding whether the system is able to infer the intended rhythm from a MIDI sequence. Analysis of the Inference module against a baseline method in 4.3 seems to suggest that the Inference module is minimally sufficient, though the sparsity of the study data makes it hard to conclude for definite.

Criterion 4, regarding whether the inferred rhythm consistantly lies in the top 25% of results returned by the database. In 4.4 all of the distance metrics apart from the Hamming distance surpass this aim, with the edit variants consistently being found in the top 10%.

Overall, I think the project can be considered a success, with 2/4 criterions being sufficiently met, and the others surpassing expectations.

5.2 Future work

5.2.1 Havester

The of number of acyclic unique patterns available in Sequence Specification (discused in 3.1.1) is limited at 2^k where $k = 8 \times 16 = 128$. Although this limit will likely never be reached as a large proportion will never be seen in songs (consider the specification of all 1s) this limit can be approached by improving the harvester using one, or both, of the following methods:

- Expand the source for transcribed drum tablature
- Improve the parser

Expanding the source for transcribed drum tablature can be done by either searching additional collaborative databases, or inviting users to transcribe more songs (updates to the collaborative database I used ceased in late 2001). The latter allows for education of the new transcribers with the common practice that the parser was build towards, thereby indirectly increasing the rate of success.

Improving the parser is an approach that will aim at increasing the rate of success directly by adding additional rules for parsing data so that less is thrown away. Examples of these rules are: if the resolution fo the pattern is not equal to 16 map scale the pattern appropiately, a second parser for a different common practice, not being as strict in the comment removal stage, etc.

5.2.2 Inference module

The inference module can be improved by adding statistical inference into deciding upon sequence specifications. At each stage in the inference pipeline a level of uncertainty can be carried forward and inferring the sequence could then be based off of Bayesian inference.

5.2.3 Querying module

As seen in figure 4.1 almost half the database is made up of duplicates. Repeats are expected as a lot of popular music is based on other music, but perhaps there is a better way of handeling cases in which they occur, than having duplicate Sequence Specifications in the database. We descovered in 4.4 that the Hamming distances are not as effective as the edit distances at querying the database, however we know that they can calculate distance in half the time ($O(n)$ vs $O(nm)$) and constast space, therefore calculating the Hamming distance is not very resource dependant. I propose that the focus for these algorithms in the future of this system is to be moved to deal with handeling database

indexing to allow for efficient look up, which could half the querying time of the algorithm (if we ignore the overhead of communication with memory).

5.2.4 Project as a whole

The project as a whole can go down several paths:

- Implementing a more complex parser to gather a larger database of Sequence Specifications
- Extending the parser to be a general compiler for drum tablature, compiling to MIDI files or Live Programming code[1]
  - Similarly compiling inferred Sequence Specifications from performance data
- Mapping the Sequence Specifications to the songs to provide an automated acoupament and/or algorithmic composition on database look-up
- Extending the system to work in real time to allow for live performance with one of the other extensions

[1] Programming languages that are used to dynamically generate music

Bibliography

[1] Wang, Avery. *The Shazam music recognition service*, Communications of the ACM, 2006.

[2] Goto, Masataka. *An audio-based real-time beat tracking system for music with or without drum-sounds*, Journal of New Music Research, 2001.

[3] Ellis, Daniel PW. *Beat tracking by dynamic programming*, Journal of New Music Reasearch, 2007.

[4] Ellis, Daniel PW; Poliner, Graham E. *Identifying cover songs' with chroma features and dynamic programming beat tracking* Acoustics, Speech and Signal Processing, 2007.

[5] Davies, Matthew EP; Plumbley, Mark D. *Context-dependent beat tracking of musical audio* Audio, Speech, and Language Processing, 2007.

[6] Mathews, Max. *The Digital Computer as a Musical Instrument*, Science, 1963.

[7] Rothstein, Joseph. *Midi: A comprehensive introduction* AR Editions, 1995.

[8] Bainbridge, David; Bell, Tim. *The challenge of optical music recognition* Computers and Humanities, 2001

[9] Dodge, Charles; Thomas A. Jerse. *Computer music: synthesis, composition and performance*, Macmillan Library Reference, 1997.

[10] Scelta, Gabriella F. *The History and Evolution of the Musical Symbol.*

[11] Weinberg, Norman. *Guidelines for Drumset Notation*, Percussive Notes, 1994.

[12] Johansen, Linn S. *Optical music recognition*, 2009.

[13] Knowles, Evan. *Parsing Guitar Tab*, http://knowles.co.za/parsing-guitar-tab/, 2013.

[14] Doornbusch, Paul. *Computer sound synthesis in 1951: the music of CSIRAC*, Computer Music Journal, 2004.

[15] Weiner, Peter. *Linear Pattern Matching Algorithms*, The Rand Corporation, 1973.

[16] Gusfield, Dan. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology* Cambridge University Press, 1999.

[17] Hamming, Richard W. *Error detecting and error correcting codes*, System technical journal, 1950.

[18] Levenshtein, Vladimir I. *Binary coddes capable of correcting deletions, insertions, and reversals*, Soviet physics doklady, 1966.

[19] Ukkonen, Esko. *On-line construction of suffix trees*, University of Helsinki, 1995.

[20] Bird, Richard S. *Two dimensional pattern matching*, Information Processing Letters, 1977.

[21] MIDI Manufacturers Association. *MIDI Specifications*, 1996.

[22] Walker, Martin. *Solving MIDI Timing Problems*, Sound on Sound, 2007.

[23] Roland. *Roland HD-1 V-Drums Lite - Owner's Manual*

[24] Wagner, Robert A.; Fischer, Michael J. *The string-to-string correction problem*, Journal of the ACM, 1974.

[25] Fox, Charles; Rezek, Iead; Roberts, Stephen J. *Drum'n'Bayes: on-line Variational Inference for Beat Tranking and Rhythm Recognition*, Proceedings of the ICMC, 2007.

[26] Tidemann, Axel; Demiris, Yiannis. *A Drum Machine That Learns to Groove*, Advances in Artificial Intelligence, 2008.

[27] Cicconet, Mercelo; Bretan, Mason; Weinberg, Gil. *Human-Robot Percussion Ensumble: Anticipation on the Basis of Visual Cues*, Robotics & Automation Magazine, 2014.

[28] Downie, Stephan J. *Music information retrieval*, Annual Review of Information Science and Technology, 2003.

[29] Delcher, Arthur L.; et al. *Alignment of whole genomes*, Nucleic acids research, 1999

[30] Baeza-Yates, Ricardo; Ribeiro-Neto, Berthier. *Modern information retrieval*, ACM press, 1999

[31] Knuth, Donald E.; Morris, James H. Jr; Pratt, Vaughan R. *Fast pattern matching in strings* SIAM jornal on computing, 1977

[32] John, Joyce; *Pandora and the Music Genome Project* Scientific Computing, 2008.

Appendix A

Percentage rankings of distance metrics on dataset 2

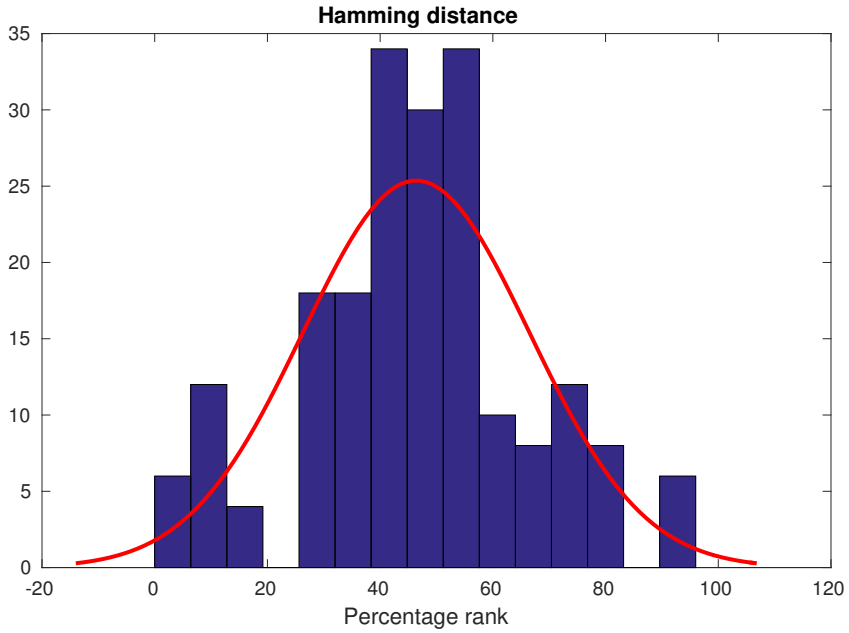

Figure A.1: Here represents the distribution of Hamming distance percentage rank. It has a mean of 46.4% and a standard deviation of 20.13%

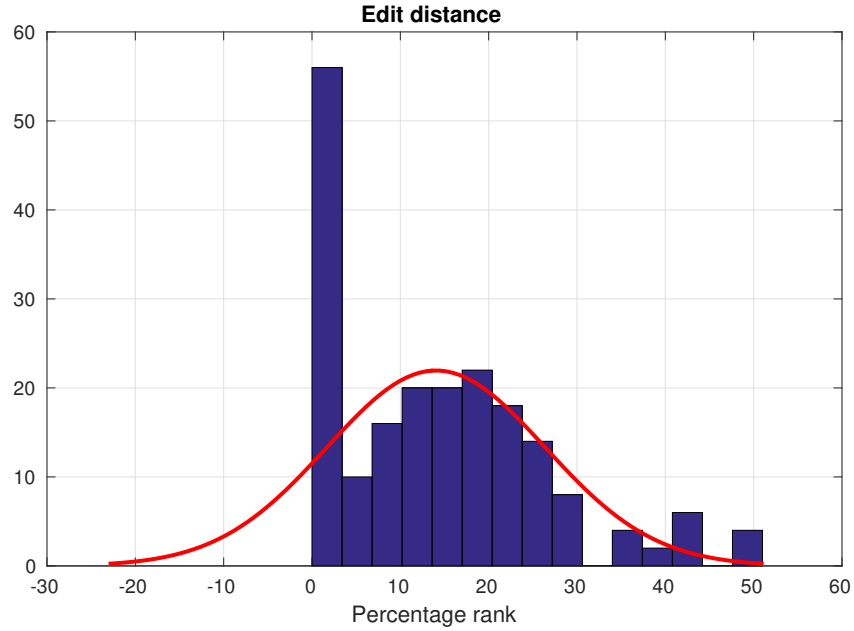

Figure A.2: Here represents the distribtion of Edit distance percentage rank. It has a mean of 14.0% and a standard deviation of 12.26%

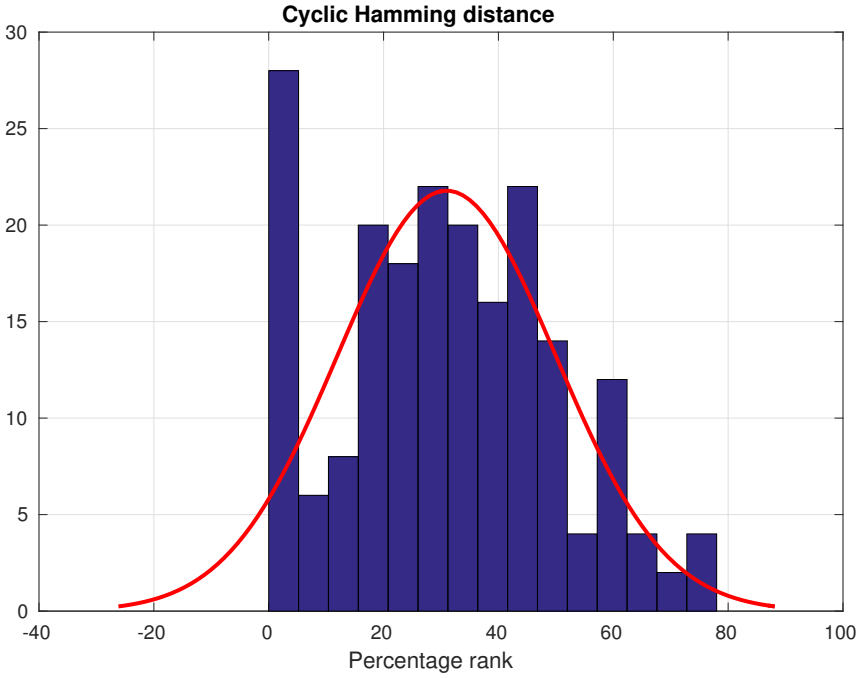

Figure A.3: Here represents the distribtion of Cyclic Hamming distance percentage rank. It has a mean of 30.0% and a standard deviation of 19.05%

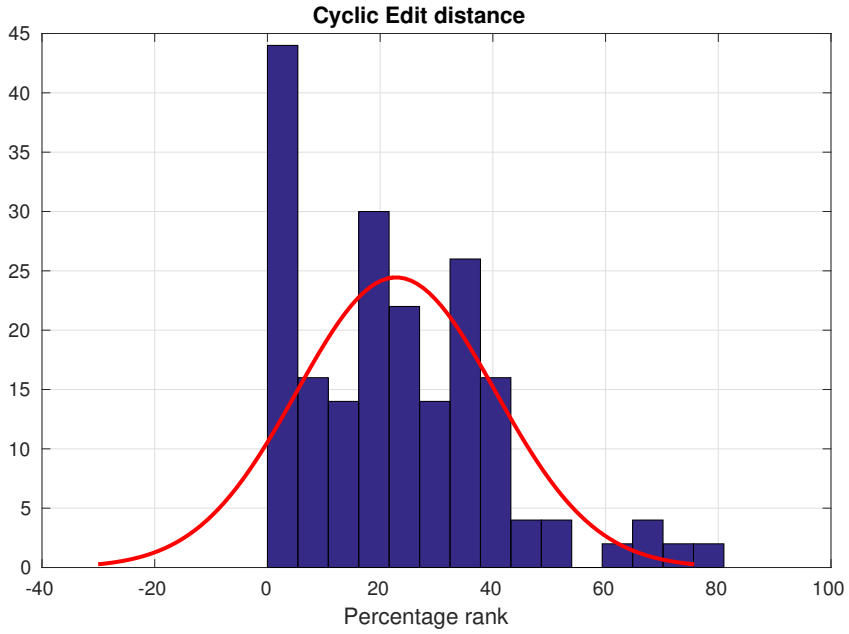

Figure A.4: Here represents the distribtion of Cyclic Edit distance percentage rank. It has a mean of 22.8% and a standard deviation of 17.62%

Appendix B

Project Proposal

Biko Agozino
St John's
ba325

Part II Project Proposal

Inferring sequence specifications from drum rhythms

October 22, 2015

Project Originator: Dr. Alan Blackwell & Dr. Sam Aaron

Resources Required: Yes, please see Resources Required section

Project Supervisor: Isak Herman

Signature:

Director of Studies: Dr Robert Mullins

Signature:

Overseers: Professor Ross Anderson & Professor Jean Bacon

Signatures:

1 Introduction and Description of Work

Drummers in many contemporary western musical ensembles play a vital role in the performance. They provide the rhythm, tempo, and pace of a song leading to the whole performance depending on the consistency of the drummer. However, Each drummer has their own nuances to their performance that manifest in the form of a slower/faster tempo or softer/harder beats. This can make it difficult to untrained musicians when attempting to reproduce the performance either on a physical drumkit or electronically without sampling. This is because the intended patterns performed by the drummer - of which their own personal transformations have affected - may not be immediately clear.

This project proposes to implement a system that infers the intended sequence of strokes on a drumkit from those that are captured, regardless of the variations in timing of the sequence and missed or incorrect beats. The system will then output the recording as the drum tablature or a MIDI[1] file for the user to either follow for their next performance or for use in their preferred music editing software.

The system will take on several steps, these are described briefly here and detailed further in the Project Structure section. The system will take in a MIDI input from an electronic drum kit and process it into a timestamped array, this will either be analysed directly or transformed into Drum Tablature before analysis.

Drum Tablature is a simplified form of Drum notation which is popular amongst drummers for its simplicity and ease of access, the contrast between Drum Tablature and standard notation can be seen in Figure 1. Drum Tablature is significant as it will be the form of information that we will harvest from a free online database[2] for use as a training set. This training set will then be used as the prior expectation when inferring the beat that the input is intending to play.

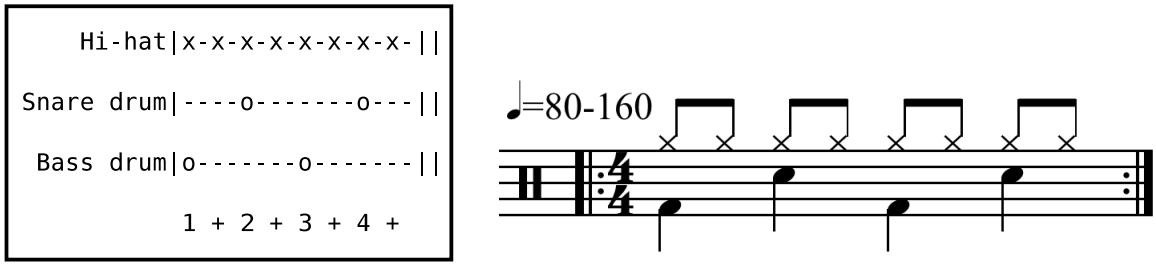

Figure 1: An example of Drum Tablature vs the standard notation (credit to Wikipedia user:Hyancinth for creating the standard notation). The x axis is essentially time and the y axis is the instrument being used. Varying symbols are used to represent different types of strike

[1] Musical Instrument Digital Interface

[2] This database lives on http://drumbum.com/drumtabs/

2 Starting point

I have experience creating web applications with a Java backend, this will be useful as the main component of the project is planned to be implemented in Java. I have also completed Java practical classes as part of Part IA and Part IB.

The Artificial Intelligence I and Mathematical Methods for Computer Science courses in Part 1B will be useful for the Machine Learning component of this project, particularly the Probability components of the Maths course. Other courses that are likely to provide key assistance are the Information Theory, Natural Language Processing, Digital Signal Processing, and Artificial Intelligence II courses in Part II.

3 Project structure

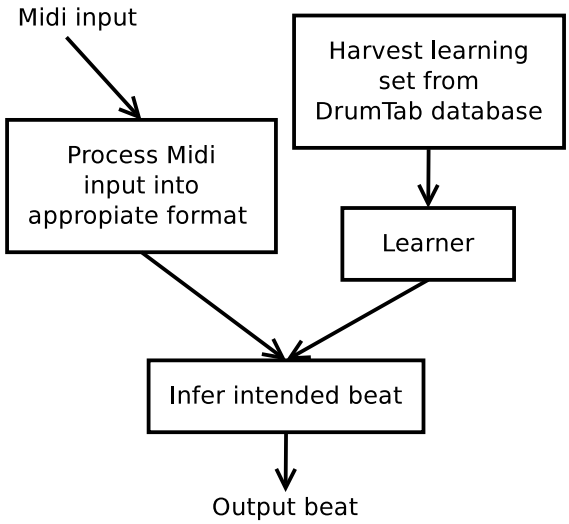

Figure 2: The core system

To allow for modular design of the system (Figure 2) the project has been split into the following steps/modules:

1. Read MIDI input data from an electronic drum kit into a time-stamped array. This step represents how the user will input into the system for later analysis by the inference model. It also allows me to get to grips with MIDI programming and prepares me for the steps to come.

2. Translate the contents of the time-stamped array into Drum Tablature format for comparison with the learning set.

3. Build the Learning set using a free Drum Tablature database, the learning set will be formed by parsing Drum Tablature into a sequence of beats.

4. Train a Hidden Markov Model (or other machine learning model if it appears more appropriate at the time) by using the sequences of beats from step 3

5. Create the inference module by using the trained model as a prior expectation in order to infer the intended sequence from the input of stage 1

3.1 Possible extensions

The primary extension and stretch goal is to integrate this inference model with the Sonic Pi live coding environment. Live coding is an emerging performing art in which the musician dynamically "codes" music. Integrating this inferred drum pattern into Sonic Pi will be a proof of application for this inference system. It could allow for further extensions in the realms of live programming and using this system as a tool for live performance. One idea is to infer what sequence of beats is likely to follow the one originally played, leading to an automatically developing baseline upon which the performer is free to worry about other details without the piece sounding monorhythmic. This opens the project to the realms of Algorithmic Composition which has been attempted many times, as detailed in this survey[1].

A further extension is to use an OMR[3] system to extend the learning set by parsing the standard drum notation (see Figure 1. OMR is a progressing field and this extension could be considered its own project, infact Yipeng Cheng[3] attempted to do just that in 2014. As such it is unlikely for this extension to be implemented, though is an area that may help with any limitations found from using Drum Tablature.

Similarly, the final extension I will detail here is using the Real-time Beat Tracking System, such as the one described by Masataka Goto[2], to both expand the learning set from audio signals as well as allowing users to input audio into the system for processing and beat inference.

4 Success Criteria

The following components are required for the successful development of the project:

1. Implementation of a system that reads MIDI and is able to translate it into Drum Tablature

2. Implementation of a system that is able to take sets of Drum Tablature as a training set for a Machine Learning model

[3]Optical Music Recognition

3. Implementation of a system that is able to infer the intended drum pattern from an input pattern

4. Evaluation of whether this inference is correct through one or both of:

    - A human study, where users determine whether the inferred pattern is correct or not
    - A statistical study based on whether the inference model is reliable or not

4.1 Evaluation

Evaluation will take place when the core aims of the project have been completed. This subsection gives more detail to the final success criterion.

- **Human Study:** I propose that the human study will take place with a selection of moderately to skilled drummers attempting a beat on the MIDI drum kit. The machine will then infer what beat it expects the user to have been intending to play. For the purposes of this study two other sequences will be selected through some random generation upon the original inferred one. All three will be fed back to the user and it is the user's task to pick the one that they intended to play, this should give us a metric of whether the user can reliably select the inferred beat.

- **Statistical Study:** This will be regression analysis on an input into the system and whether the inference model is being consistent with its inference, i.e. is the inference model consistent with its selection of beat?

5 Resources Required

I will require the following devices:

- *My Computer:* This is a Ubuntu machine with an Intel Dual Core i5 processor and 8 GB of RAM. My contingency plans to protect myself against hardware and/or software failure include using the Git version control system, and hosting all vital files on a GitHub repository. This will mean that I can download the project onto another machine or MCS machine if I needed.

- *Roland MIDI drum kit:* This will be used for integration testing as well as gathering initial test data and evaluation. The Rainbow Research Group has agreed to allow me to use theirs.

- *Alesis USB MIDI controller* Same as above.

5.1 Tools

I will be using the following tools as part of my project:

- *Java and IntelliJ:* I will use Java to write the program and the IntelliJ IDE as my development environment for which I have my own (student) license.
- *JUnit:* JUnit will be the testing framework used to test individual components.

6 Timetable and Milestones

To allow for regular targets, the remaining time until the submission deadline has be split into 2 week work sections.

6.1 23/10/15 - 5/11/15

- Read further about beat tracking and machine learning models (this will continue throughout the project, gathering a Bibliography to use)
- Write and test code for reading MIDI data into a timestamped array
- Build a framework for project
- Read further about Drum Tablature and its relation to MIDI

Target: MIDI data can now be input into the system

6.2 6/11/15 - 19/11/15

- Translate MIDI data into Drum Tablature
- Reading about how to parse Drum Tablature

Target: MIDI data can now be input into the system and translated into drum tablature

6.3 20/11/15 - 3/12/15

- Compile a learning set from a subset of the data on the DrumTab database
- Parse the learning set into a standardised format

- Prepare the machine learning model

Target: Drum Tablature has now be translated into an accepted format ready for the machine learning model

6.4 4/12/15 - 17/12/15

- Implement the machine learning model
- Draft introduction and preparation chapters of dissertation

Target: Machine Learning Model has been created, ready for the learning set. Introduction and preparation chapters of the dissertation have been drafted and submitted to Supervisor and Director of Studies for review.

6.5 18/12/15 - 31/12/15

Work in these two weeks will be slowed because of Christmas and New Year

- Read about inference algorithms
- Draft progress report

Target: Progress report up until this point drafted

6.6 1/1/16 - 14/1/16

- Implement training algorithm for the Machine learning Model
- Test training algorithm

Target: Machine learning model trained

6.7 15/1/16 - 28/1/16

- Implement chosen inference algorithm
- Test inference algorithm
- Progress report final draft

- Prepare progress presentation

Target: Machine now able to infer beat. Core aims complete. Progress report submitted to Overseers.

6.8 29/1/16 - 11/2/16

- Buffer period for any incomplete work
- Start Evaluation

6.9 12/2/16 - 25/2/16

- Sonic Pi integration extension
- Evaluation

Target: Inferred beat now able to be exported as a Sonic Pi file

6.10 26/2/16 - 10/3/16

- Write Implementation Chapter
- Any further improvements

Target: Draft of Implementation Chapter

6.11 11/3/16 - 24/3/16

- Write Evaluation and Conclusions chapters

Target: Draft of Evaluation and Conclusions chapters

6.12 25/3/16 - 7/4/16

- Update Introduction and preparation chapters where needed

6.13 8/4/16 - 21/4/16

- Final draft of Dissertation ready for review.

Target: Final Dissertation ready for review by Supervisor and Director of Studies

6.14 22/4/16 - 5/5/16

- Buffer period to fix any problems around dissertation

6.15 6/5/16 - 13/5/16

- Submit dissertation and source code

7 Bibliography

References

[1] George Papadopoulod; Geraint Wiggins, *AI Methods for Algorithmic Composition: A Survey, a Critical View and Future Prospects*, University of Edinburgh, 1999.

[2] Masataka Goto, *An Audio-based Real-time Beat Tracking System for Music With or Without Drum-sounds*, Journal of New Music Research, 30:2, 159-171, 2001.

[3] Yipeng Cheng, *Optical Music Recognition* Part II Project, University of Cambridge, 2014