An Efficient Score Alignment Algorithm and its Applications

by

Emily H. 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 26, 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 H. Zhang

Submitted to the Department of Electrical Engineering and Computer Science
on May 26, 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. This thesis creates 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

I'd like to thank my advisor, Professor Michael Cuthbert for not only advising me on this project but also for providing invaluable feedback and encouragement throughout. I am extremely grateful that I had the opportunity to combine my love of computer science and music by contributing to the `music21` project. It has been a rewarding experience working with you.

I'd like to thank the Music and Theater Arts Department for its support in both my musical education and for giving me so many opportunities to explore music from many different approaches, my thesis work being one of them.

Thank you to all who provided me with moral support, feedback, and food during this project, including Maryam K., Cathie Y., Priya P., Allison Z., Colin M., Jonathan M., Peter G., Katie G., Dan P., Michelle Q., Brian M., Harry B., and Katie L.

Finally, I'd like to my family, especially my sister Allison, whose unwavering support have helped me succeed in my endeavors at MIT.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

1.1.1 Why Compare Music?

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, generated from a scanned score. Each representation is imperfect on its own.

MIDI represents music in a 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. Many other design decisions that make MIDI very universally compatible but also lacking in definition. Moreover, since MIDI scores are generally created with human input, they tend to be correct, pitch-wise and rhythmically.

OMR takes a score that is 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 it and tries to recreate what it scans in a digital notation format. While the more commonplace optical character recognition (OCR) has been optimized and whittled to an extremely high standard, OMR is still at present only 90% correct [12]. The space that OMR is in 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".

1.1.2 Why Correct Music?

The quality of OMR should be able to be greatly improved is rules such as: (1) all measures must sum to the same duration, and (2) only notes with existing accidentals can be made natural, were taken into account. Computer-readable scores are very powerful because they can be analyzed, replicated, and distributed, and OMR is just way of producing these scores.

A big idea 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 data present in the MIDI score to correct for scanning errors in the OMR score, while retaining the richness of OMR's musicXML format.

1.2 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 discusses previous work. It will summarize related work in musical sequence representation, alignment, and correction by others in the field. It will also summarize the work I did previously that is used as a building block for my thesis work and briefly discusses some ideas that were not as promising to develop further. 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. An example of correction taken from Mozart along with performance and testing metrics are discussed in Chapter 5.

Chapter 2

Background and Related Work

2.1 Computer Musical Terminology

2.1.1 OMR

OMR, or Optical Music Recognition, is method of recognizing the characters on scanned sheet music or printed scores and interpreting them into a digital form. OMR is great for large-scale fast digitization of scores. However, the current state-of-the-art of OMR leaves much room for improvement [12].

2.1.2 MIDI

MIDI is the protocol with which electronic instruments and computers can communicate with each other. Many people associate MIDI with an idea of low-quality computer music, but it is just an instruction-like protocol, not actually an audio recording. At the basic level, MIDI messages encode notes with note-on, note-off times, and a numbers that corresponds to their frequencies. A piece of music can be encoded in MIDI as a sequence of notes. On the other hand, MIDI files contain

sequences of MIDI protocol events time-stamped for playback in a certain order.

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.

Readers familiar with the canonical representation of the string alignment problem as a dynamic programming matrix 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 [11]

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 to overlook in larger data could be much more egregious in smaller contexts.

The specific algorithm chosen to implement for solving the musical sequence alignment problem is a version of the Needleman-Wunsch dynamic programming algorithm for global sequence alignment [9]. [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, 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 from each other. If sequences are represented as an array of characters, then sequences can transform into each 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

[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.

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 excluded are all the no-change operations that look like this:

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

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

Table 2.1: Naive Operation Cost Function Table

Figure 2-2 corresponds to an alignment that looks like figure 2-2. 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.

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

Figure 2-2: 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).

One cost refinement that could be made is to slightly increase the cost of substitution. Since a substitution can be thought of as a consecutive insertion and deletion operation, but more compact, it should have a cost greater than that of a single insertion or deletion, but less than twice the cost of a single insertion/deletion.

A slightly different cost function that assigns more cost to substitution changes could look like table 2.2. Using this cost function, the 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

Yet another cost refinement is to give different subtypes of a single operation different weights. Consider if the application of trying to run a spell checker (one application of string alignment and correction algorithms) on a word that isn't recognized by the dictionary. In the particular spell check model, perhaps substitution of a vowel for a vowel isn't as costly as any other types of substitution, so vowel for vowel should only have a cost of 1.2. Using this new cost function, the 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

Considerations such as these go into the creation of a musical sequence aligner as well.

2.2.1.2 Alignment

Recall that the edit distance between two sequences is defined as the least-cost alignment. Figure 2-3 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, as it requires four substitutions, one insertion, and one deletion.

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

best alignment of `string1`$[: i]$ and `string2`$[: j]$ + best alignment of `string1`$[i :]$ and `string2`$[j :]$

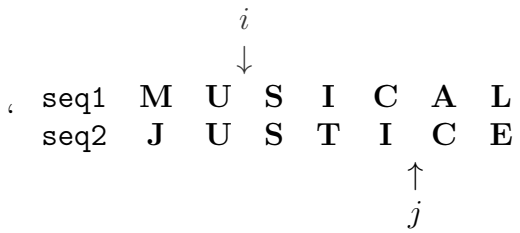

Figure 2-4: The alignment of MUSICAL split at i and JUSTICE split at j

If all the possible alignment subproblems are kept track of, it is possible to 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 the problem of aligning **MUSICAL** and **JUSTICE**, the setup would look something like in figure 2-5

Recall that the problem has been defined 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**

	-	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

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 (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), but having a simple cost operation functions, such as in tables 2.1 and 2.2, will make such problems symmetric.

In this particular sample problem, it is always the case that the word **MUSICAL** is transformed into another word, **JUSTICE**. In this particular context, OMR sequences are transformed into MIDI sequences.

After the scoring matrix is set up, it is populated 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)

In reading a scoring matrix setup like this, insertions are vertical movements, deletions are horizontal movements, and substitutions are diagonal. After setting up the initial matrix, all of the remaining slots are filled in using this update rule:

D[i][j] =min {

D[i-1][j] + insCost (the cell above),

D[i][j-1] + delCost (the cell to the left),

D[i-1][j-1] + subCost (the cell diagonal, up and to the left)

}

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 an A# quarter note in the 4th octave and a B quarter note in the 4th octave compare with an A# eighth note in the 4th octave and an A# quarter note in the 4th octave?

By massaging musical sequences into string-like objects, it becomes easier to see how techniques in string alignment can be applied to musical sequence alignment. The questions and answers posed below help identify parallels between strings and musical sequences.

- What are sequences are and what makes them up?: musical sequences can be thought of as an ordering of notes, rests, and chords.

- What is the space that sequences occupy?: This question can roughly be approximated as, what are the important properties of notes, rests, chords? Some immediate answers are pitch, duration, offset.

- What is an appropriate metric for distance?: In the previous section there were multiple different substitution functions. What kind of and how many ways are there to calculate distance between musical elements? Perhaps representing a musical element as a tuple of numbers, (pitch, duration, offset) gives us points in space that we can calculate distance.

Once musical sequences can be represented in string-like formats, we can perform string operations on them. The next chapter discusses related work by others who have explored some part of this musical sequence alignment question as well as my own previous work that relates to this thesis.

Chapter 3

Previous Work

This chapter discusses previous work in musical sequence representation, alignment, and correction. The first half of the chapter is about work by others. The second half is my own previous work that was relevant to this thesis.

3.1 Related Work by Others

In this section of work by others,

3.1.1 Typke: Music Retrieval based on Melodic Similarity

Typke's thesis on Music Retrieval based on Melodic Similarity gives an example of how music can be transformed and compressed into its most relevant features [6]. 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 [4]) and an offset in quarter notes from the beginning of the piece. This representation allows for notes to be easily compared with each other. Since

the features of notes are represented as numbers, it is convenient to do numerical operations with the representations.

3.1.2 Church and Cuthbert: Rhythmic Comparison and Alignment

Church and Cuthbert [2] describe a system that hashes scores purely on rhythm and uses heuristics of rhythm in music (e.g. a repeated rhythm should be the same every time it is expected to repeat) to correct OMR scores. This model was the original inspiration for building an alignment system with more heuristics pertaining to the qualities of music beyond just rhythm to further improve OMR accuracy.

3.1.3 IMSLP

IMSLP holds a large collection of musical scores (currently over 350,000), many of which are user contributed (i.e. not necessarily verified correct) [5]. However, its contents are free and in the public domain. My work makes use of this large dataset.

3.1.4 Viro: Peachnote, Musical Score Search and Analysis

Peachnote [14] is a project that contains a large bodyof musical scores. Peachnote performs OMR over the PDF scores in IMSLP and indexes the data for efficient querying and searching.

3.1.5 White: Statistical Properties of Music and MIDI

White's dissertation [15] describes properties of music that can be turned into heuristics for refining algorithmic analysis of music. Specifically, he builds upon Temperley's

1997 [13] paper's details of 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.

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

Rebelo et al's paper discusses the state of the art of OMR at each step of the OMR process and proposes future work in certain areas that would advance the current state of OMR [12].

3.1.7 OpenScore

OpenScore is a project that aims to liberate all public domain music using both human and computer-powered techniques. Many works in the public domain are of low quality and/or are in formats that do not lend well to analysis, sharing, or reproducing, and OpenScore plans to turn them into high-quality music in digital formats. Both humans and computer software can aid in any part of the process, from transcription to error correction. The work of my thesis can directly contribute to the computer software tools that OpenSource is looking for.

3.1.8 `music21`

`music21` is a open source Python library that is a toolkit for computer-aided musicology. It provides both a framework for representation of music as well as tools to perform analyses on music. The rhythm correctors built by Church and Cuthbert were made

using the tools that `music21` provides. The work of my thesis relies heavily on the infrastructure of`music21`, its score and stream representations and their operations.

3.2 Previous Work: A Modular Universal Hasher

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 undergraduate capstone supervised independent study (6.UAP).

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, such as notes and chords, and properties, such as pitch name or duration, are relevant 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.

3.2.1 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, this is a two step process:

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

2. hashing those elements

The details of this process can vary. For instance, in determining 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 be relevant.

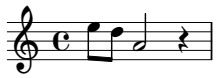

Figure 3-1: The hash of rhythmic features of this short melody would be *NoteHash(offset=0, duration=0.5), NoteHash(offset=0.5, duration=0.5), NoteHash(offset=1.0, duration=2.0) NoteHash(offset=4, duration=1.0)*

As another example, in determining whether one piece is an approximate transposition of another, only the elements with some relation pitch (e.g. notes and chords) might be relevant.

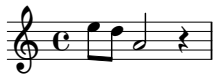

Figure 3-2: The hash of features related to transposition of this short melody would be *NoteHash(intervalFromLastNote=0), NoteHash(intervalFromLastNote=-2), NoteHash(intervalFromLastNote=-5)*

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

3.2.2 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 representation is not new. Typke in

his thesis proposes what he calls a weighted point set representation of music [6]. 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-40 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." as something like "The weight component is an assignment of a numerical value to each note in the music stream, indicating the relative importance of the different notes within the melody [4].

3.2.3 Overview of System

The `Hasher` object is found in `hasher.py`. Its purpose is to identify the relevant features of relevant elements in any music stream and to represent them in a hashed format that is easy to compute with. Relevant elements here means objects such as notes, rests, chords. Relevant features here means qualities of these elements such as pitch, duration, offset. I represent these qualities in the hash as lightweight integers, floats, and strings, which makes future computations such as comparisons, and numerical operations easy and fast.

Upon instantiation, `Hasher` does not take in any parameters. After an instance of `Hasher` is created, however, the user can set specific parameters in the `Hasher` to fit their own needs. The user must specify which relevant stream elements (notes, rests, chords) should be hashed, as well as which relevant qualities of those elements should be hashed (pitch, duration, offset, etc.).

After the hashing parameters are set, the user calls the `hashStream` method on a stream. This method strips the stream only to the relevant elements, creates a set of hash functions based on the user's selection of relevant element qualities, and

individually hashes each element with the entire set of specified hashing functions.

The `Hasher` will create either a `NoteHash` or a `NoteHashWithReference` object for each element and add it to the end of the `FinalHash` list. Both are tuples that hold the hashed qualities of a single stream element. The only difference is that a `NoteHashWithReference` stores a reference back to the original stream element. Even though this has a lot of overhead, it is necessary if ever the original streams need to referenced for context or if they need to be changed.

The output of the entire process is a list of `NoteHash` or `NoteHashWithReference` objects, stored in `finalHash`.

3.2.3.1 Pre-Hashing: High Level Hasher Settings

There are several system-wide high level settings that must first be set before the `hashStream` method can be called.

- `includeReference` - By default this is set to `False`, but if set to `True`, each element hash will be stored in a `NoteHashWithReference` object that includes a reference to the original element in its original stream. For many purposes, having this reference is not necessary and takes up extra overhead, so it is set to `False` by default.

- `validTypes` - This is a list of the types of stream elements should be hashed. By default this is `[note.Note, note.Rest, chord.Chord]` and can be changed to be any subset of those three stream elements.

- `stateVars` - This is a dictionary that keeps track of state between consecutive hashes. By default it doesn't contain anything, but can be programmed to contain *intervalFromLastNote* and *KeySignature* during the later process of preprocessing the streams and setting the appropriate hashing functions if the user-set parameters call for their use.

- `hashingFunctions` - This is the dictionary that stores maps which hashing functions should be used to hash which qualities. For example, if the user chose to hash notes and their MIDI pitch values, then the dictionary value for the key *Pitch* would be *_hashMIDIPitchName*, the name of the function that takes a note and returns its MIDI pitch.

3.2.3.2 Pre-Hashing: Low Level General 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 (measured in a multiplier of `duration.quarterLength`) .

- `_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 precisely we round. If duration and offset are rounded, they will be rounded to this granularity of beat division. For example, a granularity of 32 will round notes to the nearest 32^{nd} beat division and no further.

3.2.3.3 Pre-Hashing: Low Level Note Settings

- `_hashPitch` and `_hashMIDI` - These settings determine whether the 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 `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.

- `_hashIsAccidental` - If `True`, this will return a boolean indicating whether the pitch of the hashed element is outside of the key signature. This setting is currently not fully supported.

3.2.3.4 Pre-Hashing: Low Level Chord Hashing Settings

Chords can be thought of as multiple notes happening at the same time. In `music21`, chords are their own type of object that can be decomposed into `note` objects. In the hashing system, chords can be hashed as chords, or by their constituent notes.

- `_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.

3.2.3.5 Hashing: `hashStream`

Once the settings of a `Hasher` instance have been set, we can call its `hashStream` method on any stream. The following sections describe the what happens within a single `hashStreams` call.

3.2.3.6 Hashing: Setting up `ValidTypes` and `StateVars`

First, the method `setupValidTypesAndStateVars` is called to set up the state variables and the valid hashing types. Recall that by default `stateVars` is empty and `validTypes` is `[note.Note, note.Rest, chord.Chord]` (of course, the user has already had the option of changing `validTypes` to contain any subset of the default types). This method checks whether the user has set either of these two settings:

1. `hashIntervalFromLastNote` - if this is set, then `IntervalFromLastNote` is added to `stateVars` to keep track of it.

2. `hashIsAccidental` - if this is set, then `KeySignature` is added to `stateVars` to keep track of, and `key.KeySignature` is added to the list of `validTypes` to hash.

3.2.4 Hashing: Preprocessing the Stream

The stream that the `hashStream` method acts upon is stripped of all note ties in the `preprocessStream` if the `stripTies` setting is `True`. Additionally, this method returns the stream in a generator form by calling `recurse` on the stream.

3.2.4.1 Hashing: Creating a List of Elements to be Hashed

Next the `hashStream` method builds a list of elements that are to be hashed by filtering the recursed stream of any types that are not in `validTypes`. This is stored in `finalEltsToBeHashed`.

3.2.4.2 Hashing: Building a Set of Hashing Functions

The method `setupTupleList` is called, and it sets up `tupleList`, `hashingFunctions` and `tupleClass`, all related to each other. `tupleList` is a list of all the element

properties that should be hashed. `hashingFunctions` is a dictionary of which hashing function should be used for each property. `tupleClass` is a `namedtuple` that is constructed ad hoc based on which properties are to be hashed. The ad hoc construction accommodates for the fact that different `Hasher` instances will contain different things to hash.

3.2.4.3 Hashing: Creating a `NoteHash` for Every Element

For each of the elements in `finalEltsToBeHashed`, the `Hasher` hashes its relevant properties using the hashing function listed in `hashingFunctions`. It then creates a single `NoteHash` that stores all the hashed properties.

3.2.4.4 Hashing: Building `finalHash`

If `includeReference` is set to `False`, then each new `NoteHash` object is directly added to `finalHash`. If `includeReference` is `True`, then each `NoteHash` gets replaced with a `NoteHashWithReference` object that includes a reference to the original stream element, and that gets added to `finalHash` instead.

3.3 Previous Work: 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 be the key to significant 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, which is described in the next chapter.

Chapter 4

Implementation of `OMRMIDICorrector`

This chapter describes the final implementation of the entire OMR/MIDI musical stream alignment and correction system. First there will be an overview of the entire system. Then, there will be discussion on the final implementation of each of the large modules that is divorced from the context of the `OMRMIDICorrector` class. Afterwards, I will present an example use case of the system. I will conclude with a discussion of design tradeoffs and choices that I made in implementing the system.

The final implementation of the OMR/MIDI musical stream alignment and correction system. can be found in `omrMidiCorrector.py`. The OMR system is a combination of one or more of each of the `Hasher`, `Aligner`, and `Fixer` modules with specific parameters set.

The inputs to the system are two `music21` streams (one OMR, one MIDI). The output of the system is a corrected OMR stream.

4.1 OMRMIDICorrector

4.1.1 System Overview

The system is the `OMRMIDICorrector` class in `omrMidiCorrector.py`. This class is initialized with a `midiStream`, an `omrStream` and an optional `Hasher`. The `Hasher` parameter is optional because the system provides a default `Hasher` with some all-purpose settings if no `Hasher` is passed in. The method `processRunner` is the main method of this class that preprocesses the streams, sets up the appropriate `Hasher`, and calls the aligning and fixing methods.

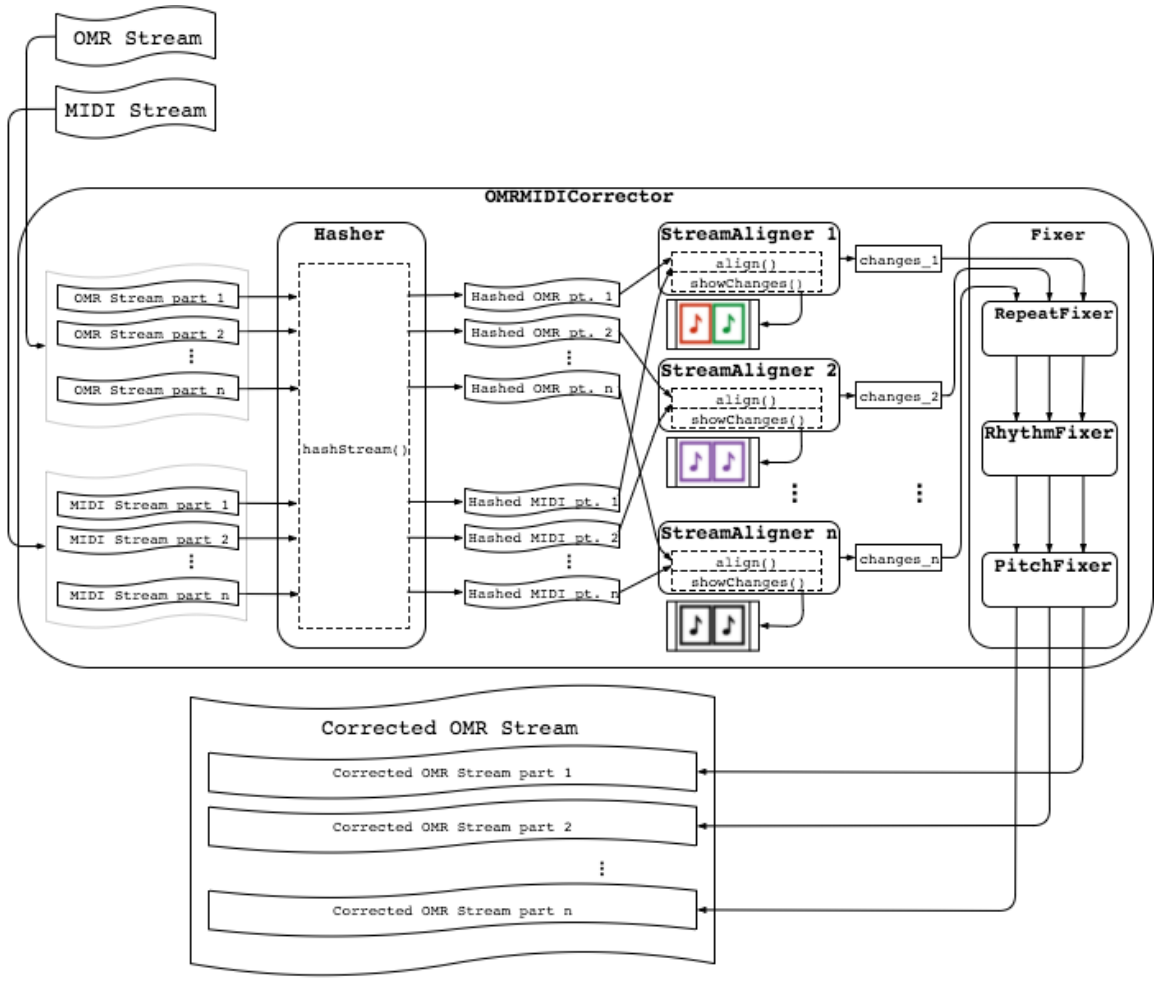

Figure 4-1: The inputs to the system are OMR and MIDI streams. The output of the system is a corrected OMR stream.

4.1.1.1 processRunner: preprocessStreams

The `processRunner` first verifies the streams that are passed in and preprocess them before being hashed, aligned, and fixed in the method `preprocessStreams`.

`preprocessStreams` begins by discretizing the parts of the stream if the user has set `discretizeParts` to be `True`. Discretizing parts means to split the input streams into their constituent parts, so that the input streams are aligned part-wise. Since music is often comprised of multiple parts, this preprocessing step allows for input streams that contain nested streams. Take for example, an OMR/MIDI alignment of a string quartet. Both the OMR and MIDI inputs are stream in and of themselves, and so are all individual instrumental parts within both of the streams.

A good alignment of the piece would involve aligning corresponding instrumental parts with each other to ensure that the beginnings and ends of corresponding parts aligned with each other. However, if we treat the inputs both as one long stream composed of four serial streams instead of four parallel streams, only the beginnings of the first instrumental parts and the ends of the last instrumental parts would be guaranteed to be aligned with each other. This guarantee comes from the global alignment method based on the Needleman-Wunsch algorithm that was implemented. Figures 4-2, 4-3, and 4-3 illustrate the phenomenon.

By default, `discretizeParts` is set to `True`. However, the user ultimately has the choice to align input streams part-wise or After a stream is split up into its constituent parts, the parts are stored in the lists `midiParts` and `omrParts`. The aligners and fixers work on all of the elements in the list, and so the entire system is agnostic to the number of parts; a stream comprised of a single part is processed in the same way as a multi-instrumental score.

Then, `preprocessStreams` checks if the number of parts in MIDI and the OMR stream are the same. If they are off by more than one, then a `OMRMIDICorrectorException` is raised and the process halts. This is a cursory check to see if the two input streams

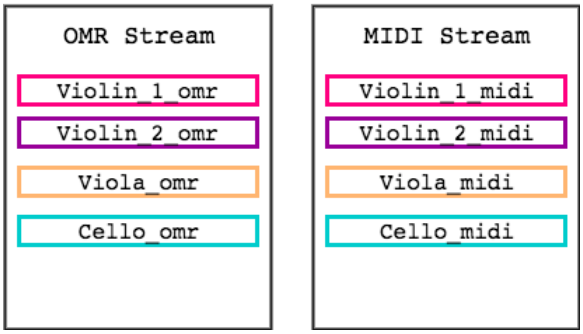

Figure 4-2: An example string quartet stream and its four parts.

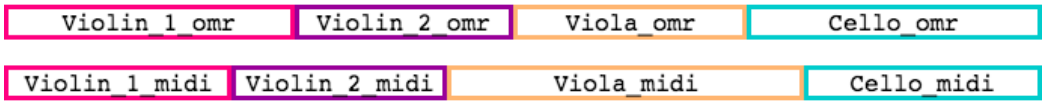

Figure 4-3: A bad alignment setup of the quartet would treat the entire piece as four serial streams and only the only guaranteed alignment would be between the beginning of the first part and the end of the last part.

Figure 4-4: A better alignment setup of the quartet would do part-wise alignment instead of treating the entire piece as a sequence of parts. This methodology guarantees that the beginnings and ends of each part are aligned.

can even reasonably be aligned.

If the numbers of parts are off by exactly one, then `preprocessStreams` spins up a lightweight `StreamAligner` object to try and see if there is an implicit bassline, where the Bass part doubles the Cello part but an octave lower. This is a common

feature of orchestral scores. In such a scenario, the OMR score would have one fewer part than the MIDI score because a bassist would read the cello line while reading sheet music, but the MIDI score would explicitly list all the instrument lines. If the similarity of the two streams is above a certain threshold, it is reasonable to assume that the Bass part doubles the Cello part and the extra MIDI part is disregarded in the alignment process. The `checkBassDoublesCello` method and its use of an `StreamAligner` object to determine whether two streams are an octave apart is a incidental example of a different use case of the `StreamAligner`.

4.1.1.2 processRunner: setupHasher

The next step in the `processRunner` method is setting up the `Hasher` object that will be used to hash both input streams. This is done in the `setupHasher` method. If user has passed in a `Hasher` object upon instantiation of an `OMRMIDICorrector` object, then that is used. Otherwise this method sets up a default `Hasher` with settings found to be generally effective in the `setDefaultHasher` method. The default `Hasher` hashes only pitch and duration and includes references to the original stream elements.

4.1.1.3 processRunner: alignStreams

This method creates a `StreamAligner` object for each pairwise MIDI part and OMR part found in `midiParts` and `omrParts` and aligns the two parts, calculates their similarity metrics, and visually shows the changes and alignment if `debugShow` is `True`. There is more detail about how the `StreamAligner` object works in section 4.3.

4.1.1.4 processRunner: fixStreams

The last step is of the entire process is to fix the OMR stream. Because the `Fixer` still remains in development, the description of this method and the `Fixer` in section 4.4 will remain at a high level.

In its final implementation, `fixStreams` should create instances of the `OMRMidiFixer` object, one per pair of aligned streams and their `changes` list. This is the base class that all other `Fixer` objects inherit from. The user would be able to choose which `Fixer` objects they want to create and use on all the pairs of streams.

Currently, in order to use the two available child `Fixer` objects, a user can manually create `OMRMidiFixer` and `DeleteFixer` and/or `EnharmonicFixer` objects using the `changes` lists that `processRunner` creates.

4.2 Hasher

The `Hasher` object used in the final implementation is a specific instance of the `Hasher` object described in section 3.2. The user can choose to pass in their own instance of `Hasher` or use the default `Hasher` provided.

4.2.1 The Default Hasher

The default `Hasher`, as described in section 4.1.1.2, hashes only pitch and duration of stream elements and set `includeReference` to be `True`. This setting is general enough to provide for a reasonable alignment for most generic alignment problems. But for more niche problems (for example, a rhythmic alignment), the user ought to define their own `Hasher` and the default `Hasher` should not be used.

4.2.2 NoteHashWithReference

One update made to the original `Hasher` was the addition of a new type of `NamedTuple` that includes a reference to the original element that is hashed and stored within the tuple. This new type provided access to the original stream elements which was necessary for three reasons:

1. It allows the user to visually see the alignment using the `showChanges` method in the `Aligner`. A list of the changes between two streams can be hard to parse and see patterns within, so this method provides a more visually appealing way of interpreting the list of changes.

2. It allows the user to access other elements in the context of the original stream element. For example, it might be helpful to be able to access the most recent key signature change in the stream for some fixing purpose.

3. In order for the `Fixer` to fix the OMR stream, it must be able to change the original elements. The reference to the original stream element allows for this.

4.3 Aligner

4.3.1 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.

4.3.2 Producing a Global Alignment

In order to produce a global alignment of two streams, the `Aligner` must hash the two streams with an instance of a `Hasher`. 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.

4.3.2.1 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, since each differs in one property. However, substituting

```
NoteHashWithReference(pitch=60, offset=2.0, duration=3.0)
```

would have a cost of 3 since all three properties are different.

4.3.2.2 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.

4.3.2.3 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 $(0, j)$.

4.3.2.4 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, previously seen in Chapter 2.

$$
\begin{aligned}
\text{D[i][j]} = &\min \{ \\
&\text{D[i-1][j]} + \text{insCost}, \\
&\text{D[i][j-1]} + \text{delCost}, \\
&\text{D[i-1][j-1]} + \text{subCost} \\
&\}
\end{aligned}
$$

where

$$
\begin{aligned}
\text{insCost} &= \text{insertCost}(\text{hashedSourceStream[0]}) \\
\text{delCost} &= \text{deleteCost}(\text{hashedSourceStream[0]}) \\
\text{subCost} &= \text{substitutionCost}(\text{hashedTargetStream[i-1]}, \text{hashedSourceStream[j-1]})
\end{aligned}
$$

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 relating the two tuples.

1. *`hashedTargetStream[i]` is an insertion* - in the case of insertion, the cost of

aligning `hashedTargetStream[i]` with `hashedSourceStream[j]` is equal to the cost of aligning `hashedTargetStream[i-i]` with `hashedSourceStream[j]` plus the cost of insertion.

2. *`hashedTargetStream[i]` is a deletion* - similar to the insertion case, for deletion, the cost of aligning `hashedTargetStream[i]` with `hashedSourceStream[j]` is equal to the cost of the subproblem of aligning `hashedTargetStream[i]` with `hashedSourceStream[j-i]` plus the cost of deletion.

3. *`hashedTargetStream[i]` is a substitution of `hashedSourceStream[j]`* - in the case of substitution, the cost of aligning `hashedTargetStream[i]` with `hashedSourceStream[j]` is equal to the cost of the subproblem of aligning `hashedTargetStream[i-i]` with `hashedSourceStream[j-1]` plus the cost of a substitution of `hashedSourceStream[j]` for `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 do it this way.

4.3.2.5 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.

4.3.2.6 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 a 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.

4.3.2.7 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 in Appendix **??** shows the visual display of alignment between the Violin 1 parts of an excerpt of *Eine Kleine Nachtmusik, K.525* by W.A. Mozart. and a MIDI recording of the same piece.

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`.

4.4 Fixer

The `Fixer` is the last part of the OMR/MIDI music stream alignment and correction system. Its developmental work can be found in `fixer.py`. There is a base class called `OMRMidiFixer` that takes as input the `changes` list that the `StreamAligners` outputs, as well as the original OMR and MIDI Streams.

Currently, there are two child classes, `EnharmonicFixer` and `DeleteFixer` that inherit from `OMRMidiFixer` and provide different fixing services.

The `EnharmonicFixer` corrects a very specific type of error that appears in OMR, classified by Byrd as a pitch error [1]. What happens in this scenario is that the MIDI score has the correct absolute pitch of a note, but might be spelled enharmonically, due to the fact that MIDI does not encode key signatures. The OMR score has been scanned incorrectly— a sharp is mistaken for a natural or vice versa. An emergent property of this scenario is that the OMR note and the MIDI note could be on different lines or spaces but probably no more than a few steps apart. The `EnharmonicFixer` accounts for the various possibilities that arise in this situation and corrects the pitch of the OMR note by transposing the note 1 or 2 half steps and by adding or removing accidentals.

The `DeleteFixer` was designed to fit the specifications of the OpenScore project [10]. The goal of the OpenScore project is to open-source music with open source software (like `music21`!). OpenScore will use a combination of computer and human power to digitize classical music scores and put them in the public domain. One idea is that software can identify wrong recognized notes in a scanned score and mark or delete the entire measure that that note is in and pass it off to a human corrector to re-transcribe the entire measure. The `DeleteFixer` could be the computer power in this method of score correction that OpenScore is using.

Any additional child Fixer classes would inherit from the parent `OMRMIDIFixer` class and also correct very specific problems. For example, in the `StaccatoFixer` skeleton code, would look for a specific pattern in the `changes` list that would indicate that the OMR process failed to recognize a passage that is marked as staccato. The `StaccatoFixer` class along with other children `Fixer` classes would look for similar patterns in the `ChangeTuples` list and correct the OMR stream based on this information.

4.5 Tradeoffs and Design Decisions

In this section, I will justify some of my design choices and discuss tradeoffs in my decisions.

4.5.1 Modularity vs. Simplicity

4.5.1.1 A Variety of Settings and a Longer Setup

All of the modules have many different settings with various parameters. The variety of settings is what makes the `Hasher` and `Aligner` so powerful— by changing just a couple of settings, both are able to solve very different problems.

However, this necessarily lends itself to the system being more complex because each module must be able to handle various combinations of settings and their edge cases. It also requires the user to be careful when selecting initial settings.

One alleviation to this problem is providing default settings to all the modules. These default settings are what I believe are good general settings for the most common uses of the modules.

4.5.1.2 Length Agnosticism

There are two instances in which I purposefully built the system to be able to handle variable length input— in building `NoteHash` and `NoteHashWithReference` tuples and in building the `OMRMIDICorrector` to be able to process a stream with any number of parts. Both of these decisions make the entire system more powerful because the system is able to process many more different types of streams this way, but it also makes the hashing and aligning process more convoluted. Fortunately for the user, all of this is hidden under the hood. There are likely some performance hits that are associated with this added complexity, but not to the point of inoperability. More demonstrations on timing are shown in section 5.3.

4.5.2 Performance vs. Space and Object Overhead

4.5.2.1 NoteHashWithReference

The `NoteHashWithReference` object contains a reference to the original stream object. Streams and their elements are complex objects, so `NoteHashWithReference` objects have much more overhead in storage than their counterparts, `NoteHash` objects. However, in setting up a `Hasher` object, the user has the decision to use either one.

4.5.2.2 Creating One Hasher and Many Aligners

During the correction process, only a single `Hasher` object is created and used and reused to generate hashes for every part in the stream passed in. There was the option of having `Hasher` object be one-time-use objects, but I chose to have the `Hasher` be able to operate on multiple streams.

On the other hand, a new `StreamAligner` object is generated for every pair of parts in the OMR and MIDI streams that are passed in.

Between generating `NoteHashWithReference` objects for every hashed stream element and `StreamAligner` objects for every pair of streams, there is a lot of space overhead taken up in even just a single alignment and correction process. However, based on experimental findings in which I tried to create these objects in C, perform the comparisons in C, and then translate the results into Python, I concluded that the limiting factor of the process was not in Pythonic object creation but rather the comparison process that calculates Levenshtein edit distance. The results and findings were discussed in more detail in section 3.3.

4.5.3 Manual User Input

The OMR and MIDI streams that are used as input to the system come from somewhere. I have left the onus on the user to source .mid files and find scanned scores. I also have left it to the user to use some kind of OMR tool to convert the scanned score and the MIDI file into a musicXML form. Once in musicXML form, the user can use the `music21` library to parse the OMR and MIDI and convert them into `music21` streams. At the time of that this thesis was written, `music21`'s MIDI parsing was having issues with quantization of streams, so I used museScore's MIDI parsing functions instead.

I leave this as a task to the user instead of automating it because with practice, humans can be much more precise and efficient with this process, and the necessity of a (probably large and complicated) computer program to do the same thing decreases.

Chapter 5

Examples and Results

This chapter will describe the results of my work using an excerpt of of W. A. Mozart's *Eine Kleine Nachtmusik, K.525- I.Allegro* (referred to by just "K.525" from here onwards) as an example. It will also use an excerpt of W. A. Mozart's *String Quartet No.7 in E-flat major, K.160- I.Allegro* (referred to by just "K.160" from here onwards) as an example where K.525 doesn't fit the specifications. This chapter will also look at timing metrics of the system.

5.1 Eine Kleine Nachtmusik, K.525- I.Allegro, W. A. Mozart

This section will go through an example of how the entire `OMRMIDICorrector` system works on W. A. Mozart's *Eine Kleine Nachtmusik, K.525- I.Allegro*.

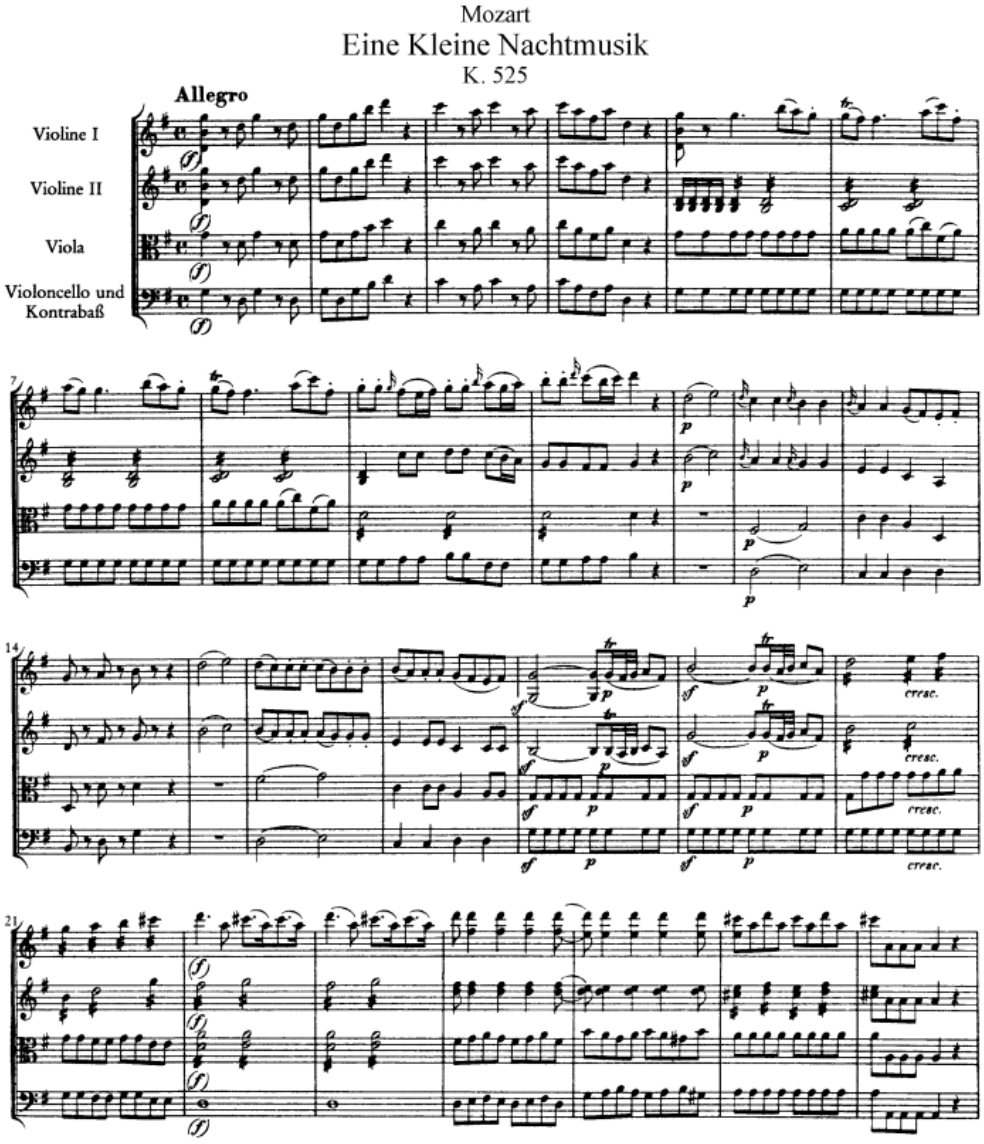

Figure 5-1: The first page of the scanned copy of the score found on IMSLP.

5.1.1 Musical Properties of K.525

I chose K.525 as the example piece for its many music properties that make it a difficult but interesting piece to align.

5.1.1.1 Bass and Cello Doubling

In the original scanned score, there are only four parts because the bass part doubles the cello part one octave lower. However, in the MIDI encoding, there are five separate voices. Thus, in the preprocessing step of the `OMRMiDICorrector`, it would be ideal for the system to recognize this. Otherwise, in instances where the `OMRMIDICorrector` could not pair up the parts between the input OMR stream and the input MIDI stream, there would be an error thrown, and alignment and correction would not happen.

5.1.1.2 Tremolos

Starting in measure 5 in the original scanned score, the second violin has tremolo notes, instead of repeated 16^{th} notes written out. The OMR parser in SmartScore is not robust enough to recognize the slashes across the stem of the note as tremolo markings and instead interprets it as a single note. The MIDI protocol has no way of indicating tremolos, so the second violin's notes in measure 5 are encoded as regular 16^{th} notes. The `Aligner` is robust enough to align tremolo measures with non-tremolo measures even though there is a huge discrepancy in the number of notes present in the measure.

5.1.2 Preparing the Raw Input

The basic raw input of the system is an OMR score and a MIDI file. A scanned copy of the score was found in IMSLP [7], the source for much free public domain sheet music. The MIDI file was sourced from the Yale MIDI Archive.

I used SmartScore's built-in OMR tool to convert the scanned score of K.525 into an ENF file. Then I exported the post-OMR file as a musicXML file.

Similarly for the K.525 MIDI file, I used SmartScore to parse the original file and exported it as a musicXML file.

The last step of preparing the raw input is to parse the musicXML files with the `music21` library so that they are `Stream` objects. This can be done by passing in the musicXML filepaths to the `parse` method in the `converter` class. We will call these two parsed streams `k525midiStream` and `k525omrStream`.

```
from music21 import *
k525MidiFilepath = "pathto/k525mvmt1/miditoxml.xml"
k525OmrFilepath= "pathto/k525mvmt1/omrtoxml.xml"
k525midistream = converter.parse(k525MidiFilepath)
k525omrstream = converter.parse(k525OmrFilepath)
```

5.1.3 Running `OMRMIDICorrector` on K.525

After the raw input has been massaged into `Stream` objects, the aligning process is simple. We create an instance of an `OMRMIDICorrector` object with the two streams as parameters. For the purposes of this example, we will use the default hasher that is built into the `OMRMIDICorrector` class. We will set the `debugShow` to `True` so that we can visualize the alignment between pairwise streams. Finally, we will call `processRunner` which will hash and align the parts in the stream. For now, we will manually create the `Fixer` objects for stream fixing. The aligner is able to correctly identify that the repeated 16^{th} notes

```
from music21 import *
OMC = alpha.analysis.omrMidiCorrector
k525omc = OMC.OMRMIDICorrector(k525midistream, k525omrstream)
k525omc.debugShow = True
k525omc.processRunner()
```

The outputs of the `showChanges` function are all in Appendix B

5.1.3.1 A Closer Look at Specifics of `processrunner`

A lot of magic happens within `processRunner`, all of which was described at a high level in sections 4.1.1.1 - 4.1.1.4. Here are some salient details of the process in the context of K.525 that I think deserve notice:

- Bass Doubles Cello Part - The OMR score has only four parts, whereas the MIDI score has 5. The `checkBassDoublesCello` method in `OMRMIDICorrector` is able to correctly identify this and thus does not reject the input on the basis of having a mismatching number of parts in the stream.

- Rhythmic Fault Tolerance: Beginning Rests - The first measure in the OMR score is a scanning and parsing error. The `Aligner` is robust enough to identify the beginning rest as a insertion error.

- Rhythmic Fault Tolerance: Propagating Rhythmic Shifts - By measure 5 in the Violin 2 part, we see that the OMR score has introduced a 16^{th} note shift in the rhythm which continues for a few measures.

- Notation Fault Tolerance: Tremolos - By measure 5 of the Violin 2 part and by measure 10 of the Viola part, the repeated 16^{th} notes get notated with tremolo slashes in the original OMR score. Of course, the MIDI protocol doesn't have a way of representing tremolo, so the notes are written out in long form. The corrector is able to match the written out repeated notes in the MIDI to the (incorrectly) scanned corresponding notes in the OMR.

5.1.4 Example 1: Similarity Metrics in K.525

Recall that the `similarityScore` is the ratio of `NoChange` operations to the total number of operations in the `changes` list. For each pair of streams in K.525, the `similarityScore` and `changesCount` are listed below:

	similarityScore	changesCount
Violin 1	0.51	{NoChange: 83, Ins: 30, Del: 10, Sub: 39}
Violin 2	0.30	{NoChange: 74, Ins: 125, Del: 9, Sub: 38}
Viola	0.72	{NoChange: 97, Ins: 21, Del: 9, Sub: 8}
Cello/Bass	0.87	{NoChange: 107, Del: 11, Sub: 5}

Table 5.1: `similarityScore` and `changesCount` for every pair of aligned streams

A visual examination of the pairs of streams suggests that these numbers and counts look correct. The Cello/Bass parts have the fewest colorations to their notes and no insertion changes at all. The least similar pair of parts is the Violin 2, which has multiple cases of the tremolo problem described above as well as propagating rhythmic shifts.

5.2 String Quartet No.7 in E-flat major, K.160-I.Allegro, W. A. Mozart

The previous section showed the alignment between parts in *Eine Kleine Nachtmusik*. In this section will use another piece by Mozart to demonstrate how the `EnharmonicFixer` works on correcting an excerpt of the Violin 1 part of K.160. This OMR score was also sourced from IMSLP [8].

5.2.1 Running `EnharmonicFixer` on K.160, Violin 1

After prepping the raw input files of K.160 in the same way as detailed above and running that through an `OMRMIDICorrector` instance, we will create an `EnharmonicFixer` instance using just the Violin 1 OMR stream and MIDI stream and the associated `changes` list.

```
from music21 import *
k160MidiFilepath = "pathto/k160mvmt1/miditoxml.xml"
k160OmrFilepath = "pathto/k160mvmt1/omrtoxml.xml"

k160midiStream = converter.parse(k160MidiFilepath)
k160omrStream = converter.parse(k160OmrFilepath)
k160omc = OMRMIDICorrector(k160midiStream, k160omrStream)
k160omc.processRunner()

V1Changes = k160omc.changes[0]
V1Midi = k160omc.midiParts[0]
V1Omr = k160omc.omrParts[0]
k160EF = fixer.EnharmonicFixer(V1Changes, V1Midi, V1Omr)
k160EF.fix()

k160omcViolin1OmrStream.show()
```

In AppendixB, the first two figures below show the alignment visualization of the Violin 1 part. The last figure shows the corrected OMR score after it has been changed by the `EnharmonicFixer`.

5.2.2 Analysis of Correction

Visual inspection of figures C-1 and C-2 suggests that the OMR failed to pick up the appropriate key signature of the movement and that is what lends to large number of changes in the first 10 or so measures. Inspection of figure C-3 shows that many of the pitch errors have been corrected using the `EnharmonicFixer`. Of course, the most ideal scenario would be to recreate the key signature, but having the correct pitch is a good start.

5.3 Results: Timing Tests

This section will evaluate the runtime on each of the three major modules of the system the `Hasher`, `Aligner`, and`Fixer` as well as the entire system, `OMRMIDICorrector` which is a combination of some number of instances of the three major modules. It will also discuss how the system would scale to different sized inputs.

For the following calculations, assume that the length of the input MIDI stream and the OMR stream are about the same, n, and that the number of parts in each of the streams is p.

5.3.1 Runtime Analysis: `Hasher`

Setup of the `Hasher` object is in $O(n)$ time. Building the list `finalEltsToBeHashed` requires scanning the input stream in full. Recall that the `tupleList` is a list of the stream element properties that will be hashed; call its length t. The runtime of the `hashStream` method is then $O(k \cdot n)$.

Thus, a single use of the `Hasher` is in $O(n) + O(k \cdot n) = O(n)$ time for small k. This is a reasonable assumption because the number of notes in a stream is likely to be far greater than the number of possible qualities of stream elements that can be hashed.

5.3.2 Runtime Analysis: `Aligner`

Setup of a `StreamAligner` object takes $O(n \cdot n) = O(n^2)$ time, dominated by the `populateDistanceMatrix` function. Traceback in the `calculateChangesList` method takes $O(n + n) = O(n)$ time. The length of the `changes` list is then also $O(n)$. The runtime of `showChanges` is proportional to the length of the `changes` list, calculated above to be $O(n)$

A single use of a `StreamAligner` object is in $O(n^2) + O(n) + O(n) = O(n^2)$ time.

5.3.3 Runtime Analysis: `Fixer`

The `OMRMIDIFixer` and the `EnharmonicFixer` both require negligible setup. The `fix` function's runtime is proportional to the length of the `changes` list input, shown above to be $O(n)$.

A single use of a `Fixer` object is in $O(n)$ time.

5.3.4 Runtime Analysis: `OMRMIDICorrector`

For a single instance of an `OMRMIDICorrector` with OMR and MIDI streams of length n and p parts, there are $2 \cdot p$ textttHasher objects, p `Aligner` objects, and p `Fixer` objects created. In total this comes out to a time of:

$$2 \cdot p \cdot O(n) + p \cdot O(n^2) + p \cdot O(n)$$

$$= p \cdot O(n^2) + 3 \cdot O(n)$$

For small p, this collapses to $O(n^2)$.

5.3.5 Empirical Results

Using the the first movement of K.160, I timed some results using the `% timeit` functionality of iPython. First I tested the major modules on a single stream, the Violin 1 part of K.160. Then I tested the OMR/MIDI Correction system on all of the K.160.

5.3.6 `%timeit`: Hasher

After setting up a `Hasher`, `% timeit` reported an average time of 621 ms per call to hash the MIDI stream and average time of 617 ms per call to hash the OMR stream. The length of the hashed MIDI stream was 662 and and the length of the hashed OMR stream was 669.

5.3.7 `%timeit`: Aligner

After setting up a `StreamAligner` object using the hashed MIDI stream and the hashed OMR stream, I ran `% timeit` on the `align` function and got an average of 2.39 s per call. Since the `align` function itself hashes input streams if not already hashed, I passed in the hashed streams. This produced more accurate measurement of the time for the alignment process, separate from the hashing process.

5.3.8 `%timeit`: Fixer

I set up a `EnharmonicFixer` object using the `changes` list from the aligning step and the original K.160 Violin 1 OMR and MIDI streams. Running `% timeit` on the `fix` function returned an average of 12.7 ms per call.

5.3.9 `%timeit`: OMR/MIDI Corrector

After testing the timing of each of the major modules, I tested the timing of the OMR/MIDI Correction system on the entire first movement of K.160. The average time to hash, align, and fix the four instrumental parts in K.160 was 12.94 s.

An entire run of the OMR/MIDI Corrector system on K.160 require 8 calls to the `Hasher`, 4 calls to the `Aligner`, and 4 calls to the `Fixer`. Using the empirical numbers

from above, this gives an estimate of 14.56 seconds, which is in the range of the 12.94 seconds that `%timeit` gives us. The speedup in the empirical time versus the calculated time likely comes from the fact that there is considerable setup that was reused in calling the `OMRMIDICorrector` once instead of its constituent modules.

5.3.10 Scalability

If a single movement of a four-part Mozart quartet takes on the order of tens of seconds to hash, align, and fix, then most multi-movement pieces for chamber ensembles should take on the order of minutes to hash, align, and fix. Larger orchestral works might take on the order of tens of minutes to hash, align, and fix. Especially for the first system of its type, these preliminary timings of the `OMRMIDICorrector` have good implications.

The bottleneck of the system lies in the `Aligner`. The method implemented in my system is a basic string alignment algorithm, and there is considerable room for speedup using techniques such as The Four Russian Algorithm [3].

5.3.11 Implications of Free Music

The lasting and tangible results of this research will hopefully be in helping providing more free and high-quality musical scores to the public. With ISMLP alone having over 350,000 scores that are freely available, there is a lot of potential for my work to liberate otherwise sub-quality scanned musical scores and turn them into digital scores that can be conveniently analyzed, replicated, and distributed. It's my hope that this work will be able to facilitate musical research, provide individuals and groups with easy access to practice material, and help bring more music to all of our lives.

Appendix A

Alignment Visualization of K.525

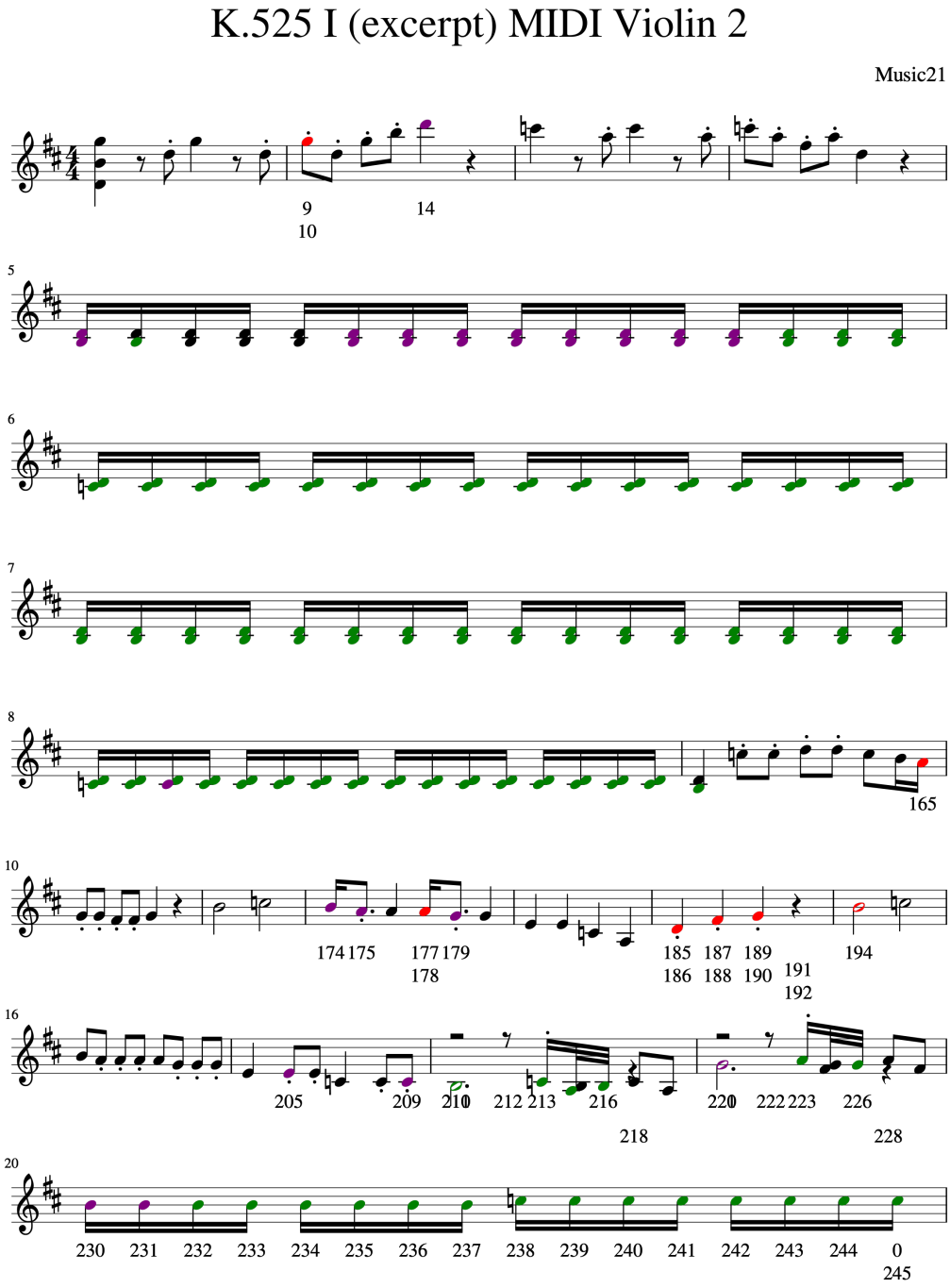

Figure A-1: Visual display of changes in Violin 1 of target (MIDI) stream.

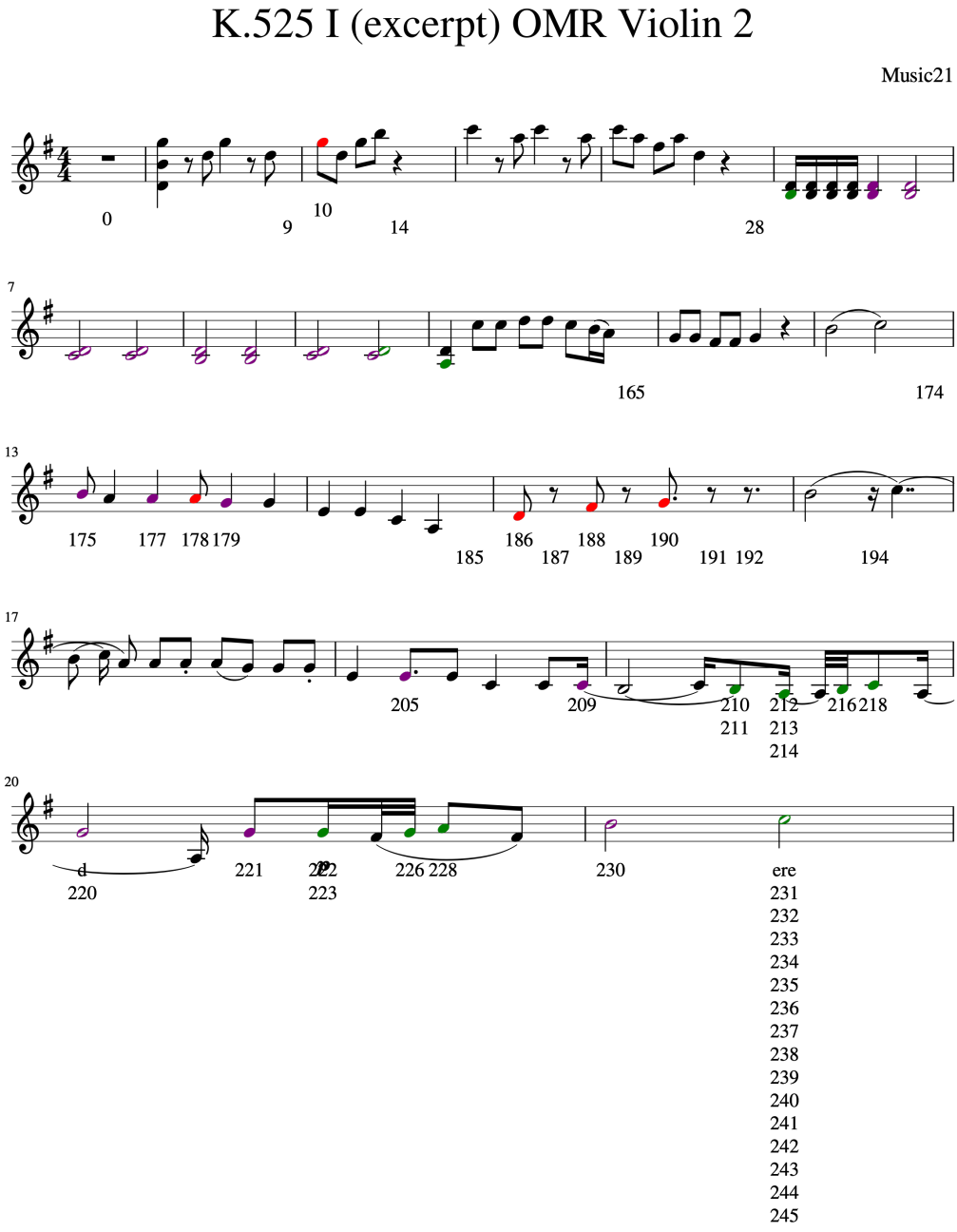

Figure A-2: Visual display of changes in Violin 1 of source (OMR) stream.

Appendix B

Alignment Visualization of K.525

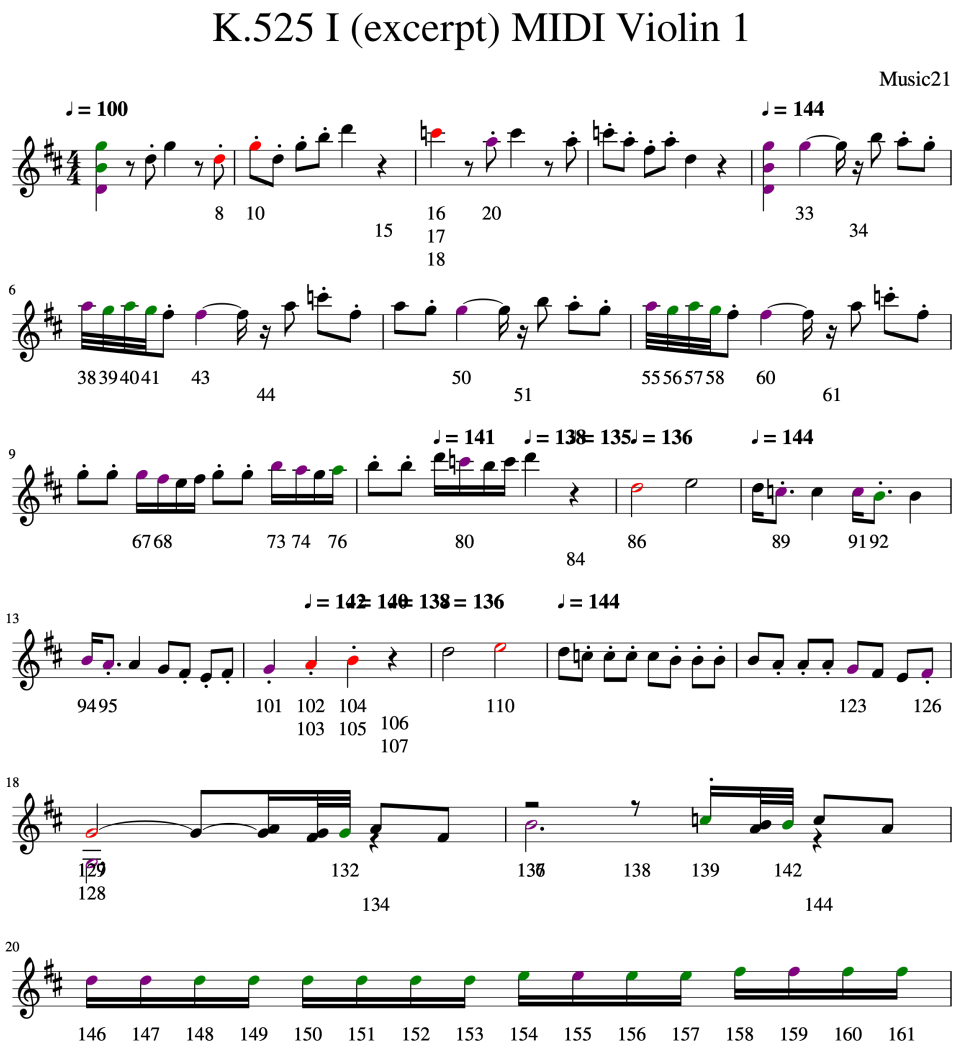

Figure B-1: Post-alignment visualization of an excerpt of the MIDI Violin 1 part in K.525-I.Allegro.

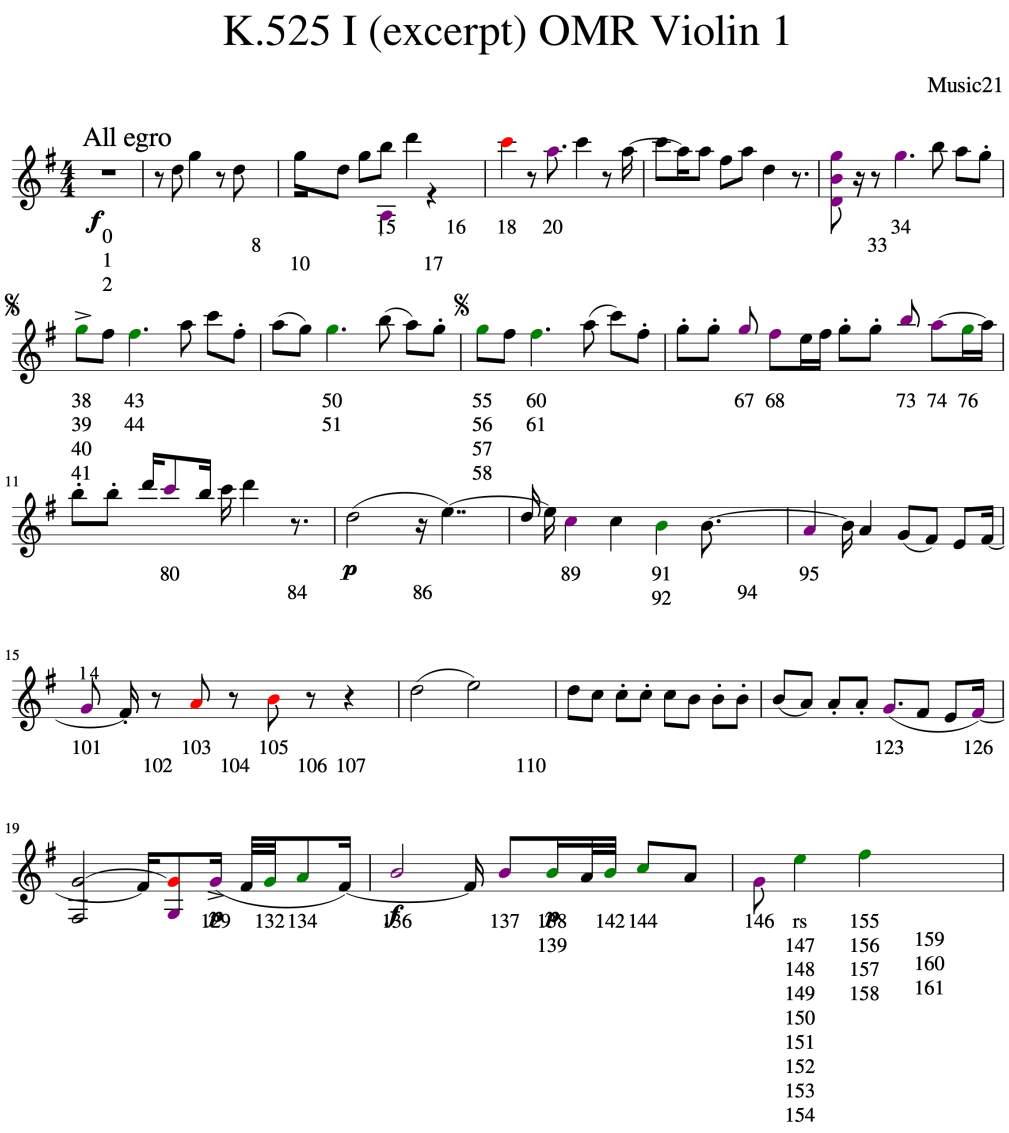

Figure B-2: Post-alignment visualization of an excerpt of the OMR Violin 1 part in K.525-I.Allegro.

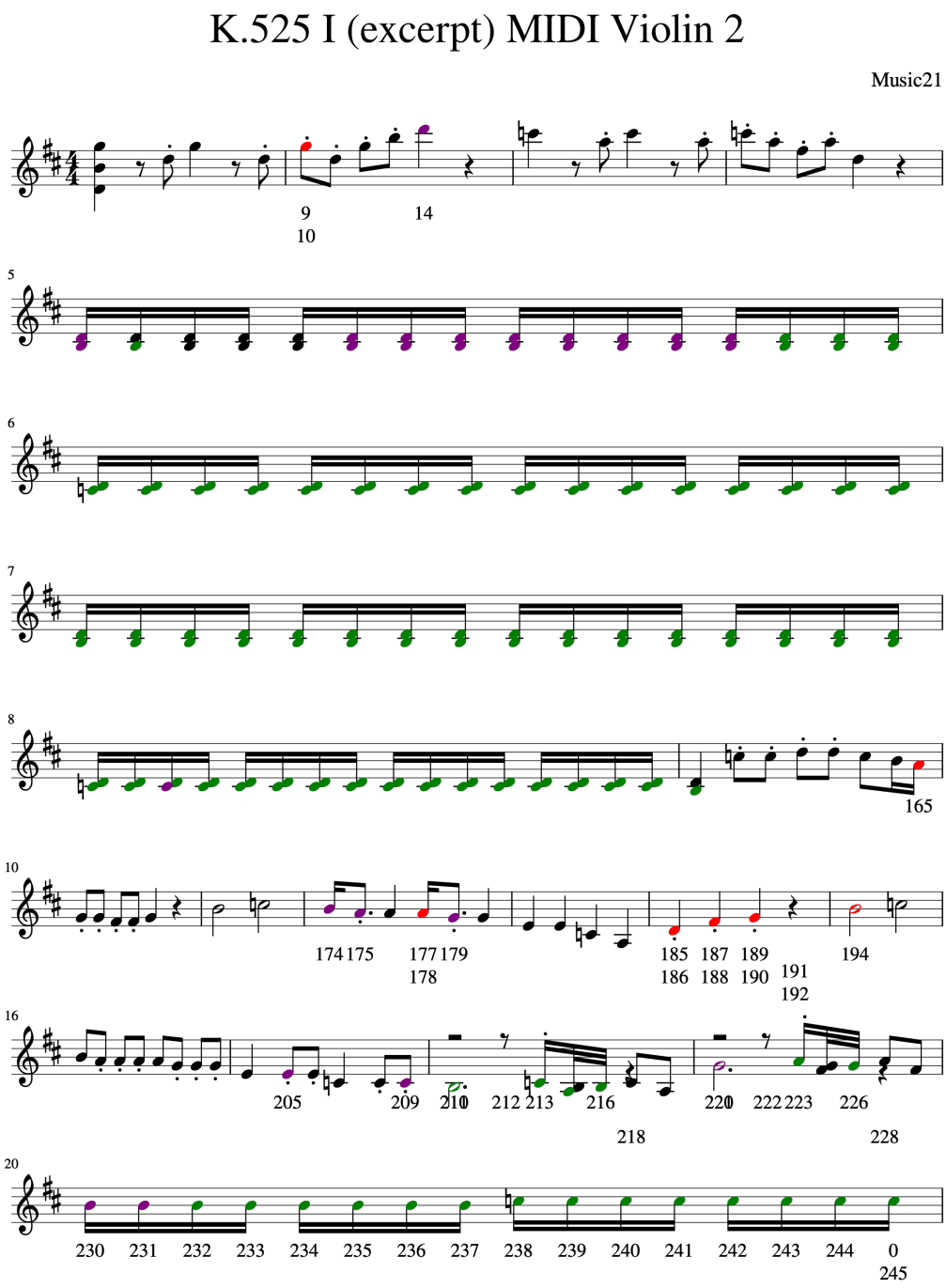

Figure B-3: Post-alignment visualization of an excerpt of the MIDI Violin 2 part in K.525-I.Allegro.

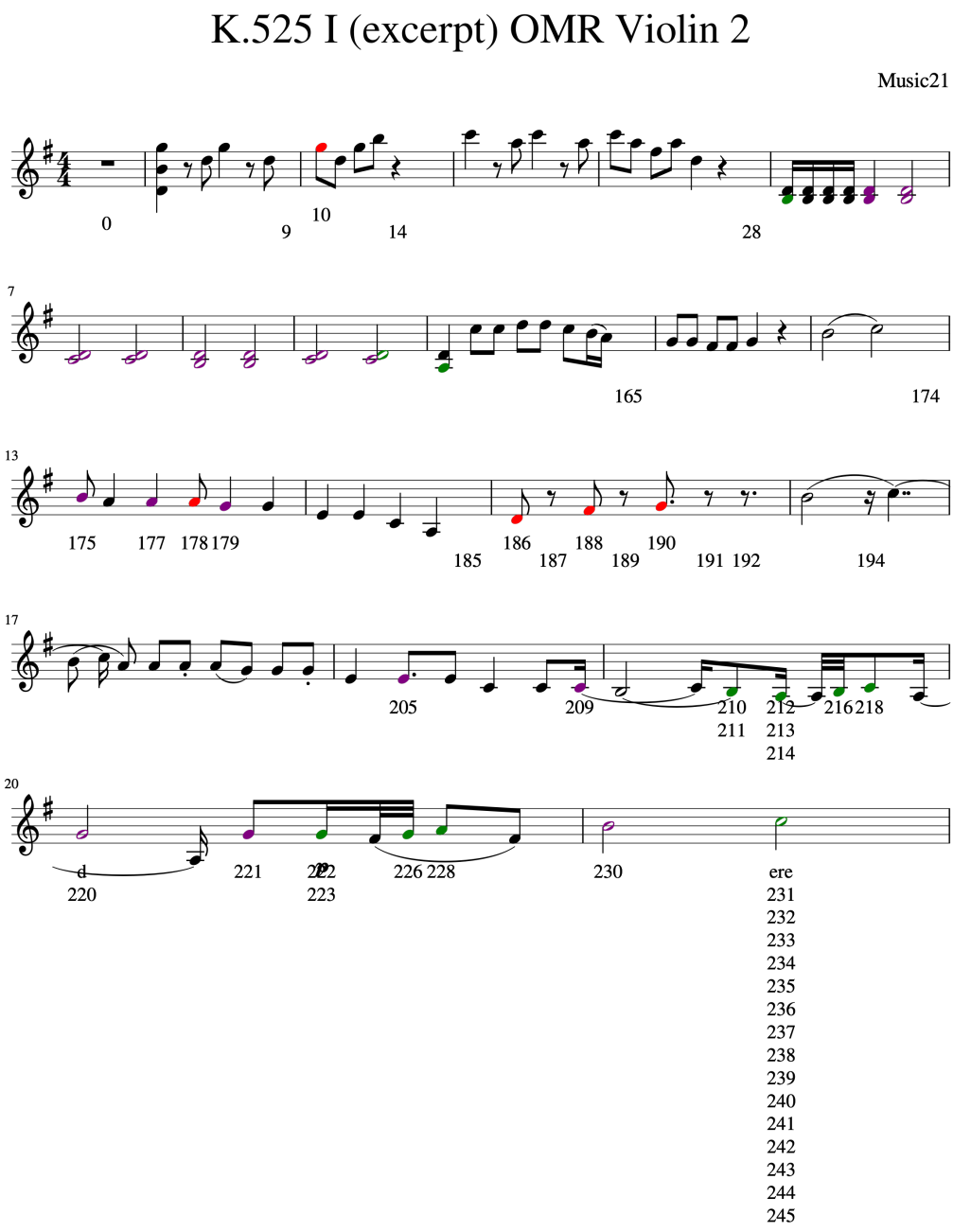

Figure B-4: Post-alignment visualization of an excerpt of the OMR Violin 2 part in K.525-I.Allegro.

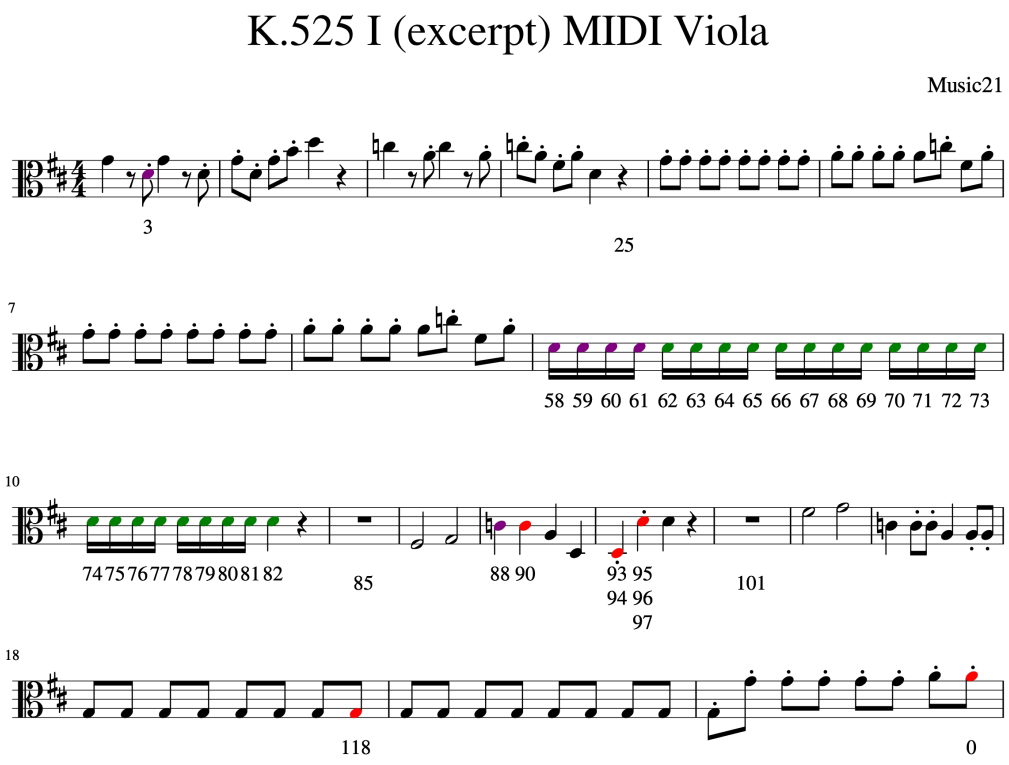

Figure B-5: Post-alignment visualization of an excerpt of the MIDI Viola part in K.525-I.Allegro.

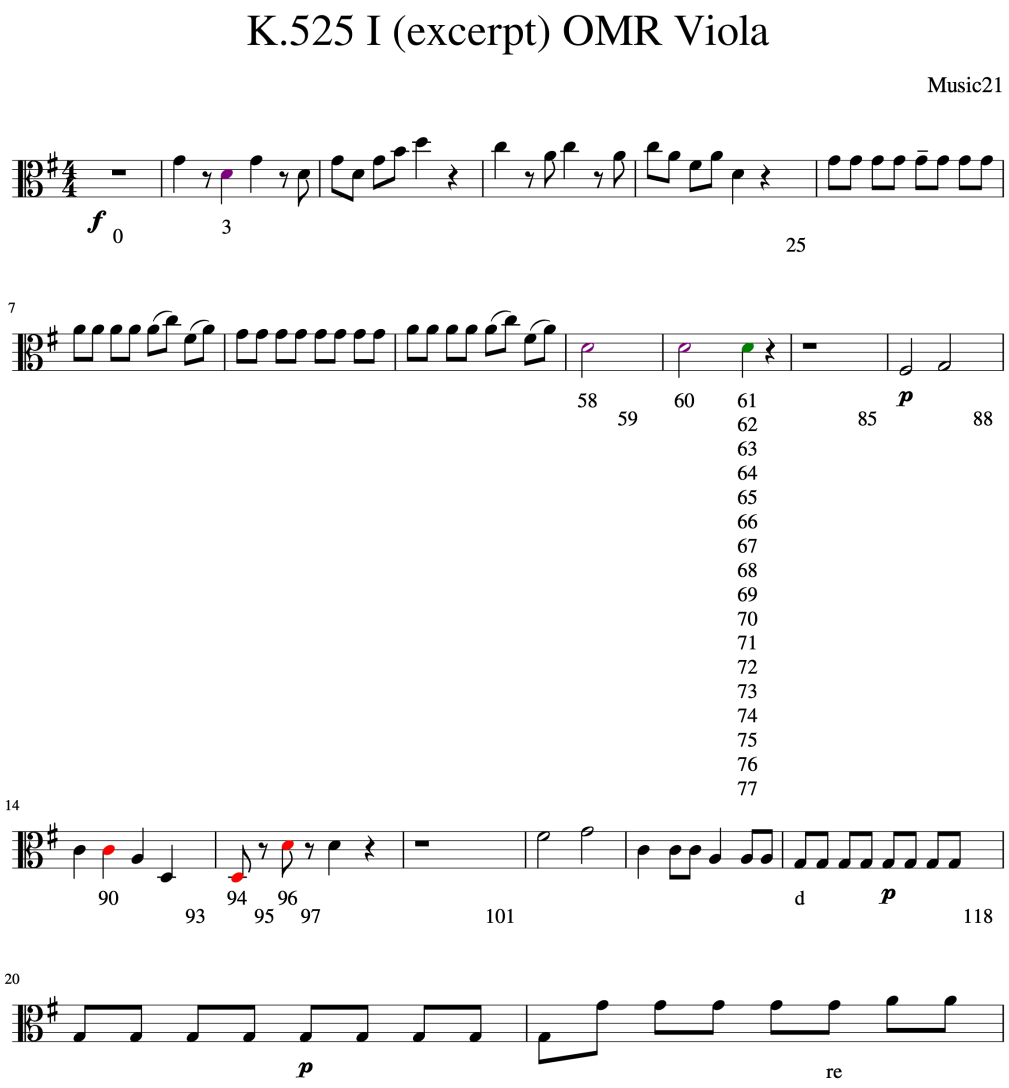

Figure B-6: Post-alignment visualization of an excerpt of the OMR Viola part in K.525-I.Allegro.

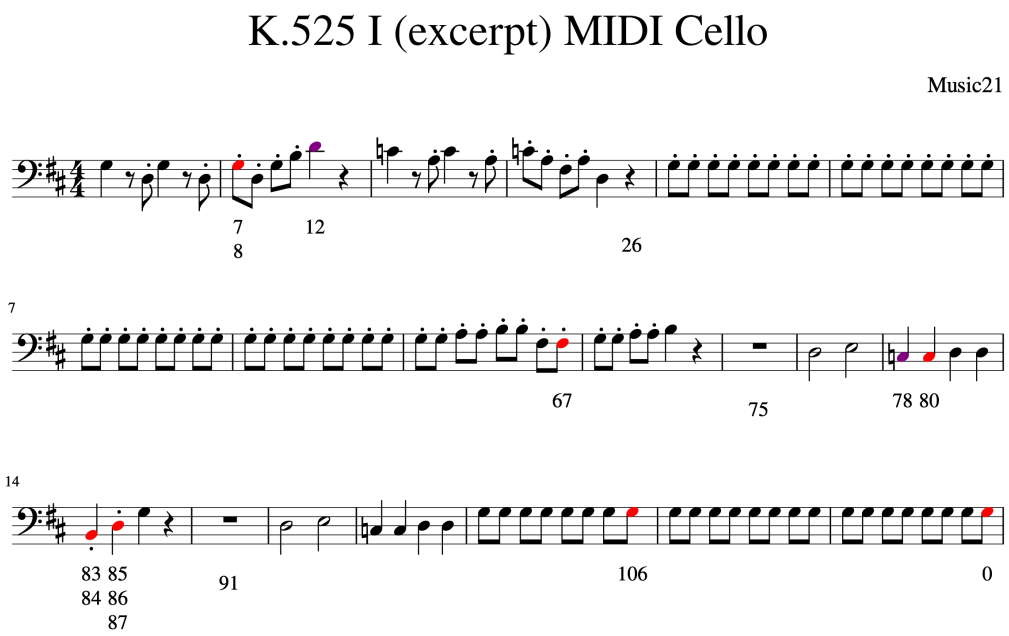

Figure B-7: Post-alignment visualization of an excerpt of the MIDI Cello part in K.525-I.Allegro.

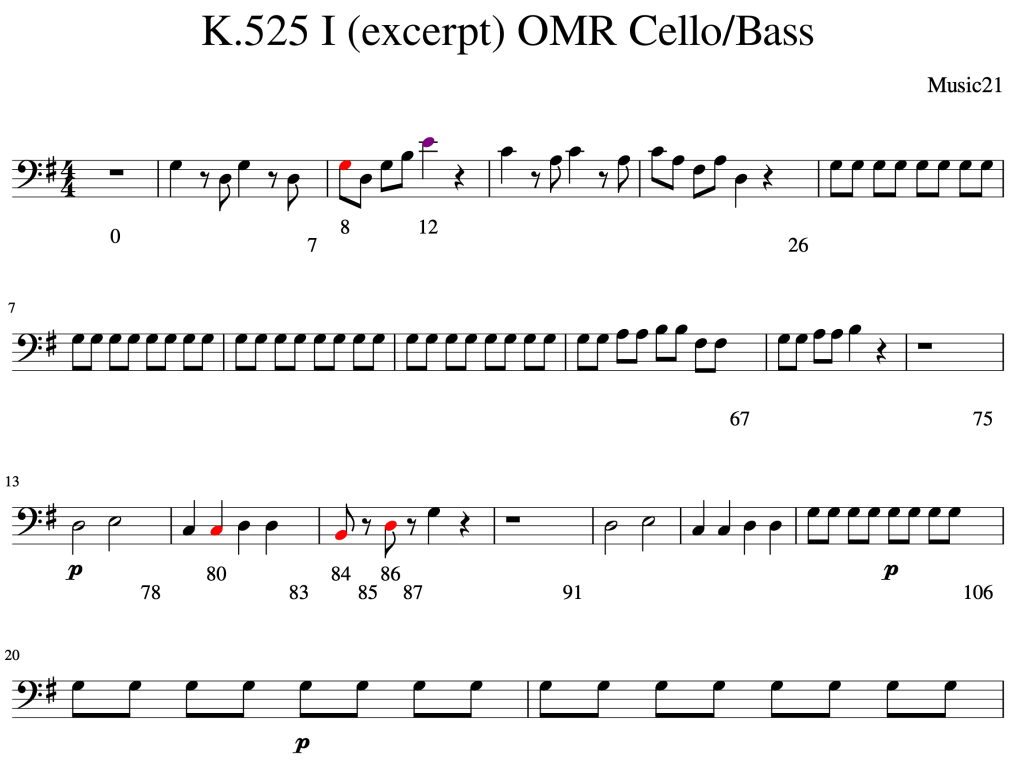

Figure B-8: Post-alignment visualization of an excerpt of the OMR Cello and Bass part in K.525.

Appendix C

Alignment Visualization of Violin 1 part in K.160

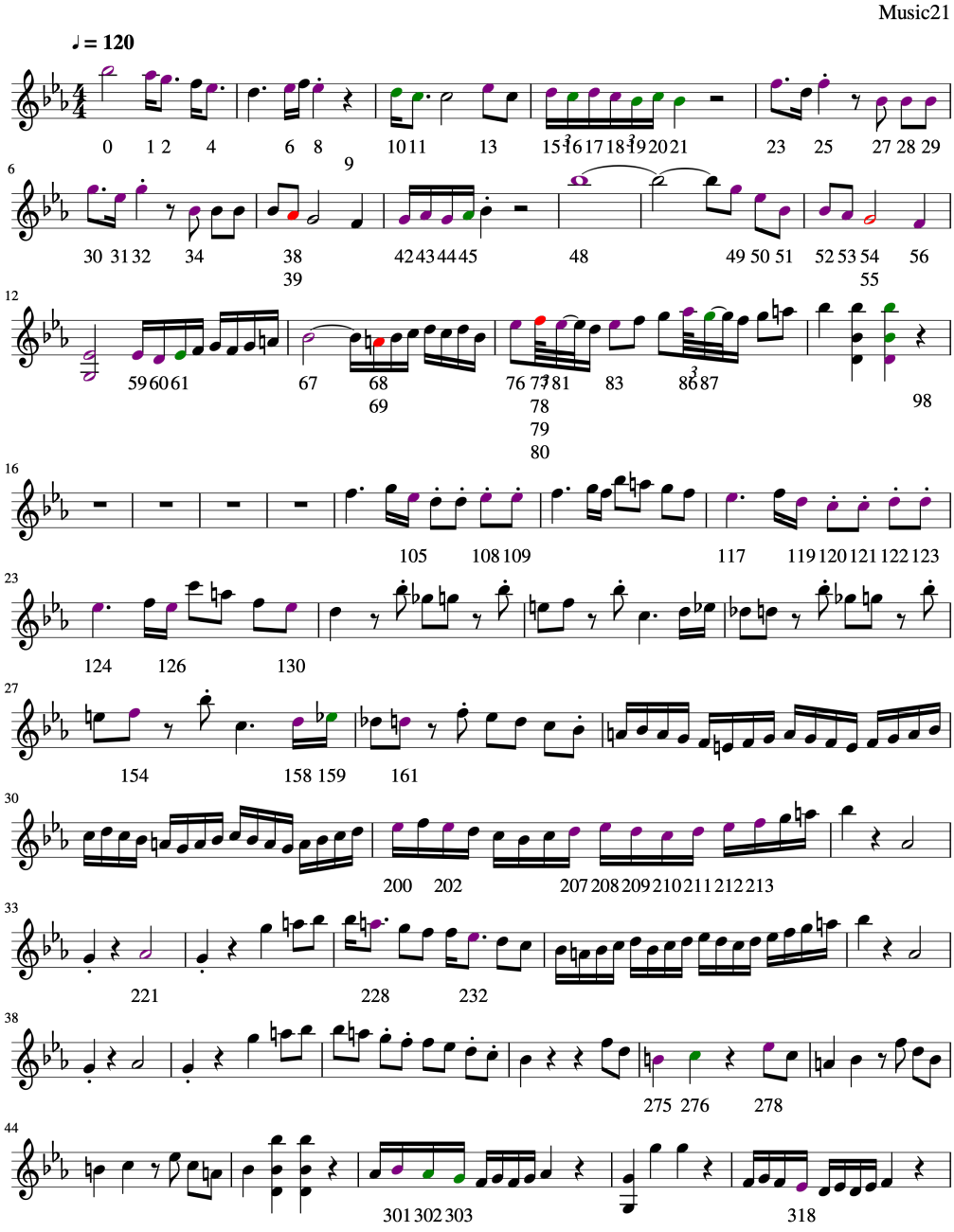

Figure C-1: Post-alignment visualization of the MIDI Violin 1 part in K.160.

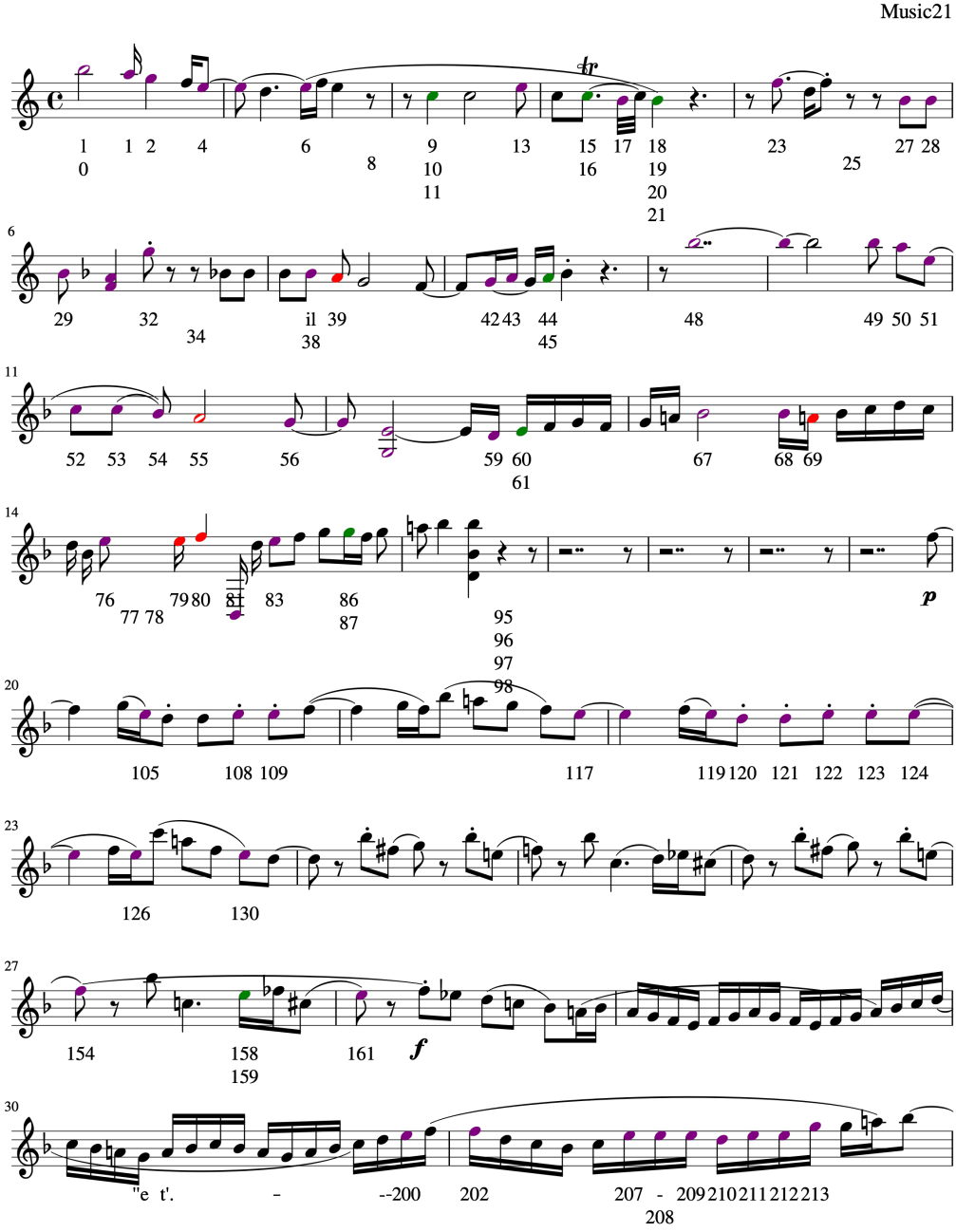

Figure C-2: Post-alignment visualization of the OMR Violin 1 part in K.160.

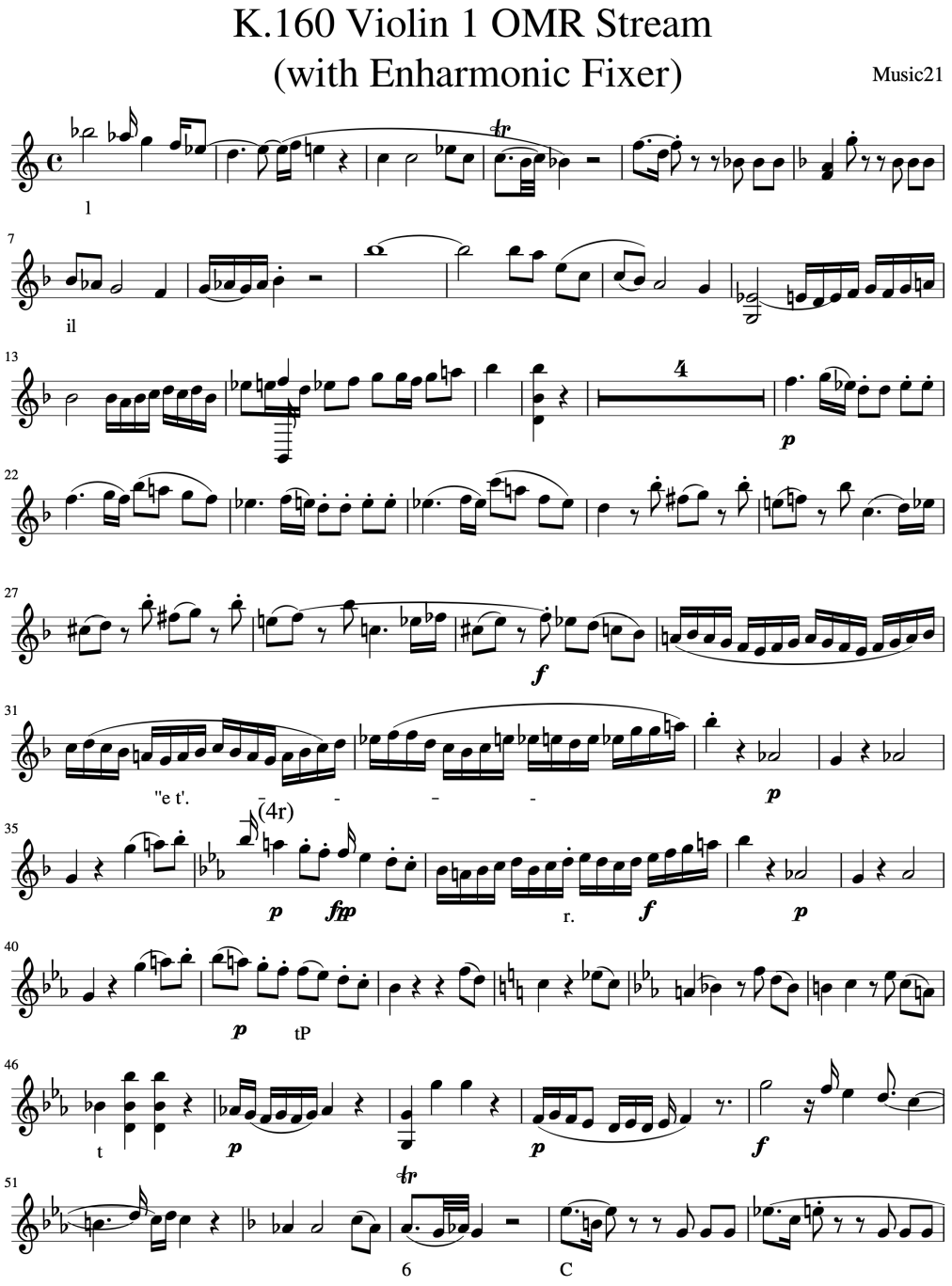

Figure C-3: Post-enharmonic and pitch fixing visualization of the OMR Violin 1 part in K.160.

Bibliography

[1] Donald Byrd and Jakob Grue Simonsen. Towards a Standard Testbed for Optical Music Recognition: Definitions, Metrics, and Page Images. *Journal of New Music Research*, 44.3:169– 195, 2015.

[2] Maura Church and Michael Scott Cuthbert. Improving Rhythmic Transcriptions via Probability Models Applied Post-OMR. In Hsin-Min Wang, Yi-Hsuan Yang, and Jin Ha Lee, editors, *Proceedings of the 15th Conference of the International Society for Music Information Retrieval*, pages 643– 647. ISMIR, 10, 2014.

[3] Algorithms for Computational Biology. Sequence Alignment and Dynamic Programming, Lecture notes for 6.096. `https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-096-algorithms-for-computational-biology-spring-2005/lecture-notes/lecture5_newest.pdf`.

[4] Walter B Hewlett. A Base-40 Number-line Representation of Musical Pitch Notation. *Musikometrika*, 4:1–14, 1992.

[5] IMSLP/Petrucci Library Free Public Domain Sheet Music. `http://imslp.org/`. Accessed: 2015-10-10.

[6] Peter Joslin. *Music Retrieval based on Melodic Similarity*. PhD thesis, Universiteit Utrecht, 2007.

[7] Wolfgang Amadeus Mozart. Eine Kleine Nachtmusik, K. 525. `http://ks.petruccimusiclibrary.org/files/imglnks/usimg/5/57/IMSLP01776-Mozart_EineKleineNachtmusik_Score.pdf`. Accessed: 2017-05-23.

[8] Wolfgang Amadeus Mozart. String Quartet No.7 in E-flat major, K.160/159a. `http://ks.petruccimusiclibrary.org/files/imglnks/usimg/f/f1/IMSLP64127-PMLP05214-Mozart_Werke_Breitkopf_Serie_14_KV160.pdf`. Accessed: 2017-05-23.

[9] Saul B. Needleman and Christian D. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology*, 48(3):443– 453, March 1970.

[10] Openscore. `https://musescore.org/en/user/57401/blog/2017/01/11/introducing-openscore`. Accessed: 2017-05-22.

[11] An excerpt of a multiple sequence alignment of tmem66 proteins. `https://commons.wikimedia.org/wiki/File:An_excerpt_of_a_multiple_sequence_alignment_of_TMEM66_proteins.png`. Accessed: 2017-05-10.

[12] Ana Rebelo, Ichiro Fujinaga, Filipe Paszkiewicz, Andre R. S. Marcal, Carlos Guedes, and Jaime S. Cardoso. Optical Music Recognition - State-of-the-Art and Open Issues. *International Journal of Multimedia Information Retrieval*, 1(3):173–190, Feb 2012.

[13] David Temperley. An Algorithm for Harmonic Analysis. *Music Perception: An Interdisciplinary Journal*, 15.1:31– 68, 1997.

[14] Vladimir Viro. Peachnote: Music Score Search and Analysis Platform. In Anssi Klapuri and Leider Kolby, editors, *Proceedings of the 12th Conference of the International Society for Music Information Retrieval*, pages 359– 362. ISMIR, 10, 2014.

[15] Christopher Wm. White. *Some statistical properties of tonality, 1650-1900*. PhD thesis, Yale University, 2013.