Biko Agozino

Inferring sequence specification from Drum Rhythms

Computer Science Tripos - Part II

St John's College

May 10, 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 that is able to infer the intended sequence of pattern of strokes on a drum kit from those that are captured. The system then aims at matching this sequence against possible popular rock songs that were originally performed before the year 2002 regardless of the variations in pressure and timing of each individual stroke 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 has been built to infer and extract a candidate sample pattern from MIDI performance data by examining for repeated sequences in a repeated bar performance. This sample pattern is then compared with the database and ranks all the patterns in the database. Multiple distance metrics have been implemented to assess the advantages and disadvantages of each in this domain.

A parser has been built to parse informal ASCII Drum Tablature that has been successful in parsing over 10,000 individual pattern of the notation to an appropriate format for use in the database that the system queries against.

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 perform 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, and 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;
- Extracts the rhythm from repeated play;
- 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

Machine learning approaches

Imitation based querying

Sequence matching

Much of this is discussed in the preparation, do I actually need this section?

Chapter 2

Preparation

(todo rewrite this intro when prep section is final)

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. I begin by discussing the task of building an adequate database, followed by a discussion on the possible querying methods. Finally, I end by discussing the tools that will be used to implement the project.

2.1 Starting point

todo discuss previous attempts at the project in a different form along with my background with the domain and unfamiliarity with the computer science techniques?

2.2 Database collection

Clearly, 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.

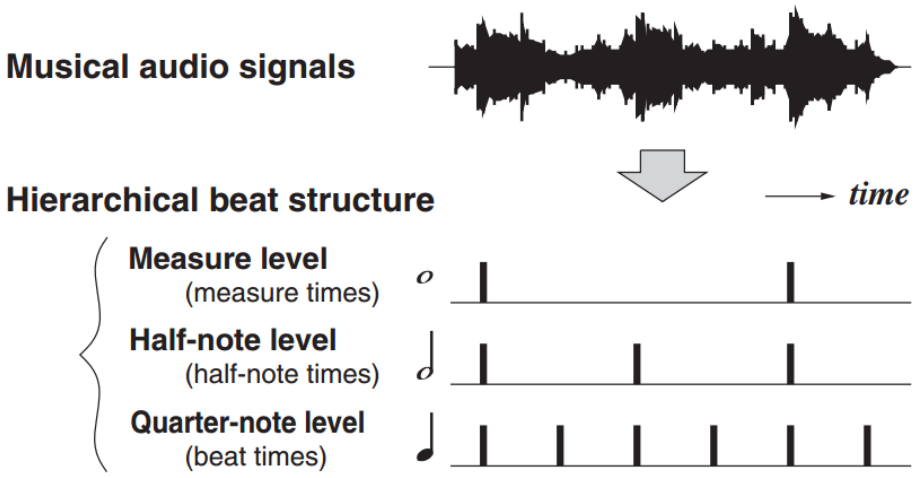

Figure 2.1: Shown here is the aim of beat tracking is to transform musical audio signals into a representation of the beat structure[2]

2.2.1 Beat tracking techniques

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 only limited to how many songs have been released in an acceptable format. In Goto's case this was compact disk's 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 coefficient, which is a measure of 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].

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

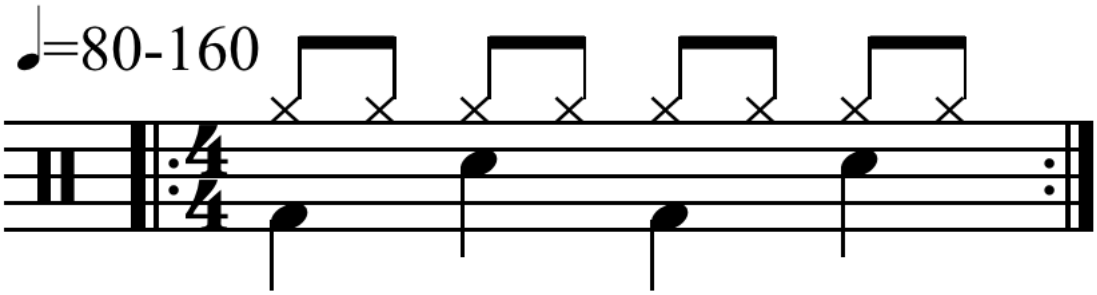

Figure 2.2: Shown here is the standard percussion notation for a popular rhythm often found in rock music

ASCII Drum Tablature Parsing

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].

[1]Music Optical Character Recognition, sometimes referred to as Optical Music Recognition
[2]http://drumbum.com/drumtabs

```
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

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 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 will have to account for this as the collaborative database will not be 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 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 be no problem writing a parser that can extract the rhythms of a subset of these songs.

2.3 Querying

Now that the data source has been decided, it is important to discuss how the querying system will work. I will start by outlining methods for inferring the bar from a sequence of beats on the drum. I will then discuss the possible ways of using this sequence to query the database.

2.3.1 Suffix trees

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. The problem with this, as discovered after some initial data collection (see Evaluation 4.4), was that the latency of the system caused by start up and shut down caused the example to be distorted. In order to tackle this problem I decided it was wise to design a system that could extract repeated structures from continuous play of the same bar.

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.

Exploring algorithms for Suffix tree construction

todo talk about the trivial O(n2) algo

todo talk about weiners algorithm

todo contrast the three

Ukonnens algorithm[19] is an on-line solution to suffix tree construction. This is important for the implementation as for the system to work in real time the suffix tree building must

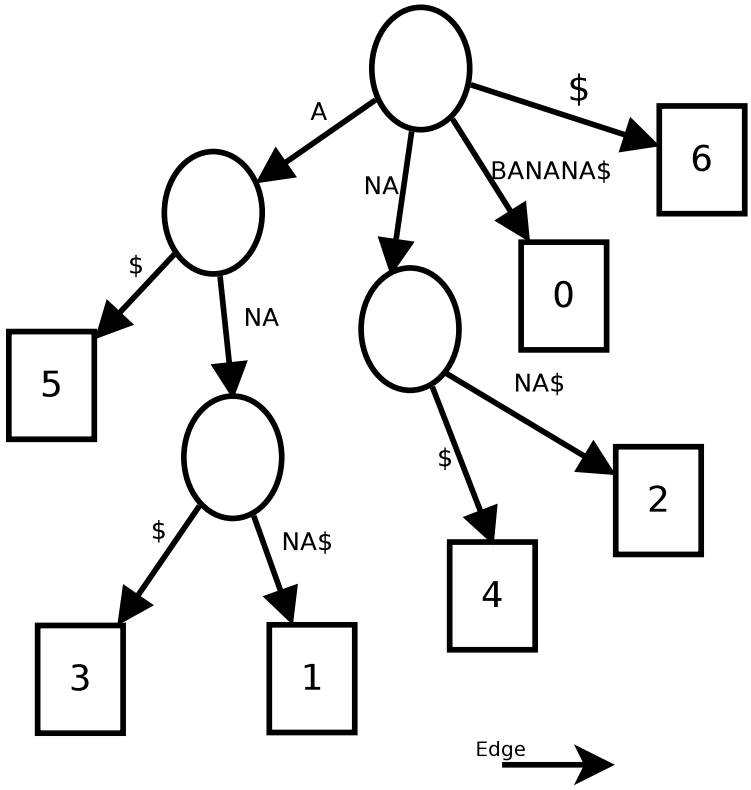

Figure 2.4: Suffix tree of the string "BANANA$"

efficiently work in real time. As shown in 3.3.2 the algorithm works in both linear time and space if run off-line which is optimal.

2.3.2 Matching techniques

If we assume both the parser and 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 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[**?**], for matching two-dimensional stings.

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 there is a good chance that the pattern inference will not be exactly correct, there is a good chance that nothing will be returned. 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.

2.4 Changes from proposal

todo

2.5 Tools Used

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

2.5.1 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 the 16 channels. Each message is one byte of data, circumfixed by a start and stop bit, transmitted serially in the format shown in figure 2.5.

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.

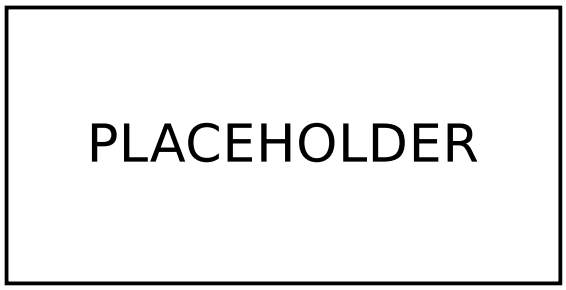

Figure 2.5: todo describe diagram of midi message

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

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

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 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 rounded to some value (see Evaluation for a discussion on what is the best value to round to).

todo list the drum components available

2.5.2 Misc

Java: The majourity of this project will be 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.

Google Tree Multiset: A package provided by Google's Guava-libraries which will allow for efficient ordering of a set of values, allowing for duplicates. I will be using this package to store the results of the distance metric matching.

2.6 Requirement Analysis

todo

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

2.7 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

todo rewrite

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.

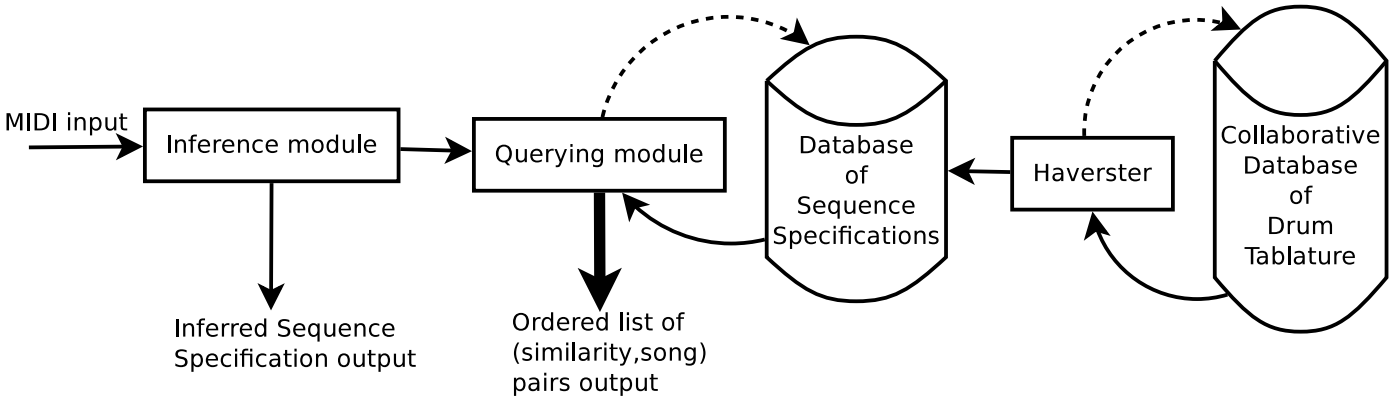

Figure 3.1: Here is a modular view of the system, solid lines represent flow of information, while the dotted lines indicate a look-up request

-parser -midi input -bar extraction -matching

3.2 Harvester

As outlined in the 2.2.3 the source for the database will 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.

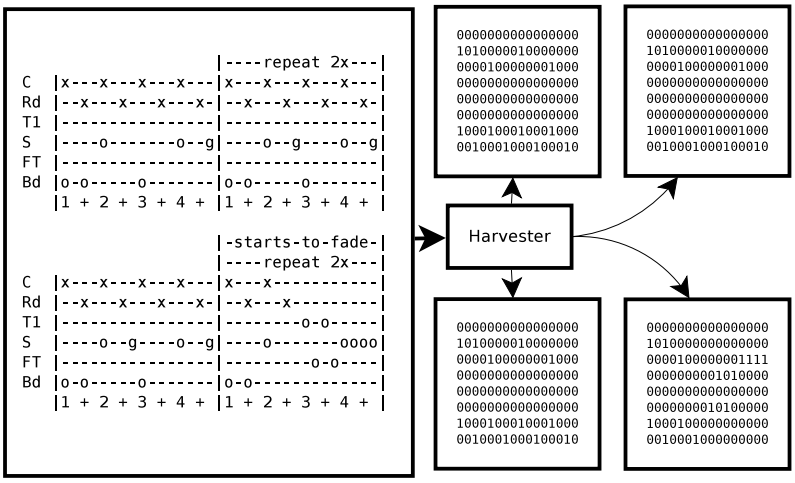

Figure 3.2: Here is an example of how the system will translate and format the drum tablature - creating Sequence Specifications that are saved in the database

When analysing the system the parser

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

todo flowchart?

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 highhat 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 high hat, is matched at a later stage. Additionally the *TrackDivider* token is used in order to provide context 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
- Whitespace - representing any whitespace in the string

Both TrackDivider and NewLine are used to provide context at a later stage, while the Whitespace was not used at any point and was only included orriginally for completeness.

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 $O(n)$.

todo perhaps include full lexing code in the appendix?

3.2.2 Pre-parser

In the aptly named 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

[1] Meaning providing lexical analysis for

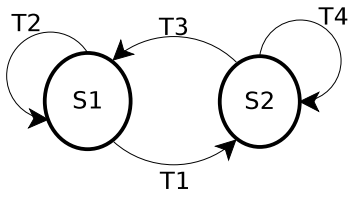

Figure 3.3: Finite State Machine for comment removal, see **Comment removal** for an explanation of the states and transition functions.

no complicated processing tasks and is merely given the job of organising the data in 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. The implementation of this can be foind in the appendix todo.

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 recursively 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.

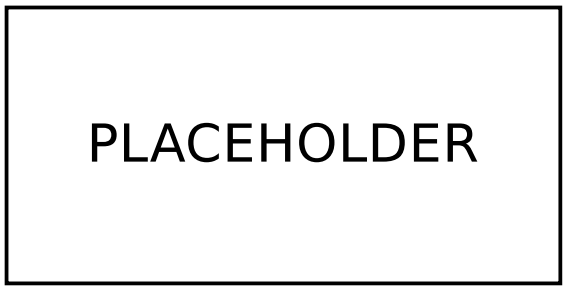

Figure 3.4: Here depicts the structure of a

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. todo possible diagram?

3.2.3 Parsing

Due to all this pre-precessing 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.

Before I outline the method for parsing the pre-processed sequences I must first outline the structure the data will be stored in.

Sequence Specifications

I introduce the concept of sequence specifications. These specifications represent a simplified view of a drum rhythm, depicted in figure 3.5.

Each specification has a constant 8 rows, each mapping to one of the 8 drum components discussed in 2.5.1. 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

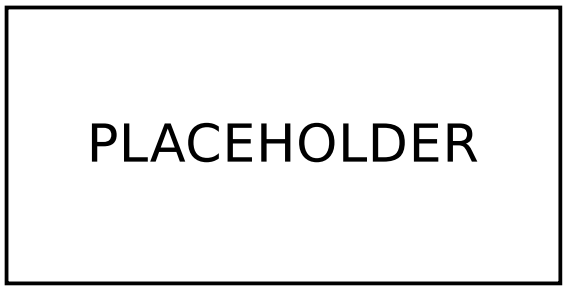

Figure 3.5: Here depicts the structure of a

that have 16 time divisions, reasons and consequences for doing so will be discussed in the Evaluation todo section?

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

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

This simplified notation therefore only depicts the *relative* timing information of the system and a lot of information is removed including power, special notes, and multiple beats within one time division are abstracted away, which again will be discussed in detail in the Evaluation chapter.

Parsing to sequence specifications

The job of the parser is now to create these sequence specifications based on the grouped data that was 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. Due to the way the nature of drum tablature (see todo relevent section) 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.

3.3 Inference Module

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

todo make into flowchart?

- Split the recording into eight note channels representing each drum component
- Calculate the time deltas between each drum beat on each channel
- Form a suffix tree, for each note channel, using these time deltas (rounded to some increment) as the alphabet
- Find the repeated sequence that fits best
- Extract the inferred pattern
- Rank the patterns in the database in order of best fit
- Return both the inferred pattern and this ordered list of matches to user

The first two steps are justified and explained in 2.5.1. The successive steps are explained in detail here.

3.3.1 Suffix Tree's

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 Tree's

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

- A tree with exactly n leaves labelled 0 to $n-1$
- Internal nodes[2] have at least two children
- Each edge is labelled with a non empty sub-string of S
- The starting character of the label of each edge coming out of a node is unique
- 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

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 $(n+1)+n+1 = 2n+2$ (todo double check this) nodes in total.

However, this definition requires $O(n^2)$ space due to the labelling of edges with the sub-strings (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: Ukonnens 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[3] 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.

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

[3]Meaning at most one step per character

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 ther 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 atleast two children. This is depicted in figure 3.6

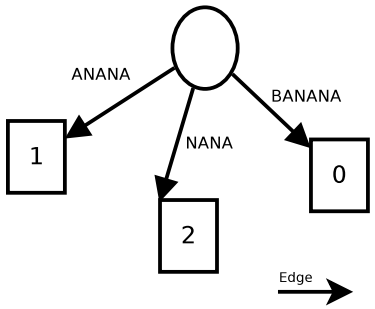

Figure 3.6: Here is the implicit suffix tree for the string BANANA. For a complete suffix tree refer to 2.4

Further to this, a 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*. This implicity happens when one of the suffix's is prefixed by another, leading to only one path where there would otherwise be two. This is important when we define the 'extendTree(int i)' function later, but for now just keep in mind that each step, i, of the algorithm constructs the implicit suffix tree T_i.

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[**?**] definitions of the concepts.

Suffix extension rules

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

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).
```

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 (todo point to this discussion) and then extends that 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. (todo diagram?)

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 orriginally detailed by Ukkonen [19].

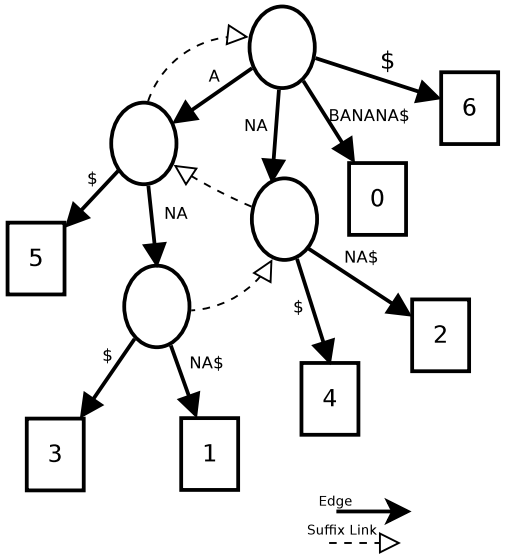

Figure 3.7: Example of the familiar BANANA$ string, shown first at 2.4, now with the suffix links explicitly shown with the dotted lines

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 (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. (todo may need to

rewrite this final paragraph)

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.

(todo if I rewrite the final paragraph of the suffix link disscussion if may be apt to include here a discussion of how efficient the algorithm is now - ie $O(n^2)$)

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 siffix 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*¿)[4]

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

Active Points

possibly discuss this concept, though implicitly discussed earlier

Complexity of algorithm

possibly discuss if needed. Maybe more appropiate in the Performance analysis section of the evaluation

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[5] corresponds to a repeated sequence. For this I implemented a simple DFS[6] 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.

[4]Of course in Java this pointer is implemented by shared references to an object that acts as an *int* wrapper

[5]Internal paths are paths starting at the root and ending at an internal node

[6]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
```

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 deeps 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 repeat 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 seen to work well in practice when we look at the bar inference step in 3.3.5.

3.3.5 Bar inferrence

Recall that we split the MIDI recording into 8 sequences (3.3) each representing one of the 8 componenets 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[7]. 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 $startTime = endTime - (repeatedStructure.length)$.

[7]defined as (length of the recording)/(Expected number of repeats)

It is then a simple matter of populating the familiar Sequence datastructure defined in 3.2.3 to complete the task of the inference module.

3.4 Distance metrics

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 Google's implementation of the *Tree Multiset* datastructure which was discussed in 2.5.2.

This module assumes the availablility of a database of Sequence 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 appropiate to consider each row of the strings as seperate to the others. Therefore the overall distance of the drum patterns are 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 (todo finish comparison).

(todo possibly more detailed explanation of how WagnerFischer works including diagrams and the works)

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

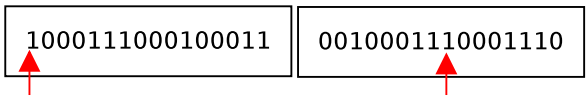

Figure 3.8: Example of two patterns that have a cyclic distance of 0 are distant when seen in isolation. The red arrow notates the start point of the cycle.

of the rhythms continiously it would be unintelligible from the other, due to that the 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 (todo insert what the edit and hamming distances would be).

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 modulu 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

Discussion on how each module can be evaluated individually

4.1 Comparison with original aims

compare with requirement analysis section of prep

4.2 Dataset collection

Discussion on how the HCI study was conducted

4.3 Evaluating the Havesting module

Discus where the parser loses information

4.3.1 Information lost

As discussed in 2.2.3 ASCII Drum Tablature has a lot of superfolous 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.

4.4 Evaluating the Inference module

repeated sequence vs single recording computing the average distance from the actual sequence and computing a t-test to see if they are significantly different

add in repeated with experimentation and discuss results

analyse the results.

discuss choice of rounding factor

todo talk about information lost in highhat

talk about using sibling lists in suffix trees and alphabet size

4.5 Evaluating the Querying module

maybe graph of how long it takes to return values for each metric

discus how there is no indexing and that this could improve performance

discus the difficiency of when a pattern is just 2x another pattern

4.6 Performance Analysis

talk about the performance of the harvester

4.7 Summary

Chapter 5

Conclusion

Discussion on how each module can be considered a seperate diliverable, with the entire system adding to the value

5.1 Lessons learnt

Importance of prior research, prototyping, and planning

5.2 Future work

How to improve the project, probabalistic stuff

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.