An Efficient Score Alignment Algorithm and its Applications

by

Emily Zhang

B.S., Massachusetts Institute of Technology (2015)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2017

Author . . . .
Department of Electrical Engineering and Computer Science
May 18, 2017

Certified by . . . .
Prof. Michael S. Cuthbert
Associate Professor
Thesis Supervisor

Accepted by . . . .
Prof. Christopher J. Terman
Chairman, Master of Engineering Thesis Committee

An Efficient Score Alignment Algorithm and its Applications

by

Emily Zhang

Submitted to the Department of Electrical Engineering and Computer Science
on May 18, 2017, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

String alignment and comparison in Computer Science is a well-explored space with classic problems such as Longest Common Subsequence that have practical application in bioinformatic genomic sequencing and data comparison in revision control systems. In the field of musicology, score alignment and comparison is a problem with many similarities to string comparison and alignment but also vast differences. In particular we can use ideas in string alignment and comparison to compare a music score in the MIDI format with a music score generated from Optical Musical Recognition (OMR), both of which have incomplete or wrong information, and correct errors that were introduced in the OMR process to create an improved third score. My thesis focuses on creating a set of algorithms that align and compare MIDI and OMR music scores to produce a corrected version of the OMR score that borrows ideas from classic computer science string comparison and alignment algorithm but also incorporates optimizations and heuristics from music theory.

Thesis Supervisor: Prof. Michael S. Cuthbert
Title: Associate Professor

Acknowledgments

This is the acknowledgements section. You should replace this with your own acknowledgements.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Comparing music is seminal to the field of Music Information Retrieval (MIR). Questions such as “Is this piece of music plagiarized from Bach?", “Is this rhythmically more similar to Philip Glass or Scott Joplin? Melodically?", “Is this newly discovered musical transcription an entirely new folk song or just a variation of an old one?" can be answered by comparing the pieces in question and applying a quantitative metric to determine that, “Yes, this piece quotes Bach more than other pieces of similar time period and style", or, “This piece has 74% melodic similarity to Joplin's repertoire, 10% similarity to Glass, but rhythmically shares 64% similarity with Glass and only 12% with Joplin."

Computational music comparison, a subset of classical computer science string comparison, comes in many forms and each domain has slightly different specifications for alignment and comparison.

The domain of music comparison that I will be studying and building tools for will be comparison between a MIDI score, which represents music in a restricted manner and its OMR counterpart, possibly a scanned score. Each representation is imperfect on its own.

MIDI represents music in a very limited manner. This format encodes enharmonic notes as the "same"; note lengths are relative to each other, not necessarily in relation to a defined time signature; there is freedom in representing duration, as long as the total length is correct; and many other design decisions that make MIDI very universally compatible but also lacking in definition. Furthermore, since MIDI scores are generally created with some kind of human input, they tend to be correct, pitch-wise and rhythmically.

OMR takes a score that likely to be correct (the idea being this is a printed and published work, and also likely proofed for errors by the composer or publisher), scans its and tries to recreate what it scans in a digital notation format. While the more commonplace OCR has been optimized and whittled to an extremely high standard, OMR is still at present only 90% correct [?]. One can imagine that the "alphabet" that OMR deals with is much larger and less well-defined than the 52 capital and lowercase letters and punctuation that OCR has been optimized for. Therefore, much of the optimizations for OCR do not directly apply to OMR. Some common errors in OMR include incorrectly recognizing a sharp as a natural and vice versa, mistaking rests as notes and vice versa, and other errors that any musician trained in the basics of music theory could identify as "wrong". Simple rules such as: all measures must sum to the same duration, only notes with existing accidentals can be made natural, should be able to great improve the quality of OMR if taken into account.

The idea here is that we can compare MIDI scores and OMR scores, properly align them for as many points in the music as possible, and then use the correct metadata present in the MIDI to score to correct for scanning errors in the OMR score.

1.2 Project Overview

1.3 Chapter Summary

Chapter 2 will discuss background information that is important to understanding my work, including basic musical terms, an overview of canonical sequence alignment methods, as well as related work in musical sequence representation, alignment, and correction. Chapter 3 will summarize the work I did previously that is used as a building block for my thesis work. It also briefly discusses some ideas that were not as promising to explore. Chapter 4 will be a deep dive into the final implementations of the three main modules of my work and provides a concrete example of how the entire system would work. Performance and testing metrics are discussed in Chapter 5. Finally, Chapter 6 concludes with areas to explore in future work.

Chapter 2

Background and Related Work

2.1 Musical Terminology

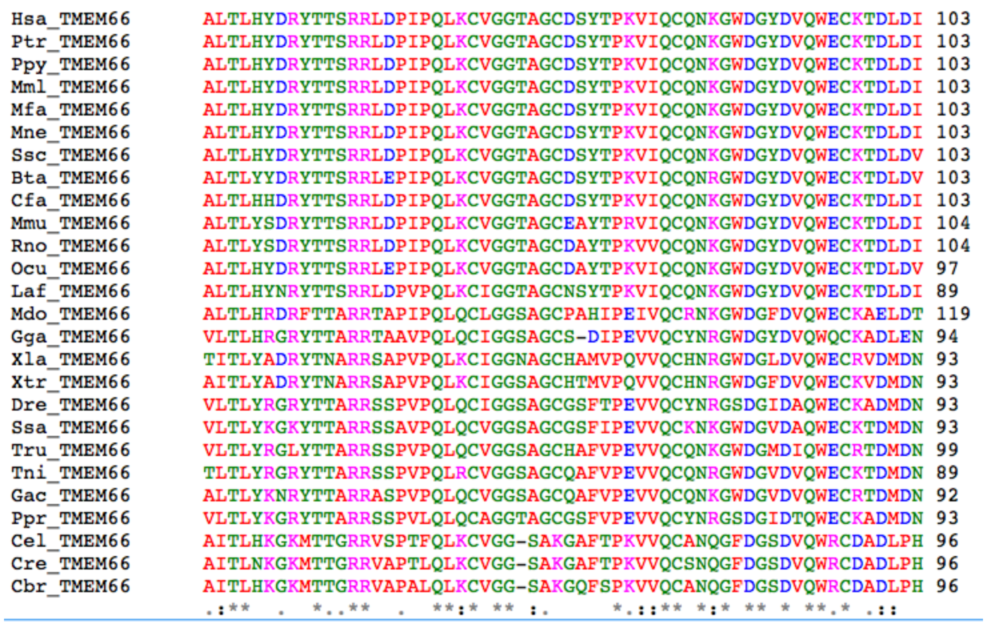
2.2 Non-musical Work in Sequence Alignment

It is far easier to understand sequence alignment in a non-musical context (here the context is plain string alignment), understand the rules and assumptions made in the non-musical context, generalize them, and then recreate music-specific rules and assumptions.

If you are familiar with the canonical representation of the string alignment problem as a dynamic programming matrix, you can skip this section.

2.2.1 An Overview of Sequence Alignment

Sequence alignment in bioinformatics is an application of well-studied problem. In this field, researchers want to identify similar regions of DNA, RNA, or protein sequences to study functional, structural, or evolutionary relationships between the two sequences.



```
Hsa_TMEM66       ALTLHYDRYTTSRRLDPIPQLKCVGGTAGCDSYTPKVIQCQNKGWDGYDVQWECKTDLDI 103
Ptr_TMEM66       ALTLHYDRYTTSRRLDPIPQLKCVGGTAGCDSYTPKVIQCQNKGWDGYDVQWECKTDLDI 103
Ppy_TMEM66       ALTLHYDRYTTSRRLDPIPQLKCVGGTAGCDSYTPKVIQCQNKGWDGYDVQWECKTDLDI 103
Mml_TMEM66       ALTLHYDRYTTSRRLDPIPQLKCVGGTAGCDSYTPKVIQCQNKGWDGYDVQWECKTDLDI 103
Mfa_TMEM66       ALTLHYDRYTTSRRLDPIPQLKCVGGTAGCDSYTPKVIQCQNKGWDGYDVQWECKTDLDI 103
Mne_TMEM66       ALTLHYDRYTTSRRLDPIPQLKCVGGTAGCDSYTPKVIQCQNKGWDGYDVQWECKTDLDI 103
Ssc_TMEM66       ALTLHYDRYTTSRRLDPIPQLKCVGGTAGCDSYTPKVIQCQNKGWDGYDVQWECKTDLDV 103
Bta_TMEM66       ALTLYYDRYTTSRRLEPIPQLKCVGGTAGCDSYTPKVIQCQNRGWDGYDVQWECKTDLDV 103
Cfa_TMEM66       ALTLHHDRYTTSRRLDPIPQLKCVGGTAGCDSYTPKVIQCQNKGWDGYDVQWECKTDLDI 103
Mmu_TMEM66       ALTLYSDRYTTSRRLDPIPQLKCVGGTAGCEAYTPRVIQCQNKGWDGYDVQWECKTDLDI 104
Rno_TMEM66       ALTLYSDRYTTSRRLDPIPQLKCVGGTAGCDAYTPKVVQCQNKGWDGYDVQWECKTDLDI 104
Ocu_TMEM66       ALTLHYDRYTTSRRLEPIPQLKCVGGTAGCDAYTPKVIQCQNKGWDGYDVQWECKTDLDV 97
Laf_TMEM66       ALTLHYNRYTTSRRLDPVPQLKCIGGTAGCNSYTPKVIQCQNKGWDGYDVQWECKTDLDI 89
Mdo_TMEM66       ALTLHRDRFTTARRTAPIPQLQCLGGSAGCPAHIPEIVQCRNKGWDGFDVQWECKAELDT 119
Gga_TMEM66       VLTLHRGRYTTARRTAAVPQLQCIGGSAGCS-DIPEVVQCYNRGWDGYDVQWQCKADLEN 94
Xla_TMEM66       TITLYADRYTNARRSAPVPQLKCIGGNAGCHAMVPQVVQCHNRGWDGLDVQWECRVDMDN 93
Xtr_TMEM66       AITLYADRYTNARRSAPVPQLKCIGGSAGCHTMVPQVVQCHNRGWDGFDVQWECKVDMDN 93
Dre_TMEM66       VLTLYRGRYTTARRSSPVPQLQCIGGSAGCGSFTPEVVQCYNRGSDGIDAQWECKADMDN 93
Ssa_TMEM66       VLTLYKGKYTTARRSSAVPQLQCVGGSAGCGSFIPEVVQCKNKGWDGVDAQWECKTDMDN 93
Tru_TMEM66       VLTLYRGLYTTARRSSPVPQLQCVGGSAGCHAFVPEVVQCQNKGWDGMDIQWECRTDMDN 99
Tni_TMEM66       TLTLYRGRYTTARRSSPVPQLRCVGGSAGCQAFVPEVVQCQNRGWDGVDVQWECKTDMDN 89
Gac_TMEM66       ALTLYKNRYTTARRASPVPQLQCVGGSAGCQAFVPEVVQCQNKGWDGVDVQWECRTDMDN 92
Ppr_TMEM66       VLTLYKGRYTTARRSSPVLQLQCAGGTAGCGSFVPEVVQCYNRGSDGIDTQWECKADMDN 93
Cel_TMEM66       AITLHKGKMTTGRRVSPTFQLKCVGG-SAKGAFTPKVVQCANQGFDGSDVQWRCDADLPH 96
Cre_TMEM66       AITLNKGKMTTGRRVAPTLQLKCVGG-SAKGAFTPKVVQCSNQGFDGSDVQWRCDADLPH 96
Cbr_TMEM66       AITLHKGKMTTGRRVAPALQLKCVGG-SAKGQFSPKVVQCANQGFDGSDVQWRCDADLPH 96
                 .:**  .  *..**  .  **:* ** :.     *.::** *:* ** * **.* .::
```

Figure 2-1: An example of protein alignment

[[figure out how to cite this image: `https://commons.wikimedia.org/wiki/File:An_excerpt_of_`

There are various computational algorithms that can be used to solve sequence alignment including slower but more accurate dynamic programming algorithms and faster but less accurate probabilistic or heuristic-based algorithms. In my work, it makes more sense to use the slower but more accurate methods, as our datasets are not so large and small mistakes that might be permissible and overlook-able in larger data could be much more egregious in smaller contexts.

The specific algorithm I chose to implement for solving the musical sequence alignment problem is a version of the Needleman-Wunsch dynamic programming algorithm for global sequence alignment. [1]

At a high level, the Needleman-Wunsch algorithm and its family of dynamic programming sequence alignment algorithms calculate the edit distance between two sequences by producing a least-cost alignment of the sequences. An alignment is an assignment of pairwise characters and edit operations. Overall cost of an alignment (i.e. its **edit distance**) is determined by summing the individual costs of insertion,

[1]This algorithm has a history of multiple invention. Needleman-Wunsch was first published in the context of protein alignment in bioinformatics. People with a more theoretical computer science bent might know the same algorithm by a different name, Wagner-Fisher.

deletion, substitution, or no-change operations between pairwise characters in order to morph one sequence into the other.

The next sections discuss coming up with edit distance calculations and then the method that Needleman-Wunsch uses to find the best alignment.

2.2.1.1 Edit Distance

The term edit distance is a way of quantifying how dissimilar two sequences are to each other. If you think of sequences as being made up of characters, you can also think of one sequence transforming into the other through a series of insertion, deletion, substitution, and no-change operations, with each operation "costing" a certain amount. Edit distance is the smallest possible total cost associated with the transformation that is a series of operations that turns one sequence into another. As an example, consider the calculation of the edit distance between the two strings **MUSICAL** and **JUSTICE** that uses this series of operations for transforming **MUSICAL** into **JUSTICE**:

1. M $\rightarrow$ J (substitution)
2. $\epsilon \rightarrow$ T (insertion)
3. A $\rightarrow$ E (substitution)
4. L $\rightarrow \epsilon$ (deletion)

Note that implicitly I am excluding all the no-change operations that look like this:

1. U $\rightarrow$ U (no-change)

This corresponds to an alignment that looks like this. The green pair of characters indicate an insertion operation, red is deletion, and purple is substitution. These colors are also used in the visualization of OMR/MIDI stream alignment.

Operation	**Cost**
Insertion	1
Deletion	1
Substitution	1
No-change	0

Table 2.1: Naive Operation Cost Function Table

seq1	**M**	**U**	**S**	_	**I**	**C**	**A**	**L**
seq2	**J**	**U**	**S**	**T**	**I**	**C**	**E**	_

Figure 2-2: Alignment of MUSICAL and JUSTICE

You would say that the two strings have an edit distance of 4, because it requires four operations to completely morph one string into the other (discounting no-change operations).

However, you also might think that a substitution should ”cost” more than either an insertion or deletion; after all, a substitution can be thought of as a consecutive insertion and deletion operation.

A slightly different cost function that assigns more cost to substitution changes could look like this. Using this cost function, our edit distance of changing **MUSICAL** into **JUSTICE** is 5.

Operation	**Cost**
Insertion	1
Deletion	1
Substitution	1.5
No-change	0

Table 2.2: Operation Cost Table With Costlier Substitutions

But wait, should certain subtypes of a single type of operation have differently weights? Consider if you were trying to run a spell checker (one application of string alignment and correction algorithms) on a word that wasn't recognized by the dictionary. In your particular spell check model, maybe substitution of a vowel for a vowel isn't as costly as any other types of substitution. Let's say that substituting a vowel for a vowel should only have a cost of 1.2. Using this new cost function, our new edit distance is 4.7.

Operation	**Cost**
Insertion	1
Deletion	1
Substitution - vowel for vowel	1.2
Substitution - all others	1.5
No-change	0

Table 2.3: Operation Cost Table With Different Substitution Costs

2.2.1.2 Alignment

Recall that the edit distance between two sequences come from the least-cost alignment. The figure below shows a valid alignment, but it is certainly not the least-cost alignment, and therefore does not correspond to the edit distance between between the two sequences.

seq1	_	**M**	**U**	**S**	**I**	**C**	**A**	**L**
seq2	**J**	**U**	**S**	**T**	**I**	**C**	**E**	_

Figure 2-3: A bad alignment of MUSICAL and JUSTICE

If we have two sequences, of length n and m, then we have $\binom{n+m}{m}$ possible alignments. Put into context, consider a two melodies of 10 notes each. Naively aligning just the note pitches, we have 184,765 different possible alignments! Clearly this isn't scalable, which is why we rely heavily on this piece of insight: A globally optimal alignment contains locally optimal alignments! This means that we can reuse the computations of subproblems to solve larger problems. What follows from this piece of insight is that for two strings split at (i, j), the best alignment is:

$$\text{best alignment of } \texttt{string1}[:i] \text{ and } \texttt{string2}[:j]$$
$$+ \text{ best alignment of } \texttt{string1}[i:] \text{ and } \texttt{string2}[j:]$$

If we can keep track of all the possible alignment subproblems, we can recursively build an optimal solution using the answers from the subproblems. One way to represent this is with a scoring matrix indexed by i and j. For our problem of aligning **MUSICAL** and **JUSTICE**, the setup would look something like this:

$$\begin{array}{lccccccccccccc}
 & & & & i & & & & & & & & & \\
 & & & & \downarrow & & & & & & & & & \\
\texttt{seq1} & \mathbf{M} & & \mathbf{U} & & \mathbf{S} & & \mathbf{I} & & \mathbf{C} & & \mathbf{A} & & \mathbf{L} \\
\texttt{seq2} & \mathbf{J} & & \mathbf{U} & & \mathbf{S} & & \mathbf{T} & & \mathbf{I} & & \mathbf{C} & & \mathbf{E} \\
 & & & & & & & & & & \uparrow & & & \\
 & & & & & & & & & & j & & &
\end{array}$$

Figure 2-4: The alignment of MUSICAL split at i and JUSTICE split at j

[[much more here on the steps of alignment. 2) populating distance matrix 3) backtracing from bottom right corner using the cost function we have previously defined]]

	-	**J**	**U**	**S**	**T**	**I**	**C**	**E**
-								
M								
U								
S								
I								
C								
A								
L								

Figure 2-5: Initial distance matrix setup for aligning MUSICAL and JUSTICE

Recall that we have defined our problem in three different ways:

1. Aligning **MUSICAL** and **JUSTICE**

2. Finding the edit distance between **MUSICAL** and **JUSTICE**

3. Finding a series of operations that transforms **MUSICAL** into **JUSTICE**

All of these problem definitions are (almost) one and the same. Calculating the edit distance will also provide a good alignment of the two words and a good alignment necessitates a series of character change operations. It is important to know that that (3) has an implicit directionality built into the problem statement. It won't always be the case that alignments and transformations are symmetric (i.e. cost the same and have the same series of operations), and having a simple cost operation functions, such as in tables and will make such problems symmetric.

In our particular sample problem, we will always be transforming **MUSICAL** into **JUSTICE**. In the context of my thesis, we will always be transforming OMR sequences in MIDI sequences.

After we've set up is the scoring matrix, we will populate it with initial values:

1. 0 in $(0, 0)$
2. $i - 1 \cdot insertioncost$ for i in $(i, 0)$ (this is the first column)
3. $j - 1 \cdot deletioncost$ for j in $(0, j)$ (this is the first column)

[[somehow there needs to be an explanation that insertions are vertical movements, deletions are horizontal movements, and this is all relative to how you define the problem of transforming X->Y or Y->X]]

After we've set up the initial matrix, we can go through and fill in all of the remaining slots using this update rule:

[[update rules here– find the min (moving up + insertion, right + deletion, diagonal + subst)]]

2.2.2 Formal Definitions in Sequence Alignment

This section is a more mathematical and formal guide to sequence alignment in a matrix representation that uses the cost functions we have been slowly refining.

[[this section might not be necessary if the previous sections are rigorous enough...]]

[[no, there's a really cool way of explaining the set up of the problem in space transformation language... see notes!]]

2.3 Turning Sequence Alignment into a Musical Problem

Finding the edit distance of a musical sequence as opposed to a string of ASCII characters is a little trickier. First, there isn?t an intuitive ”space” of music elements the way that there is a ”space” of all characters. Second, even after identifying the musical elements, there isn?t an intuitive metric of distance between elements. For example, how would the distance between [Ab quarter note, 4th octave] and [Bb quarter note, 4th octave] compare with [Ab either note, 4th octave] and [Ab quarter note, 4th octave]? [pictures, maybe an even more ambiguous example]

Going back to [original model] and the questions we posed and answered for string alignment,

- What sequences are and what makes them up: We can think of musical sequences as an ordering of notes, rests, and chords

- The space that sequences occupy: This question can roughly be approximated as, what properties of notes, rests, chords do we care about? Some immediate answers are pitch, duration, rhythm.

- Coming up with an appropriate metric for finding distance: In the previous section we had two different substitution functions. What kind of and how many ways do we want to calculate distance in musical elements?

2.4 Related Work

2.4.1 Typke: Music Retrieval based on Melodic Similarity

[[re-edit to talk about feature extraction as an important step, even though i don't necessarily call it feature extraction]] Another instance in which music is transformed and compressed into its most relevant features before comparison is in Typke?s thesis on Music Retrieval based on Melodic Similarity [**?**]. Here, Typke proposes encoding music into what he calls a ?weighted point set? in a two-dimensional space of time and pitch before measuring any sort of melodic similarity between music. A note in a piece is represented as point set of pitch (in the Hewlett 40 system [**?**]) and an offset in quarter notes from the beginning of the piece.

2.4.2 Church and Cuthbert: Rhythmic Comparison and Alignment

2.4.3 Viro: Peachnote, Musical Score Search and Analysis and IMSLP

2.4.4 White: Statistical Properties of Music and MIDI

White's dissertation [?] describes properties of music that can be turned into heuristics for refining algorithmic analysis of music. Specifically, he builds upon Temperley's 1997 [?] paper's details to harmonic analysis of MIDI and refers to other research that has been constructive towards Temperley's model. White does say that Temperley's algorithm fails when faced with ambiguity, such as in MIDI differentiating between enharmonic tones.

The hope is that by combining the ambiguous parts of MIDI with the probabilistic and rigorous details of OMR scores, ambiguities in MIDI tonality can be resolved. In addition, White also cites a large repository of high quality MIDI files.

2.4.5 Rebelo et al: Optical Music Recognition - State-of-the-Art and Open Issues

Chapter 3

Previous Work

The work of my thesis uses the universal Hasher for digital representations of music tool. This work was largely done in the Spring of 2015 as part of my 6.UAP. Additionally, experimental results in a related project focused on speeding up musical comparison by converting Python objects to C objects provide evidence that the bottleneck in musical sequence alignment is not in the comparison step. Both subjects are described in more detail below but only to the extent necessary to understand the work of my thesis.

3.1 A Modular Universal Hasher

Prior to my work on building a musical sequence aligner and fixer, I built a hasher with the specific intention that it would be able to adapt according to different specification settings decided by the programmer. The programmer decides which kinds of music21 stream elements and properties important to their hash, and selects them to build a specific instance of a Hasher object. This Hasher instance can then be used to produce a hash from any music21 stream.

3.1.1 Why Musical Comparison and Alignment Needs a Modular Hashing Function

A hash function maps pieces of data of arbitrary size into data of fixed size and standard representation. If musical sequences could be discretized into elements and if these elements could be hashed, then the processes of comparison and alignment could be made deterministic and quick.

Depending upon the problem statement that calls for a hash in the first place, the details of this two step process:

1. discretizing a musical sequence into a list of relevant elements

2. hashing those elements can vary

For instance, in determining the the rhythmic similarity between two musical sequences, only the musical elements with durations (e.g. notes, chords, rests) in each sequence would be relevant. In hashing these elements, only properties such as duration and offset might only be relevant.

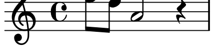


Figure 3-1: The hash of rhythmic features of this short melody would be *NoteHash(offset=0, duration=0.5), NoteHash(offset=0.5, duration=0.5), NoteHash(offset=1.0, duration=2.0) NoteHash(offset=4, duration=1.0)*

Another example instance in determining whether one piece was an approximate transposition of another, only the elements with some relation to pitch would be relevant (e.g. notes and chords) would be relevant.

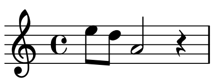


Figure 3-2: The hash of features related to transposition of this short melody would be *NoteHash(intervalFromLastNote=0), NoteHash(intervalFromLastNote=2), NoteHash(intervalFromLastNote=5)*

Once musical sequences are discretized and represented in a hash sequence, each individual tuple in a hash sequence can be thought of as a discrete point in the space of all combinations of possible element properties. Then, canonical edit distance or string alignment algorithms could be used to come up with a measure of similarity or an alignment between the two original musical sequences.

3.1.2 Previous Work on Modular Hashing Functions

The idea of extracting qualities of music specific to the needs of distance analyses and encoding that information in a lower-dimensional and representation is not new. Typke in his thesis proposes what he calls a weighted point set representation of music [1]. His method of creating a weighted point set for a stream of music involves only taking melodic information (i.e. notes) and representing each discrete note as a three-tuple of *(*time, pitch, weight). The time component encodes information about the note's offset from the beginning. The pitch component encodes the note's pitch using the Hewlett's base-4 system (this is a numerical encoding similar to MIDI pitch but is also able to encode more information than just absolute frequency). The weight component is an assignment of how important the particular note is to the melody of the music stream.

3.1.3 Overview of System

The hashing system is found in `hasher.py`. It takes in a musical stream and outputs a hash based based on what hashing settings are specified. The user will first initialize the hasher to the appropriate settings, and then call the `hashStream` method, the core of the Hasher system. The `hashStream` method strips the stream only to the elements that are to be hashed, creates a dictionary of hash functions to use based on the user settings, and individually hashes each element with the entire set of specified hashing functions.

The final output is stored in `finalHash`.

3.1.3.1 Pre-Hashing: Initializing the Hasher

3.1.3.2 Pre-Hashing: High Level Hasher Settings

- NoteHash vs NoteHashWithReference

3.1.3.3 Pre-Hashing: Low Level Hasher Settings

- General Low Level Hashing Settings These settings affect more than one of notes, rests, chords.

  - `_hashOffset` - If True, this will include in the hash a number that is associated with the offset of a musical element from the beginning of the stream.
  - `_hashDuration` - If True, this will include in the hash a number that is associated with its duration
  - `_roundDurationAndOffset` - This is a boolean for determining whether offsets and durations should be rounded. This is useful for files (e.g. MIDI recordings without rounding) where beginnings of notes, rests, and chords may not fall exactly on the beat (e.g. beat 2.985 instead of 3)
  - `granularity` - A number that determines how granularly we round. If duration and offset are rounded, they will be rounded to this granularity. For example, a granularity of 32 will round notes to the nearest 32nd beat division and no further.

- Note Low Level Hashing Settings

  - `_hashPitch` and `_hashMIDI` - These settings determine whether pitch of a note will be hashed, and if it is, how it will be represented. If `hashMIDI` is `True`, then the MIDI pitch value is hashed. Otherwise, its name representation is hashed (e.g.“C##"). This setting is useful depending upon whether the user would want to differentiate between B# and C.
  - `_hashOctave` - If True, this includes in the hash the number octave that a note is in. This could be useful for an envelope-like following of pitches.
  - `_hashIntervalFromLastNote` - If set to`True`, this returns a number corresponding to the interval between the current note and the last note when applicable. This setting is a useful alternative to hashing pitch when trying to identify transpositions of pieces.

- Chord Low Level Hashing Settings
  - `_hashPrimeFormString` - If `True`, this includes in the hash of chord a string representation of its prime form.
  - `_hashChordNormalOrderString` - If `True`, this includes in the hash of a chord a string representation of its normal order.

3.1.3.4 Pre-Hashing: Building a Set of Hashing Functions from Settings

3.1.3.5

3.2 Low-level Object Comparison Optimizations

Work during the fall of 2015 focused on optimizing the comparison process with variable-length data. The underlying idea was that Python objects tend to have more overhead than C objects, and if comparison and alignment algorithms were going to be doing a lot of comparisons between NoteHash objects produced by the Hasher, then converting these Python objects into C objects of appropriate size could save a lot of time. Here, "appropriate size" refers to the idea that NoteHash objects with only a few hashed properties would correspond to smaller C objects, and NoteHash objects with more hashed properties would correspond to larger C objects.

A smart implementation of the most efficient method, Yeti Levenshtein, already exists in C, so my work was to cleverly convert variable-length NoteHash Python objects to C objects to use the Yeti Levenshtein library, which I hoped would be faster than using any Python edit distance algorithm.

I observed that the speedup we gained in converting variable-length NoteHash Python objects to C objects to use this library was a negligible cost compared to the run time of the comparison algorithm. This empirical evidence suggested that massive parallelization and speedup of the comparison process is difficult and that optimizations in alignment would provide non-negligible impact of speedup.

Thus, I concluded that musical comparison was already near the boundaries of its optimization and my work turned to focus on creating a robust alignment algorithm.

Chapter 4

Final Implementation

4.1 Hasher

4.2 Aligner

4.2.1 System Overview

`Aligner.py` contains the `StreamAligner` class that is the engine of the Alignment step. `StreamAligner` accepts as input two `music21` streams, one specified as a source stream, one as a target stream. The main functionality of `StreamAligner` is that it produces a global alignment between the source and the target streams in the form of a list of Change Tuples that maps each element in the source stream to an element in the target stream via one of the four change operations (insertion, deletion, substitution, no change). Additionally, `StreamAligner` also outputs a basic measurement of similarity between the two stream inputs `similarityScore`. Lastly `StreamAligner` has a `show` function that visually displays the alignment between the source and target streams.

4.2.2 Producing a Global Alignment

The main objective of the Aligner is to produce a global alignment of two streams. In order to do so, it must hash the two streams with an instance of a Hasher (either passed in as a parameter during instantiation of a `StreamAligner` or using a default Hasher built into the `StreamAligner` class). After the two streams are hashed, `StreamAligner` sets up a distance matrix, a la the method described in Chapter 2 for classic string alignment, populates the matrix, and then performs a backtrace of the matrix starting the lower right corner to produce the alignment.

4.2.2.1 Pre-Alignment: Setting Appropriate Parameters

In the interest of being a modular system, I chose to give the programmer the option of setting their own Hasher. Upon instantiation of a `StreamAligner`, the programmer can choose to pass in an instance of a Hasher. If no Hasher is set initially, then `StreamAligner` uses the default Hasher that is set to hash Notes, Rests, and Chords, and their MIDI pitches and durations. The default Hasher is generic enough to be able to work with general alignment, but a more niche problem would require the programmer to use a Hasher more specific to their problem

I also chose to leave the cost of Insertion, Deletion, and Substitution easily substitutable as well. By default, the cost of both Insertion and Deletion is equal to the length of one of the `NoteHashWithReference` tuples, which is exactly the number of properties that are hashed of any musical element. (So in the case of the default Hasher, Insertion and Deletion both have cost 2.) The cost of Substitution between two `NoteHashWithReference` tuples is equal to the number of properties they don't have in common with each other. For example, the cost of substituting

```
NoteHashWithReference(pitch=59, offset=1.0, duration=2.0)
```

with either

```
NoteHashWithReference(pitch=59, offset=1.0, duration=3.0)
```

or

```
NoteHashWithReference(pitch=60, offset=1.0, duration=2.0)
```

would have a cost of 1.

4.2.2.2 Pre-Alignment: Hashing the Streams

The alignment algorithm can only be applied after the streams are represented in a hashed format as a list of `NoteHashWithReference` tuples. Both input streams will be hashed and stored as the `hashedTargetStream` and the `hashedSourceStream`. The hashed format is the analog to a string and each `NoteHashWithReference` tuple is the analog to a character in classic string alignment.

It recommended that the hasher used to the hash the streams be set to include references to the original stream by using `NoteHashWithReference` tuples in the creation of the hash (as opposed to the `NoteHash` tuple that has much less overhead), and in the default hasher, This ensures that the `show` function and future fixers have access to the original objects that each hash came from. This makes it easier to color the original objects in the case of the show function and to extract any necessary metadata that wasn't encoded in the hash for future fixers.

There is also an option to indicate that the streams passed in are already hashed, but in the context of my thesis work, this should never be the case.

4.2.2.3 Alignment: Initializing the Distance Matrix

The first step in the alignment process is initializing the distance matrix. We set up an empty $n+1 \times m+1$ matrix, where n is the length of the hashed target stream and the m is the length of the hashed source. The extra column and extra row are populated with initial costs that correspond to just Insertion or just Deletion operations. In the leftmost column, the value $i \times insertCost$ is put into entries $(i, 0)$. In the topmost row, the value $j \times deleteCost$ is put into entries $(0, j)$.

4.2.2.4 Alignment: Populating the Distance Matrix

The next step in the alignment process is to fill in the remainder of the distance matrix with values generated with these update rules.

```
D[i][j] =min {
        D[i-1][j] + insCost,
        D[i][j-1] + delCost,
        D[i-1][j-1] + subCost
      }
```

where

insCost = insertCost(hashedSourceStream[0])

delCost = deleteCost(hashedSourceStream[0])

subCost = substitutionCost(hashedTargetStream[i-1], hashedSourceStream[j-1])

Each entry A[i][j] in the distance matrix stores the lowest cost for aligning `hashedTargetStream[i]` with `hashedSourceStream[j]`. There are 4 possible ways of aligning the two tuples.

1. ***hashedTargetStream[i]*** *is an insertion* - in this case the cost of aligning `hashedTargetStream[i]` with `hashedSourceStream[j]` is the same as the cost of the smaller sub problem of aligning `hashedTargetStream[i-i]` with `hashedSourceStream[j]` plus the cost of insertion.

2. ***hashedTargetStream[i]*** *is a deletion* - in this case the cost of aligning `hashedTargetStream[i]` with `hashedSourceStream[j]` is the same as the cost of the smaller sub problem of aligning `hashedTargetStream[i]` with `hashedSourceStream[j-i]` plus the cost of deletion.

3. *Completely substituting one tuple for the other* - in this case the cost of align-

ing `hashedTargetStream[i]` with `hashedSourceStream[j]` is the same as the cost of the smaller sub problem of aligning `hashedTargetStream[i-i]` with `hashedSourceStream[j-1]` plus the cost of substituting `hashedSourceStream[j]` with `hashedTargetStream[i]`.

4. *No change between the two tuples i.e. the two tuples are the same* - same as above, where substitutionCost(`hashedTargetStream[i]`, `hashedSourceStream[j]`) is 0.

The method `populateDistanceMatrix` fills in all the entries of the distance matrix using the rules and calculations listed above. It is important to note that in my work, all changes are made relative to `TargetStream`. That is, whenever possible, `SourceStream` is the stream that is being transformed into `TargetStream`. This invariant holds because in OMR/MIDI correction, the approach and direction I take is to change OMR into MIDI. Therefore, the MIDI stream is always the `TargetStream` and the OMR stream is always the `SourceStream`. It is certainly possible to go in either direction, but this paper will be done this way.

4.2.2.5 Alignment: Backtrace to Find the Best Alignment and Create the Changes List

Once the distance matrix has been completely filled in, we use a backtrace starting from the bottom right hand corner of the matrix (i.e. A[i][j]) to find the path of least of cost.

Starting from the value found at A[i][j], we look at the values directly above, to the left, and diagonal in the up-left direction. That is, we look at the values in A[i-][j], A[i][j-1], and A[i-1][j-1]. Among these three values, we choose the minimum value. The combination of direction and value that we moved in tells us what kind of operation was performed to align NoteHashWithReference tuples `hashedSourceStream[i]` and `hashedSourceStream[j]`:

1. if the direction is up, then regardless of the value, this corresponds to an insertion operation.

2. if the direction is left, then regardless of the value, this corresponds to an insertion operation.

3. if the direction is diagonal up-left, and the value at A[i-1][j-1] is *different* from the value at A[i][j], then this corresponds to a substitution operation.

4. if the direction is diagonal up-left, and the value at A[i-1][j-1] is the *same* as the value at A[i][j], then this corresponds to a no-change operation.

We continue this backtrace until we end up back at A[1][1]. Since we are using a global alignment technique, we should always end up back at this entry. If at the end of backtrace we end up in another part of the distance matrix, the method throws an `AlignmentException`.

Additionally, at every step of the backtrace, we create a `ChangeTuple` that holds references to the original musical elements that are represented in the original `SourceStream` and `TargetStream` and the kind of operation (insertion, deletion, substitution, no-change) that links them. Then we insert this `ChangeTuple` into the beginning of the `changes` list.

This is all performed in the `calculateChangesList()` method.

4.2.2.6 Post-Alignment: Measures of Similarity

After the backtrace to find the alignment, there is enough data to calculate some basic metrics of similarity.
`changesCount` is Counter object that provides a count of how many of each of the four different change operations appear in the `changes` list.

`similarityScore` is a float between 0 and 1.0 that is the ratio of `NoChange` operations to the total number operations in the `changes` list.

4.2.2.7 Post-Alignment: Visual Display of Alignment

The `showChanges` method provides a visual display of how the two streams are related via the `changes` list. For each `ChangeTuple` in the list, this method goes back to the initial reference and changes the color of the element in reference and adds in an id number as a lyric to that element if the change operation is not a `NoChange`. The color is determined by the type of change operation. Green corresponds to an insertion, red to deletion, and purple to substitution. The id number is the index of the `ChangeTuple` in the list.

The figures below shows the visual display of alignment between a scanned score of an excerpt of *String Quartet No.7 in E-flat major, K.160: I. Allegro* and a MIDI recording of the same piece as well as the original score that the OMR came from.

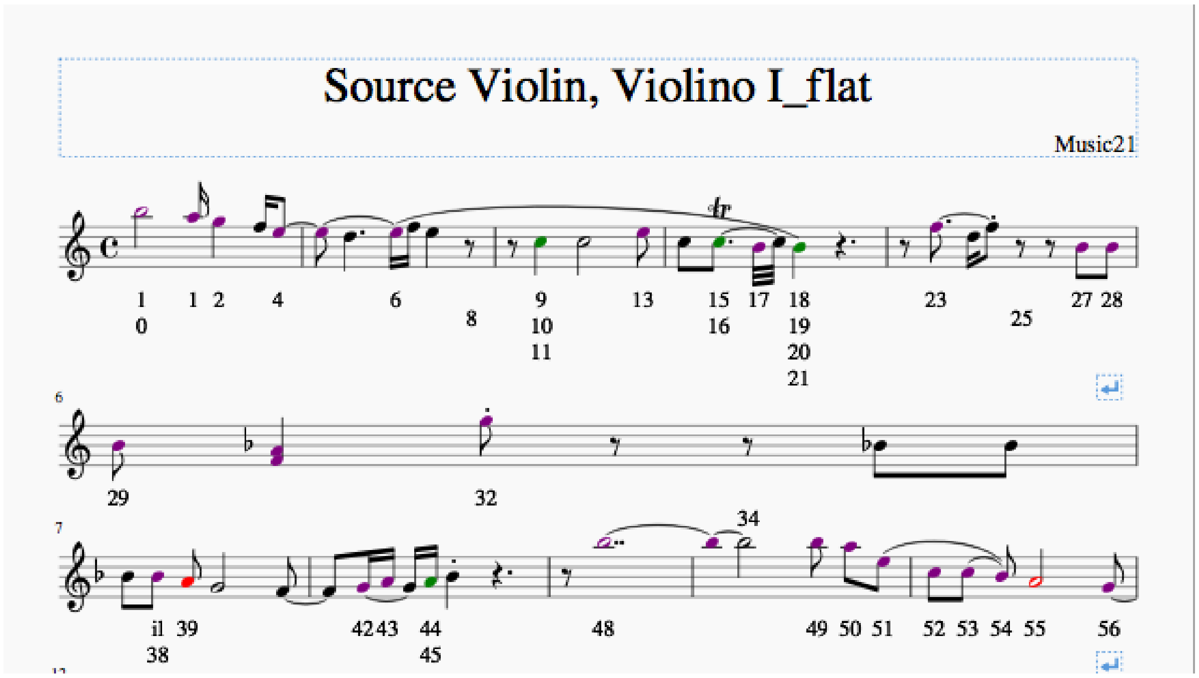


Figure 4-1: Visual display of changes in source (OMR) stream

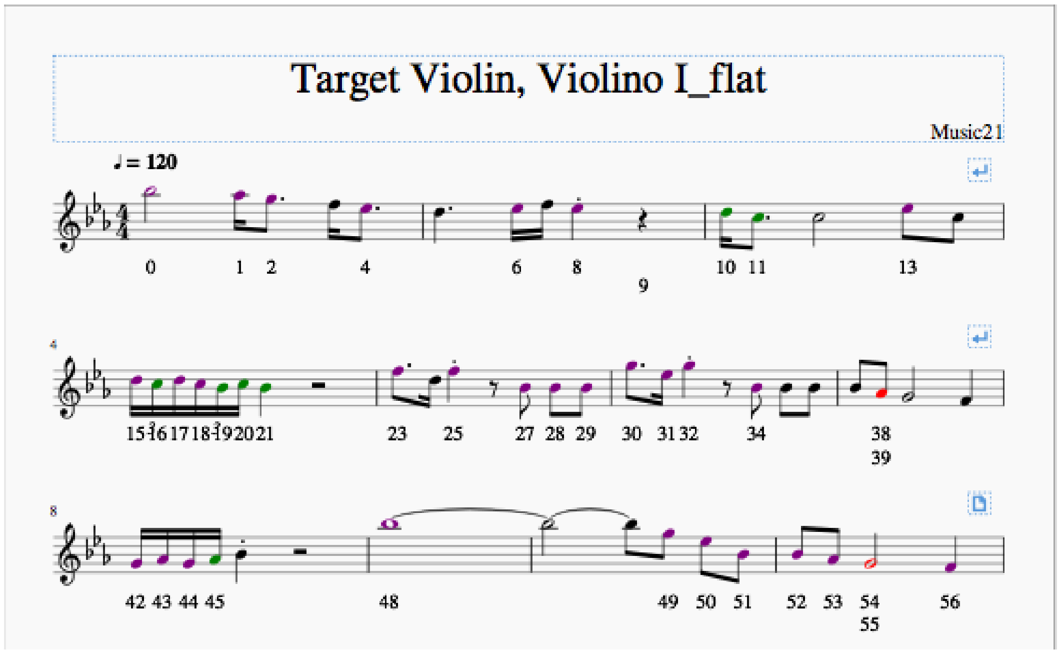


Figure 4-2: Visual display of changes in target (MIDI) stream

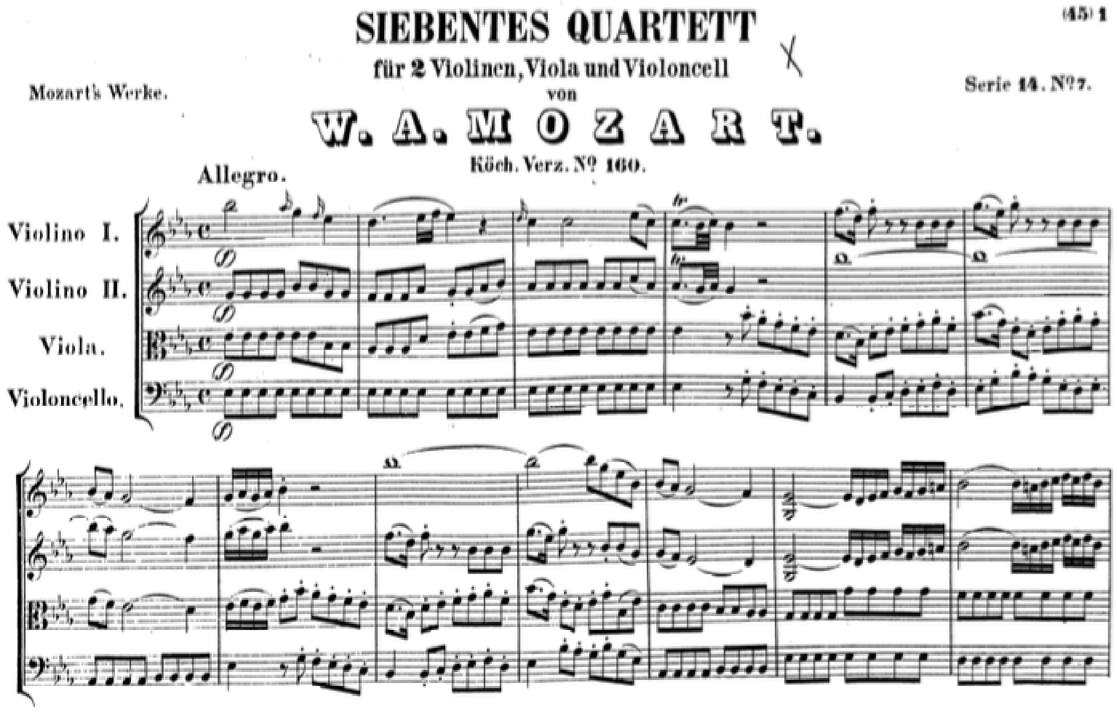


Figure 4-3: Excerpt of original score taken from IMSLP

Since this method has a lot of overhead, it is by default set not to run. It can be set to run for every alignment by passing in the argument `show=True`.

4.3 Fixer

4.4 A Use Case

Chapter 5

Testing

5.1 Existing Music

5.2 Timing

Chapter 6

Future Work

6.1 Future Fixers

Appendix A

Tables

Table A.1:

Appendix B

Figures

Figure B-1:

Figure B-2:

Bibliography

[1] Rainer Typke. *Music Retrieval based on Melodic Similarity.* PhD thesis, 2007.