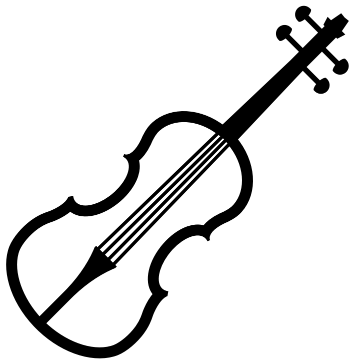
Open Strings

Fingering inference for violin music



Joseph Morag

Submitted in partial fulfillment of the requirements for the
Master of Music in Music Technology
in the Department of Music and Performing Arts Professions
Steinhardt School
New York University

Advisor: Dr. Brian McFee
Reader: Dr. Tae Hong Park
April 21, 2021

Abstract

Choosing fingerings for violin music is a hard problem, both for humans and computers. Though there is a great deal of pedagogical literature by famed teachers and performers on the subject, most violinists wind up developing the skill of picking a fingering only through years of study and trial and error. Besides being a hard problem, choosing fingerings is one with no agreed upon optimal solution. What distinguishes a good fingering from a bad one is highly individualized to a specific musician's tastes.

This thesis describes OpenStrings, an expert system style inference engine for fingerings with configurable rules to account for the individual preferences of the user. Evaluation by expert violinists of the generated fingerings concluded that the results are competitive with human generated fingerings.

Acknowledgments

I've wanted to do a project like this since the end of college. After such a long time of this being an idea floating around my head, I'm ecstatic to have something tangible to show for it. There are so many people who had a hand in making this thesis possible.

To my family, thank you for your constant and loving support throughout this and all of my educational endeavors, and for always believing, no, *knowing* that I would be able to finish this, even when I didn't.

To my wonderful partner, Lauren, thank you for helping me through our marathon editing sessions, and for actually reading the entire thing. I love you.

To my advisor, Dr. Brian McFee, thank you for providing such valuable insights and advice, and for spending so much time with me in office hours talking about everything from algorithms to graph theory to programming languages.

To the Haskell community, thank you for creating the libraries which allowed me to write this thesis in my favorite programming language. Special thanks go to Ed Kmett for writing `lens` and answering my questions about it on reddit, and to Michael Snoyman for `yesod`.

To Simon Schmid, thank you for working with me to add string numbers on grace notes to OpenSheetMusicDisplay. I probably spent more time on that than I should have, but seeing those tiny Roman numerals appear in the right place was worth it.

To my coworkers at Lantern, thank you for your enthusiasm for hearing me talk about this project, and for teaching me enough about the internet so that I could actually implement it.

To my friends Sophia, Itamar, Nicholas, Camille, Gabrielle, Victor, Emelia, and Ian, fantastic musicians all, thank you for being the first to use this and for your crucial feedback.

Lastly, to my beloved violin teachers, Anna Pelekh and Aaron Rosand, thank you for teaching me what I wanted on the violin and how to get it. I wish you could have gotten to see this finished. You would have enjoyed it.

Contents

1 Introduction 6

2 Literature Review 8
- 2.1 Pedagogical Approaches to Violin Fingering . . . 8
- 2.2 Algorithmic Approaches to Violin Fingering . . . 11

3 Methodology 14
- 3.1 Fingering Database . . . 14
- 3.2 User interface . . . 15
  - 3.2.1 Upload a passage and render it as sheet music . . . 15
  - 3.2.2 Conveniently edit fingerings and persist them to the database . . . 15
  - 3.2.3 Query the server for an inferred fingering, respecting any entries made by the user . . . 16
- 3.3 Fingering inference . . . 17
- 3.4 Evaluation . . . 24

4 Results 25
- 4.1 Fingering samples . . . 25
  - 4.1.1 Generated fingerings . . . 25
  - 4.1.2 Control group fingerings . . . 25
- 4.2 User responses . . . 26
- 4.3 Comparison with prior work . . . 27

5 Conclusion 33
- 5.1 Discussion . . . 33
- 5.2 Future work . . . 33

5.2.1 Longer term memory 33
5.2.2 User interface improvements 34
5.2.3 Fingering data collection 34
5.2.4 Extensibility 35

A Computer Representations of Symbolic Music **36**

B Selected code samples **38**
B.1 Merge multiple voices into time steps 38
B.2 Fingering penalties 40

List of Figures

1.1 An excerpt from Beethoven's last sonata, Op. 96, with fingerings from five editions. Reproduced from Ysaÿe, *Exercises et Gammes*. 6

2.1 How Flesch and Galamian would finger the same arpeggio 10

3.1 Entity Relationship Diagram for OpenStrings 15
3.2 A snapshot of the fingering editor. The blue notehead is currently selected. 17
3.3 Transformation of multiple voices into a single voice 18
3.4 A graphical representation of the fingering inference problem 19

4.1 Prokofiev: Violin Concerto No.2, Op.63, I. Allegro moderato mm. 101-104 26
4.2 Bartók: Violin Concerto No. 2, III. Allegro molto mm. 185-189 26
4.3 Ysaÿe: Solo Sonata No. 3, Op. 27 "Ballade" mm. 76-87 27
4.4 Strauss: Don Juan, Op. 20 mm. 31-36 28
4.5 Bruch: Scottish Fantasy, Op. 46, I. Introduction mm. 66-82 28
4.6 Mozart: Violin Concerto No. 4, K. 218, I. Allegro mm. 42-56 29
4.7 How comfortable do you find this fingering to play? 29
4.8 How expressive do you find this fingering allows a violinist to be? 30
4.9 How idiomatic do you find this fingering? 30
4.10 Would you say this fingering was generated by a human or by a computer? 31
4.11 Comparison of OpenStrings and prior work by Nagata et al. 31
4.12 Comparison of OpenStrings and prior work by Maezawa et al. 32

Chapter 1

Introduction

Choosing fingerings is an integral part of any violinist's skill set. A good fingering can greatly simplify the execution of a passage of music, whereas a bad one can render it impossible to play. Prolonged use of an awkward fingering can even lead to pain and eventual injury. Fingerings are highly personal; the astute listener can distinguish Heifetz from Kreisler simply by how they approach shifting. Even within the space of several measures of a "simple" scalar passage from Beethoven's last sonata (see Figure 1.1), we see that though there is agreement in the first two measures, there are *at least* four equally valid ways to play the rest of the excerpt.

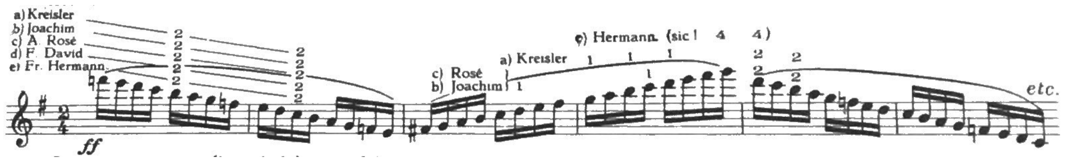


Figure 1.1: An excerpt from Beethoven's last sonata, Op. 96, with fingerings from five editions. Reproduced from Ysaÿe, *Exercises et Gammes.*

Devising a fingering is not only important for great violinists, but for players at all skill levels. Beginning students usually rely entirely on their teachers for fingerings, and it generally takes many years before they feel comfortable designing their own fingerings that suit their hands. Unfortunately, many violinists may never develop the necessary intuition to figure out good fingerings, relying on suggestions from their teachers, friends, or printed editions for their entire careers.

Why is this the case? Since the instrument's standardization in the 18th century, the average level of violin playing has only improved, due to a combination of more accessible learning resources, new technical discoveries, and a steadily increasing number of practicing violinists. One only need observe that the vast majority of professionals and many advanced students can play pieces by Paganini or Wieniawski that, at the time they were composed, were only executable by a handful of the greatest violinists in the world. Looking even further back in history, the concertos of de Beriot, chamber violinist to King Charles X of France, which he wrote for a violinist of his own ability, are today given to young children to perform.[1]

Although knowledge of violin playing has come so far in the centuries since the instrument's creation, devising fingerings remains a difficult problem. For one, most performers do not share their fingerings publicly with the rest of the violin-playing community. This is due to a combination of the desire to guard "trade secrets" and the lack of a platform on which to easily publish such information. After all, there are only a handful of violinists who edit published editions. The other reason is simply because the problem is almost intractably hard. Most notes on the violin can be played on all four strings with all four fingers. For an n note passage, this leads to a naive upper bound of $\approx 16^n$ possible fingerings. Problems with exponential search spaces are common in computer science and artificial intelligence, and there are algorithms for dealing with them that are vastly more efficient than brute-force search. While there exists prior work that takes advantage of these methods, no existing system generates fingerings comparable to those generated by an expert human or enjoys widespread use in the performing community. This thesis addresses both of these concerns by creating a website for violinists to easily share fingerings with their colleagues, and integrating a novel inference engine that generates fingerings comparable to those made by human practitioners.

Chapter 2 provides an overview of prior work in pedagogical and algorithmic approaches to fingering assignment. Chapter 3 describes the implementation of the user interface and the inference engine. Chapter 4 summarizes the results of a survey of expert violinists judging the quality of generated fingerings and compares them with previous algorithmic systems.

[1]https://www.youtube.com/watch?v=NAGazAcwCm8

Chapter 2

Literature Review

2.1 Pedagogical Approaches to Violin Fingering

The study of violin fingering methodology is an ancient one, far predating the advent of modern computers. One of the oldest examples in the literature is the *Violinschule* by Leopold Mozart, father of the legendary Wolfgang Amadeus (Mozart et al., 1756/1985). Although it originally appeared in 1756, the elder Mozart's advice on fingerings is still quite useful to this day. He advocates for the judicious use of extensions to reach higher notes on the E string while remaining in "Whole" (what today we would call "Third") Position, as well as for the need to remain in a single position as much as possible during technical passages in order to minimize shifting. He is even aware of the technique of changing fingers on repeated instances of the same note in order to prepare the hand for an upcoming passage. Some of his writing, however, has not aged as well. By calling Third Position "Whole" and Second Position "Half" as well as the order in which he presents them in his treatise, he has engendered an aversion among many violinists at all levels to using Second Position for any reason whatsoever. Even today, many professional violinists take extraordinary measures to avoid all even positions in passages where using other fingerings would ostensibly incur more difficulties. In fairness to Mozart, much of the music he would have played at court was in D or A major, keys in which using even positions is awkward and unnecessary. In modern repertoire, with music written in all keys, and quite often no key at all, violinists stand to benefit greatly from the use of even positions.

During the 20th Century, violin technique grew rapidly, with many pedagogues forming competing schools of thought on how the instrument should be played. By this point, innovations such as the modern Tourte bow and the chinrest were in universal use by violinists, obsoleting many of the practices championed by Leopold Mozart's *Violinschule*. In Western Europe, one of the preeminent violin teachers was Carl Flesch, whose students included superstars Ivry Gitlis, Ida Haendel, and Joseph Hassid. He was one of the first to point out that in many passages, it is desirable to play them higher on the G string than more comfortably on the D or A strings, as the sound on the G is more brilliant and penetrating (Flesch et al., 1924/2000; Flesch et al., 1966/1978). During his lifetime, the steel E string came into prominence, displacing the traditionally used plain gut E. Flesch was a strong opponent of this change, advocating that the open E be replaced in most cases by the same pitch on the A string, due to the former's harsh sound. Today, the plain gut E is almost non-existent, but Flesch's admonition to avoid the open steel E is still heeded by many violinists, especially in orchestral settings.

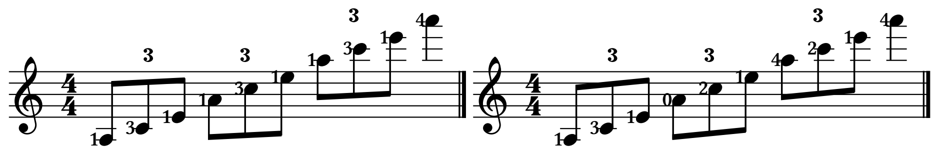
In America, violin pedagogy over the last 70 years has been dominated by the teachings of Ivan Galamian, who produced such luminaries as Itzhak Perlman, Pinchas Zukerman, and Kyung Wha Chung. One of Galamian's primary contributions to the art of violin fingering was his insight that the hand's frame remain intact to as great an extent as possible, in order to preserve intonation. To that end, he believed in the paramount importance of practicing octaves, as they set the hand's position perfectly in all but the highest reaches of the fingerboard (Galamian, 1962). Another of Galamian's innovations was the use of the "creeping" method in moving between positions. Essentially, instead of only stretching or shifting, one can combine the two motions by extending the first or fourth finger to a note and then using it as an anchor to drag the rest of the hand into position.

Unlike Flesch, Galamian did not have an aversion to the open E string, and recommended its use rather frequently in his *Contemporary Violin Technique* (Galamian & Neumann, 1966). Both Flesch and Galamian recognized the utility of the half step shift, i.e. using the same finger for two consecutive notes a half step apart, in facilitating smooth transitions between positions while minimizing undesirable noise from shifting, although Galamian recommended its use in many more scenarios than Flesch.

Peter Myers presents very thorough comparison of the fingering guidelines of Flesch and Galamian in his thesis (Myers, 2011), which compiles a huge corpus of editions fingered by the two teachers and distills them into concise, general guidelines for various types of passages. Summarizing Myers' work, the most pronounced differences in Flesch's and Galamian's recommendations are that

- Very broadly speaking, Flesch places more emphasis on technical considerations in choosing fingerings while Galamian insists that musical considerations be the more important determining factor.

- In technical passages, Galamian favors shifting while Flesch favors string crossings, but in lyrical passages, their opinions are reversed.
- In arpeggios, Flesch advocates using different fingers for consecutive notes, while Galamian advises shifting with the same finger (see Figure 2.1 for an example).
- In the matter of oblique crossings, where over two consecutive notes the same finger moves a string over but stays at the same distance from the nut, Galamian allows for them if the notes are separated by a bow change or other suitable break in the sound, whereas Flesch is against their use in nearly all cases.
- Flesch recommends the use of harmonics much more liberally than Galamian.
- Galamian permits the use of open strings in lyrical passages whereas Flesch forbids it.
- Flesch shies away from using the fourth finger for expressive notes while Galamian encourages violinists to develop its strength so that it can vibrate as well as the other three fingers.
- Flesch favors *portamento* much more than Galamian.
- Galamian advises different fingerings for Baroque and Romantic passages, while Flesch adheres to the same general guidelines no matter the era of the music.



(a) Carl Flesch **(b)** Ivan Galamian

Figure 2.1: How Flesch and Galamian would finger the same arpeggio

Although much of the contemporary violin methodology is derived from the teachings of Flesch and Galamian, there are several other pedagogues and violinists who made very important contributions to the fingering literature. One of the preeminent figures in the Soviet violin school, I. M. Yampolsky, puts forth the notion of an anchoring finger to secure the hand during awkward passages in order to ensure good intonation (Yampolsky et al., 1980). In addition to reiterating many of the points of Galamian and Flesch, Yampolsky also posits that bowings must be taken into account as well as just the notes when determining fingerings. Otherwise, Yampolsky's guidelines are fairly close to what Flesch recommends.

The great Belgian violinist, Eugène Ysaÿe, best known for his 6 solo sonatas, presents several novel ideas absent from the works of Flesch and Galamian in his oft-overlooked *Exercises et Gammes*. These include the use of harmonics in fast passages to ease shifting and the nearly ubiquitous use of even positions and half step shifts in order to facilitate technical comfort (Ysaÿe & Szigeti, 1967). Max Rostal, in his foreword and supplementary fingerings for Flesch's *Scale System* (Flesch & Rostal, 1942/1987), advocates for the one finger half step shift to be the most used transition mechanism presented in the book, though, when playing chromatic scales which contain exclusively half steps, modern practice recommends the "newer" fingering method of 1-2-1-2-3-4 on a single string over repeated half-step shifts, as it sounds much cleaner. Ruggiero Ricci, widely known for his thrilling renditions of the most challenging virtuoso works in the solo repertoire, advocates for an extreme version of Galamian's creeping method he calls "positionless" violin playing, using *glissandi* exclusively to traverse the instrument (Ricci & Zayia, 2007). Given this wide range of opinions, we see that though there is broad agreement on some foundational aspects of violin fingering, there is no "one-size-fits-all" solution.

2.2 Algorithmic Approaches to Violin Fingering

The first work in algorithmic fingering for string instruments is by Samir Sayegh in his Master's thesis (S. Sayegh, 1988). In it, Sayegh lays out the mathematical foundation of the problem as a graph searching algorithm over a network of nodes. The network is defined as a series of layers, with each layer representing a single note in the passage and the individual nodes in the layer representing a possible fingering for that note. Between nodes in consecutive layers there is a transition penalty which describes the difficulty in moving the hand between two notes on the finger or fretboard. Finding a good fingering is therefore reduced, in Sayegh's framework, to finding a path through the graph that minimizes the total transition penalty accumulated during the traversal.

Sayegh proposes two distinct methods for finding this path. The first, the use of an expert system, relies on musicians to supply rules concerning fingerings to the computer, which can then use these rules to minimize the overall transition penalty. The second is to use Viterbi's algorithm (VA) to mathematically optimize the path. VA is an extremely general technique that, given a set of states, a set of transition probabilities between those states, and an ordered list drawn from the state set, returns the most likely path through the list using the provided transitions (Forney, 1973). It has been successfully applied in many domains, including parsing, speech recognition, and biological sequence alignment (Klein & Manning, 2003; Rabiner, 1989; Zhihui Du et al., 2010).

Each system has its advantages and disadvantages. The optimization approach is rather straightforward to implement and Viterbi's algorithm runs in linear time with the size of the input passage, which is as fast as theoretically possible (it is impossible to decide on a fingering without at least looking at the entire passages from beginning to end). However, the algorithm assumes that the problem of fingering is a Markov process, i.e. that the fingering of the next note in the passage can always be decided by considering only the immediately preceding note. This is mostly a reasonable assumption, but there certain cases where having a greater context window is desirable. For example, passages with sequences, i.e. units of music repeated several times, each time in a different register, often benefit from each instance of the unit being fingered identically. This aids in memorization, ease of execution, and musical expression, as the uniformity of shifts reinforces the similarity between units. However, without lookahead context, the algorithm considers only one interval at a time and the fact that a passage contains a sequence is lost to it. The optimization approach also renders the decision process somewhat opaque to the user of the fingering system, as raw transition weight data is impenetrable to anyone not intimately familiar with the inner workings of the system. The expert system, on the other hand, lends itself to introspection by the user quite naturally, as the expert rules used in making decisions would be easily understood by violinists who possess vast expert knowledge themselves. However, this poses a problem in itself, as true experts are few and far between, and expressing their knowledge in a form usable by a computer is a monumental task, not to mention that experts often disagree with each other. Sayegh also argues that expert systems quickly become overly domain specific, but that is not a problem as long as one is interested in only one particular domain.

Since the 80s, there have been several improvements to Sayegh's original work. Radicioni et. al extend the method by first splitting a given passage into musically separate phrases and then fingering those (Radicioni et al., 2004). As the problem's complexity is exponential in the length of the input passage without the Markov process assumption discussed above, shortening the passage is a marked improvement, as it allows for the assumption to be weakened without disastrous performance implications.

One of the primary difficulties in automated fingering is the myriad considerations aside from simply the notes and rhythms indicated in the written score that can go into a musician's fingering decisions. The previously discussed methods of constraint optimization have no provisions to account for dynamics or articulation. For example, there are often places in music where it is natural to take a breath or make a slight pause, not indicated by a rest. These places are excellent locations to make a shift, as it will be inaudible, but the computer has no knowledge of this feature of the music and cannot use it. “Teaching” an automated system to understand music on a deep enough level to develop human-like intuition for where to place musical breaks just by looking at the score is a Herculean task. Maezawa et al. circumvent this difficulty entirely by supplementing the score with an audio recording to recreate the fingering used in that performance (Maezawa et al., 2012). They augment the Viterbi model of Sayegh with additional constraints derived from analyzing the bow changes, shifts and string crossings present in the recording.

While Maezawa et al. account for individuality among violinists by incorporating audio into their algorithm, Nagata et. al do so by incorporating the expressiveness of certain fingerings, defined as some function of string number and finger number, into their purely symbolic model (Nagata et al., 2014). The authors make the assumption that a violinist’s skill level correlates directly with their ability to play shorter notes more expressively. Under these premises, professional violinists are recommended fingerings in higher positions on lower strings while beginner violinists are recommended to stay in lower positions and use more open strings. This trend is similar to the recommendations of most teachers. Nagata et al. achieve fingerings very similar to textbook examples for the beginner level dataset with their model tuned to output beginner fingerings, but at higher skill levels, the inferred fingerings do not correspond to those commonly used by professionals (see 4.3 for details).

While these results are promising, it is unclear if they scale to longer and more difficult passages. It is also impossible to actually use these systems without acquiring the code from the respective authors. This prohibits usage by primarily non-technical violinists, though distribution of the system is less of a research problem than a software engineering one. The other problem with these existing systems is that they do not allow for fine-grained control of the algorithm by the user. One can provide a specific recording or set the skill level parameter, but those methods of configuration are not precise enough in constraining the algorithm to respect specific guidelines like avoiding oblique finger crossings.

Chapter 3

Methodology

The user interfacing contribution of this thesis, open-strings.com, is a monolithic web application consisting of a backend server connected to a PostgreSQL database, serving HTML, JavaScript, and CSS to a client web browser. It is written using the Yesod[1] web framework in the Haskell programming language. The backend, in addition to managing HTTP requests, runs the fingering inference algorithm.

3.1 Fingering Database

The model for the database uses common design practices. Users, identified by email, can upload and view excerpts of pieces that are registered in the database. Metadata for many common works on IMSLP[2] has been already scraped and entered, but a form is provided to add previously unknown works, composers, and movements to their respective tables. A music Entry consists of the sheet music from an excerpt of a work, with or without fingering annotations, and a textual description. The notation in Figure 3.1 where entity A ("train tracks") points to entity B ("ring with a crow's foot") indicates a relationship in which *exactly one* of entity A is related to *zero or more* of entity B.

[1]https://www.yesodweb.com/
[2]https://imslp.org

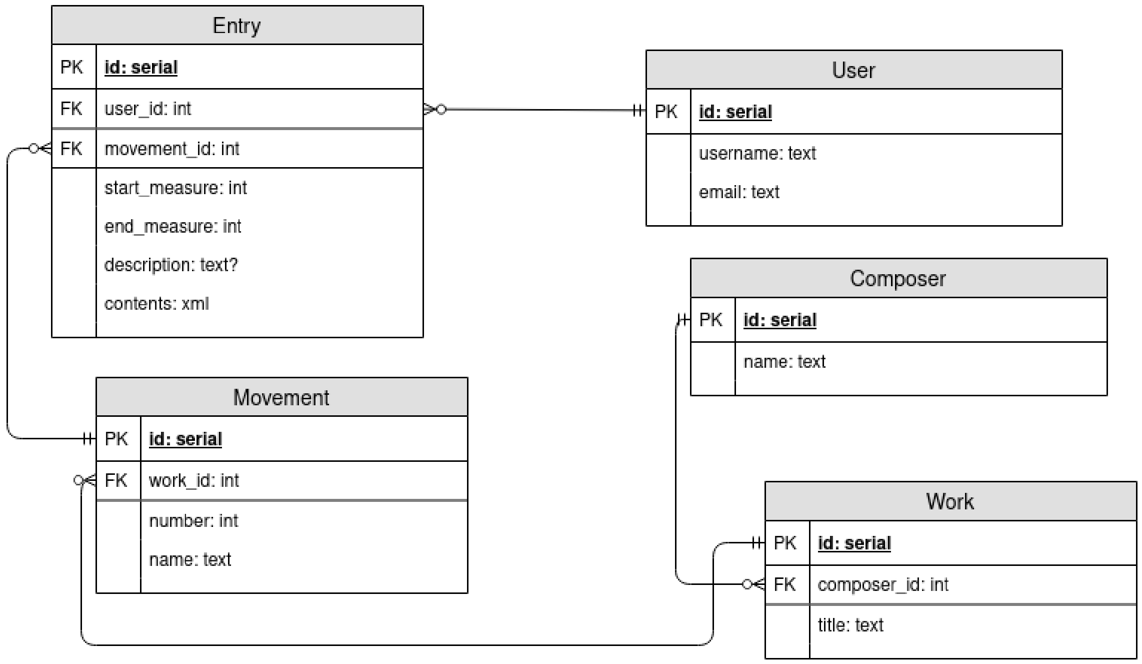


Figure 3.1: Entity Relationship Diagram for OpenStrings

3.2 User interface

At minimum a user of OpenStrings must be able to have the ability to perform the following operations:

3.2.1 Upload a passage and render it as sheet music

We render MusicXML (Good, 2009) documents in the browser using the excellent *OpenSheetMusicDisplay* (OSMD)[3] JavaScript library. MusicXML is by no means the *only* way to represent sheet music on a computer, but the fact that it is commonly exported by most score-writers makes it attractive for our use-case. For a discussion of other approaches to rendering symbolic music, see Appendix A.

3.2.2 Conveniently edit fingerings and persist them to the database

Since OSMD does not at this time support direct modification of a MusicXML document without re-rendering the entire page, this requires working directly on the generated *Scalable Vector*

[3]https://opensheetmusicdisplay.org/

Graphic (SVG). We provide four basic operations.

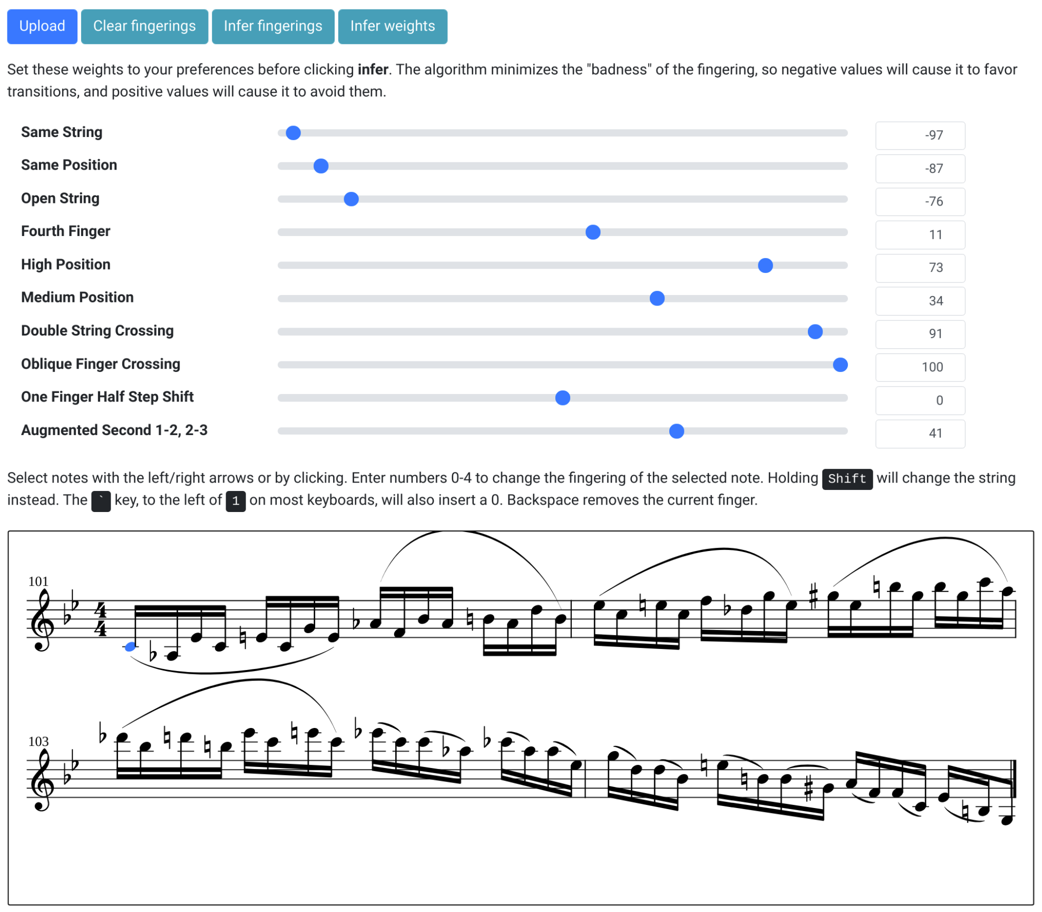
1. Select a note (arrow keys or the mouse)
2. Enter a finger (number keys)
3. Enter a string (shifted number keys)
4. Remove annotations (backspace)

As re-calculating note positions after the addition or removal of a fingering would duplicate OSMD's rendering algorithm, we instead force it to draw the note positions as if every note already had a finger and string annotated simply by adding dummy fingerings and strings to the MusicXML where there are absent and hiding the corresponding nodes in the SVG. Fortunately, the SVG and MusicXML note node orderings are identical except for grace notes, so calculating this correspondence is straightforward.[4] After this transformation, we call the OSMD renderer to produce the SVG and collect all notehead, fingering, and string lyric nodes. Mutable references to these SVG elements are then updated in response to the user navigating between noteheads and changing fingering annotations via the arrow and number keys, respectively. Changes are immediately persisted to the MusicXML document as the user enters them so that they are not lost in the event of a re-render, which can happen if the browser window is resized during the session. Uploading a fingering once it is entered happens when the user presses the "Upload" button.

3.2.3 Query the server for an inferred fingering, respecting any entries made by the user

The user's main interaction with this element is adjusting the penalty weights to suit their preferences. This is accomplished by presenting a group of sliders labeled with a particular weight, such as "shift aversion" or "string crossing aversion". The benefit of sliders over a number entry field is that the weight range can be restricted to an interval that is known to output reasonable fingerings. Clicking "Infer fingerings" sends the chosen weights and MusicXML to the server, which responds with a copy of the MusicXML with every note annotated.

[4]Early versions of OSMD did not support drawing string numbers, so we used to add them in as lyrics, but this has since been fixed upstream. See the corresponding issue and pull request



Set these weights to your preferences before clicking **infer**. The algorithm minimizes the "badness" of the fingering, so negative values will cause it to favor transitions, and positive values will cause it to avoid them.

Same String	-97
Same Position	-87
Open String	-76
Fourth Finger	11
High Position	73
Medium Position	34
Double String Crossing	91
Oblique Finger Crossing	100
One Finger Half Step Shift	0
Augmented Second 1-2, 2-3	41

Select notes with the left/right arrows or by clicking. Enter numbers 0-4 to change the fingering of the selected note. Holding `Shift` will change the string instead. The `` ` `` key, to the left of `1` on most keyboards, will also insert a 0. Backspace removes the current finger.

Figure 3.2: A snapshot of the fingering editor. The blue notehead is currently selected.

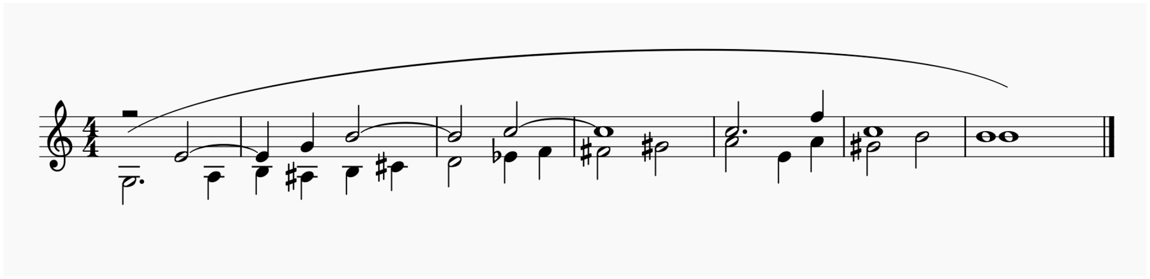
3.3 Fingering inference

Since our sheet music is represented by an XML document, the interesting part of inferring a fingering annotation boils down to a pure function of this form:

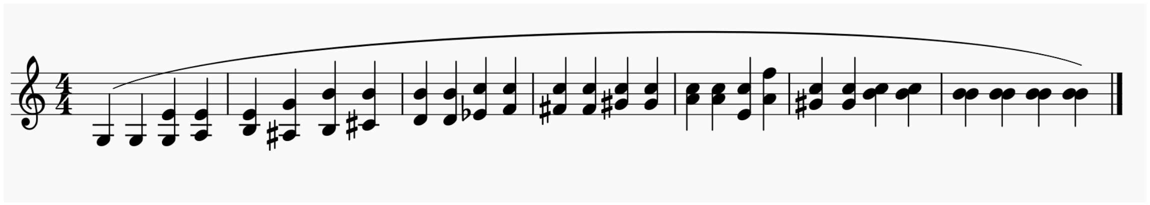
```haskell
-- `Document` is the type of an XML document
inferFingerings :: Document -> Document
inferFingerings = ...
```

We could construct a Haskell datatype to represent a MusicXML document, but with 364 elements, 276 attributes, 179 complex types, and 116 simple types — only a fraction of which are relative to sheet music involving one string instrument — parsing and validating the MusicXML schema[5] would be prohibitively time consuming and error-prone. Instead, we operate directly on the `Document` type, as provided by the `xml-conduit`[6] library.

The first order of business is to reify the possibly multiple voices present in a passage into a single voice containing chords at each "timestep", where a timestep is the shortest duration note appearing in the passage. For example, the following passage from the first movement of Prokofiev's first violin concerto, (rehearsal 19-20), originally written as in figure 3.3a, is transformed into the setting in figure 3.3b (see Appendix B.1 for details).



(a) Passage written in two voices



(b) Passage re-written in one voice

Figure 3.3: Transformation of multiple voices into a single voice

Having thus "gridded" the passage, we have further reduced the problem of fingering inference to a function of type

```haskell
type UnassignedStep = Step Set
type AssignedStep = Step Identity
infer :: [UnassignedStep] -> [AssignedStep]
infer = ...
```

If we take Sayegh's (S. I. Sayegh & Tenorio, 1989) assertion that fingering is a *Markov* process, i.e. that the fingering for a given time step is, given the immediately preceeding time step, indepen-

[5]https://usermanuals.musicxml.com/MusicXML/Content/XS-MusicXML.htm
[6]https://hackage.haskell.org/package/xml-conduit

dent of all previous fingerings, then this is equivalent to choosing a path from t_0 to t_n over the graph of the form in figure 3.4 where each vertex corresponds to a choice of fingering and each vertical group corresponds to a single `TimeStep`.

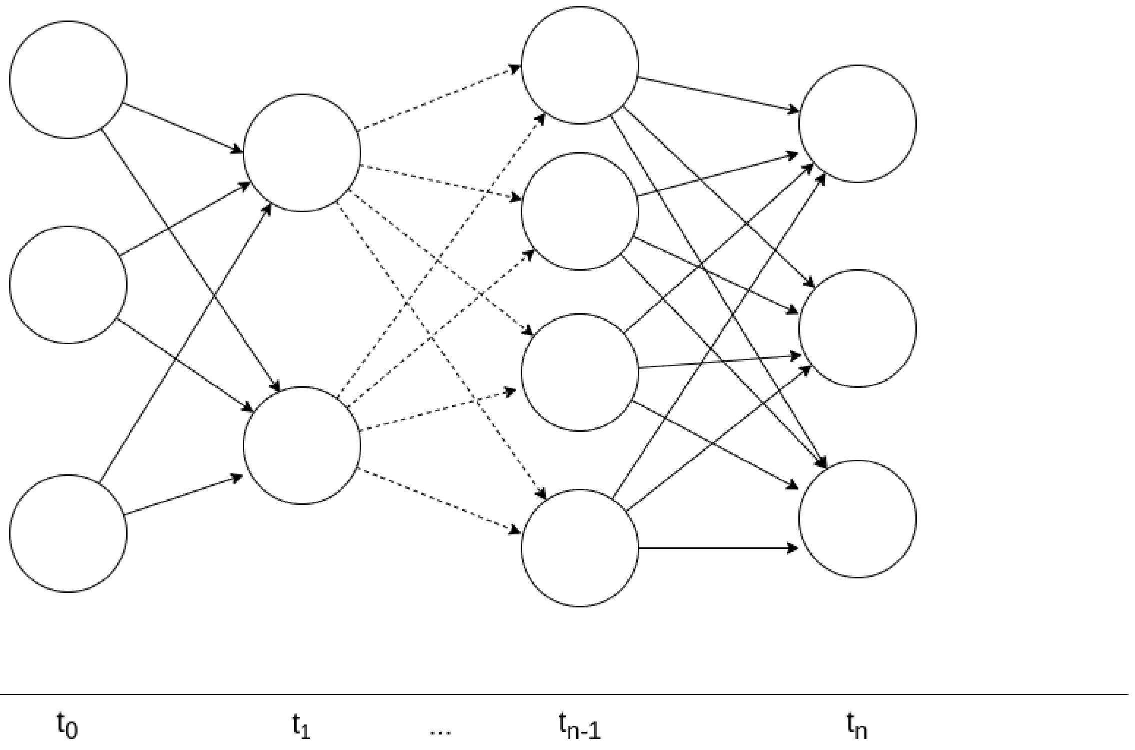


Figure 3.4: A graphical representation of the fingering inference problem

At this point, one would typically "learn" weight assignments for the graph edges by providing Viterbi's algorithm (Forney, 1973) with a corpus of fingered passages, in a similar manner to (S. I. Sayegh & Tenorio, 1989). However, to the best of my knowledge, there is no collection of violin fingering data already in MusicXML form, and in the absence of reliable Optical Music Recognition, transcribing existing editions by hand would be the labor of several decades. Moreover, fingering choices of many of the pedagogues responsible for creating well-known editions differ greatly, so feeding those fingerings to a learning algorithm would produce confusing results. Instead, we construct cost functions manually by codifying agreed-upon rules for shifts and string crossings and allow users to adjust the degree to which each cost function affects the inferred fingering.

Under the Markov assumption, we can only consider two classes of penalty functions: ones that take a either a single `AssignedStep` or two adjacent `AssignedSteps` into account.

```haskell
data Penalty step cost = P
  { _pName :: Text -- assign a name for display purposes
  , _pCost :: step -> cost -- fixed
  , _pWeight :: cost -- adjustable by user
  }

type Penalty1 = Penalty AssignedStep

type Penalty2 = Penalty (AssignedStep, AssignedStep)
```

Objects of type `Penalty1` are not particularly interesting. They generally reflect mechanical impossibilities when executing certain chords. For example, all chords *must* be played on adjacent strings,[7] so the corresponding penalty is

```haskell
chordAdjacent :: Num a => Penalty1 a
chordAdjacent = P "chords on adjacent strings" cost high
  where
    cost step =
      case sort (getStrings step) of
        [] -> 0 -- rest
        [_] -> 0
        [G, D] -> 0
        [D, A] -> 0
        [A, E] -> 0
        [G, D, A] -> 0
        [D, A, E] -> 0
        [G, D, A, E] -> 0
        _ -> infinity
```

We can also penalize the use of the fourth finger, something recommended to many beginners.

```haskell
fourthFinger :: Num a => Penalty1 a
fourthFinger = P "fourth finger" cost 0 -- initialize the weight to 0
  where
    cost (Step (Single n) _) = if (getFinger n == Four) then 1 else 0
    cost _ = 0
```

An example of a `Penalty2` is the check for a half-step shift with one finger. In my own playing,

[7]actually artificial harmonics would be classified as double stops under our gridding scheme, but we ignore them for simplicity here

I employ this technique often, so I assign it a negative weight to bias inference towards using it. However, this is only an initial suggestion, and it is easily adjustable in the UI.

```haskell
oneFingerHalfStep :: Num a => Penalty2 a
oneFingerHalfStep = P "one finger half step shift" cost (- medium)
  where
    cost (Step (Single n1) _, Step (Single n2) _) =
      if _m2 n1 n2
         && getFinger n1 == getFinger n2
         && getString n1 == getString n2
        then 1 else 0
    cost _ = 0
```

Given a list of these penalties (see Appendix B.2 for the full list), we calculate the edge weights by summing the cost of each individual penalty. Then, we retrieve the fingering assignment indicated by the shortest path through the graph.

```haskell
p1s :: [Penalty1]
p1s = [chordAdjacent, staticThird, ...]

p2s :: [Penalty2]
p2s = [oneFingerHalfStep, ...]

cost :: AssignedStep -> AssignedStep -> Double
cost s1 s2 = sum (map p1Cost p1s) + sum (map p2Cost p2s)
  where
    p1Cost p = ((p.pCost) s1 + (p.pCost) s2) * p.pWeight
    p2Cost p = (p.pCost) (s1, s2) * p.pWeight
```

The graph is acyclic and nodes are ordered chronologically by the time steps during which their corresponding notes are sounded. Thus, the shortest path can be calculated in $\mathcal{O}(V + E)$ using the `DAG-SHORTEST-PATHS` algorithm described in Cormen et al., 2003/2009, Chapter 6, Section 24.2, with the added benefit that the natural chronological ordering of nodes means that the graph is already topologically sorted. We codify this assumption at the type level by representing our graph as a list of non-empty states (`[NonEmpty State]`), where each `NonEmpty state` represents the possible states (fingerings) for a single time step. The algorithm in Cormen et al., 2003/2009 relies heavily on mutation to update costs as more of the graph is searched, which we accomplish in Haskell by transforming the list of `NonEmpty state` into a mutable array and mutating it in place locally using the ST monad (Launchbury & Peyton Jones, 1998). It also assumes a "single source," whereas there are potentially many possible fingerings for the first note in the passage. To get around this limitation, we run the single source algorithm for each of the candidate fingerings for the first timestep and pick the one with the lowest cost.

```haskell
import qualified Data.Vector as V
import qualified Data.Vector.Mutable as VM

-- Introduction to Algorithms, Chapter 6 (Cormen, Leiserson, Rivest, Stein)
--------------------------------------------------------------------------------

-- G ~ [NonEmpty (GraphNode state a)]
data GraphNode state a = Node
  { vertex :: state
  , dist :: a
  , staticCost :: a
  , previous :: Maybe (GraphNode state a)
  }
  deriving (Show, Eq, Ord)

getPath :: (Ord a, Num a) =>
  Vector (NonEmpty (GraphNode state a)) -> (a, [state])
getPath vec =
  let end = minimumBy (compare `on` dist) (V.last vec)
   in (dist end, reverse (vertex end : go (previous end)))
  where
    go Nothing = []
    go (Just p) = vertex p : go (previous p)

-- INITIALIZE-SINGLE-SOURCE(G, s):
-- for each vertex v in G.V
--   v.d = ∞
--   v . = NIL
-- s.d = 0
initialize :: (Show state, Num a, Ord a) =>
  [NonEmpty state] -> (state -> a) -> [NonEmpty (GraphNode state a)]
initialize graph singleCost = map (pruneVertices . fmap mkNode) graph
  where
    mkNode f = Node f infinity (singleCost f) Nothing
    pruneVertices (s :| ss) =
      case filter (\node -> staticCost node < infinity) (s : ss) of
        [] -> error "No possible fingering for note"
        (s' : ss') -> s' :| ss'

```

```
-- DAG-Shortest-Paths(G, w, s):
--   INITIALIZE-SINGLE-SOURCE(G)
--   for each vertex u, taken in topologically sorted order
--     for each vertex v in G.adj(u)
--       RELAX(u,v,w)
shortestPath :: forall state a. (Ord state, Num a, Ord a, Show state) =>
  [NonEmpty state] -> (state -> a) -> (state -> state -> a) -> (a, [state])
shortestPath [] _ _ = (0, [])
shortestPath (fs : rest) singleCost transitionCost =
  minimumBy (compare `on` fst) (fmap mkGraph fs)
  where
    mkGraph f =
      let graph = V.fromList $ (Node f 0 (singleCost f) Nothing :| []) :
                               initialize rest singleCost
       in getPath $ V.modify go graph

    -- RELAX(u, v, w):
    -- if v.d > u.d + w(u,v)
    --   v.d = u.d + w(u,v)
    --   v . = u
    relax u v =
      -- add the static cost of the next node when calculating the new distance
      let dist' = dist u + transitionCost (vertex u) (vertex v) + staticCost v
       in if dist' < (dist v) then v {dist = dist', previous = Just u} else v

    go :: forall s. VM.MVector s (NonEmpty (GraphNode state a)) -> ST s ()
    go g = forM_ [1 .. VM.length g - 1] \i -> do
      prev <- VM.read g (i - 1)
      forM_ prev \u -> VM.modify g (fmap (relax u)) i
```

Given a corpus of fingered passages, we can reverse this process to infer weights by running stochastic gradient descent to minimize the objective

$$\text{PathCost}(\text{passage}) + \lambda \sum_{w \in \text{weights}} w^2$$

where the PathCost is the sum of costs between all pairs of adjacent steps. The second term, $\lambda \sum w^2$, biases the weights towards 0 to be more "neutral." This corresponds to the notion that well-rounded violinists tend not to prefer a single type of shift to the exclusion of all others and makes the problem more numerically tractable. Currently, there is not enough data to reliably set the hyper-parameter, λ, so we just leave it at 1.

3.4 Evaluation

The most valuable metric in assessing the quality of generated fingerings is the opinions of the violinists we expect to use them in performance. To this end, we present a group of survey respondents with several fingered passages and ask them to answer the following questions on a 5-point Likert scale (Preedy & Watson, 2010). Results of the survey are found in Section 4.2.

- How comfortable do you find this fingering to play?
- How expressive do you find this fingering allows a violinist to be?
- How idiomatic do you find this fingering?
- Would you say this fingering was generated by a human or by a computer?

Of the presented passages, half have fingerings generated by a human (me), and half have fingerings generated by the method presented above.

Chapter 4

Results

4.1 Fingering samples

4.1.1 Generated fingerings

The first bar and a half of the Prokofiev (Figure 4.1) is extremely promising, as it is identical to what I use in practice and has been recommended to me by teachers. However, the last two bars are somewhat awkward; measure 103 uses repeated 4-2 instead of 3-1 and measure 104 shifts back and forth between first and third positions whereas a more comfortable solution would be to just stay in third position at the expense of an extra string crossing. Several free responses in the user survey confirm this analysis. The Bartók (Figure 4.2) is certainly playable, but the 4-2 shifts are undesirable and the passage would be made easier by using the open E-string for the written F-flat. Unfortunately, the Ysaÿe (Figure 4.3), aptly described by one of the respondents, "is a mess." Ironically, given the name of this project, the primary failing seems to be that the algorithm makes the incorrect assumption that using *any* finger after an open string is equally acceptable. While it is true that using open strings to mask a *necessary* shift can be very convenient at times, the algorithm has abused this fact to such an extreme as to render the generated fingering for the Ysaÿe unusable.

4.1.2 Control group fingerings

We include familiar excerpts by Richard Strauss (Figure 4.4), Bruch (Figure 4.5), and Mozart (Figure 4.6) as a control group representing human-generated fingerings. Of course, these examples are idiosyncratic and represent my opinions and preferences, but they have the desirable property of

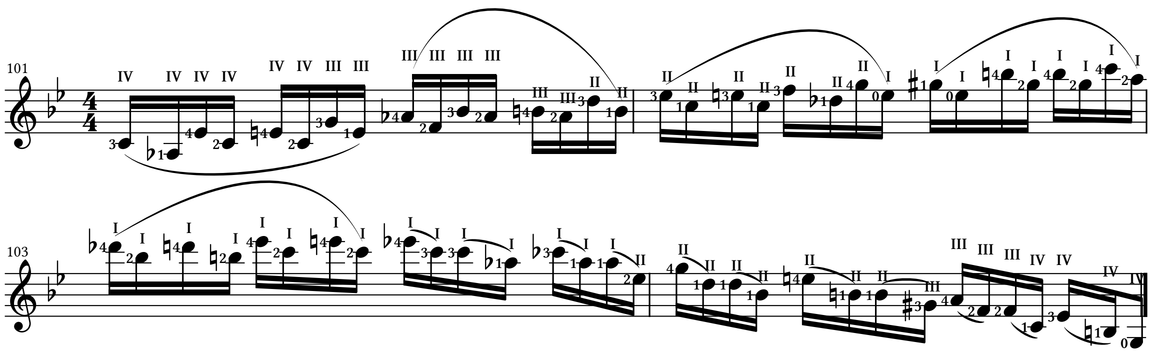


Figure 4.1: Prokofiev: Violin Concerto No.2, Op.63, I. Allegro moderato mm. 101-104

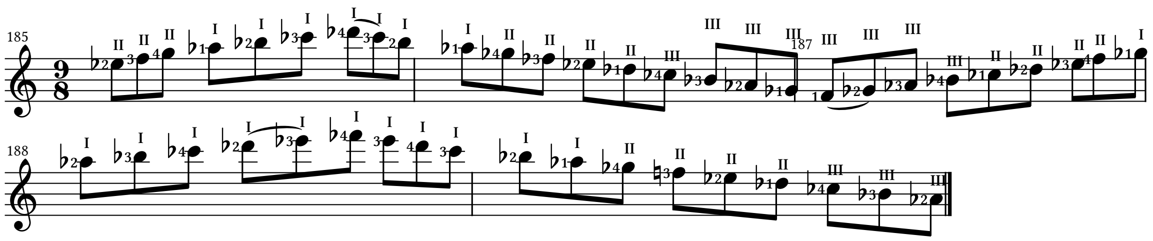


Figure 4.2: Bartók: Violin Concerto No. 2, III. Allegro molto mm. 185-189

having been used by at least one violinist in an actual performance.

4.2 User responses

Responses to the questions in 3.4 are tabulated in Figures 4.7, 4.8, 4.9, and 4.10. Each column represents a passage and each row, a gradation on the Likert scale. The number of dots indicates the number of responses in that category.

The responses mostly agree with the analysis above. In every metric, the Bartók (Figure 4.2) had the most positive responses, followed closely by the Prokofiev (Figure 4.1), although if we excluded the last two measures of the Prokofiev, the positions would probably have been reversed. Additionally, the Bartók succeeded in passing for human-generated, with all violinists answering “Unsure” or “Probably a human” when asked who made the fingering. Though the excerpt involves mostly step-wise motion, this is still an important milestone in demonstrating the potential effectiveness of the algorithm.

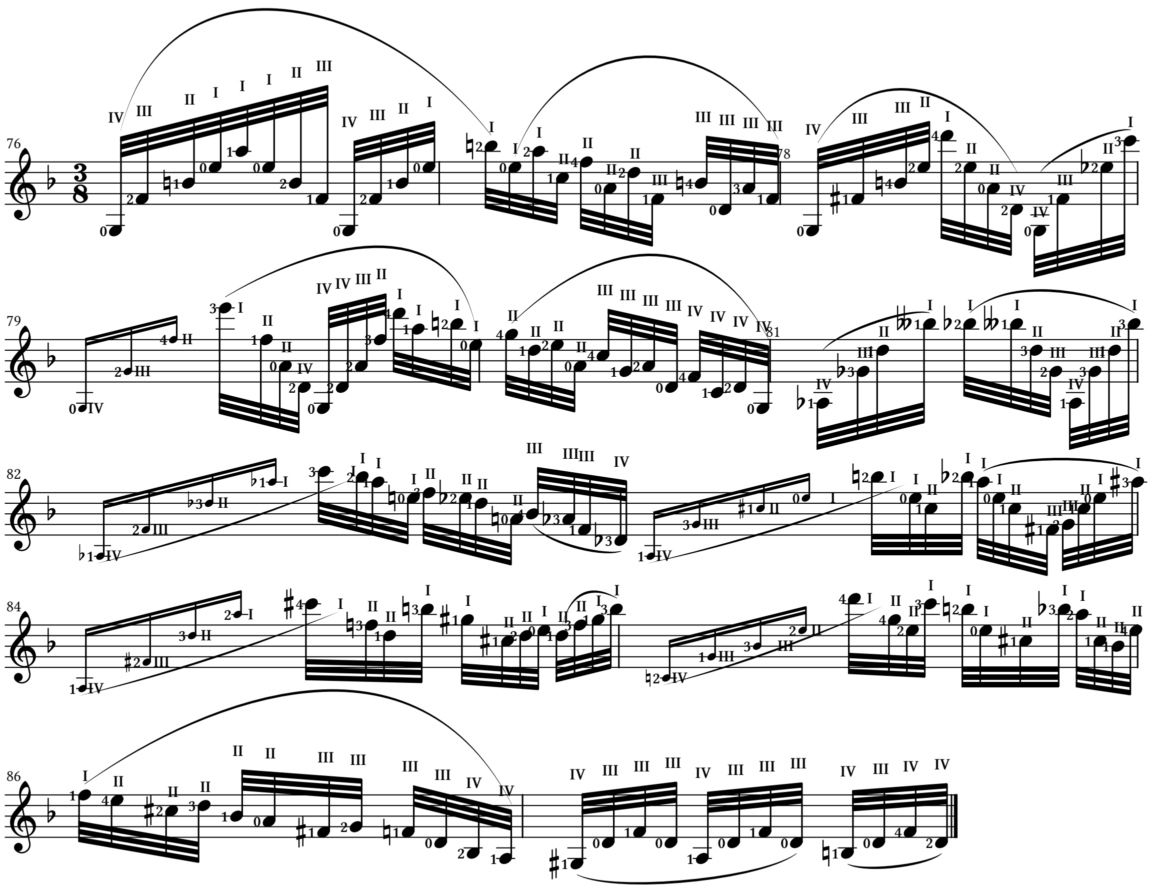


Figure 4.3: Ysaÿe: Solo Sonata No. 3, Op. 27 "Ballade" mm. 76-87

4.3 Comparison with prior work

Nagata et al., 2014 demonstrate their inference algorithm for a simple passage, producing both a beginner (4.11a) and an advanced (4.11b) fingering. Without altering the default weight setting, our algorithm produces the result in 4.11c. Starting measure 5 with the second finger is a poor choice, brought on by the same defect of no position memory across rests or open strings that plagued the Ysaÿe excerpt, but for the other notes, the OpenStrings fingering out-performs the advanced one inferred by Nagata et al., 2014. Shifting to third position on the downbeat of measure 6 is more natural than shifting in the middle of the bar and, using the open string to shift to first position in measure 7 is much cleaner than the awkward 4-3 downward slide. Adjusting the weights to favor beginner tendencies produces the result in Figure 4.11d. For a novice, our

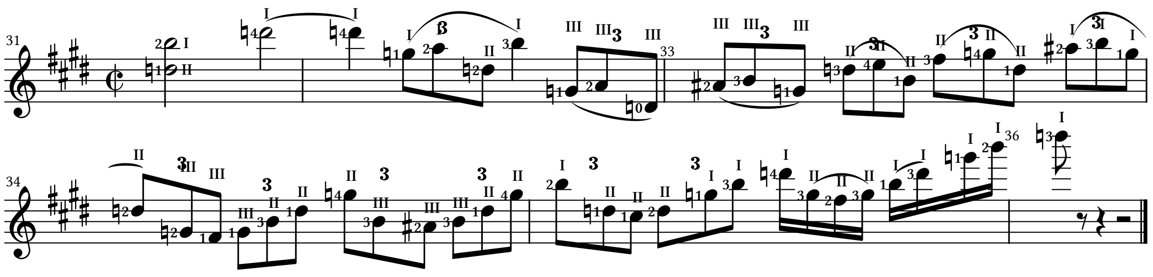


Figure 4.4: Strauss: Don Juan, Op. 20 mm. 31-36

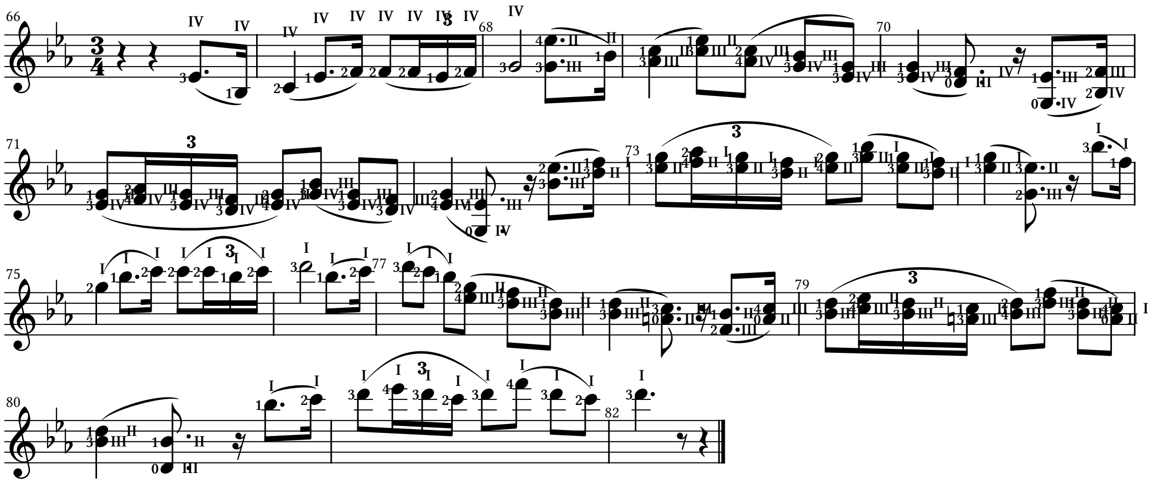


Figure 4.5: Bruch: Scottish Fantasy, Op. 46, I. Introduction mm. 66-82

fingering is not as good as 4.11a, which fastidiously avoids shifting of any kind and makes only minimal use of the left pinkie. In principle, we could add weights specifically penalizing second position and biasing inference towards something like Nagata et al., 2014's beginner result, but that has yet to be implemented.

Maezawa et al., 2012, estimating fingerings for *Romanze in C*, with the aid of a Joachim recording, obtained the fingering in Figure 4.12a. For the same passage, we obtain the result in Figure 4.12b. The authors speculate "that the actual fingering [in the Joachim recording] may have been very different from that estimated by [their] method." This is true. Joachim would not have employed Maezawa et al., 2012's fingering in practice. The shifts are awkward and it is difficult to play in tune and with good tone on the highest part of the D string for an extended period. While the OpenStrings algorithm's fingering also has questionable shifts, it always succeeds in picking the range of the instrument that most human violinists would use.

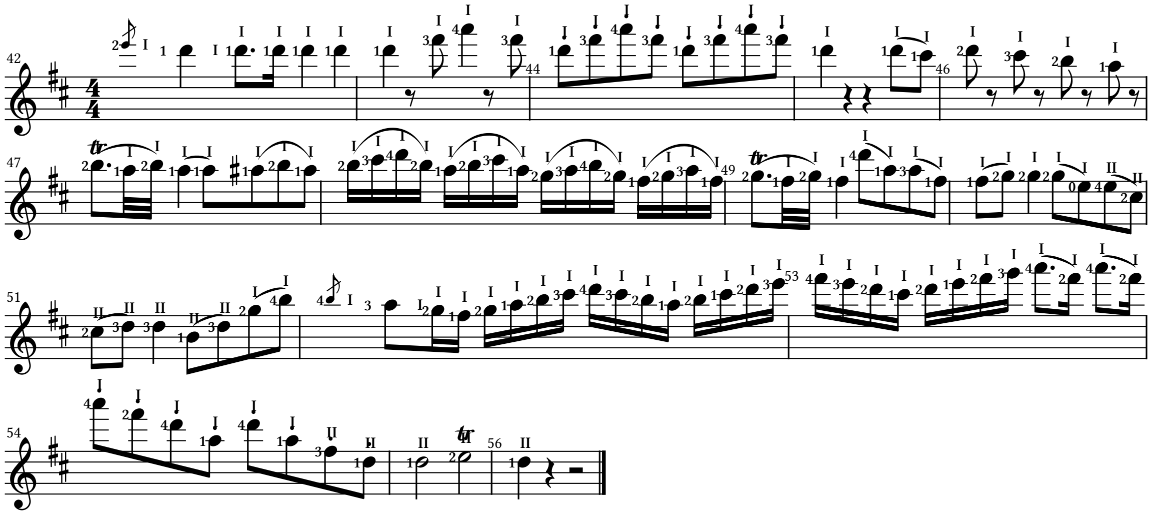


Figure 4.6: Mozart: Violin Concerto No. 4, K. 218, I. Allegro mm. 42-56

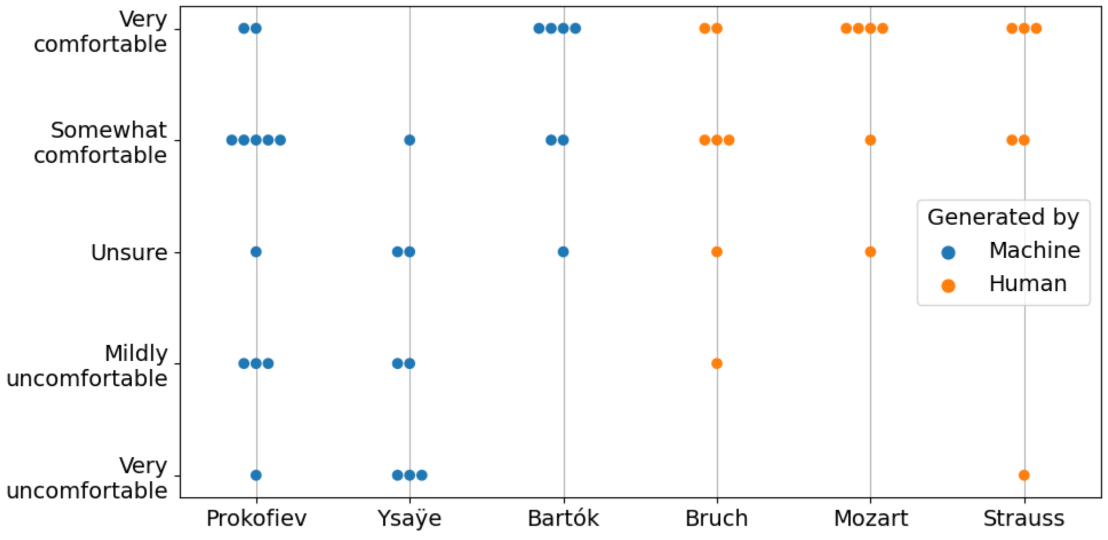


Figure 4.7: How comfortable do you find this fingering to play?

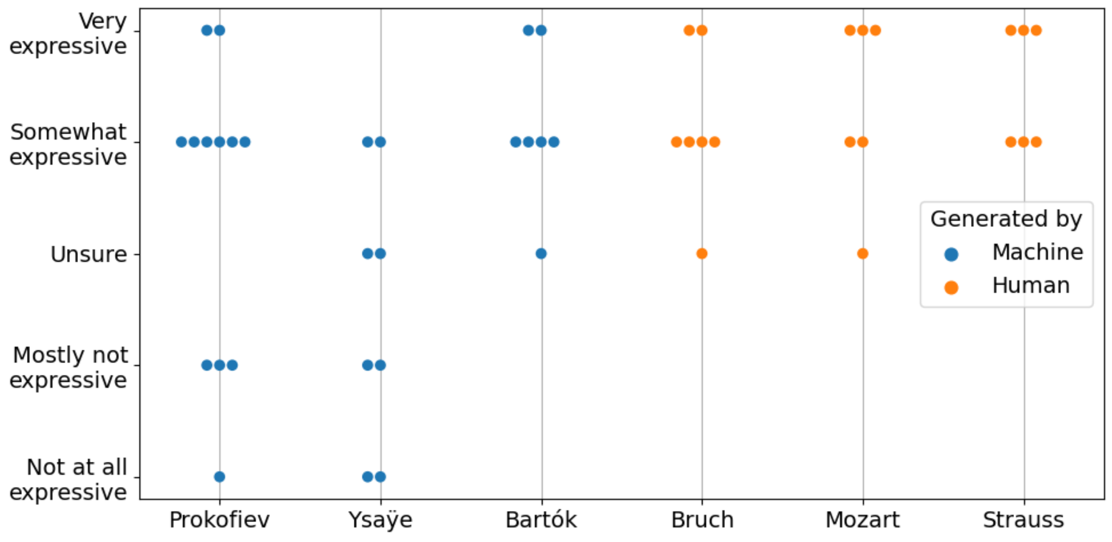


Figure 4.8: How expressive do you find this fingering allows a violinist to be?

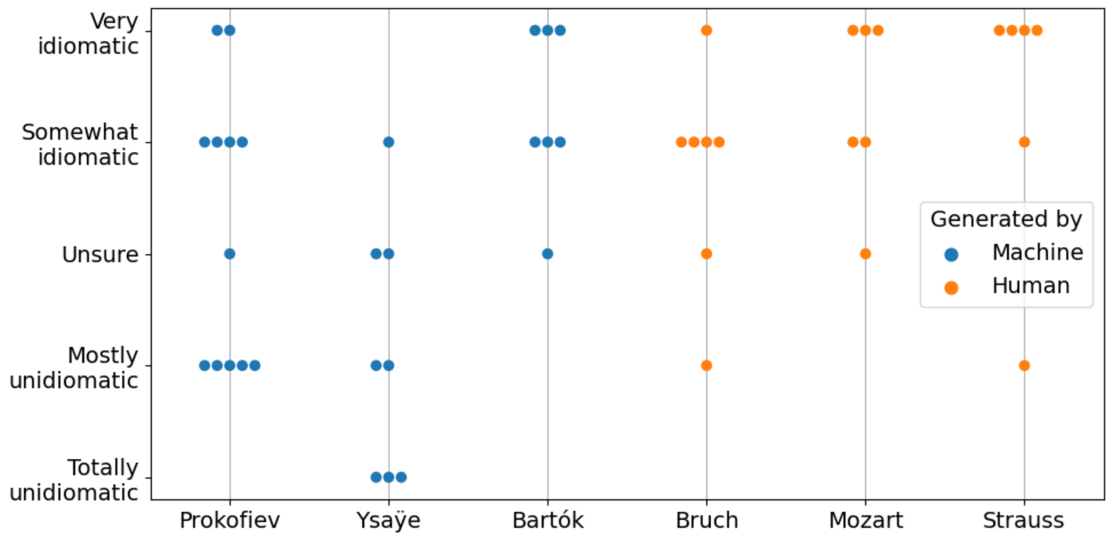


Figure 4.9: How idiomatic do you find this fingering?

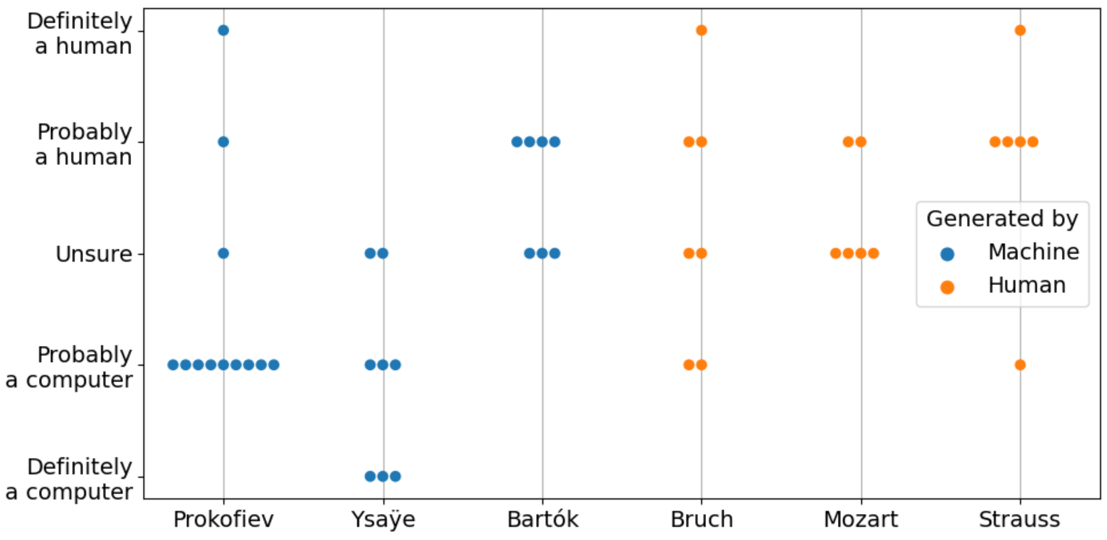
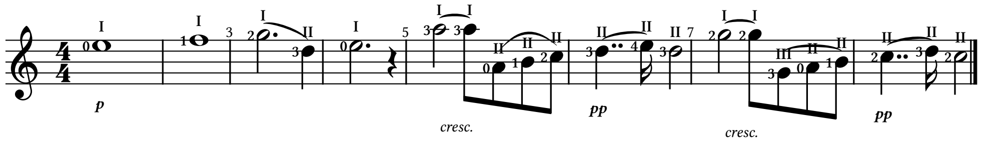
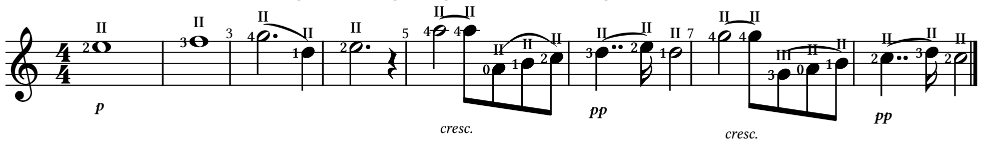


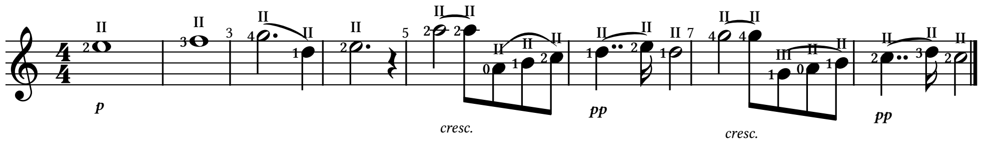
Figure 4.10: Would you say this fingering was generated by a human or by a computer?



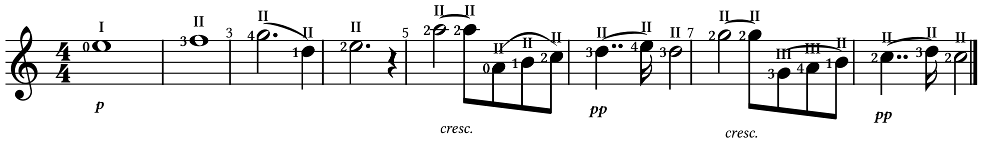
(a) Beginner fingering inferred by Nagata et al., 2014



(b) Advanced fingering inferred by Nagata et al., 2014

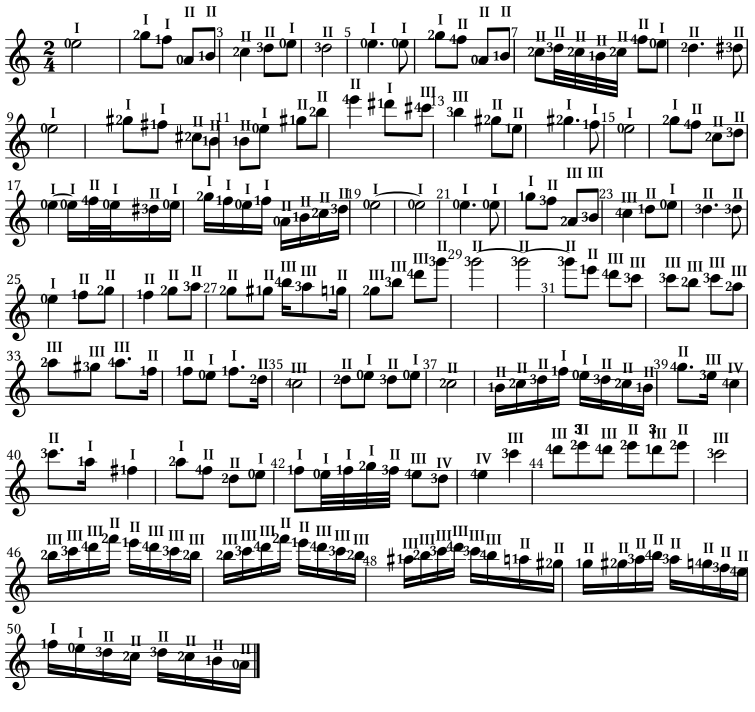


(c) Default fingering inferred by OpenStrings



(d) Beginner fingering inferred by OpenStrings

Figure 4.11: Comparison of OpenStrings and prior work by Nagata et al.


(a) Fingering estimated by Maezawa et al., 2012

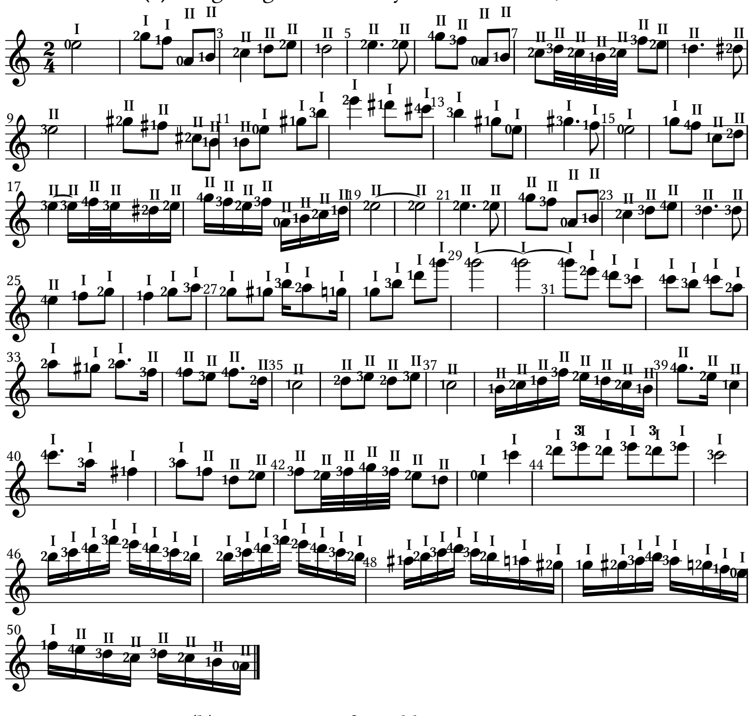

(b) Fingering inferred by OpenStrings

Figure 4.12: Comparison of OpenStrings and prior work by Maezawa et al.

Chapter 5

Conclusion

5.1 Discussion

OpenStrings narrows the gap between previous attempts at automated fingering selection and what humans are capable of producing, but by no means closes it. For simple enough passages, experienced violinists are comfortable with the fingerings assigned by the algorithm, but as musical complexity increases, results degrade. Could more complicated passages be handled better with the addition of more fine-grained rules under the current system (with some improvements discussed below in 5.2.1), or is there no amount of manually written rules that can approach the human intuition involved in choosing fingerings? Perhaps, given enough data, a deep neural network could work out the rules better than I could translate my understanding of them into code. Indeed, in fields with enough samples to train on, deep learning has obliterated classical AI expert systems like this one. However, though we could imagine a neural network eventually generating better fingerings than OpenStrings, its inner workings, like those of most deep learning models, would be a black box. The advantage of manual feature extraction is in its easy inspection and comprehension by violinists, who, though they may not be programmers, are capable of understanding the process of setting costs on semantically meaningful transitions.

5.2 Future work

5.2.1 Longer term memory

The biggest flaw in the inference procedure is that it "forgets" hand position between notes. Instead of weakening the Markov assumption that each fingering depends only on the one preceding it, we should generalize the notion of fingering to include the whole hand, as opposed to the fingers pressing on the string. This will allow the algorithm to "remember" what position it was in while playing an open string so that it doesn't shift afterwards without good reason. Additionally, this could permit some version of conservation of momentum to be introduced, penalizing rapid shifts back and forth more heavily than a series of shifts in the same direction.

5.2.2 User interface improvements

Instead of inferring the single best fingering for a particular weight setting, we can infer the k best fingerings, as it is still a computationally tractable problem at $\mathcal{O}(|E| + k|V|)$ (Eppstein, 1997). This would be a marked improvement to the user interface, as it would allow users to see several fingerings at a time, instead of waiting for the algorithm to finish running after a weight adjustment. On a related note, the runtime performance of the inference, although algorithmically optimal, leaves much to be desired, taking up to 30 seconds to decide fingerings for a passage of 10 bars. Possible solutions include

- implementing sub-selection in the UI, i.e. constraining inference to run on a contiguous subset of the passage without having to manually upload a new XML file
- more aggressively pruning the fingering graph to remove poor transitions
- caching often used information about XML nodes like pitch to reduce the runtime of the inner loop (the need for this has not yet been verified by profiling)
- relaxing the requirement of finding the optimal path through the fingering graph in favor of finding a heuristically "good" one faster

The current flow for uploading a new piece of music requires the use of an external score writer for initial typesetting. This has the advantage of delegating the difficult task of engraving music to a specialized program, but can be rather inconvenient when uploading many short passages. It may be expedient to implement a basic sheet music editor inside of the user interface that outputs MusicXML so that the entire flow could be accomplished inside one program.

5.2.3 Fingering data collection

Another area for improvement is the lack of fingering data on which to run standard machine learning procedures. Barring large advances in Optical Music Recognition, the problem becomes a social one, in that the only way forward is for people to spend time typing out passages and uploading them to OpenStrings. To incentivize users to commit their efforts to this task, we could construct a bounty system for uploads. User A would pledge n currency to whoever typesets measures 200-300 of, say, the Tchaikovsky Violin Concerto, and then user B, upon uploading the XML, would receive the n currency towards placing their own future bounties. Users would be able to purchase this currency with real dollars, which would go towards maintaining the servers and administering the website.

5.2.4 Extensibility

With the current design of OpenStrings, if a user wants to add a rule to the inference engine, they can open a pull request and wait for it to be merged and deployed. This process is accessible to everyone, but the feedback time between writing a rule and seeing it in action on a passage is untenably long for anyone not running a copy of the website locally. Instead, we could bake only a few core rules into the engine at compile time and read the rest at runtime, thereby giving all users the same level of control over the inference process as the author. There are very few pieces of end user software that allow for programmatic extensibility at runtime, a notable exception being emacs (Yegge, 2007). Allowing for users to upload code to run on one's server is especially dangerous. However, for our purposes, we need only allow for users to upload Haskell functions of types `Penalty1` or `Penalty2`. Haskell's purity makes it very easy to audit, and there is already literature on running untrusted Haskell code in an application (see Pang et al., 2004), so the task of adding extensibility seems feasible.

Appendix A

Computer Representations of Symbolic Music

When designing an automated system to aid working musicians in preparing fingerings for practice and performance, it is very important to consider the underlying representation of the musical data. Sheet music has evolved into an incredibly information dense format over the past 700 years and computers still struggle to properly represent its complexity and nuance.

Perhaps the most common representation format for computer music is MIDI, a protocol designed for the transition of musical information between synthesizers (of Musical Electronics Industry AMEI & MMA, 2020). It encodes information about pitch, note onset and offset times, and note velocities. The format is extremely widespread, but it is more optimized for synthesized performance than for symbolic analysis, although there is a piano fingering dataset that stores fingerings with midi-like note onset and velocity information, along with accompanying human-readable PDFs for each fingered piece (Nakamura et al., 2019).

Aside from MIDI, there are several other contending formats for symbolic music storage. The three most prevalent are MusicXML (Good, 2009) , Music Encoding Initiative (MEI) (Crawford & Lewis, 2016) , and Lilypond (Lilypond, 2019). Other, less established formats include Score (Sapp, 2015), Humdrum (Huron, 2002), Guido (Daudin et al., 2009), and MuseData. We omit discussion of these formats because they are relatively unpopular and have been subsumed by the other, more modern representations. For a more in depth analysis of the vast landscape of computer music formats, see Selfridge-Field, 1997.

Most musicians are probably most familiar with MusicXML out of all non-MIDI formats. MusicXML is the *lingua franca* of scorewriting software, serving as the interchange format between programs such as Finale and Sibelius. It embeds music into a hierarchical XML document containing both raw musical information, such as pitch and rhythm, as well as score layout information so that the rendered output looks the same between different pieces of software. One of its chief advantages is its ubiquity and that it can be rendered both locally on a user's desktop if they own a scorewriting application or in the browser with OpenSheetMusicDisplay, as we do for OpenStrings.

MEI is very similar to MusicXML, except that its intended use is as a musicology tool rather than an interchange format. Unlike MusicXML, it is a purely semantic format with the ability to represent editorial information and discrepancies between various editions in a single document within the XML schema. The format embeds no rendering information but can be viewed online using Verovio.[1] The fact that it contains only musical information rather than conflating music with rendering concerns is appealing, but because it is not as widespread as MusicXML, there are fewer options for interoperability between programs.

Lilypond is very different from MEI and MusicXML, as it is the only format intended to be written and read by humans. It's main focus is on producing the best possible publication quality score output, which it does very well. Writing music in Lilypond is very similar to writing mathematics in LaTeX. This is both an advantage and a disadvantage of the format, as with practice, one can become very proficient at transcribing music quickly into Lilypond, but the novelty factor of writing plain text makes it difficult to learn initially, especially for most musicians used to entering music into scorewriters with the mouse. There is some work converting a limited subset of Lilypond to MEI so that it can be rendered in the browser using Verovio as well (Liska et al., 2015). Although music can be converted between all three of these formats as well as Humdrum and Score, the conversion is never lossless (Nápoles López et al., 2019), so it is advisable to pick a single format to store fingerings in order to maintain the semantic integrity of the data.

[1]https://www.verovio.org/index.xhtml

Appendix B

Selected code samples

All of the code for the latest version of OpenStrings can be found online at https://github.com/jmorag/open-strings.

B.1 Merge multiple voices into time steps

In order to merge multiple voices into a single line as in the transformation from figure 3.3a to figure 3.3b, we group MusicXML `note` elements into single notes, double/triple/quadruple-stops, or rests.

```haskell
-- the type variable `f` parametrizes if a note has many fingerings
-- (undecided) or just one (decided)
data Note f = Note {_xmlRef :: XmlRef, _fingerings :: f Fingering}
data TimeStep f
  = Single (Note f)
  | DoubleStop (Note f) (Note f)
  | TripleStop (Note f) (Note f) (Note f)
  | QuadrupleStop (Note f) (Note f) (Note f) (Note f)
  | Rest
  deriving (Show, Eq)
```

Given a list of relevant xml elements, we calculate the total duration of the passage by summing the `duration` attributes, create an empty `Vector` of rests, and then traverse the element list while keeping track of our position in the vector. The mechanism by which we perform vector mutation efficiently in a pure language using the ST monad is described in (Launchbury & Peyton Jones, 1998).

```haskell
readTimeSteps :: [Element] -> Vector (TimeStep Set)
readTimeSteps es = V.create do
  vec <- VM.replicate (totalDuration es) Rest
  foldM_ (readTimeStep vec) 0 (zip [0 ..] es)
  pure vec

readTimeStep :: VM.MVector s (TimeStep Set) -> Int -> Element -> ST s Int
readTimeStep vec t ref =
  case e ^?! name of
    "note" -> do
      let t' = maybe t (const (t - xmlDuration)) (e ^? deep (el "chord"))
      forM_ [t' .. t' + xmlDuration - 1] $
        VM.modify vec (maybe Rest Single (mkNote ref) <>)
      pure (t' + xmlDuration)
    "backup" -> pure (t - xmlDuration)
    "forward" -> pure (t + xmlDuration)
    n -> error $ "Impossible timestep element " <> show n
  where
    e = deref ref
    xmlDuration = e ^?! dur
```

The `mkNote :: Element -> Maybe (Note Set)` tags all `note` elements with every possible fingering that they could be executed with. It is elided here for brevity. Conveniently, `TimeSteps` form a monoid, with identity `Rest` and binary operation `(<>)`, so we can combine whatever element was previously in the vector at the current position with the next one in the list.

```haskell
instance Semigroup (TimeStep f) where
  Single n1 <> Single n2 = DoubleStop n1 n2
  Single n1 <> DoubleStop n2 n3 = TripleStop n1 n2 n3
  Single n1 <> TripleStop n2 n3 n4 = QuadrupleStop n1 n2 n3 n4
  DoubleStop n1 n2 <> Single n3 = TripleStop n1 n2 n3
  TripleStop n1 n2 n3 <> Single n4 = QuadrupleStop n1 n2 n3 n4
  DoubleStop n1 n2 <> DoubleStop n3 n4 = QuadrupleStop n1 n2 n3 n4
  n <> Rest = n
  Rest <> n = n
  n1 <> n2 = error $
    "Unsatisfiably large constraint - too many notes to cover at one time: "
    <> show n1 <> " | " <> show n2
```

Since there might be adjacent identical `TimeSteps` in the vector, we collapse it into a list of `Step Sets`, where a `Step` is a `TimeStep` with a duration.

```
data Step f = Step {_timestep :: TimeStep f, _duration :: Int}
  deriving (Show, Eq)

coalesceTimeSteps :: Vector (TimeStep f) -> [Step f]
coalesceTimeSteps = fmap (\ts -> Step (NE.head ts) (length ts)) . NE.group
```

B.2 Fingering penalties

A penalty is a record containing a name, cost function, and numerical weight.

```
data Penalty step a = P
  { _pName :: Text
  , _pCost :: step -> a
  , _pWeight :: a
  }

type Penalty1 = Penalty AssignedStep

type Penalty2 = Penalty (AssignedStep, AssignedStep)

type Weights a = Map Text a
```

For convenience, we define predicates on diatonic intervals and some starting values for costs and weights.

```
halfSteps :: Note f -> Note f -> Int
halfSteps p1 p2 = abs (pitch p2 - pitch p1)
_p1 p1 p2 = halfSteps p1 p2 == 0
_p4 p1 p2 = halfSteps p1 p2 == 5
_p5 p1 p2 = halfSteps p1 p2 == 7
_p8 p1 p2 = halfSteps p1 p2 == 12
_m2 p1 p2 = halfSteps p1 p2 == 1
_M2 p1 p2 = halfSteps p1 p2 == 2
_m3 p1 p2 = halfSteps p1 p2 == 3
_M3 p1 p2 = halfSteps p1 p2 == 4
_a4 p1 p2 = halfSteps p1 p2 == 6
_m6 p1 p2 = halfSteps p1 p2 == 8
_M6 p1 p2 = halfSteps p1 p2 == 9
_m7 p1 p2 = halfSteps p1 p2 == 10
_M7 p1 p2 = halfSteps p1 p2 == 11
```

```
-- minor 9th or higher
_m9 p1 p2 = halfSteps p1 p2 > 12
second p1 p2 = _m2 p1 p2 || _M2 p1 p2
third p1 p2 = _m3 p1 p2 || _M3 p1 p2
sixth p1 p2 = _m6 p1 p2 || _M6 p1 p2
seventh p1 p2 = _m7 p1 p2 || _M7 p1 p2

infinity, high, medium, low :: Num a => a
infinity = 1000000000000000
high = 100
medium = 50
low = 1
```

The penalties on single steps consist of the following.

```
p1s :: Num a => [Penalty1 a]
p1s =
  singles
    <> doubleStops
    <> [ trill
       , chordAdjacent
       , staticTripleStop
       , staticQuadrupleStop
       ]

singles, doubleStops :: Num a => [Penalty1 a]
singles = [highPosition, mediumPosition, fourthFinger, openString]
doubleStops =
  [ staticUnison
  , staticSecond
  , staticThird
  , staticThirdAdjacentFingers
  , staticMajorThird4_3
  , staticFourth
  , staticFifth
  , staticSixth
  , staticMajorSixth
  , staticSeventh
  , staticMinorSeventh
  , staticMajorSeventh
  , staticOctave
  , staticTenth
  ]

binarize :: Num a => Bool -> a
binarize b = if b then 1 else 0

trill :: Num a => Penalty1 a
trill = P "trill" cost high
  where
    cost step = case step ^. timestep of
      Rest -> 0
      Single n ->
        case n
          ^? xmlRef . to deref
            . deep (failing (ell "trill-mark") (ell "wavy-line")) of
          Just _ -> case n ^. fgr of
            Open -> high
            One -> 0
            Two -> 0
            Three -> if step ^. duration >= 4 then infinity else high
            Four -> infinity
          Nothing -> 0
      DoubleStop n1 n2 ->
        cost (set timestep (Single n1) step) +
        cost (set timestep (Single n2) step)
      -- there should never be trills on triple/quadruple stops...
      _ -> 0

highPosition :: Num a => Penalty1 a
highPosition = P "high position" cost medium
  where
    cost step = binarize $
                  anyOf (notes . fingering . to position . traversed)
                  (>= EighthAndUp) step

mediumPosition :: Num a => Penalty1 a
mediumPosition = P "medium position" cost low
  where
    cost step = binarize $
                  anyOf (notes . fingering . to position . traversed)
                  (>= Fourth) step

```

```haskell
fourthFinger :: Num a => Penalty1 a
fourthFinger = P "fourth finger" cost 0
  where
    cost step = binarize $ step ^? timestep . _Single . fgr == Just Four

openString :: Num a => Penalty1 a
openString = P "open string" cost 0
  where
    cost step = binarize $ anyOf (notes . fgr) (== Open) step


--------------------------------------------------------------------------
-- Double Stops
--------------------------------------------------------------------------
chordAdjacent :: Num a => Penalty1 a
chordAdjacent = P "chords on adjacent strings" cost high
  where
    cost step =
      case sort $ step ^.. notes . str of
        [] -> 0 -- rest
        [_] -> 0
        [s1, s2] | dist s2 s1 == 1 -> 0
        [G, D, A] -> 0
        [D, A, E] -> 0
        [G, D, A, E] -> 0
        _ -> infinity

staticMajorThird4_3 :: Num a => Penalty1 a
staticMajorThird4_3 = P "static major third 4-3" cost medium
  where
    cost (Step (DoubleStop n1 n2) _)
      | _M3 n1 n2 =
        let higherThan pos =
              anyOf
                (traversed . fingering . to position . traversed)
                (>= pos)
                [n1, n2]
         in case (n1 ^. fgr, n2 ^. fgr) of
              (Four, Three) -> if higherThan Fourth then 1 else high
              _ -> 0
    cost _ = 0

```

```
staticThirdAdjacentFingers :: Num a => Penalty1 a
staticThirdAdjacentFingers = P "static third nonstandard" cost high
  where
    cost (Step (DoubleStop n1 n2) _) | third n1 n2 =
      case (n1 ^. fgr, n2 ^. fgr) of
        (Three, Two) -> high
        (Two, One) -> high
        _ -> 0
    cost _ = 0

staticThird :: Num a => Penalty1 a
staticThird = P "static third" cost high
  where
    cost (Step (DoubleStop n1 n2) _) | third n1 n2 =
      case (n1 ^. fgr, n2 ^. fgr) of
        (Three, One) -> 0
        (Four, Two) -> 0
        (_, Open) -> 0
        (Open, _) -> 0
        (f1, f2) | f1 <= f2 -> infinity
        _ -> 0
    cost _ = 0

staticSixth :: Num a => Penalty1 a
staticSixth = P "static sixth" cost high
  where
    cost (Step (DoubleStop n1 n2) _) | sixth n1 n2 =
      case (n1 ^. fgr, n2 ^. fgr) of
        (One, Two) -> 0
        (Two, Three) -> 0
        (Three, Four) -> 0
        (_, Open) -> low
        (Open, _) -> low
        (f1, f2) | f1 >= f2 -> infinity
        _ -> 0
    cost _ = 0

staticMajorSixth :: Num a => Penalty1 a
staticMajorSixth = P "static major sixth" cost high
  where
    cost (Step (DoubleStop n1 n2) _) | _M6 n1 n2 =
      case (n1 ^. fgr, n2 ^. fgr) of
        (One, Three) -> low
        (Two, Four) -> low
        (One, Four) -> high
        _ -> 0
    cost _ = 0

staticOctave :: Num a => Penalty1 a
staticOctave = P "static octave" cost high
  where
    cost (Step (DoubleStop n1 n2) _) | _p8 n1 n2 =
      case (n1 ^. fgr, n2 ^. fgr) of
        (One, Four) -> 0
        (Open, _) -> 0
        (One, Three) -> if n2 ^. fingering . distance >= G7 then 0 else low
        (Two, Four) -> if n2 ^. fingering . distance >= G7 then 0 else low
        _ -> infinity
    cost _ = 0

staticTenth :: Num a => Penalty1 a
staticTenth = P "static tenth (or anything greater than an octave)" cost high
  where
    cost (Step (DoubleStop n1 n2) _) | _m9 n1 n2 =
      case (n1 ^. fgr, n2 ^. fgr) of
        (One, Four) -> 0
        (Open, _) -> 0
        (_, Open) -> high
        _ -> infinity
    cost _ = 0

staticUnison :: Num a => Penalty1 a
staticUnison = P "static unison" cost high
  where
    cost (Step (DoubleStop n1 n2) _)
      | _p1 n1 n2 =
        let f1 = n1 ^. fingering
            f2 = n2 ^. fingering
         in -- Make sure that f1 refers to the lower string
            case (min f1 f2 ^. finger, max f1 f2 ^. finger) of
              (Four, One) -> 0
              (Open, _) -> 0
```

```
                (_, Open) -> 0
                _ -> infinity
    cost _ = 0

staticSecond :: Num a => Penalty1 a
staticSecond = P "static second" cost high
  where
    cost (Step (DoubleStop n1 n2) _) | second n1 n2 =
      case (n1 ^. fgr, n2 ^. fgr) of
        (Four, One) -> 0
        (Open, _) -> 0
        (_, Open) -> 0
        _ -> infinity
    cost _ = 0

staticFourth :: Num a => Penalty1 a
staticFourth = P "static fourth" cost high
  where
    cost (Step (DoubleStop n1 n2) _) | _a4 n1 n2 || _p4 n1 n2 =
      case (n1 ^. fgr, n2 ^. fgr) of
        (Two, One) -> 0
        (Three, Two) -> 0
        (Four, Three) -> 0
        (Open, _) -> 0
        (_, Open) -> 0
        _ -> infinity
    cost _ = 0

staticFifth :: Num a => Penalty1 a
staticFifth = P "static fifth" cost high
  where
    cost (Step (DoubleStop n1 n2) _) | _p5 n1 n2 =
      case (n1 ^. fgr, n2 ^. fgr) of
        (f1, f2) | f1 == f2 -> 0
        (Open, _) -> 0
        (_, Open) -> 0
        _ -> infinity
    cost _ = 0

staticSeventh :: Num a => Penalty1 a
staticSeventh = P "static seventh" cost high
```

```
  where
    cost (Step (DoubleStop n1 n2) _) | seventh n1 n2 =
      case (n1 ^. fgr, n2 ^. fgr) of
        (Open, _) -> 0
        (_, Open) -> 0
        (f1, f2) | f1 >= f2 -> infinity
        _ -> 0
    cost _ = 0

staticMinorSeventh :: Num a => Penalty1 a
staticMinorSeventh = P "static minor seventh" cost high
  where
    cost (Step (DoubleStop n1 n2) _) | _m7 n1 n2 =
      case (n1 ^. fgr, n2 ^. fgr) of
        (One, Three) -> 0
        (Two, Four) -> 0
        (One, Two) -> low
        (Two, Three) -> low
        (Three, Four) -> medium
        _ -> 0
    cost _ = 0

staticMajorSeventh :: Num a => Penalty1 a
staticMajorSeventh = P "static major seventh" cost high
  where
    cost (Step (DoubleStop n1 n2) _) | _M7 n1 n2 =
      case (n1 ^. fgr, n2 ^. fgr) of
        (One, Three) -> 0
        (Two, Four) -> 0
        _ -> 0
    cost _ = 0

-----------------------------------------------------------------
-- Triple/Quadruple Stops
-----------------------------------------------------------------

staticTripleStop :: Num a => Penalty1 a
staticTripleStop = P "static triple stop" cost high
  where
    cost step =
      case sortOn (view str) $
```

```haskell
        step ^.. timestep . _TripleStop . each of
        ns@[n1, n2, n3] ->
          applyP1s mempty doubleStops (set timestep (DoubleStop n1 n2) step)
            + applyP1s mempty doubleStops (set timestep (DoubleStop n2 n3) step)
            + let [f1, _, f3] = ns ^.. traversed . fgr
               in if f1 == f3 && f1 /= Open then infinity else 0
        _ -> 0

staticQuadrupleStop :: Num a => Penalty1 a
staticQuadrupleStop = P "static quadruple stop" cost high
  where
    cost step =
      case sortOn (view str) $
        step ^.. timestep . _QuadrupleStop . each of
        ns@[n1, n2, n3, n4] ->
              let [f1, f2, f3, f4] = ns ^.. traversed . fgr
               in if any
                    (\(x, y) -> x == y && x /= Open)
                    [(f1, f3), (f2, f4), (f1, f4)]
                    then infinity
                    else 0
        _ -> 0
```

For adjacent time steps, we have

```haskell
p2s :: Num a => [Penalty2 a]
p2s =
  [ oneFingerHalfStep
  , samePosition
  , sameString
  , augmentedSecond
  , doubleStringCrossing
  , obliqueFingerCrossing
  , fourOneUpShift
  ]

oneFingerHalfStep :: Num a => Penalty2 a
oneFingerHalfStep = P "one finger half step shift" cost (- medium)
  where
    cost (Step (Single n1) _, Step (Single n2) _)
      | and [_m2 n1 n2, n1 ^. fgr == n2 ^. fgr, n1 ^. str == n2 ^. str] = 1
    cost _ = 0
```

```haskell

samePosition :: Num a => Penalty2 a
samePosition = P "same position" cost (- high)
  where
    cost (Step (Single n1') _, Step (Single n2') _) =
      let [n1, n2] = sortOn pitch [n1', n2']
          (f1, s1, f2, s2) = (n1 ^. fgr, n1 ^. str, n2 ^. fgr, n2 ^. str)
       in binarize $
            if
                | second n1 n2 ->
                  or
                    [ dist f2 f1 == 1 && s2 == s1
                    , dist s2 s1 == 1 && (f1, f2) == (Four, One)
                    , Open `elem` [f1, f2]
                    ]
                | third n1 n2 ->
                  or
                    [ dist f2 f1 == 2 && s2 == s1
                    , dist s2 s1 == 1 && dist f2 f1 == 2
                    , Open `elem` [f1, f2]
                    ]
                | _p4 n1 n2 || _a4 n1 n2 ->
                  or
                    [ (f1, f2) == (One, Four) && s2 == s1
                    , dist s2 s1 == 1 && dist f2 f1 == 3
                    , Open `elem` [f1, f2]
                    ]
                | _p5 n1 n2 ->
                  or
                    [ f1 == f2 && dist s2 s1 == 1
                    , Open `elem` [f1, f2]
                    ]
                | _m6 n1 n2 ->
                  or
                    [ dist f2 f1 == 1 && dist s2 s1 == 1
                    , dist s2 s1 == 2 && (f1, f2) == (Four, One)
                    , Open `elem` [f1, f2]
                    ]
                | _M6 n1 n2 ->
                  or
                    [ dist f2 f1 `elem` [1, 2] && dist s2 s1 == 1
```

```
                      , dist s2 s1 == 2 && (f1, f2) == (Four, One)
                      , Open `elem` [f1, f2]
                      ]
                  | seventh n1 n2 ->
                    or
                      [ dist f2 f1 == 2 && dist s2 s1 == 1
                      , dist s2 s1 == 2 && dist f2 f1 == 2
                      , Open `elem` [f1, f2]
                      ]
                  | _p8 n1 n2 ->
                    or
                      [ dist s2 s1 == 1 && dist f2 f1 == 3
                      , dist s2 s1 == 2 && dist f1 f2 == 1
                      , Open `elem` [f1, f2]
                      ]
                  | otherwise ->
                    not . null $
                      F.foldr1
                        L.intersect
                        (map position [n1 ^. fingering, n2 ^. fingering])
    cost steps =
      let positions = steps ^.. both . notes . fingering . to position
       in binarize $ not (null (F.foldr1 L.intersect positions))

augmentedSecond :: Num a => Penalty2 a
augmentedSecond = P "augmented second 1-2, 2-3" cost medium
  where
    cost (x, y) = case (x ^. timestep, y ^. timestep) of
      (Single n1, Single n2) ->
        binarize $
          _m3 n1 n2 && (n1 ^. fgr, n2 ^. fgr)
            `elem` [(One, Two), (Two, One), (Two, Three), (Three, Two)]
      _ -> 0

sameString :: Num a => Penalty2 a
sameString = P "same string" cost (- high)
  where
    cost (x, y) = case (x ^. timestep, y ^. timestep) of
      (Single n1, Single n2) -> binarize $ n1 ^. str == n2 ^. str
      (DoubleStop n11 n12, DoubleStop n21 n22) ->
        binarize $ n11 ^. str == n21 ^. str && n12 ^. str == n22 ^. str
      (Single n1, DoubleStop n21 n22) ->
        binarize $ (n1 ^. str) `elem` [n21 ^. str, n22 ^. str]
      (DoubleStop n11 n12, Single n2) ->
        binarize $ (n2 ^. str) `elem` [n11 ^. str, n12 ^. str]
      _ -> 0

doubleStringCrossing :: Num a => Penalty2 a
doubleStringCrossing = P "double string crossing" cost high
  where
    cost (Step (Single n1) _, Step (Single n2) _) =
      binarize $ abs (dist (n1 ^. str) (n2 ^. str)) > 1
    cost _ = 0

obliqueFingerCrossing :: Num a => Penalty2 a
obliqueFingerCrossing = P "oblique finger crossing" cost high
  where
    cost (Step (Single n1) _, Step (Single n2) _) =
      binarize $
        abs (dist (n1 ^. str) (n2 ^. str)) == 1 && n1 ^. fgr == n2 ^. fgr
    cost _ = 0

fourOneUpShift :: Num a => Penalty2 a
fourOneUpShift = P "shift up a step with 4-1" cost high
  where
    cost (Step (Single n1) _, Step (Single n2) _)
      | and
          [ second n1 n2
          , pitch n1 < pitch n2
          , n1 ^. fgr == Four
          , n2 ^. fgr == One
          , n1 ^. str == n2 ^. str
          ] =
        1
    cost _ = 0
```

Bibliography

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2003/2009). *Introduction to algorithms*. MIT Press. https://edutechlearners.com/download/Introduction_to_algorithms-3rd%20Edition.pdf

Crawford, T., & Lewis, R. (2016). Review: Music encoding initiative. *Journal of the American Musicological Society*, *69*(1), 273–285. https://doi.org/10.1525/jams.2016.69.1.273

Daudin, C., Fober, D., Letz, S., & Orlarey, Y. (2009). The guido engine - a toolbox for music scores rendering.

Eppstein, D. (1997). Finding the k shortest paths. https://doi.org/10.1137/S0097539795290477

Flesch, C., Rosenblith, E., & Mutter, A.-S. (1924/2000). *The art of violin playing*. Carl Fischer.

Flesch, C., & Rostal, M. (1942/1987). *Scale system*. Ries & Erler; Carl Fischer.

Flesch, C., Schwarz, B., & Menuhin, Y. (1966/1978). *Violin fingering its theory and practice*. Barrie & Jenkins.

Forney, G. D. (1973). The viterbi algorithm. *Proceedings of the IEEE*, *61*(3), 268–278.

Galamian, I. (1962). *Principles of violin playing and teaching*. Simon; Schuster Company.

Galamian, I., & Neumann, F. (1966). *Contemporary violin technique*. Galaxy Music Corporation.

Good, M. D. (2009). Using musicxml 2.0 for music editorial applications. *Digitale Edition zwischen Experiment und Standardisierung*, 157–173. https://doi.org/10.1515/9783110231144.157

Huron, D. (2002). Music information processing using the humdrum toolkit: Concepts, examples, and lessons. *Computer Music Journal*, *26*, 11–26. https://doi.org/10.1162/014892602760137158

Klein, D., & Manning, C. D. (2003). A parsing: Fast exact viterbi parse selection. *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1*, 40–47. https://doi.org/10.3115/1073445.1073461

Launchbury, J., & Peyton Jones, S. (1998). Lazy functional state threads. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, *29*. https://doi.org/10.1145/773473.178246

Lilypond, D. T. (2019). Lilypond - essay on automated music engraving. http://lilypond.org/doc/v2.19/Documentation/essay-big-page.html

Liska, U., Bjuhr, P., & Solomon, M. Interfacing mei and gnu lilypond. In: *Music encoding conference 2015*. 2015, May. http://lilypondblog.org/wp-content/uploads/2015/06/mei2ly.pdf

Maezawa, A., Itoyama, K., Komatani, K., Ogata, T., & Okuno, H. G. (2012). Automated violin fingering transcription through analysis of an audio recording. *Computer Music Journal*, *36*, 57–72.

Mozart, L., Knocker, E., & Einstein, A. (1756/1985). *A treatise on the fundamentals of violin playing*. Oxford University Press. https://archive.org/details/GrndlicheViolinschule1787

Myers, P. T. (2011). *Guidelines for violin fingerings based on editions of ivan galamian and carl flesch* (Doctoral dissertation).

Nagata, W., Sako, S., & Kitamura, T. (2014). Violin fingering estimation according to skill level based on hidden markov model. *ICMC*.

Nakamura, E., Saito, Y., & Yoshii, K. (2019). Statistical learning and estimation of piano fingering. *CoRR*, *abs/1904.10237*. http://arxiv.org/abs/1904.10237

Nápoles López, N., Vigliensoni, G., & Fujinaga, I. (2019). The effects of translation between symbolic music formats: A case-study with humdrum, lilypond, mei, and musicxml. *Music Encoding Conference*.

of Musical Electronics Industry AMEI, A., & MMA, M. M. A. (2020). Midi 2.0 specification. http://www.midi.org

Pang, A., Stewart, D., Seefried, S., & Chakravarty, M. M. T. (2004). Plugging haskell in. *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, 10–21. https://doi.org/10.1145/1017472.1017478

Preedy, V. R., & Watson, R. R. (2010). *Handbook of Disease Burdens and Quality of Life Measures*, 4288–4288. https://doi.org/10.1007/978-0-387-78665-0_6363

Rabiner, L. R. (1989). A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, *77*(2), 257–286.

Radicioni, D., Anselma, L., & Lombardo, V. (2004). An algorithm to compute fingering for string instruments.

Ricci, R., & Zayia, G. H. (2007). *On glissando*. Indiana University Press. https://books.google.com/books?id=iDQZAQAAIAAJ

Sapp, C. (2015). Graphic to symbolic representations of musical notation. In M. Battier, J. Bresson, P. Couprie, C. Davy-Rigaux, D. Fober, Y. Geslin, H. Genevois, F. Picard, & A. Tacaille (Eds.), *Proceedings of the first international conference on technologies for music notation and representation, tenor 2015, paris, france, may 28-30, 2015* (pp. 124–132). Institut de Recherche en Musicologie. https://doi.org/10.5281/zenodo.923829

Sayegh, S. (1988). *Artificial intelligence approach to string instrument fingering.* (Doctoral dissertation).

Sayegh, S. I., & Tenorio, M. F. (1989). Fingering for string instruments with the optimum path paradigm.

Selfridge-Field, E. (Ed.). (1997). *Beyond midi: The handbook of musical codes*. MIT Press. https://doi.org/10.5555/275928

Yampolsky, I. M., Lumsden, A., & Oistrakh, D. (1980). *The principles of violin fingering*. Music Department Oxford University Press.

Yegge, S. (2007). The pinocchio problem. http://steve-yegge.blogspot.com/2007/01/pinocchio-problem.html

Ysaÿe, E., & Szigeti, J. (1967). *Exerices et gammes pour violon*. Schott.

Zhihui Du, Zhaoming Yin, & Bader, D. A. (2010). A tile-based parallel viterbi algorithm for biological sequence alignment on gpu with cuda. *2010 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW)*, 1–8. https://doi.org/10.1109/IPDPSW.2010.5470903