

Judah Daniels

# **Inferring Harmony from Free Polyphony**

Computer Science Tripos – Part II

Clare College

July, 2023

## Declaration of originality

I, Judah Daniels of Clare College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose. I am content for my dissertation to be made available to the students and staff of the University.

Signed Judah Daniels

Date May 8, 2023

# Proforma

---

Candidate Number: **2200D**  
Project Title: **Inferring Harmony from Free Polyphony**  
Examination: **Computer Science Tripos – Part II, July, 2023**  
Word Count: **8900<sup>1</sup>**  
Code Line Count: **2672<sup>2</sup>**  
Project Originator: **Christoph Finkensiep**  
Supervisor: **Dr Peter Harrison**

## Original Aims of the Project

## Work Completed

## Special Difficulties

There were no special difficulties encountered in this project

---

<sup>1</sup>This word count was computed by `texcount -v1 -sum -sub=chapter main.tex`

<sup>2</sup>This code line count was computed by using `cloc`

# Contents

---

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Previous Work . . . . .	1
1.2	Contributions of my Project . . . . .	2
<b>2</b>	<b>Preparation</b>	<b>4</b>
2.1	Starting Point . . . . .	4
2.1.1	Relevant courses and experience . . . . .	4
2.2	Inferring Harmony . . . . .	5
2.3	The Protovoice Model . . . . .	8
2.3.1	Outer Structure: Slices and Transitions . . . . .	8
2.3.2	Inner Structure: Notes and Edges . . . . .	9
2.3.3	Generative Proto-voice Operations . . . . .	10
2.4	Overview of Project Aims . . . . .	11
2.5	Requirements Analysis . . . . .	12
2.6	Software Engineering Techniques . . . . .	13
2.6.1	Development model . . . . .	13
2.6.2	Languages, libraries and tools . . . . .	13
<b>3</b>	<b>Implementation</b>	<b>15</b>
3.1	Repository Overview: . . . . .	16
3.2	Proto-voice Fitness Function . . . . .	17
3.2.1	Motivation of Fitness Function . . . . .	17
3.2.2	Implementation of Fitness Function . . . . .	18
3.3	Harmony Module . . . . .	21
3.3.1	Chord Labels . . . . .	22
3.3.2	Inferring Harmony . . . . .	22
3.3.3	Optimisations . . . . .	23

3.4	The Proto-Voice Harmony Parser . . . . .	24
3.4.1	Overview of Harmony Parser Design . . . . .	24
3.4.2	Parse States . . . . .	25
3.4.3	Parsing Operations . . . . .	26
3.4.4	Conserving Segment Boundaries . . . . .	27
3.5	Core Algorithms . . . . .	28
3.6	Extension: Heuristic Search Algorithms . . . . .	30
3.6.1	Stochastic Beam Search . . . . .	30
3.7	Testing . . . . .	32
<b>4</b>	<b>Evaluation</b>	<b>33</b>
4.1	Harmony Model . . . . .	34
4.2	Heuristic Search . . . . .	36
4.2.1	Prediction Accuracy . . . . .	36
4.2.2	Interpretability . . . . .	37
4.2.3	Scalability . . . . .	38
4.3	Success Criteria . . . . .	39
<b>5</b>	<b>Conclusions</b>	<b>40</b>
	<b>Bibliography</b>	<b>40</b>
<b>A</b>	<b>Additional Information</b>	<b>46</b>
<b>B</b>	<b>Project Proposal</b>	<b>47</b>
B.1	Abstract . . . . .	2
B.2	Substance and Structure . . . . .	2
B.2.1	Core: Search . . . . .	2
B.2.2	Core: Evaluation . . . . .	3
B.2.3	Extension . . . . .	3
B.2.4	Overview . . . . .	3
B.3	Starting Point . . . . .	4
B.4	Success Criteria . . . . .	4
B.5	Timetable . . . . .	6
B.6	Resources . . . . .	7
B.7	Supervisor Information . . . . .	7

## Acknowledgements

I'd like to thank Christoph Finkensiep and Peter Harrison for being awesome.

# Chapter 1

## Introduction

---

The problem of inferring the harmonic structure of a piece of music, represented by a sequence of chords, is a fundamental task in the analysis and understanding of Western tonal music. Free polyphony refers to one of the most general forms of western music, wherein multiple independent melodic lines (or voices) are combined without adhering to strict rules or constraints. Symbolic approaches to Automatic Chord Estimation (ACE) usually solve such problems by taking a sequence of notes as the input, and generating a sequence of chord labels as the output. In this project, the input, called the *surface*, is a sequence of notes with precise descriptions of when each note begins (onset) and ends (offset). The output is a sequence of chord labels, each describing a *chord segment*, a group of notes that sound simultaneously or in close succession.

This project proposes a novel approach to ACE that integrates the proto-voice model [15], a recent model of note-level structure, with a chord segment reduction process. This contributes an **interpretable** framework for inferring sequences of chord labels by providing an explicit explanation of how those labels relate to the surface notes. Furthermore, I design and implement a **novel fitness function** and **efficient heuristic search algorithms** to improve on the computational complexity of the parsing algorithm provided in the proto-voice paper [15] from **exponential to linear time** with respect to the length of the piece.

### 1.1 Previous Work

Automatic chord estimation systems have ranged from handcrafted grammar/rule-based approaches [34] [61], to the development of optimisation algorithms [42]. In more recent years, deep learning methods have risen in popularity, exploiting large datasets and improved compute power. Examples of deep architectures used include recurrent neural networks (RNNs) [1] and long short-term memory (LSTM) networks [5]. Some systems also make use of audio rather than symbols as an input, utilising convolutional neural networks (CNNs), and most recently convolutional recurrent neural networks (CRNNs) [62] [9]. These architectures have found success due to their ability to capture temporal dependencies in the music.

A broad limitation of these existing ACE approaches is that they do not specify the precise

relationship between the surface and the inferred chord labels. Deep-learning approaches typically involve a black box inference of the chord labels based on the raw input. For example, a recent chord classification model described by McLeod [35] consists of a neural network that takes a sequence of notes represented as one-hot feature vectors and outputs a distribution over a set of 1540 chord labels. Earlier probabilistic optimisation approaches consider the surface as being generated by a noisy process. Pardo and Birmingham’s influential algorithm compares the notes in each segment to a set of templates describing common chords [42]. There is no explicit relationship described between the notes in each segment and the nearest template.

The recent paper *Modeling and Inferring Proto-voice Structure in Free Polyphony* [15] presents the proto-voice model, a generative model that represents a musical piece as a result of recursively applying primitive operations on notes. The combination of these operations forms a hierarchical structure which encodes explicit relations between all the notes in a piece. The paper presents a *chart parsing* algorithm which can parse a piece of music according to the grammar outlined by the model, returning a list of possible *derivations*, that is, all the ways the piece of music could have been created through the recursive application of these simple rules.

In this project, I build upon the proto-voice model by creating a new *proto-voice harmony parser* to enable the generation of a *reduction* of the musical surface that can be used to directly infer chord labels. In music analysis, a reduction refers to the process of simplifying a complex musical surface to reveal its underlying harmonic structure. The new parser is used to reduce the piece, with the goal of finding a reduction that preserves the harmony of the piece, while eliminating extraneous notes. This reduction facilitates the inference of descriptive chord labels and provides an *explicit explanation* through the corresponding proto-voice derivation, which has not been attempted before.

While the chart parsing algorithm provided [12] could in theory be used to generate harmonic annotations, the naive exhaustive parse strategy would be prohibitively time-consuming in practice for all but the shortest musical extracts; one half measure can have over 100,000 valid derivations [13]. Moreover, not all derivations are created equally. Each one corresponds to a specific interpretation of the note-level structure within a given piece of music. These reductions align with intuitive or theoretical musical interpretations to differing extents, with most being implausible. Determining the relative plausibility of derivations remains an *open question*.

I provide answers to these questions by proposing and implementing a **novel heuristic fitness function** for proto-voice derivations and address the **exponential search space** using heuristics and pruning techniques to significantly reduce the time and space complexity from  $\mathcal{O}(2^{n \cdot m})$  to  $\mathcal{O}(n \cdot m)$ , where  $n$  is the length of the piece and  $m$  is the maximal number of notes in each slice. I show that my heuristic search algorithm produces a **statistically significant improvement** in accuracy over a random parse for a variety of pieces, a qualitative improvement in **interpretability**, and **scales linearly** with the input dimensions.

## 1.2 Contributions of my Project

The implementation of this project achieved all of the original aims. In doing so, the key contributions include:



1. Developing a novel approach to Automatic Chord Estimation that integrates the proto-voice model with a chord segment reduction process (Section 3). This not only provides an interpretable and explainable framework for Automatic Chord Estimation but also has the potential to contribute to a more comprehensive understanding of the underlying structure of Western tonal music.
2. Proposing and implementing an informed heuristic-based fitness function to evaluate proto-voice derivations, which decomposes into a estimated score for each reduction step (Section 3.2.1). This enables the identification of musically meaningful interpretations of the piece while discarding implausible derivations.
3. Designing and developing the proto-voice harmony parser which can generate reductions of the musical surface to facilitate chord label inference (Section 3.4). This has the benefit over previous approaches of providing an explicit explanation through the proto-voice derivation, with potential to be incorporated in more complex systems to improve their interpretability.
4. Introducing a novel heuristic search strategy to efficiently explore the large space of possible derivations, using a beam search and pruning techniques to significantly reduce the computational complexity (Section 3.6.1).
5. Demonstrating the effectiveness of this approach on a diverse set of musical pieces in terms of accuracy, performance and interpretability (Section 4).

# Chapter 2

## Preparation

---

This Chapter outlines the essential concepts, models and representations used in the implementation of this project. First, the approach taken to infer labels from pitches is described (Section 2.2). Next, the proto-voice model is introduced, explaining the structures and operations used to manipulate the musical surface (Section 2.3). Subsequently, an description of the existing parser is presented, highlighting the limitations and innovations required (Section 2.4). Finally, a discussion and analysis of project requirements is provided (Section 2.5) followed by a description of the software engineering techniques used throughout the project (Section 2.6).

### 2.1 Starting Point

This project builds on the codebase of Finkensiep, the `protovoices-haskell` repository [12], which contains an implementation of the proto-voice model, types and functions for representing and working with protovoice derivations, and an implementation of a chart parser and greedy parser. Everything else in the codebase is written by me, including the new harmony parser, a harmony module, the fitness function and heuristic search algorithms.

#### 2.1.1 Relevant courses and experience

*IB Data Science* and *Artificial Intelligence* provided some of the Machine Learning background required and *Formal Models of Language* introduced some of the ideas and terminology used in the proto-voice model.

Although I had ample experience coding in Python from personal projects, I had only a few months' experience with Haskell before starting this project, being introduced to the language during an internship in the summer of 2022. The existing `protovoices-haskell` repository is a large and complex codebase, and the parser provided is non-trivial. Completing this project involved learning many features of Haskell, including gaining familiarity with Monad Transformers, the subtleties of lazy execution and its rich type system.

## 2.2 Inferring Harmony

The approach taken to infer harmony is to take a *multi-set* of pitches as an input, describing the counts of the pitches present in a chord segment, and predict the chord label that best describes that chord segment using *probabilistic chord-profiles*, probability vectors describing the relationship between chord labels and pitches classes. Each label is inferred independently for each chord segment, ignoring contextual information.

The notation most commonly used for Western tonal music is called a *score*, a symbolic abstraction of a piece of music based on a *2-dimensional axis*. The marks on the score represent notes, with the *pitch* of the note represented by its position on the vertical axis, and the notes' placement in time represented by its position on the horizontal axis. Chord labels are placed above the segment they describe, shown in Figure 2.1.

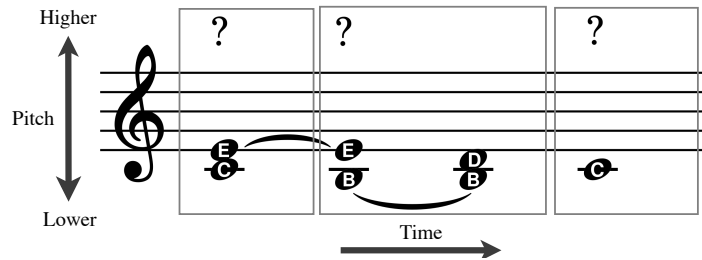


Figure 2.1: An example of music notation showing a short phrase. The goal is to find chord labels that describe each segment.

### Pitch Classes

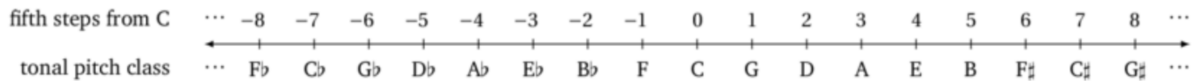


Figure 2.2: Tonal pitch class representation. The reference pitch class is C.

The representation used for pitches is the *tonal pitch class*, of which an implementation is provided in the `haskell-musicology` library [11]. The central object of this representation is the *interval*, the ‘distance’ between pitch classes, which is described by an integer representing the distance between pitches classes along the *line of fifths* (See Figure 2.2). Pitches classes are then derived from intervals by interpreting them with respect to a reference pitch class. This is similar to the relation between vectors (intervals) and points (pitch classes) [11]. Although the line of fifths is theoretically infinite, in this project the reference pitch class is set to C, and the intervals can range from  $-14$  to  $14$ , resulting in 29 unique tonal pitch classes (octaves are ignored).

### Chord Labels

Each chord label  $l \in \mathcal{L}$  comprises a root note and a chord-type (e.g. major, minor), and is thus described as a product  $\mathcal{L} = \mathcal{P} \times \mathcal{C}$  where  $\mathcal{P}$  is the set of pitch classes and  $\mathcal{C}$  is the set of

chord-types. The set of chord-types used in this project corresponds to the Digital Cognitive Musicology Lab (DCML) annotation standard, and consists of 14 unique types. As a result, there are  $14 \times 29 = 406$  unique chord labels considered. I provide an implementation of chord labels using these chord-types as part of the Harmony model.

## Slices

**Definition 2.2.1** (Multi-set). A *multi-set* is a set that allows multiple instances for each of its elements, formally defined as an ordered pair  $(A, m)$  where  $A$  is the *underlying set* of the multiset, and  $m : A \rightarrow \mathbb{Z}^+$  gives the *multiplicity*, such that the number of occurrences of  $a$  in  $(A, m)$  is given by  $m(a)$ .

Slices are multi-sets of pitches, used describe sets of pitches that sound simultaneously or in close succession, i.e. a set of pitches that sound within a ‘slice’ of time, or single time-frame.

**Definition 2.2.2** (Slice). A *slice*  $s \in \mathcal{S}$  is defined as a multi-set of pitches  $(\mathcal{P}, m)$ .

$$s = \{p_1^{m(p_1)}, \dots, p_n^{m(p_n)}\} \quad (2.1)$$

**Definition 2.2.3** (Slice Vector). The *slice vector* for a given slice is defined as 29-wide-vector containing the counts of each corresponding pitch class present:

$$s = [m(\text{C}\flat\flat), m(\text{G}\flat\flat) \dots, m(\text{B}\flat), m(\text{F}), m(\text{C}), m(\text{G}), m(\text{D}), \dots, m(\text{F}\sharp\sharp), m(\text{C}\sharp\sharp)] \quad (2.2)$$

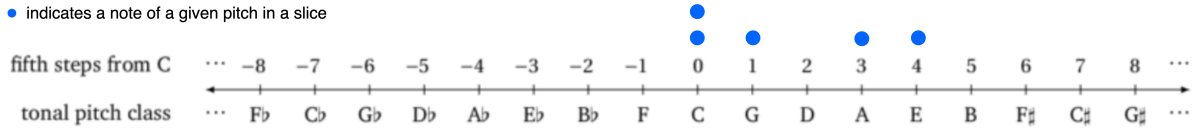


Figure 2.3: A representation of a slice showing the counts of each pitch.

## Chord Profiles

**Chord profiles**, also known as chord templates, are often used in Automatic Chord Estimation (ACE) systems to find the chord label which best matches a given slice.

Chord profiles describe the *intervals* with respect to the *root note* of the chord that are typically present within a segment described by the given chord type. Figure 2.4 gives example chord profiles for major, minor and minor 7 chord-types, and shows how the profiles describe which notes are present for a given chord, by interpreting the intervals of the chord profile with respect to the root note. Intuitively, for a given chord-type, shifting the corresponding profile along the axis shifts the root-note of the chord. In Figure 2.4, the notes in the slice match the chord profile for an *A minor 7* chord exactly, and almost matches the profile for a *C Major* chord.

Chord labeling is inherently ambiguous; different chord labels can consist of similar (or even identical) groups of notes, and not all notes directly relate to harmony. There are cases where the notes in a given segment match a chord exactly, but an analyst would decide that one of

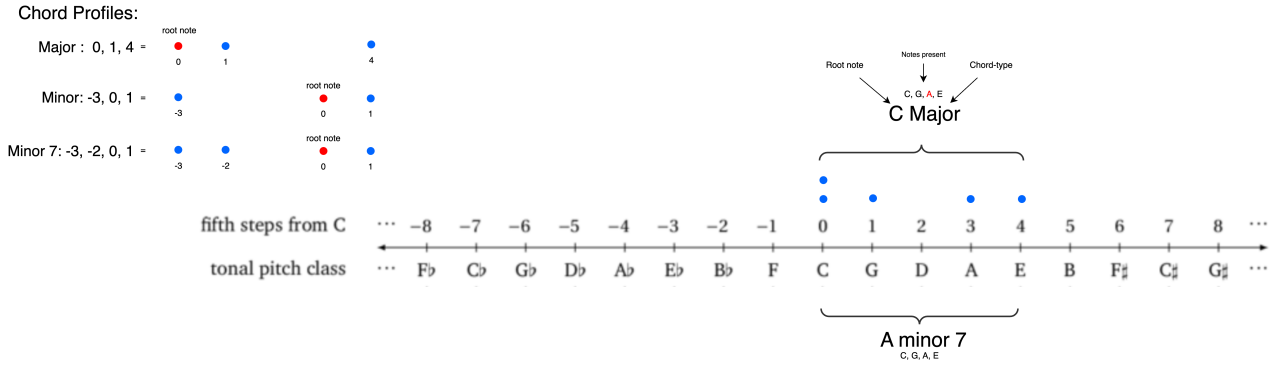


Figure 2.4: Application of chord profiles. The dots represent the presence of a pitch.

those notes is in fact an ornament and the remaining notes exemplify a different chord. For example, the notes A, C, E and G match the profile for an *Am7* chord exactly, but there are many cases where the correct interpretation would decide that the A is an ornament, and the chord-tones C, E and G exemplify a C major chord.

## Probabilistic Chord Profiles

Probabilistic chord profiles aim to address this limitation of standard chord profiles by assigning each note a probability value indicating the likelihood of its presence in a particular chord. In the paper *A Bayesian Model of Extended Chord Profiles* [14], Finkensiep extends this idea by inferring two different distributions for each chord type. Notes in a given segment are categorised into chord-tones or ornaments, where the chord-tones are the notes that directly relate to the chord. The paper presents a *chord-tone distribution* and an *ornament distribution* for each chord-type, inferred from a labelled dataset. In Figure 2.5 the most common intervals are 0, 1 and 4, which corresponds exactly to the major chord profile shown in Figure 2.4.

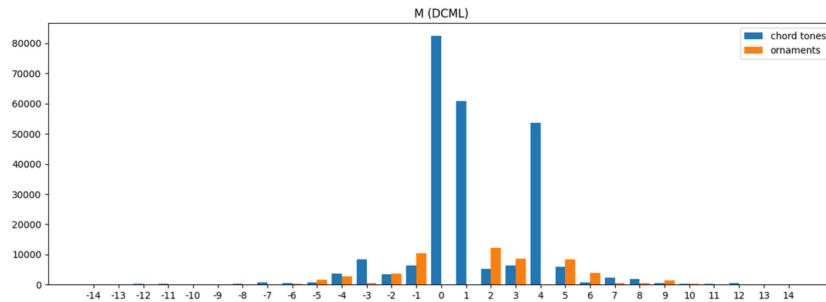


Figure 2.5: Relative counts of intervals relative to the root note in a major chord segment, distinguished by chord-tones and ornaments. Graph taken from the *chord-types and ornamentation repository* [14] **TODO: replace this with two graphs side by side showing the chord-tone and ornament distribution**

**Definition 2.2.4** (Probabilistic Chord Profiles). The probabilistic chord profiles for a given chord-type  $c$  are a pair of probability vectors,  $(\mathbf{p}_{\text{chord-tone}}^{(c)}, \mathbf{p}_{\text{ornament}}^{(c)})$ , where  $\mathbf{p}_{\text{chord-tone}}^{(c)}$  and  $\mathbf{p}_{\text{ornament}}^{(c)}$  represent the chord-tone distribution and ornament distribution respectively. Each probability vector has the same size as the slice vectors described above, and sum of all values in each probability vector equals 1.

## 2.3 The Protovoice Model

The novel approach taken by this project is to integrate the protovoice model with a chord segment reduction process. The protovoice model is used to generate a sequence of *slices* for each chord segment to be labelled, wherein each slice clearly denotes the underlying harmony, and extraneous notes are *explained away*. By performing this reduction, the chord label inference is guided towards the most plausible interpretation for each segment.

The protovoice model is a generative model for tonal music based on a set of primitive operations that are applied recursively to create a hierarchical structure representing a musical surface. These operations are derived from a set of intuitive and theoretically motivated transformations, and was proposed by Finkensiep in the paper *Modeling and Inferring Protovoice Structure in Free Polyphony* [15]. The model is primarily concerned with the analysis of Western Classical music, although its expressiveness and generality means it could be applied to different musical styles including jazz or some popular western music [13].

The next two sections provide a description of the data structures used to represent the musical surface in the protovoice model, followed by a description of the generative protovoice operations.

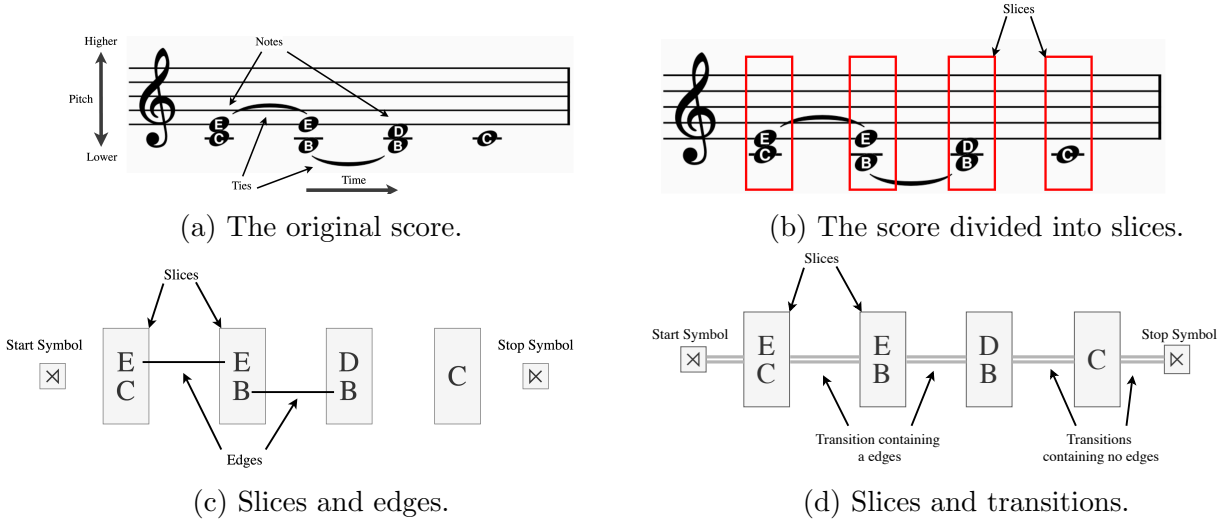
### 2.3.1 Outer Structure: Slices and Transitions

#### Slices

Groups of notes are stored as *slices*, initially representing the maximal durations in which a single group of pitches sound, shown in Figure 2.6b. Figure 2.6a shows a score representing a short musical phrase. Notes have an onset describing when they begin, and a duration describing how long they sound for. Ties are used to indicate that the same note continues sounding, rather than ending and starting again subsequently. As notes have different durations, notes that are simultaneous with several non-simultaneous notes are split among the corresponding slices, such as the note E in Figure 2.6a. In this case *edges* connect each of these notes, which ensures that a single surface note is generated through a single generation process [15], as shown in Figure 2.6c.

#### Transitions

Transitions are introduced to relate adjacent slices with a configuration of edges, connecting notes in the adjacent slices. The unreduced surface contains *regular* edges, in particular, *repetitions* which connect single surface notes that span multiple slices, such as the edges shown in Figure 2.6c. Other types of edges exist only during the generation (or reduction) of a piece, and are described in the Section 2.3.2. An unreduced surface (e.g. Figure 2.6d) contains only *frozen* transitions. Frozen transition are *terminal* (analogous to CFGs), meaning no more generative operations can be applied. When all transitions are frozen, this means the generative process has completed, which is the case with the original surface.



**Definition 2.3.1** (Transition). A *transition*  $t \in \mathcal{T}$  relates two adjacent slices,  $s_l$  and  $s_r$ , with a configuration of edges  $e$ .

$$t = (s_l, e, s_r) \quad (2.3)$$

**Definition 2.3.2** (Path). A *path* is the data structure used to represent an alternating sequence of transitions and slices, where the *head* of the path is accessible in constant time, defined inductively as:

$$P = t \mid t s P \quad t \in \mathcal{T}, s \in \mathcal{S} \quad (2.4)$$

As slices and transitions contain notes and edges respectively, slices and transitions are called *outer structure*, and the notes and edges contained therein are called *inner structure*.

### 2.3.2 Inner Structure: Notes and Edges

The following is a restatement of the core of the proto-voice model, as described in the original paper by Finkensiep [15].

Internally, proto-voices are represented as a directed graph with one vertex for each note contained in a slice, a vertex each for the beginning ( $\bowtie$ ) and end ( $\bowtie$ ) of the piece, and edges that indicate step-wise connections between notes, which are contained within transitions. A *proto-voice* is a path within this graph. The protovoice model is characterised by stepwise generative operations on notes. *Regular edges* indicate a sequential connection between two notes, which may be *elaborated* by introducing a repetition or a neighbour of either parent note, or both if the parents have the same pitch. The interval along a regular edge is always within a range of a step. *Passing edges* indicate connections between two notes with an interval larger than a step, introducing a new subordinate proto-voice. These passing edges must be filled with stepwise passing notes from either end [15].

The generation of a piece begins with the empty piece ( $\bowtie \rightarrow \bowtie$ ) and is defined by the recursive application of elaboration rules.

These elaboration rules relate new child notes to one or two existing *parent* notes.

*Single-sided* rules attach a new repetition or neighbour note with an edge connected to a single parent note:

$$\begin{aligned}
 x &\Longrightarrow n \rightarrow x && \text{left-neighbor} \\
 x &\Longrightarrow x \rightarrow n && \text{right-neighbor} \\
 x &\Longrightarrow x' \rightarrow x && \text{repeat-before} \\
 x &\Longrightarrow x \rightarrow x' && \text{repeat-after}
 \end{aligned} \tag{2.5}$$

*Double-sided* rules are represented by *edge replacement*.

$$\begin{aligned}
 \bowtie \rightarrow \bowtie &\Longrightarrow \bowtie \rightarrow x \rightarrow \bowtie && \text{root-note} \\
 x_1 \rightarrow x_1 &\Longrightarrow x_1 \rightarrow x' \rightarrow x_2 && \text{full-repeat} \\
 x \rightarrow y &\Longrightarrow x \rightarrow y' \rightarrow y && \text{repeat-before'} \\
 x \rightarrow y &\Longrightarrow x \rightarrow x' \rightarrow y && \text{repeat-after'} \\
 x_1 \rightarrow x_1 &\Longrightarrow x_1 \rightarrow n \rightarrow x_2 && \text{full-neighbor}
 \end{aligned} \tag{2.6}$$

*Passing* rules, finally, fill passing edges with notes from the left or right until the progression is fully stepwise.

$$\begin{aligned}
 x \dashrightarrow y &\Longrightarrow x \rightarrow p \dashrightarrow y && \text{passing-left} \\
 x \dashrightarrow y &\Longrightarrow x \dashrightarrow p \rightarrow y && \text{passing-right} \\
 x \dashrightarrow y &\Longrightarrow x \dashrightarrow p \rightarrow y && \text{passing-final}
 \end{aligned} \tag{2.7}$$

The inner structure provided by protovoices captures the sequential and functional organisation of notes, but does not capture when notes are simultaneous. To model simultaneity, notes and edges are integrated into the outer structure of slices and transitions as described in Section 2.3.1.

### 2.3.3 Generative Proto-voice Operations

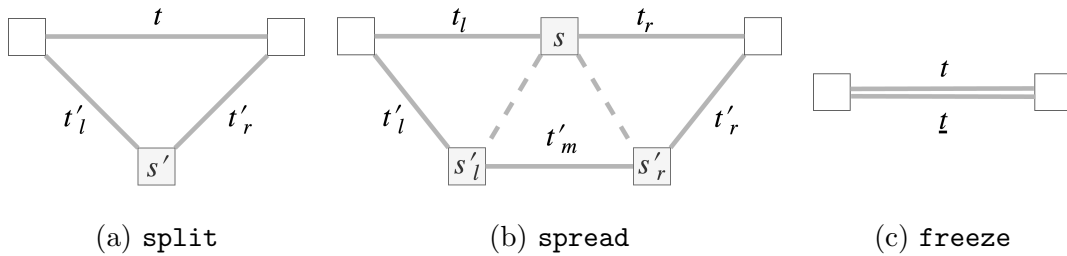


Figure 2.7: The three operations on outer structure. The original slices and transitions are shown at the top, while the generated structure is shown underneath. Figure reproduced from original paper [13]

The outer structure is generated through the recursive application of three production rules:

- A **split** replaces a parent transition  $t$  by inserting a new slice  $s'$  and two surrounding transitions  $t_l$  and  $t_r$ . One or more inner operations can be applied to each of the edges



in  $t$ , and the resulting edges can be discarded or kept to form the new edges of  $t_l$  and  $t_r$ .

$$t \rightarrow t'_l \ s' \ t'_r \quad (2.8)$$

- A **spread** replaces a parent slice  $s$  by distributing its notes to two child slices  $s'_l$  and  $s'_r$ . This allows a vertical configuration of notes (i.e a slice) to become sequential, thereby generating implied harmonies such as those produced by broken chords, chords played without all the notes beginning simultaneously. These latent harmonies can also exist within a single melodic line, thus a single melody can be generated from an implied harmony. During a spread, passing edges can be introduced between any two of the child slices.

$$t_l \ s \ t_r \rightarrow t'_l \ s'_l \ t'_m \ s'_r \ t'_r \quad (2.9)$$

- A **freeze** marks a transition as terminal, preventing any further application of operations to its edges.

$$t \rightarrow \underline{t} \quad (2.10)$$

## 2.4 Overview of Project Aims

This project aims to address the **exponential time complexity** of the original proto-voice parser by developing a **novel heuristic** used to guide an efficient beam search algorithm to find **excellent solutions** in **linear time**.

There are two input dimensions in this problem. First is the number of notes in each slice, referred to as the size of the slice,  $m$ . The worst case is dominated by the number of possible **unsplit** reductions at a single step, which corresponds to the number of configurations of inner operations involved. The number of possible inner operations is dominated in the worst case by the number of double-sided operations,  $\mathcal{O}(|parents|^2 \cdot |children|) = \mathcal{O}(m^3)$ . As a **split** operation can consist of any configuration of these inner operations, the number of **split** operations is  $\mathcal{O}(2^{m^3}) = \mathcal{O}(2^m)$  where  $m$  is the maximal number of notes in each slice involved and  $c$  is a constant. The second input is the number of slices in the original surface, referred to as the length of the surface,  $n$ . In the worst case, the total number of complete reductions is given by  $\mathcal{O}(2^{m^n}) = \mathcal{O}(2^{m \cdot n})$ . Thus the number of possible reductions is **exponential** with respect to **two inputs**, the size of each slice  $m$  and the length of the surface  $n$ .

I propose and implement a novel stochastic beam search algorithm to handle these dimensions of complexity in Section 3.6.1. In order to achieve this, a heuristic fitness function is developed that provides an approximate score for each partial reduction. This fitness function can be decomposed into each reduction step, allowing it to be used to score each reduction step, giving more musically meaningful reductions a greater fitness score. Finally, a new harmony parser is developed that adds constraints to ensure chord segments are preserved throughout the reduction.

## 2.5 Requirements Analysis

The Success Criteria are given in the Project Proposal. During the preparation phase, the Success Criteria were refined in light of increased clarity from reading the literature related to the project. Concretely, the final Success Criterion given in the Proposal describing the development of heuristic search algorithms has been divided into two criteria, developing the heuristic, and developing the search algorithms.

This project will be deemed a success given it achieves:

- A harmony module that can use a reduced surface to infer chord labels, and quantitatively evaluate its accuracy against the ground truth annotations.
- A heuristic fitness function that can be used to judge the relative plausibility of derivations.
- An implementation of a new parser for the protovoice model which finds possible reductions of a musical surface.
- Extension: One or more search algorithms that make use of the developed heuristic to inform the search, dealing with the two dimensions of exponential complexity.

Table 2.1: Project Deliverables

ID	Deliverable	Priority	Risk
core1	Harmony Model	High	Low
core2	End to End Pipeline	High	Medium
core3	Proto-voice Harmony Parser	High	High
base1	Templating Algorithm	High	Low
algo1	Random Parse	High	Low
ext1	Heuristic Design	Medium	High
ext2	Heuristic Search	Low	Very High

### Risk Analysis

This project has a high general risk factor as it involves a **novel** approach to inferring harmony. Table 2.1 shows a list of project deliverables with associated priorities and risk, denoted qualitatively. The task with the greatest risk attached is designing and implementing the protovoice harmony parser, as this requires understanding and building on a complex existing codebase, and my proposed adaptation of the proto-voice parser has not been implemented before. Designing a bottom-up heuristic has a high risk factor as this is also a novel task, requiring creativity, research and iterative development. The baseline inference deliverable implements a standard chord templating method used commonly for ACE [42], posing minimal risk. Finally, the design and optimisation of efficient search algorithms is also a substantial task, which runs the risk of sinking a practically infinite amount of time.

In order to mitigate the risks posed by this project, the heuristic design and heuristic search tasks were set as extensions, so that only one high risk task was in the core part of this project.

## 2.6 Software Engineering Techniques

### 2.6.1 Development model

Based on the risk analysis (Table 2.1), a plan was created describing which modules to implement in which order, with a list of milestones on a 2 week basis. Notion was used to maintain a list of core tasks and corresponding subtasks with associated priorities, facilitating the selection of the next tasks to work on. The development strategy chosen drew from the Agile methodology, involving two-week long sprints with regular re-evaluations of the plan informed by experimental data and testing. GitHub’s continuous integration features were used to run a test suite on the repository after every commit.

#### Dataset selection

The datasets used contain bodies of work by a single composer as shown below, these were selected to exhibit a range of different styles. Each dataset contains an entire set work by a composer consisting of between 15 – 50 annotated scores, each with varying lengths. Each dataset was split into training, validation, and test sets, using a 60:20:20 split with *stratified sampling* in order to maintain a balanced representation of the different composers. In order to choose hyperparameters, the training and validation sets from each of the five datasets were combined into a single training pool, for use in cross-validation. For the final evaluation, the performance of all developed algorithms will be evaluated on the hold-out test set (20%). This is to provide an unbiased estimate of each algorithm’s performance on **unseen** data.

### 2.6.2 Languages, libraries and tools

Table 2.2 shows a justified list of the key languages, libraries and tools used in the project. The licensing agreements for all the tools used in the project were determined and analysed. For the most part, these are all permissive licenses, guaranteeing freedom to use, modify and redistribute as well as permitting proprietary derivative works.

The code was developed using Vim for Haskell and Visual Studio Code for Python notebook development, on my personal laptop (16’ MacBook Pro 2022, M1 Max, 32GB).

I used GitHub for all my notes, development and dissertation writing. Finally, this dissertation was written in Vim with VimTeX.

Table 2.2: Languages, libraries and tools

Tool	Purpose	Justification	License
<i>Languages</i>			
Haskell	Main language used for the core, baseline and extension implementations	Protovoice model implementation is in Haskell. Functional and amenable to parser development.	GHCL
Python	Secondary language for experiments and analysis	Powerful library ecosystem for running experiments and creating plots	PSFL
<i>Libraries</i>			
Musicology	Haskell Library with data-types for pitches	Contains a robust implementation of spelled pitch classes, which would be tedious to reimplement.	BSD-3.0
Timeit	Lighweight wrapper to show the used CPU time of a monadic computation	This is used to measure the runtime of the algorithms as part of analysis	BSD-3.0
Dimcat	Python library: Digital Musicology Corpus Analysis Toolkit	This library was written to work with the datasets used in this project	GPL-3.0
Numpy	Python library used for preprocessing and analysis	Powerful standard library that is used in conjunction with Seaborn to run analysis and visualise data	BSD-3.0
Pandas	Python library for preprocessing and analysis	This is a standard library for data manipulation and processing	BSD-3.0
Seaborn	Python data visualisation library used for analysis	Creates high quality graphs and charts	BSD-3.0
<i>Tools</i>			
Docker	Containerised software service used to run repeatable experiments	Protects code from breaking changes and allows code to be executed on different devices without manually installing dependencies	Free/Paid
Git	Version Control, Continuous Integration	Provides natural backups and allows for reverts to previous commits if necessary	GPL-3.0
GitHub	Hosting source code	Free, reliable hosting	GPL-3.0
GHC	Compiling and profiling.	This is the standard Haskell compiler.	BSD-3.0
Stack	Haskell building and testing	Creates reliable builds, and includes a powerful testing framework.	BSD-3.0
Undotree	Vim Plugin: stores all past actions as a tree	Solves the problem of linear undo history being lost. Protects code between commits.	BSD-3.0
MuseScore	Music notation software	The raw inputs are in the MusicXML format, which is used by MuseScore 3	GPL-3.0
PAT	Protovoice Annotation Tool, Used to view protovoice derivations on a web browser	The protovoice derivations are huge and very complex, so it's vital to have a viewing tool for use in analysis and iterative development	GPL-3.0

# Chapter 3

## Implementation

---

This chapter provides a high-level overview of the project structure (Section 3.1), followed by a detailed description of the key components of the implementation. First, the design and implementation of the heuristic fitness function are presented and justified (Section 3.2.1). Next, the implementation details of the harmony module is discussed, including the inferences used in the heuristic function (Section 3.3). This is followed by an exposition of the novel proto-voice harmony parser (Section 3.4), followed by discussion of the core algorithms implemented (Section 3.5), including an implementation of the template matching algorithm [42] as a standard ACE baseline. Then, a description of the heuristic search algorithms developed is given (Section 3.6.1). Finally, the test strategies used evaluate the effectiveness of the proposed algorithms are outlined (Section 3.7).

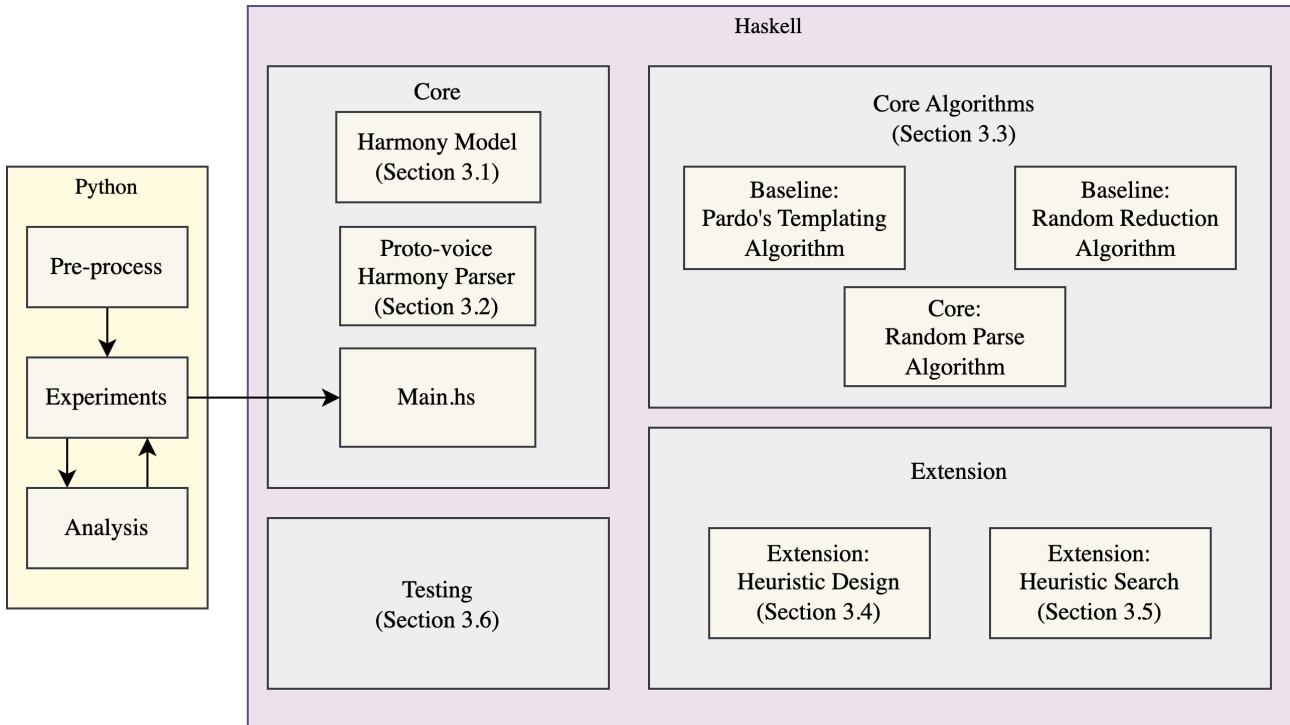


Figure 3.1: Diagram of project components

### 3.1 Repository Overview:

Table 3.1: Repository Overview

File/Folder	Description	LOC
protovoices-haskell/	Root directory	2272
├── src/		
│   ├── HarmonyParser.hs	Harmony Parser (Section 3.4)	470
│   ├── Harmony.hs	Harmony Model (Section 3.3)	121
│   │   ├── Harmony/		
│   │   │   ├── ChordLabel		
│   │   │   └── Profiles		
│   ├── Algorithm.hs	Core Algorithms (Section 3.5)	383
│   │   ├── Algorithm/		
│   │   │   ├── TemplateMatching.hs, RandomSample.hs, InformedReduction.hs		188
│   │   │   ├── RandomParse.hs	Core Algorithm (Section 3.5)	
│   │   │   └── HeuristicSearch/	Extension Algorithms (Section 3.6.1)	
│   │   │       ├── BestFirst.hs, Beam.hs, StochasticBeam.hs		
│   ├── Heuristics.hs	Heuristic Implementation (Section 3.2)	431
│   ├── FileHandling.hs	Utilities	
│   └── Probability.hs		115
│   ├── Common.hs	Existing code	
│   ├── GreedyParser.hs		
│   ├── PVGrammar.hs, PVGrammar/		
│   └── Display.hs		611
├── scripts/	Experiment Pipeline (Section 3.7)	
│   ├── preprocess.py		
│   ├── experiments.ipynb		
│   └── analysis.ipynb		
├── data/		
│   ├── inputs/		
│   └── outputs/		
├── app/		
│   └── MainFullParse.hs		
└── tests/	Unit and Integration Tests (Section 3.7)	

## Repository Justification

The repository has broadly been split into five main modules, as illustrated in Figure 3.1. Everything shown has been written during this project except for the files shown in blue.

- Firstly, the `HarmonyParser.hs` contains the first core contribution, an adaptation of the greedy parser (shown in blue) which integrates the protovoice model with the chord segment reduction process.
- Next, the `Harmony` module contains functions and data-types for dealing with chord labels, and profiles, as well as performing probabilistic inference.
- The `Algorithm` module provides a generic framework to run chord segment reduction algorithms. This module contains implementations of all the core algorithms, including Pardo and Birmingham’s template matching algorithm [42]. Within this, the `HeuristicSearch/..` algorithms are those that make use of the heuristic fitness function.
- `Heuristics.hs` contains the novel heuristic fitness function developed.
- The `experiments/` folder contains all the python code that is used for this project. Experiments were conducted by running the Haskell executable parameterised by the input piece and algorithm to use, and results are stored using `Json`. The experiments consist of three stages, as described by the three main files: `preprocess.py`, `experiments.py` and `analysis.py`. Splitting these stages up prevents wasteful computation, as all the pre-processing can be done just once, while experiments are run on the processed data iteratively alongside algorithm development.
- Finally, the `test/` folder contains unit tests for all of the Haskell modules, using the `Spec` testing framework which is used in Continuous Integration.

## 3.2 Proto-voice Fitness Function

To address the **exponential time** complexity of the naive parser and enable the efficient identification of musically meaningful proto-voice derivations, I propose a **novel fitness function** that evaluates a plausibility score for each derivation. The fitness function decomposes into a score for each reduction step and is used to guide the efficient heuristic search algorithm designed in Section ??, allowing **excellent solutions** to be found in **linear time**.

### 3.2.1 Motivation of Fitness Function

The proto-voice fitness function is motivated by defining a probability distribution for proto-voice derivations,  $P(\vec{d})$ , factored into each derivation step. Given a derivation  $\vec{d} = d_0 \dots d_N$  define:

$$P(\vec{d}) = \frac{1}{Z} \prod_i \phi(d_i) \quad (3.1)$$

Where  $Z$  is the normalisation constant, and  $\phi$  is a heuristic function that defines a relative probability for each reduction step, consisting of two factors corresponding to the generative proto-voice operations.

$$\phi(d_i) = \phi_{\text{parents}}(d_i) \cdot \phi_{\text{children}}(d_i) \quad (3.2)$$

The **parents** factor evaluates the relative plausibility of the parent notes chosen to be elaborated in the operation, and the **children** factor evaluates the relative plausibility of the child notes introduced from those parents.

Conditioning this distribution on the input sequence  $S$ , it follows from Bayes' Rule:

$$P(\vec{d}|S) = \frac{P(S|\vec{d}) \cdot P(\vec{d})}{P(S)} = P(S|\vec{d}) \frac{1}{P(S)} \frac{1}{Z} \prod_i \phi(d_i) \quad (3.3)$$

where  $P(S|\vec{d}) = \begin{cases} 1 & \text{if } \vec{d} \text{ produces } S \\ 0 & \text{otherwise} \end{cases}$

It is guaranteed (see Section 3.7) that any derivation  $\vec{d}$  found by the proto-voice harmony parser produces  $S$ , so  $P(S|\vec{d})$  is always 1. Computing the normalisation constant,  $\frac{1}{P(S) \cdot Z}$ , is intractable, but for the purpose of maximisation it can be ignored. It therefore suffices to compute  $P(\vec{d}) = \frac{1}{Z} \prod_i \phi(d_i)$ .

### Justification

It is informative to consider how a true generative probability distribution for proto-voice derivations would be structured. The probability of a derivation  $\vec{d}$  in the generative direction is naturally factored into each derivation step as follows:

$$p(\vec{d}) = \prod_i^N p(d_i | d_0, \dots, d_{i-1}) \quad (3.4)$$

Where  $d_0$  is the first generation step. A generative model would factorise this expression into conditionals that correspond to the generation steps, thus providing a guess of the distribution  $p(\vec{d})$ . Instead, the fitness function approximates this overall distribution based on local plausibility properties, corresponding to  $\phi$ . This approximation allows the fitness function to remain computationally tractable while capturing essential aspects of the true generative distribution.

### 3.2.2 Implementation of Fitness Function

In this section, Algorithm 1 is walked through step-by-step, discussing any relevant implementation details and justifying design decisions where appropriate. At a high level, the fitness function provides a score for partial derivations by partial application of each derivation step during the search.



---

**Algorithm 1** Proto-voice Fitness Function

---

**Require:** derivation, surface**Output:**

```

1: Initialise fitness score  $F \leftarrow 1$ 
2: for each reduction step  $d_i$  in derivation do
3:   if  $d_i$  is a freeze operation then
4:      $P(d_i) \leftarrow 1$ 
5:   else if  $d_i$  is a split or spread operation then
6:     Identify parent notes  $\vec{p}$  and child notes  $\vec{n}$ 
7:     Estimate chord label  $\hat{l}$  for each slice
8:     Evaluate parent and child note plausibilities
9:     Compute joint probability distribution
10:    Calculate plausibility score  $P(d_i)$ 
11:   end if
12:   Update fitness score:  $F \leftarrow F * P(d_i)$ 
13: end for
14: return  $F$ 

```

---

**Initialise Fitness Score**

The fitness score always begins with 1. Throughout the implementation log probabilities are used in order to improve performance and to avoid floating point errors that occur with very small values. As a result, line 12 is implemented by adding the log-plausibility score to the previous fitness score, rather than the multiplication as shown.

**Scoring each Derivation Step**

The high-level fitness function is decomposed into each reduction step by implementing the contents of the **for** loop in Algorithm 1 as a function `scoreOperation :: top -> op -> fitness`. This function takes the partially reduced surface, `top` and the operation applied, `op` and calculates the log plausibility score  $\log P(\text{op})$  which is subsequently added to the fitness score of the derivation up to that point.

**Scoring Freeze Operations**

The freeze operation  $t \rightarrow t'$  marks a transition as *terminal*. Freeze operations are given a plausibility score of 1 as they do not generate any child notes, meaning that there is no cost in fitness as a result. A full reduction will always result in the same number of freeze operations in the derivation as each transition can only be unfrozen once.

**Scoring Split Operations**

The plausibility of a split operation is given by the joint plausibility score of the child notes and parent notes. Recall that `Split` operations elaborate parent edges by introducing new child

notes that are within an interval of a *step* from one or two parent notes (Section 2.3.2). The plausibility scores for both parent and child notes are computed by assessing how well they match the appropriate chord-tone or ornament profiles of the slices they reside in. The joint probability distribution for single sided elaborations is  $P(\vec{p}, \vec{n})$  is factored as:

$$\begin{aligned} P(\vec{p}, \vec{n}) &= \sum_l P(\vec{n}|\vec{p}, l) \cdot P(\vec{p}|l) \cdot P(l) \\ &\approx P(\vec{n}|\vec{p}, \hat{l}) \cdot P(\vec{p}|\hat{l}) \end{aligned} \quad (3.5)$$

Where the distribution is approximated by replacing the marginalisation sum with the maximum-likelihood estimate of  $l$ , given by  $\hat{l}$ . This is motivated by modelling the generative process as follows. The parent slice has a latent chord label  $l \in \mathcal{L}$  which is not known. The parent notes  $\vec{p}$  are drawn based on this chord label, then the child notes  $\vec{n}$  are drawn based on the parent notes  $\vec{p}$  as well as the latent chord label  $\hat{l}$ .

The first step is to identify the parent notes elaborated  $\vec{p}$  and child notes  $\vec{n}$  introduced in the operation.

A `split` operation is implemented in the existing `PVGrammar.hs` as follows:

---

```

1  data Split n = SplitOp
2    { splitReg :: Map (Edge n) [(n, DoubleOrnament)]
3      -- ^ Maps every regular edge to a list of ornamentations.
4      , splitPass :: Map (InnerEdge n) [(n, PassingOrnament)]
5      -- ^ Maps every passing edge to a passing tone.
6      , fromLeft :: Map n [(n, RightOrnament)]
7      -- ^ Maps notes from the left parent slice to lists of ornamentations.
8      , fromRight :: Map n [(n, LeftOrnament)]
9      -- ^ Maps notes from the right parent slice to lists of ornamentations.
10     , keepLeft :: HashSet (Edge n)
11     -- ^ The set of regular edges to keep in the left child transition.
12     , keepRight :: HashSet (Edge n)
13     -- ^ The set of regular edges to keep in the right child transition.
14     , passLeft :: MultiSet (InnerEdge n)
15     -- ^ Contains the new passing edges introduced in the left child
16       transition
17     -- (excluding those passed down from the parent transition).
18     , passRight :: MultiSet (InnerEdge n)
19     -- ^ Contains the new passing edges introduced in the right child
20       transition
21     -- (excluding those passed down from the parent transition).
22   }
23   deriving (Eq, Ord, Generic, NFData)

```

---

Listing 3.1: Split Operation

In this code, an `Inner Edge` is a type synonym for a tuple of notes `(n,n)`, and an `Edge` is a type synonym for `(StartStop n, StartStop n)`, where `StartStop` is a container-type that augments the set of notes with the start ( $\bowtie$ ) and stop ( $\bowtie$ ) symbol. The left parents  $\vec{p}_l$  are identified using the union of the key-set of `fromLeft` and the notes on the left of the edges in the ket-sets of `splitReg` and `splitPass`. The right parents  $\vec{p}_r$  are identified analogously. The child notes  $\vec{n}$  are identified by taking the union of first element of each tuple in the value of the four maps, `splitReg`, `splitPass`, `fromLeft` and `fromRight`.

The second step is to identify the most likely chord label for each involved slice. This is precomputed and stored within an container-type for slices that augments slices with the MAP estimate of the chord label  $\hat{l}$ . The implementation details for this inference is discussed in section on the Harmony Module (Algorithm 2), and the optimisation is discussed in (Section 3.3.3).

The third step is to evaluate the plausibility score of the parents,  $P(\vec{p}|\hat{L})$ , which models the probability of the parent notes being expressed as chord-tones of the guessed chord label  $\hat{L}$ . This is computed by computing the multinomial probability density function at the value given by the parent slice vector, parametrised by the chord-tone profile vector of the guessed chord label. The functions for finding the parent slice vectors, chord-tone profile vectors and the multinomial pdf are all discussed in the Harmony Module (Section 3.3).

The fourth step is to evaluate the plausibility score of the children,  $P(\vec{n}|\vec{p}, \hat{L})$ , found by evaluating the multinomial probability density function at the value given by the note slice vector, parametrised by the chord-tone or ornament profile vector of the guessed chord label, depending on the type of elaboration. **Repetition** notes are evaluated as being generated from the chord-tone profile corresponding to the parent slice, and **Neighbour** notes are evaluated as being generated from the ornament profile. For *double-sided* elaborations such as **repeat-after**, the child notes are evaluated using a *mixture* model of the corresponding profiles for both parents. The implementation of this evaluation is discussed in the next section (Section 3.3)

### Scoring spread operations

Recall the **spread** rule :

$$t_l s_r \rightarrow t'_l s'_l t'_m s'_r t'_r$$

**Spread** operations include elaborations that introduce child notes which are scored in the same fashion as **split** operations. The spread operation has the music theoretical function of *prolonging* the parent slice, that is, the notes in the child slices are both subsets of the parent slice, exemplifying the same chord label. As a result, the regular edges introduced in a spread operation are all repetitions.

## 3.3 Harmony Module

The harmony module implementation is responsible for conducting inferences based on pitches and chord labels, and consists of two key components:

- **Harmony.hs**: Data structures and algorithms for representing and manipulating pitches, pitch classes, and chord labels.
- **ChordLabel.hs**, **ChordProfile.hs**: Functions for computing slice vectors from input data. Functions for rotating and aligning probabilistic chord profiles for given chord labels.

### 3.3.1 Chord Labels

Chord Labels are represented as a product of chord-type and root-note,  $\mathcal{L} = \mathcal{C} \times \mathcal{P}$ . Chord-types are represented at the type level as a sum-type of all 14 chord-types with the addition of `NoChord`, which is used to represent regions of music with no chord label. The root-note is stored as an `SPC` (Spelled Pitch Class), from the `haskell-musicology` library.

### 3.3.2 Inferring Harmony

The function `mostLikelyLabelGivenSlice :: Slice SPitch -> ChordLabel` is implemented to find the most probable chord label  $l$  given a slice  $s$ , shown in Algorithm 2.

---

**Algorithm 2** Inferring Harmony

---

**Require:** Slice  $s$ , set of chord labels  $\mathcal{L}$ , probabilistic chord-tone profiles  $p_{\text{chord-tone}}$

**Output:** Most probable chord label  $\hat{l}$

- 1: **for** each chord label  $l \in \mathcal{L}$  **do**
  - 2:     Translate the probabilistic chord profiles for  $l$  to align with the root note
  - 3:     Compute the log probability of the slice given the rotated profiles
  - 4:     Add the log probability for the chord-type of  $l$
  - 5:     Store the log probability and label in a list
  - 6: **end for**
  - 7: Find the label with the highest log probability in the list
  - 8: **return** the most probable label  $\hat{l}$
- 

#### Aligning chord profiles

The first step is to translate the profiles of the label  $l = (c, r)$  to align with the root-note. This is achieved by implementing the function `pChordtones :: ChordLabel -> Vector Double` which calls `translateVector :: Int -> Vector Double -> Vector Double` to translate the chord-tone profile  $r_{\text{fifths}}$  places to the left, aligning the chord-profile with the correct root-note,  $r$ , on the line of fifths.

#### Chord-type Probabilities

This module exports the probability vector `pChordType` that contains the relative probabilities  $p(c)$  for each of the 14 chord-types. I make use of a study of observed chord-types [14] in order to define  $p_{\text{chord-type}}$ , where  $p_{\text{chord-type}}^{(c)}$  gives the relative probability of the chord-type  $c$ . Chord labels are modelled as i.i.d categorical variables, so  $p_{\text{chord-type}}^{(c)}$  is found by taking the MLE of the posterior distribution for chord labels found the study.

### Maximising Label Probability

The function `sliceVector :: Slice ns -> Vector Double` is implemented to generate the 29-wide vector comprising counts of each pitch-class in the given slice. We model the generation of a slice  $s$  from the chord-tone profile of  $l$ . Each chord label  $l$  is characterised by a chord-tone profile,  $\mathbf{p}_{\text{chord-tone}}^{(l)}$ , a vector of probabilities for each pitch-class. Given a label  $l$ , the slice  $s$  is generated by drawing a set of pitches according to  $\mathbf{p}_{\text{chord-tone}}^{(l)}$  and forgetting their order. For a slice  $s$  containing  $n$  pitch instances, this defines a *multinomial* distribution:

$$s \sim \text{Multinomial}(n, |\mathcal{P}|, \mathbf{p}_{\text{chord-tone}}^{(l)}) \quad (3.6)$$

where  $|\mathcal{P}|$  is the number of pitch-classes.

In order to find the best-fitting chord label, the label  $l = (c, r)$  is chosen to maximise the conditional probability  $P(l|s)$ :

$$\begin{aligned} \hat{l} &= \arg \max_l P(l|s) \\ &= \arg \max_l \underbrace{\frac{1}{P(s)}}_{\text{constant}} P(s|l) \cdot P(l) && (\text{Bayes' Rule}) \\ &= \arg \max_l P(s|l) \cdot P(l) \\ &= \arg \max_l f(\mathbf{v}(s); \mathbf{p}_{\text{chord-tone}}^{(l)}) \cdot \frac{1}{Z} p_{\text{chord-type}}^{(c)} \end{aligned} \quad (3.7)$$

where  $f(\mathbf{x}; \mathbf{p})$  computes the multinomial probability density of the vector  $\mathbf{x}$  parameterised by the probability vector  $\mathbf{p}$ , given by:

$$f(x_1, \dots, x_n; p_1, \dots, p_n) = \frac{\Gamma(\sum_i x_i + 1)}{\prod_i \Gamma(x_i + 1)} \prod_{i=1}^n p_i^{x_i} \quad (3.8)$$

Note that  $s$  is fixed within the context of the maximisation so  $\frac{1}{P(s)}$  is a constant that can be ignored. To mitigate floating point errors, it is convenient to instead maximise the log of the conditional probability,  $\log P(l|s)$ , which is equivalent to maximising  $P(l|s)$  as the logarithm is a monotonically increasing function of its argument.

#### 3.3.3 Optimisations

A number of optimisations are implemented in order to avoid redundant computation of probability density functions.

#### Slice Wrapper

The data-type for a `Slice` is augmented by implementing a wrapper data-type, `SliceWrapper` which stores the MAP estimate for latent chord label,  $\hat{l}$ , along with the conditional probability

$P(\hat{l}|s)$ . This provides a substantial reduction (by a constant factor) of the computation required to compute the heuristic.

The original `slice` data-type and the wrapper `SliceWrapped` are shown below:

---

```

1      newtype Slice ns = Slice
2          { sContent :: ns }
3
4      data SliceWrapped ns = SliceWrapped
5          { sWContent :: ns
6            , sLbl      :: ChordLabel
7            , sLblProb  :: Double }

```

---

Listing 3.2: Slice Wrapper

This allows `SliceWrapped` to be used in place of the original `Slice`. A new type, `SliceWrapper` allows the inference of the additional data to be parameterised through the function `wrapSlice :: Slice ns -> SliceWrapped ns`. Pulling the MAP chord label estimate to the type level guarantees that the MAP estimate is always and only computed when a slice is created or modified.

## 3.4 The Proto-Voice Harmony Parser

The original proto-voice parser finds full derivations of a given surface from the empty piece, but this cannot be used to infer harmony as the chord segment boundaries are not conserved. The harmony parser is a new parser for the protovoice model that preserves segment boundaries such that the parse completes with a reduction that consists of a single slice per chord segment.

This section first describes the design of the `HarmonyParser` before discussing relevant implementation details.

### 3.4.1 Overview of Harmony Parser Design

Formally, a derivation  $D$  is defined as a pair  $(top, ops)$ , where the *surface* is derived by starting with the fully reduced *top* and applying each operation in *ops* in *left-most derivation order*.

It is informative to consider the *generation order* and *parse order* of a proto-voice derivation. In generation order, we begin with the reduced surface *top*, consisting of only chord-tones, with a pointer at the left-most transition and apply each `split` or `spread` operation to the two left-most non-terminal transitions in the path graph, generating new slices and transitions. The `freeze` operation marks a transition as terminal thus shifts the pointer to the right. Operations are applied until the pointer is right-most, and all transitions are terminal, resulting in the fully elaborated *surface* and a derivation  $D = (top, ops)$ , where *ops* is the sequence of operations applied,  $ops = [d_N \dots d_0]$ , resulting in the  $surface = d_0 \circ \dots \circ d_N(top)$ .

Reduction is the inverse of generation: the parse begins with the elaborated surface consisting of only frozen transitions with a pointer at the right-most frozen transition. During the parse, we can either `unfreeze` the transition at the pointer, shifting the pointer to the left, or apply

a `unsplit` or `unspread` reduction to the two transitions to the right of the pointer. Once the pointer is left-most, all transitions are unfrozen and there is a *single reduced slice* for each chord segment at the top of the derivation, the resulting derivation is  $D = (top, ops)$ , where the reduced surface is  $top = (d_0^{-1} \circ \dots \circ d_N^{-1})(surface)$ .

This is similar to a *shift-reduce* parser, making one pass across the surface, right to left. Shift-reduce parsers are normally used for *deterministic grammars*, where at most one parse exists. As the proto-voice grammar is non-deterministic, the `HarmonyParser` has to deal with ambiguity in addition.

Finally, the `HarmonyParser` needs to conserve segment boundaries throughout the reduction it finds. In order to achieve this, the parser needs to keep track of where the boundaries are, disallow specific reductions that lead to dead states, and propagate boundary information throughout the reduction.

There are many potential ways to parse according to the proto-voice grammar, representing states in a different manner to as described above. For example, the proto-voice reduction rules could be applied to any pair of open transitions rather than those nearest to the parse pointer, which would allow the exploitation of long-range dependencies. I choose not to pursue this at this point due to the associated combinatorial explosion, and the benefits of being able to represent a proto-voice derivation as a sequence of left-most reductions, but this could be a topic for further research.

### 3.4.2 Parse States

The state of parse is represented by a sum-type of three sets of states that distinguish between the position of the parse pointer. The first type of state, `SSFrozen`, describes when the pointer is right-most, and all transitions are frozen. The second set of states is `SSSemiOpen`, which describes all the states where the pointer is neither left-most or right-most. In a `SSSemiOpen` state, some transitions are frozen, and some are open. The final edge case is the `SSOpen`, describing states where the pointer is left-most, and all transitions have been unfrozen. This idea of how to describe states is borrowed from Finkensiep’s implementation of the `GreedyParser` in the original `protovoices-haskell` repository.

---

```

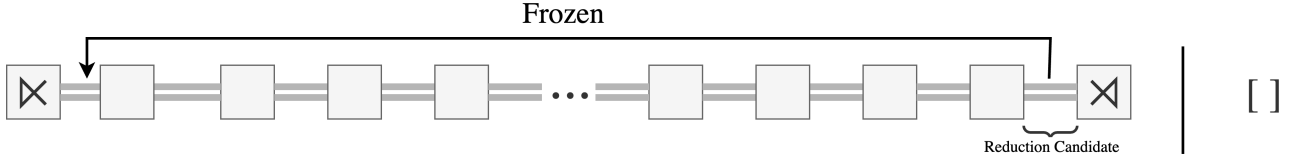
1  data ParseState es es' ns o
2    = psFrozen !(Path (Maybe es', Bool) (Slice ns))
3    -- ^ Beginning of parse - all frozen transitions
4    | psSemiOpen -- <- Mid-parse
5      { _psFrozen :: !(Path (Maybe es', Bool) (Slice ns))
6        -- ^ frozen transitions and slices from current point leftward
7        , _psMidSlice :: !(SliceWrapped ns)
8        -- ^ the slice at the current pointer
9        , _psOpen :: !(Path (Trans es) (SliceWrapped ns))
10       -- ^ non-frozen transitions and slices from current point rightward
11       , _psDeriv :: ![o]
12       -- ^ derivation from current reduction to original surface
13     }
14    | psOpen !(Path (Trans es) (SliceWrapped ns)) ![o]
15    -- ^ Final Parse States - path with unfrozen transitions, slices and
      derivation

```

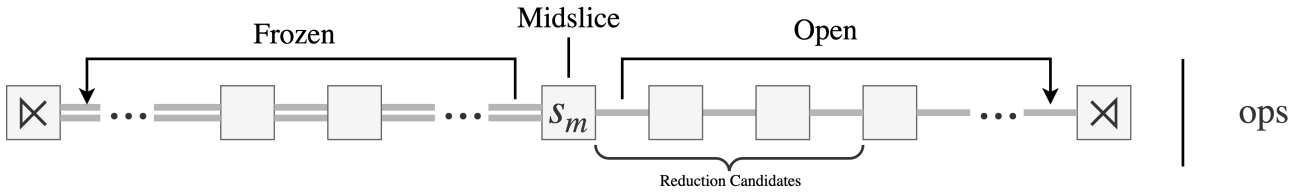
---

Listing 3.3: Harmony Parser States

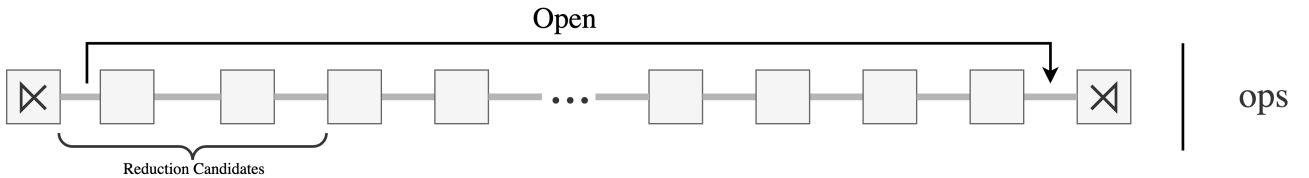
In the initial parse state, **pSFrozen**, the surface is represented as a **path** from the end to the beginning of the piece, with the right-most transition at its head. The start( $\times$ ) and stop( $\times$ ) symbols are not explicitly stored in the **path**, but are implied. All transitions are initialised as frozen, indicated by the double lines in Figure 3.2. The **path** begins on the right. In this initial state the only option is to unfreeze the rightmost transition, moving to the **pSSemiOpen** state.

Figure 3.2: **pSFrozen**

The **pSSemiOpen** represents the majority of the parse. The pointer is represented by **midSlice**, and points to the rightmost frozen transition. The **open** path contains all unfrozen transitions (denoted by a single line) from the right of the pointer to the end of the piece, and the **frozen** path contains all frozen transitions from the pointer to the start of the piece.

Figure 3.3: **pSSemiOpen**

In the final state, **pSOpen**, the pointer is at the beginning of the piece, and comprises a single path **open** containing only unfrozen transitions from the beginning to the end of the piece.

Figure 3.4: **pSOpen**

Each state also stores a list of operations, **ops** that when applied to the current reduction, results in the original surface. This is empty at the beginning of the parse; for each reduction applied, the corresponding generative operation **op** is consed to the list: **ops' := op : ops**.

These parse states define a **directed acyclic graph** where the parse states are nodes and the reductions are edges. Traversing this graph results in either reach a valid derivation or dead state.

### 3.4.3 Parsing Operations

There is a separation between parsing the outer structure and inner structure. This allows different kinds of inner structure to be parsed while the outer structure stays the same. The outer structure parsing is conducted in the **HarmonyParser**, while the inner structure parsing



is encapsulated in the *evaluators*. The `protoVoiceEvaluator` [15] provides methods that are used to enumerate the possible inner operations for each reduction type <sup>1</sup>.

$$\begin{aligned}
 \text{evalUnfreeze} : & \quad t \rightarrow [(t', op)] \\
 \text{evalUnsplit} : & \quad (t_l, s_l, t_m) \rightarrow [(t_{top}, op)] \\
 \text{unSpreadLeft} : & \quad (s_l, t_l) \rightarrow s_{top} \rightarrow [t_{topl}] \\
 \text{unSpreadRight} : & \quad (s_r, t_r) \rightarrow s_{top} \rightarrow [t_{topr}] \\
 \text{unSpreadMiddle} : & \quad (s_l, t_{top}, s_r) \rightarrow \text{Maybe } (s_{top}, op)
 \end{aligned} \tag{3.9}$$

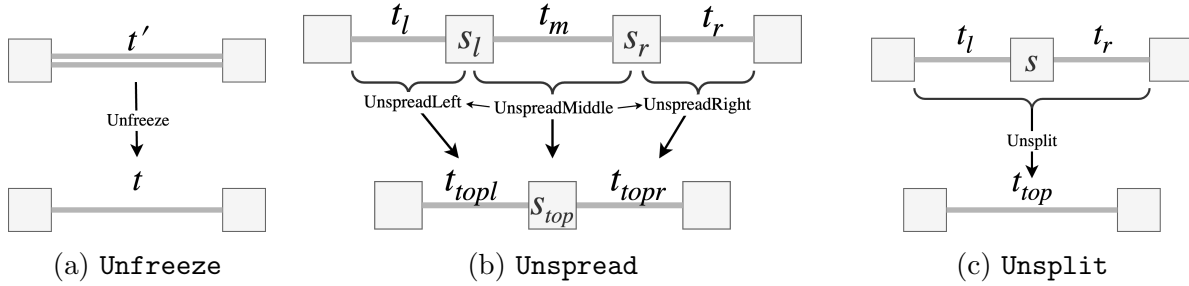


Figure 3.5: Reduction operations.

The `evalUnfreeze` and `evalUnsplit` functions allow the associated generative operation, `freeze` and `split` to be determined trivially, but the possible `spread` operations need to be derived using `unSpreadLeft`, `unSpreadRight`, and `unSpreadMiddle`. This is achieved by considering all combinations of left parent transitions and right parent transitions if the middle operation exists.

### 3.4.4 Conserving Segment Boundaries

Recall that the goal of the parse is to reduce the original surface such that there is one slice per segment. In order to achieve this, constraints are imposed on the reduction operations dependent on adjacent segment boundaries. The boundary values are integrated into the `HarmonyParser` by augmenting the transition data-type, `Trans` with a boolean tag indicating whether or not the transition crosses a boundary.

---

```

1  data Trans es = Trans
2    { tContent :: !es
3      -- ^ The value inside the transition, i.e. protovoice edges
4      , t2nd    :: !Bool
5      -- ^ Marks if this transition is the 2nd right parent of a spread
6      , tBoundary :: !Bool
7      -- ^ Marks whether or not this transition is at a boundary between two
          segments
8    }

```

---

Listing 3.4: Transition Data-type

<sup>1</sup>Note that the notation used here for functions combines type declarations with variable names. For example, the function `evalUnsplit` has type `evalUnsplit :: (transition, slice, transition) → [(transition, operation)]`, but type names (e.g. *transition*) have been substituted by variable names (e.g.  $t_l, t_r, t_{top}$ ) for expository purposes.

Let  $B_t : t \rightarrow \{\mathbf{True}, \mathbf{False}\}$ , such that  $B_f$ ,  $B_l$ ,  $B_m$ , and  $B_r$  denote the boundary values of  $t_f$ ,  $t_l$ ,  $t_m$  and  $t_r$  respectively, as shown in Figure 3.6. As an **unspread** operation ( $t_l s_l t_m s_r t_r \rightarrow t_{topl} s_{top} t_{topr}$ ) merges two slices  $s_l$  and  $s_r$ , removing  $t_m$ , the constraint  $\neg B_m$  is imposed to prevent two segments from merging. Similarly, an **unsplit** operation (eg.  $t_l s_l t_m \rightarrow t_{top}$ ) combines two transitions, thus we impose the constraint  $\neg(B_l \wedge B_m)$ . Finally, the **unfreeze** operation shifts the context to the left. If a boundary transition is unfrozen, no more reduction operations can be applied to its right, hence it is a necessary condition that the segment has been fully reduced. Thus the constraint imposed on an **unfreeze** operation is  $\neg B_f \vee (B_l \wedge B_m \wedge B_r)$ . Figure 3.6 provides an illustration of these constraints. The boundary must also be propagated to parent transitions following a reduction, defined by the union of each child transitions' boundary values for a given reduction.

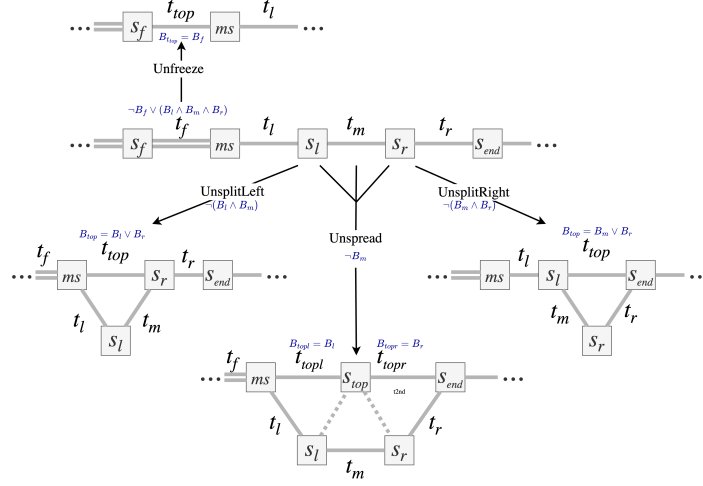


Figure 3.6: Boundary reduction conditions and propagation

### 3.5 Core Algorithms

All algorithms implemented in this project are encapsulated within the **Algorithm** module. The **ParseAlgo** type-class provides an interface for running these algorithms. The result is wrapped in a **Maybe** to describe the case when no full reduction is found, and is in the **IO** monad to allow randomness.

The input is implemented as a sum-type, **AlgoInput**, which is either pure or impure, where the **PureEval** is a evaluator for inner proto-voice operations that returns the list of all possible inner operations for each type of reduction. The **ImpureEval** is an adapted evaluator that instead samples from the possible operations without enumerating them, allowing an exponential to linear reduction in time-complexity with respect to the size of each slice.

The output is implemented as **AlgoResult**, a sum-type of **BlackBoxResult**, **ReductionResult** or **PVResult**. The **BlackBoxResult** contains just the inferred chord labels with no explanation, and is used to represent the output of the baseline Templating algorithm. The **ReductionAnalysis** contains a reduced surface represented as a list of slices. The **ProtoVoiceAnalysis** comprises a pair (**top**, **ops**) where **top** represents the reduced surface as a path from the beginning of the piece to the end, and **ops** is the sequence of derivation steps in generation order.

---

```

1  data AlgoType
2    = Templating
3    | InformedReduction
4    | RandomReduction
5    | RandomParse
6    ...
7    | StochasticBeamSearch
8    deriving (Show, Read, Eq)
9
10 class ParseAlgo algo where
11   runParse :: algo -> AlgoInput -> IO (Maybe AlgoResult)
12
13 data AlgoInput
14   = AlgoInputPure    PureEval    [InputSlice] [ChordLabel]
15   | AlgoInputImpure  ImpureEval  [InputSlice] [ChordLabel]
16
17 data AlgoResult
18   = BlackBoxResult    [ChordLabel]
19   | ReductionResult   ReductionAnalysis
20   | PVResult          ProtoVoiceAnalysis
21   deriving (Show)

```

---

Listing 3.5: Algorithm Module Overview

**Template Matching:** As a baseline for evaluating the efficacy of the reduction process, I implemented *Pardo and Birmingham’s* Templating algorithm [42], a standard ACE algorithm for predicting chord labels from sequences of slices. This algorithm is parameterised by *templates*, which are equivalent to binary chord-tone profiles. The templates used correspond to the chord-tone profiles used in the Harmony model and the heuristic search algorithms, allowing to a fair comparison to be made against the templating baseline. The algorithm works by choosing the chord label with the highest score for each chord segment, given by the formula  $S = P - (M - N)$ . The Positive Evidence,  $P$ , is the number of notes in the segment whose pitch class matches a template element. The Negative Evidence,  $N$ , is the number of notes in the segment whose pitch class does not match any template element. The Misses,  $M$ , is the number of template elements which are not matched by any note in the segment. Finally a sequence of rules are used to break ties when multiple labels are assigned the same score.

**Informed Reduction:** The Informed Reduction algorithm reduces each segment by using removing non-chord-tones according to the ground-truth probabilistic chord profiles, corresponding to an idealised reduction. The purpose of this algorithm is to test the efficacy of the chord segment reduction process and the Harmony Model by showing how well chords can be inferred given a ‘perfect’ reduction of the piece.

**Random Reduction:** The Random Reduction algorithm services as a baseline for reductions that make use of proto-voice derivations. This algorithm works by combining all notes in each segment into a single slice, and random removing notes until there are  $n$  notes in each segment, where  $n$  is sampled from a Poisson distribution,  $n \sim \text{Poisson}(\lambda)$ . The parameter  $\lambda$  is taken from the empirical study *Chord Types and Ornamentation* [14].

**Random Parse:** The first core algorithm is the **Random Parse** algorithm which reduces the surface via a proto-voice derivation by randomly traversing the graph of parse states. The proto-voice model does not make judgements based on harmony or note function explicitly, but by comparing this algorithm to the other baselines, it can be determined whether the constraints

enforced by using a proto-voice derivation has any effect on the quality of the reduction.

## 3.6 Extension: Heuristic Search Algorithms

All the core code described before this section was implemented in the early stages of this project. The bulk of the time since involved an iterative process of refining the heuristic fitness function, reading research papers and experimenting with different heuristic search algorithms.

### 3.6.1 Stochastic Beam Search

The final iteration of my heuristic search experimentation is the **Stochastic Beam Search** algorithm. This algorithm achieves a reduction of time complexity from the original proto-voice Chart Parser from  $\mathcal{O}(2^{n \cdot m})$  to  $\mathcal{O}(\beta \cdot \gamma \cdot n \cdot m)$ , where  $n$  is the number of surface slices,  $m$  is maximal number of notes in each slice, and  $\beta, \gamma$  are parameters. By making use of pruning techniques and the fitness function developed in Section 3.2.2, I show a qualitative (Section 4.2.2) and a statistically significant quantitative improvement in chord labelling accuracy over the Random Parse for a variety of pieces (Section 4.2).

This algorithm was developed by necessity, **solving three key problems**. First is that the fitness function used naively with a greedy algorithm would result in **local optima**, with limited improvement over the random search. Interestingly, the greedy search resulted in an increased chance of reaching a dead state, leading to an incomplete parse. This problem was mitigated by using a beam search, which maintains a bounded number of best search states evaluated using the fitness function.

The second problem was that of **diversity** in the types of reductions performed. The fitness function approximates the relative plausibilities of each operation independently but their fitnesses cannot be compared against each-other directly. This problem was solved taking up to  $\beta$  successors of each reduction type, shown in Line 6, as well as adjusting the relative operation weightings of the fitness function to align **spreads** and **splits** more closely. Future work could include developing a more sophisticated *diversity metric* that as added to the fitness function

The third was the biggest roadblock I ran into in this project which was the **combinatorial blowup** that arose due to the number of possible **unsplit** reductions at each step. For slices of size  $m$ , the number of **freeze** operations is  $\mathcal{O}(1)$ , the number of **unspread** operations is  $\mathcal{O}(m)$ , but the number of **unsplit** operations is  $\mathcal{O}(2^m)$ , as it is defined combinatorially, choosing any configuration of  $\mathcal{O}(m^3)$  possible elaborations, as each elaboration requires a choice of two parent notes and a child note.

In order to implement this algorithm a sizeable change had to be made to the existing `protoVoiceEvaluator`. The existing function `evalUnsplit` takes a slice and two neighbouring transitions  $(t_l, s, t_r)$  and returns a list of all possible parent transitions, with the corresponding **split** operation,  $[(t_{top}, op)]$ . This is all done using the *list monad*, which is analogous to the workings of non-deterministic finite automaton. Within the list monad, each action is chosen from a list "non-deterministically", so that in the end all combinations have been considered. In order to adapt this code I created a new monad, the `RandomChoice` monad, which works

in a similar way to the list monad, but instead samples from the possible actions at each decision point instead. This was implemented in a new `Probability` module and allowed the `evalUnsplitImpure` to be written by adapting the existing `evalUnsplit` with minimal changes. This has the potential to be a useful Haskell module for probabilistic inference in other contexts.

---

**Algorithm 3** Stochastic Beam Search
 

---

```

1: hyper-parameters:  $\beta \leftarrow \text{BeamWidth}, \gamma \leftarrow \text{SampleSize}$ 
2: initialise:  $\text{open} \leftarrow \{\text{initialState}\}, \text{costs} \leftarrow \{\text{initialState} : 0\}$   $\triangleright |\text{open}| = 1$ 
3: while  $\text{open}$  does not contain any goal states do  $\triangleright$  Loop runs  $2n - k + 1$  times
4:    $\text{freezes} \leftarrow \bigcup_{s \in \text{open}} \text{UNFREEZES}(s)$   $\triangleright |\text{freezes}| \leq \beta$ 
5:    $\text{spreads} \leftarrow \bigcup_{s \in \text{open}} \text{UNSPREADS}(s)$   $\triangleright |\text{spreads}| \leq \beta$ 
6:    $\text{splits} \leftarrow \bigcup_{s \in \text{open}} \text{SAMPLEUNSPLITS}(s, \gamma)$   $\triangleright |\text{splits}| \leq \gamma \cdot \beta$ 
7:   Compute heuristic cost for each state in  $\text{freezes}$ 
8:   Compute heuristic cost for each state in  $\text{splits}$ 
9:   Compute heuristic cost for each state in  $\text{spreads}$ 
10:   $\text{open}, \text{costs} \leftarrow \bigcup_{\text{freezes}, \text{spreads}, \text{splits}} \text{Best } \beta \text{ states and costs.}$   $\triangleright |\text{open}| \leq 3\beta$ 
11: end while
12: return Best state in open

```

---

### Complexity Analysis

Finally, I provide a proof of algorithmic complexity for the Stochastic Beam Search algorithm by carefully analysing the algorithm line-by-line, proving that this algorithm improves on the complexity of the parser in the original paper from  $\mathcal{O}(2^{n \cdot m})$  where  $n$  is the number of slices, and  $m$  is the maximal number of notes in each slice, to  $\mathcal{O}(\beta \cdot \gamma \cdot n \cdot m)$ , where  $\beta$  is the beam-width and  $\gamma$  is the sample-size. Let  $n$  be the number of slices in the input, and  $k$  the number of chord segments, where  $k \leq n$ ,  $k, n \in \mathbb{N}^+$ . Let  $m \in \mathbb{N}^+$  be the maximal number of notes in the input slices.

I first attest that  $|\text{open}| \leq 3 \cdot \beta$  throughout the execution of the algorithm. During the first iteration,  $|\text{open}| = 1$  as it is initialised to contain only *initailState* on Line 2. The set *open* is only updated at the end of the loop, on line 10, where *open* is reassigned to the union of the best  $\beta$  states from three sets of states, which is strictly less than or equal to  $3\beta$ .

Next, I show that the number of computations within each iteration of the while loop in Line 3 is  $\mathcal{O}(m \cdot \beta \cdot \gamma)$ , then show that the number of loop iterations is  $\mathcal{O}(n)$ , thus proving that the complexity of the algorithm is  $\mathcal{O}(\beta \cdot \gamma \cdot n \cdot m)$ .

Lines 4 and 5 each compute the union of all the *unfreeze* and *unspread* successors of the states in *open*, respectively. There are zero or one unfreeze successors for each state, and each successor is computed in constant time so  $|\text{freezes}| \leq \beta$  and Line 4 requires  $\mathcal{O}(\beta)$  computation steps. There are also zero or one unspread successors for each state, but each successor is computed by computing the maximal slice of the two child slices. While  $|\text{spreads}| \leq 3\beta$ , computing *spreads* requires  $\mathcal{O}(\beta \cdot m)$  computation steps. Finally Line 6 computes the union of samples of all the *unsplit* successors of the states in *open*. Each sample consists of making a random choice

between  $\mathcal{O}(m)$  options, a constant number of times. Given that  $\gamma$  operations are sampled for each state in *open*, Line 6 involves  $\mathcal{O}(\gamma \cdot \beta \cdot m)$  computation steps.

Next, Lines 7,8 and 9 compute the heuristic costs for freezes, splits and spreads respectively. The computation of the heuristic involves up to three evaluations of the multinomial probability density function, which scales linearly in  $m$ . For unSpreads the algorithm needs to perform inference of the latent chord label for the new parent slice. This involves a multinomial evaluation for each of a finite vocabulary (120) of chords, which is also linear in  $m$ . The number of computation steps in lines 7 to 9 is  $\mathcal{O}(\gamma \cdot \beta \cdot m)$

Line 10 involves finding the best  $\beta$  states for each of the three sets of states. This was achieved by writing an efficient function `minElems` which finds the best  $\beta$  states in one pass, resulting in  $\mathcal{O}(\beta^2)$  computation steps.

Each iteration of the loop on line 3 increments the number of derivation steps stored each state in *open* by exactly 1. It follows that the number of iterations of the loop on line 3 in the worst case is defined by the maximal number of derivation steps in a full chord segment reduction, as this the definition of a goal state. Consider a single path through the search tree. Each step through the search tree comprises either an unfreeze, unsplit or unspread operation. At the beginning of the parse, all transitions are marked as frozen, and for the parse to complete each transition is unfrozen once and only once. As no transition is reduced before it is unfrozen, and there is one more transition than there are slices, the number of unfreeze operations is  $n + 1$ . Secondly, both unsplit and unspread only reduce unfrozen transitions, and each decrease the number of slices in the reduced surface by 1. As the `Harmony Parser` ensures that the parse results in exactly one slice per segment, the number of unsplit and unspread operations in one complete reduction is equal to the number of excess slices,  $n - k$ . Therefore the number of iterations of the while loop on line 3 is  $2n - k + 1 = \mathcal{O}(n)$ .

It then follows that as the complexity each iteration of the while loop is  $\mathcal{O}(\gamma \cdot \beta \cdot m)$  and the number of iterations is  $\mathcal{O}(n)$ , the complexity of the `StochasticBeamSearch` algorithm is  $\mathcal{O}(\gamma \cdot \beta \cdot n \cdot m)$ , linear in both input dimensions and both parameters.

## 3.7 Testing

Continuous integration was used throughout this project. Tests were written using the `HSpec` testing framework for each module. This proved useful as it ensures that changes in the code base didn't have any unexpected consequences. The `Harmony.spec` test file was particularly useful as it contains a large number of potential slices and expected chord label outputs, which facilitated an effective iterative test-driven development cycle; more complicated examples were added to the test specification as the harmony model became more sophisticated.

# Chapter 4

## Evaluation

---

This chapter justifies all claims made in the introduction of this project by describing quantitative and qualitative evaluations of the work, demonstrating that the core and extension Success Criteria were met. This includes both the evaluation of the music theoretical assumptions behind the fitness function and harmony module, as well as the performance, scalability and interpretability of the developed algorithms.

Table 4.1: Evaluation Overview

Section	Deliverable	Axes of Evaluation
Section 4.1	Harmony Module	Accuracy
Section 4.2	Random Parse, Harmony Parser	Accuracy, Scalability
Section ??	Heuristic Search, Fitness function	Interpretability, Accuracy, Scalability

### Evaluation Metrics

Two numeric metrics are used to evaluate label classification performance. Recall that a chord label is described as a tuple of root-note and chord-type. **Label accuracy** is the proportion of completely correct chord label predictions, and **root accuracy** is the proportion of correct root-note predictions, regardless of the chord-type. Other metrics used include chord-label and root-note **recall** for each chord-type, used to evaluate how well different chord-types are classified, and **running time**, which is used to empirically evaluate time complexity of the search algorithms. Finally, **qualitative** judgements are made to evaluate the interpretability of the proto-voice fitness function.

### Evaluation Pipeline

Early in the implementation a full end-to-end pipeline from piece to chord label predictions was developed. This consists of the scripts in the `scripts/` folder, `preprocess.py`,

`runExperiment.py` and `analysis.ipynb`. The repository was then containerised using *Docker*, so that experiments could be run and analysed using a remote server. Experiments were executed on the server, initialised via *ssh*, and *tmux* was used to maintain the server environment across connections, with automated scripts to pull the latest changes from GitHub. A *jupyterlab* environment was set up on the server, and scripts were written to fetch the results of the latest experiments and produce plots and tables. Algorithms were first run on my laptop, then later run on a server provided by the EPFL Digital Cognitive Musicology Lab (Dell PowerEdge R740XD Server, 2x Xeon Gold 625R, 768GB).

## 4.1 Harmony Model

**Question 1:** Does the Harmony Model effectively infer chord labels from multi-sets of corresponding chord-tones?

The harmony model is shown to effectively infer chord labels by evaluating the upper bound on prediction accuracy (given an oracle heuristic), and comparing this *Pardo and Birmingham's* Templating algorithm [42], a standard ACE algorithm for predicting chord labels from sequences of slices.

This is achieved by running the Harmony Model on the reduction produced by the **Informed Reduction** algorithm. This algorithm reduces each segment by using removing non-chord-tones according to the ground-truth probabilistic chord profiles, corresponding to an idealised reduction.

### Discussion of Results

Table 4.2: Harmony Model vs Templating Baseline

Composer	Algorithm	Label Accuracy	Root Accuracy
Chopin	Informed Reduction	<b>0.89</b>	<b>0.96</b>
	Templating	0.65	0.70
Grieg	Informed Reduction	<b>0.82</b>	<b>0.92</b>
	Templating	0.52	0.62
Schumann	Informed Reduction	<b>0.85</b>	<b>0.91</b>
	Templating	0.71	0.75

Table 4.2 shows that the harmony model paired with an informed reduction algorithm leads to excellent results, out-performing the baseline Templating algorithm for three sets of pieces. As the accuracy varies significant accross different pieces and composers, a paired-t test was run for the two algorithms, resulting in a paired-t value of X, showing that the Informed Reduction algorithm has a higher accuracy with X% confidence. The excellent prediction performance using the Informed Reduction algorithm demonstrates that the chord segment reduction method can be used effectively to infer chord labels given that the non-chord-tones can be removed, affirming the underlying theory behind this project.



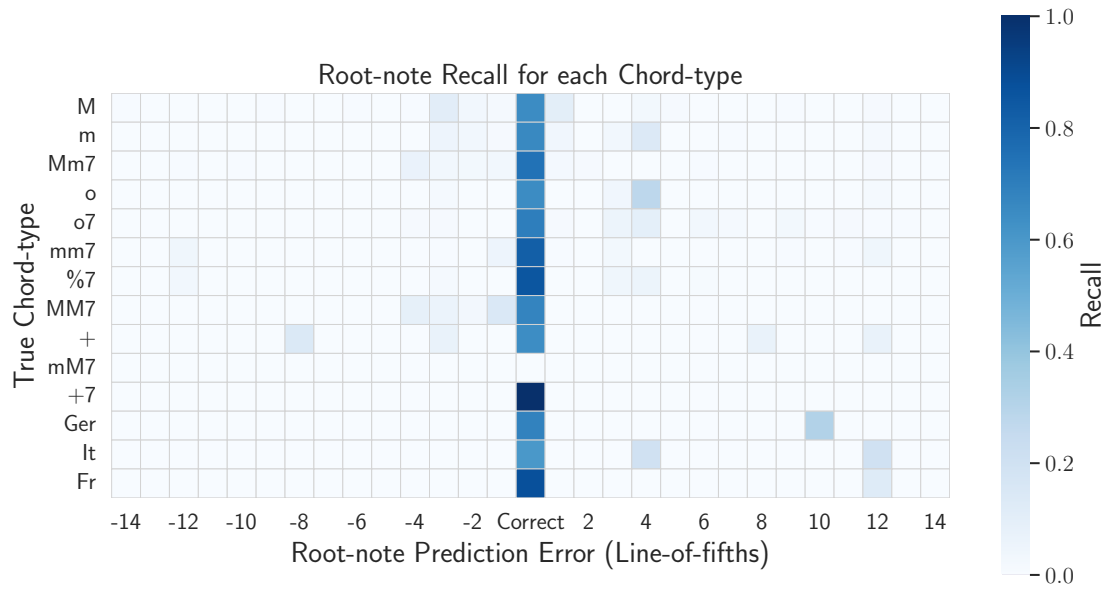


Figure 4.1: Root note confusion matrix for the Informed Reduction. The proportion of correct root note predictions are shown in the central column.

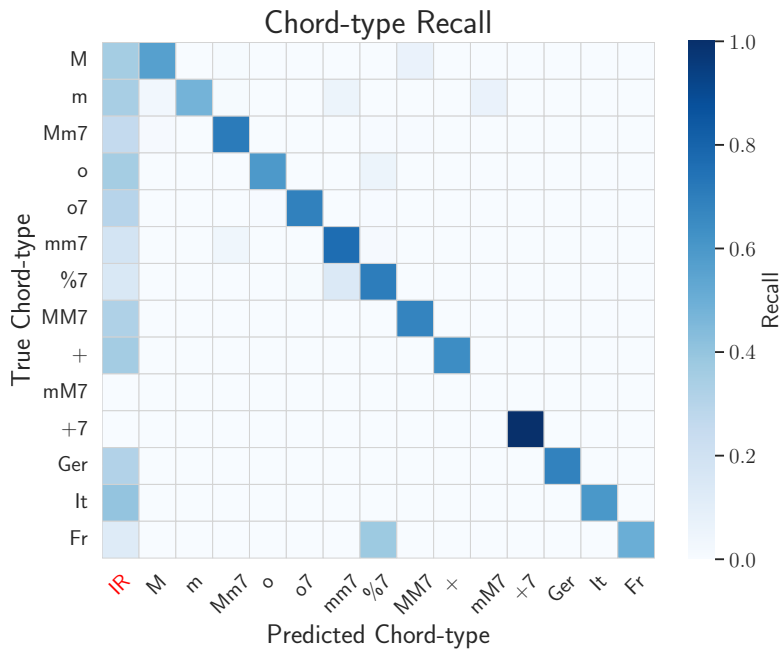


Figure 4.2: Chord-type confusion matrix for the Informed Reduction.

Figure 4.1 and Figure 4.2 show the root-note and full label prediction recall for specific chord-types. In Figure 4.1 we see that the root note prediction is excellent for the most common chord-types, with some systematic errors for the less common types.

## Limitations

**Flat vs hierarchical classification:** The task of inferring harmony has been framed as a *flat* classification problem where the output is a single predicted class. This assumes that each class

is mutually exclusive. In truth, chord labelling is a hierarchical system. For example, in most Western music most chord labels are loosely either major or minor; more complex chord-types (e.g. minor 7th) can be described through elaborations of these simpler ones.

**Incompleteness:** There are a huge number of features that relate the musical surface to chord labels which are ignored in the harmony model. The only features considered are the pitch-classes, the relative timing information of the notes and the segment boundaries. Gotham showed that the Other musical features include metre, dynamics, rhythm, form and texture and all influence the chord labelling process.

**Subjectivity:** Annotating music with chord labels is a subjective task which depends on the style and period of the music and each annotator’s subjective judgement. It has been noted that subjectivity has a significant effect on chord estimation accuracy [40] although the application of annotator subjectively in current research is limited. The DCML annotation standard used in this project is designed specifically to address the problems of subjectivity by providing a precise set of annotation rules and requiring a peer review process.

## 4.2 Heuristic Search

### 4.2.1 Prediction Accuracy

**Question 2:** Does the fitness function developed provide a meaningful proxy for the true accuracy of a proto-voice reduction?

**Question 3:** Does the developed heuristic search algorithm achieve an improvement in accuracy compared to the random parse?

To answer both questions, I show that the **Stochastic Beam Search** guided by the heuristic fitness function results in a statistically significant improvement in accuracy over both the **Random Parse** and **Random Reduction** for a varied selection of pieces.

### Discussion of results

Table 4.3: Random Parse vs Stochastic Beam Search

Data-set	Algorithm	Accuracy	Root-note Recall
Chopin	Stochastic Beam Search	<b><math>0.67 \pm 0.02</math></b>	<b><math>0.74 \pm 0.04</math></b>
	Random Parse	$0.54 \pm 0.04$	$0.67 \pm 0.04$
	Random Reduction	$0.55 \pm 0.03$	$0.68 \pm 0.04$
Grieg	Stochastic Beam Search	<b><math>0.48 \pm 0.02</math></b>	<b><math>0.62 \pm 0.03</math></b>
	Random Parse	$0.45 \pm 0.02$	$0.57 \pm 0.04$
	Random Reduction	$0.43 \pm 0.04$	$0.58 \pm 0.03$
Schumann	Stochastic Beam Search	$0.57 \pm 0.01$	$0.63 \pm 0.01$
	Random Parse	$0.60 \pm 0.01$	$0.68 \pm 0.01$
	Random Reduction	$0.55 \pm 0.05$	$0.66 \pm 0.05$

Table 4.3 shows the average full chord-label and root-note prediction accuracies for three datasets, with 95% confidence intervals. For Chopin and Grieg the heuristic search algorithm leads to a statistically significant improvement, however a statistically significant improvement is not achieved with the Schumann dataset. Regardless, this provides evidence that the combination of the fitness function and the Stochastic Beam Search algorithm leads to more accurate chord label predictions, and there's space for improvement with further refinement of the fitness function and search algorithm.

### 4.2.2 Interpretability

While the prediction accuracy results show that the fitness function guides the search towards reductions that lead to more accurate predictions, it is important that it does so in an interpretable manner, rather than making use of statistical regularities that do not coincide with human intuition or music theory practice.

**Question:** Does the fitness function developed provide a meaningful proxy for the music theoretic soundness of a proto-voice derivation?

I take an example segment reduction from the StochasticBeamSearch search example and compare it with one from the random parse, qualitatively evaluating if the heuristic search finds a derivation that corresponds to a more musically meaningful interpretation of the segment of music.

### Discussion of Results

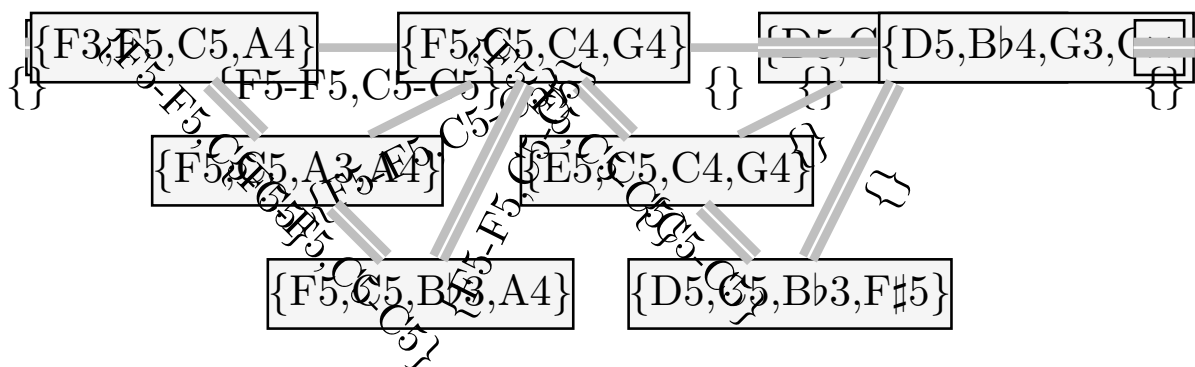


Figure 4.3: Example derivation (I will need to recreate this manually)

Figure 4.3 shows. .... While it did this thing well.. it isn't perfect though..fitness function can be refined further.

### 4.2.3 Scalability

**Question 5:** Does the Stochastic Beam Search algorithm achieve linear time complexity with respect to the input dimensions?

**Question 6:** Does the Stochastic Beam Search scale well with respect to its parameters?

**Question 7:** How is the trade-off between accuracy and run-time for the Stochastic Beam Search characterised?

### Discussion of Results

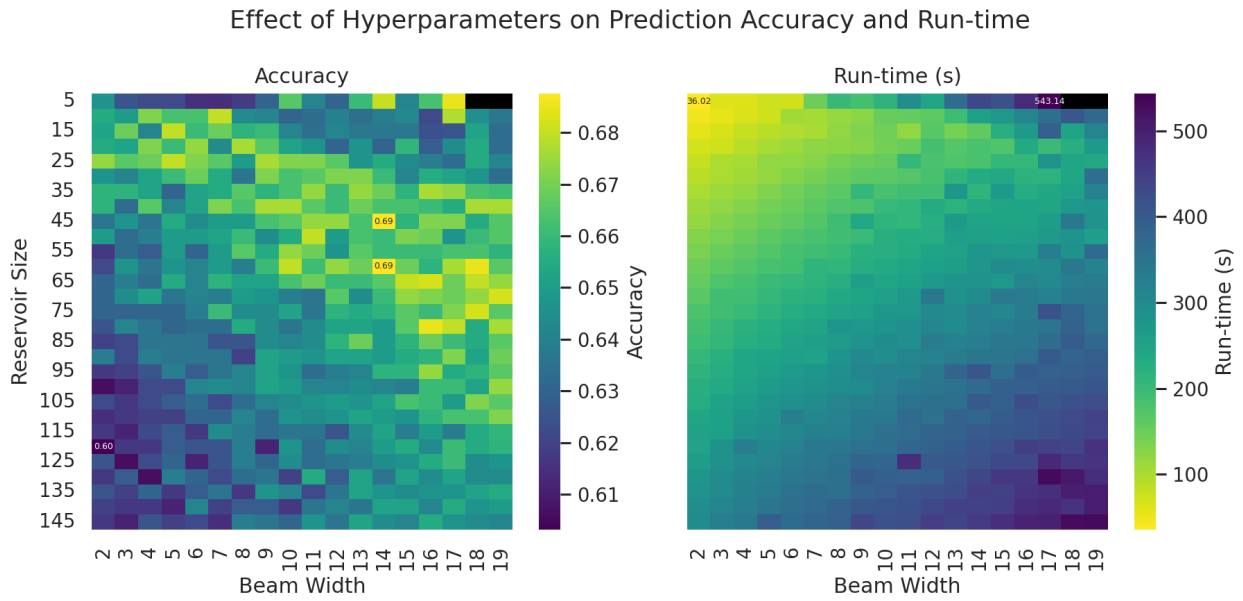


Figure 4.4: Note I guess these are actually just parameters

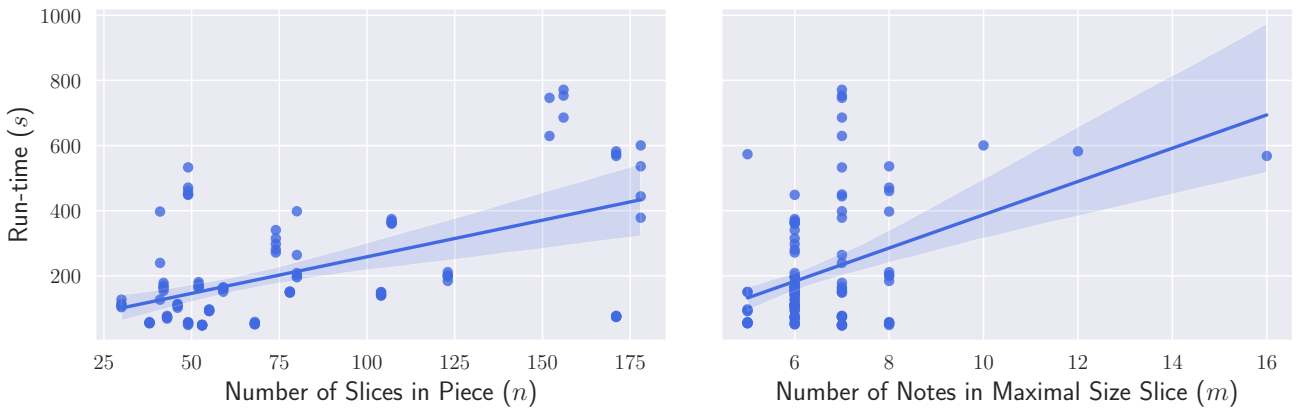


Figure 4.5: Run time with respect to the input dimensions,  $n$  and  $m$

## 4.3 Success Criteria

[TODO: explain how the evaluation has shown that the success criteria have been met.]

# Chapter 5

## Conclusions

---

This project aimed to and it has been demonstrated that this was achieved with great success. This chapter provides a reflection on the achievements of the project, the lessons learned, and possible avenues of future work.

- Reiterate how each of the questions asked on the outset have been answered.
- Use of visualisations for development very complex algorithm and output.
- Logging at different verbosity levels proved very helpful.
- Different Domains
- Further heuristic algorithms/ genetic algorithms
- Diversity - Context
- Non left-most parse
- Longer range dependencies
- Multi-stage process all in an explainable way? - get trendy and combine with large language model lol

# Bibliography

---

- [1] CHEN, T.-P., AND SU, L. Functional Harmony Recognition of Symbolic Music Data with Multi-task Recurrent Neural Networks, Sept. 2018.
- [2] CHEN, T.-P., AND SU, L. Harmony Transformer: Incorporating Chord Segmentation into Harmony Recognition. In *International Society for Music Information Retrieval Conference* (2019).
- [3] COHEN, D. E. “The Imperfect Seeks Its Perfection”: Harmonic Progression, Directed Motion, and Aristotelian Physics. *Music Theory Spectrum* 23, 2 (Oct. 2001), 139–169.
- [4] DAVIDSON-PILON, C. Bayesian Methods for Hackers, Mar. 2023.
- [5] DENG, J., AND KWOK, Y.-K. Large vocabulary automatic chord estimation using bidirectional long short-term memory recurrent neural network with even chance training. *Journal of New Music Research* 47, 1 (Jan. 2018), 53–67.
- [6] DONG, L., AND LAPATA, M. Language to Logical Form with Neural Attention. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)* (Berlin, Germany, 2016), Association for Computational Linguistics, pp. 33–43.
- [7] EBCIOĞLU, K. An Expert System for Harmonizing Four-Part Chorales. *Computer Music Journal* 12, 3 (1988), 43–51.
- [8] FAN, X., LIU, S., AND HENDERSON, T. C. Explainable AI for Classification using Probabilistic Logic Inference, May 2020.
- [9] FARD, H. H. Automatic Chord Recognition with Fully Convolutional Neural Networks.
- [10] FEISTHAUER, L., BIGO, L., GIRAUD, M., AND LEVÉ, F. Estimating keys and modulations in musical pieces. In *Sound and Music Computing Conference (SMC 2020)* (June 2020).
- [11] FINKENSIEP, C. Haskell-musicology: Scientific music types in Haskell. DCMLab, 2019.
- [12] FINKENSIEP, C. Protovoices - A Model of Tonal Structure. Digital and Cognitive Musicology Lab, Nov. 2021.
- [13] FINKENSIEP, C. *The Structure of Free Polyphony*. PhD thesis, EPFL, Lausanne, 2023.

- [14] FINKENSIEP, C., ERICSON, P., KLASSMANN, S., AND ROHRMEIER, M. Chord Types and Ornamentation - A Bayesian Model of Extended Chord Profiles. *Open Research Europe* (Apr. 2023).
- [15] FINKENSIEP, C., AND ROHRMEIER, M. Modeling and Inferring Proto-voice Structure in Free Polyphony. In *Proceedings of the 22nd ISMIR Conference* (Online, Nov. 2021).
- [16] GJERDINGEN, R. Cognitive Foundations of Musical Pitch Carol L. Krumhansl. *Music Perception: An Interdisciplinary Journal* 9 (July 1992), 476–492.
- [17] GOODMAN, J. Semiring Parsing. *Computational Linguistics* 25, 4 (1999), 573–606.
- [18] GOTHAM, M., KLEINERTZ, R., WEISS, C., MÜLLER, M., AND KLAUK, S. What if the 'When' Implies the 'What'? Human harmonic analysis datasets clarify the relative role of the separate steps in automatic tonal analysis. In *Proceedings of the 22nd International Society for Music Information Retrieval Conference, ISMIR 2021, Online, November 7-12, 2021* (2021), J. H. Lee, A. Lerch, Z. Duan, J. Nam, P. Rao, P. van Kranenburg, and A. Srinivasamurthy, Eds., pp. 229–236.
- [19] GRANROTH-WILDING, M. Harmonic Analysis of Music Using Combinatory Categorical Grammar. *The University Of Edinburgh* (2013).
- [20] HARASIM, D. Harmonic Syntax in Time: Rhythm Improves Grammatical Models of Harmony. In *Proceedings of the 20th ISMIR Conference* (2019), T. J. O'Donnell and M. A. Rohrmeier, Eds., ISMIR.
- [21] HUMPHREY, E. J., AND BELLO, J. P. Four Timely Insights on Automatic Chord Estimation.
- [22] ISLAM, R., ANDREEV, A. V., SHUSHARINA, N. N., AND HRAMOV, A. E. Explainable Machine Learning Methods for Classification of Brain States during Visual Perception. *Mathematics* 10, 15 (Jan. 2022), 2819.
- [23] JOHANNES, H. Ms3 - Parsing MuseScore 3. <https://github.com/johentsch/ms3>, 2021.
- [24] KIDNEY, D. O., AND WU, N. Algebras for weighted search. *Proceedings of the ACM on Programming Languages* 5, ICFP (Aug. 2021), 1–30.
- [25] KOOPS, H. V., DE HAAS, W. B., BRANSEN, J., AND VOLK, A. Automatic chord label personalization through deep learning of shared harmonic interval profiles. *Neural Computing and Applications* 32, 4 (Feb. 2020), 929–939.
- [26] KOOPS, H. V., DE HAAS, W. B., BURGOYNE, J. A., BRANSEN, J., KENT-MULLER, A., AND VOLK, A. Annotator subjectivity in harmony annotations of popular music. *Journal of New Music Research* 48, 3 (May 2019), 232–252.
- [27] KORZENIOWSKI, F., AND WIDMER, G. Improved Chord Recognition by Combining Duration and Harmonic Language Models, Aug. 2018.
- [28] KRUMHANSL, C. L., AND KESSLER, E. J. Tracing the dynamic changes in perceived tonal organization in a spatial representation of musical keys. *Psychological Review* 89 (1982), 334–368.



- [29] LERDAHL, F., AND JACKENDOFF, R. *A Generative Theory of Tonal Music*, repr. ed. MIT Press, Cambridge, Mass., 2010.
- [30] LU, S., MAO, J., TENENBAUM, J. B., AND WU, J. Neurally-Guided Structure Inference, Aug. 2019.
- [31] MARSDEN, A. Schenkerian Analysis by Computer: A Proof of Concept. *Journal of New Music Research* 39, 3 (Sept. 2010), 269–289.
- [32] MASADA, K., AND BUNESCU, R. Chord Recognition in Symbolic Music: A Segmental CRF Model, Segment-Level Features, and Comparative Evaluations on Classical and Popular Music, Oct. 2018.
- [33] MAUCH, M., MULLENSIEFEN, D., DIXON, S., AND WIGGINS, G. Can Statistical Language Models be used for the Analysis of Harmonic Progressions?
- [34] MAXWELL, H. J. An expert system for harmonizing analysis of tonal music. In *Understanding Music with AI: Perspectives on Music Cognition*. MIT Press, Cambridge, MA, USA, Aug. 1992, pp. 334–353.
- [35] MCLEOD, A., AND ROHRMEIER, M. A Modular System for the Harmonic Analysis of Musical Scores using a Large Vocabulary. In *Proceedings of the 22nd International Society for Music Information Retrieval Conference, ISMIR 2021, Online, November 7-12, 2021* (2021), J. H. Lee, A. Lerch, Z. Duan, J. Nam, P. Rao, P. van Kranenburg, and A. Srinivasamurthy, Eds., pp. 435–442.
- [36] MCLEOD, A., SUERMONDT, X., RAMMOS, Y., HERFF, S., AND ROHRMEIER, M. A. Three Metrics for Musical Chord Label Evaluation. In *Proceedings of the 14th Annual Meeting of the Forum for Information Retrieval Evaluation* (Kolkata India, Dec. 2022), ACM, pp. 47–53.
- [37] MEARNs, L. The Computational Analysis of Harmony in Western Art Music.
- [38] NEUWIRTH, M., HARASIM, D., MOSS, F. C., AND ROHRMEIER, M. The Annotated Beethoven Corpus (ABC): A Dataset of Harmonic Analyses of All Beethoven String Quartets. *Frontiers in Digital Humanities* 5 (July 2018), 16.
- [39] NI, Y., MCVICAR, M., SANTOS-RODRIGUEZ, R., AND DE BIE, T. An end-to-end machine learning system for harmonic analysis of music, July 2011.
- [40] NI, Y., MCVICAR, M., SANTOS-RODRÍGUEZ, R., AND DE BIE, T. Understanding Effects of Subjectivity in Measuring Chord Estimation Accuracy. *IEEE Transactions on Audio, Speech, and Language Processing* 21, 12 (Dec. 2013), 2607–2615.
- [41] OUDRE, L., FEVOTTE, C., AND GRENIER, Y. Probabilistic Template-Based Chord Recognition. *IEEE Transactions on Audio, Speech, and Language Processing* 19, 8 (Nov. 2011), 2249–2259.
- [42] PARDO, B., AND BIRMINGHAM, W. P. Algorithms for Chordal Analysis. *Computer Music Journal* 26, 2 (June 2002), 27–49.
- [43] PEARL, J. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. The Addison-Wesley Series in Artificial Intelligence. Addison-Wesley Pub. Co, Reading, Mass, 1984.

- [44] PICKENS, J. *Harmony Modeling for Polyphonic Music Retrieval*. PhD thesis, University of Massachusetts, 2004.
- [45] QUINN, I. Are Pitch-Class Profiles Really “Key for Key”? *Zeitschrift der Gesellschaft für Musiktheorie [Journal of the German-Speaking Society of Music Theory]* 7 (Jan. 2010).
- [46] RADICIONI, D. P., AND ESPOSITO, R. BREVE: An HMPerceptron-Based Chord Recognition System. In *Advances in Music Information Retrieval*, J. Kacprzyk, Z. W. Raś, and A. A. Wieczorkowska, Eds., vol. 274. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 143–164.
- [47] RAFFEL, C., MCFEE, B., HUMPHREY, E. J., SALAMON, J., NIETO, O., LIANG, D., AND ELLIS, D. P. W. Mir\_eval: A Transparent Implementation of Common MIR Metrics. In *Proceedings of the 15th International Conference on Music Information Retrieval* (2014).
- [48] RAPHAEL, C., AND STODDARD, J. Functional Harmonic Analysis Using Probabilistic Models. *Computer Music Journal* 28, 3 (Sept. 2004), 45–52.
- [49] ROCHER, T., ROBINE, M., HANNA, P., AND STRANDH, R. Dynamic Chord Analysis for Symbolic Music.
- [50] SAPP, C. Visual hierarchical key analysis. *Computers in Entertainment* 3 (Oct. 2005), 1–19.
- [51] SAPP, C. 6 Computational Chord-Root Identification in Symbolic Musical Data: Rationale, Methods, and Applications.
- [52] SARICA, A., QUATTRONE, A., AND QUATTRONE, A. Explainable machine learning with pairwise interactions for the classification of Parkinson’s disease and SWEDD from clinical and imaging features. *Brain Imaging and Behavior* 16, 5 (Oct. 2022), 2188–2198.
- [53] SIDOROV, K., JONES, A., AND MARSHALL, D. MUSIC ANALYSIS AS A SMALLEST GRAMMAR PROBLEM.
- [54] TEMPERLEY, D. An Algorithm for Harmonic Analysis. *Music Perception* 15, 1 (Oct. 1997), 31–68.
- [55] TEMPERLEY, D. A Bayesian Approach to Key-Finding. In *Music and Artificial Intelligence*, G. Goos, J. Hartmanis, J. van Leeuwen, C. Anagnostopoulou, M. Ferrand, and A. Smaill, Eds., vol. 2445. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002, pp. 195–206.
- [56] TEMPERLEY, D. A Bayesian Approach to Key-Finding. In *Music and Artificial Intelligence* (Berlin, Heidelberg, 2002), C. Anagnostopoulou, M. Ferrand, and A. Smaill, Eds., Lecture Notes in Computer Science, Springer, pp. 195–206.
- [57] TEMPERLEY, D. A Unified Probabilistic Model for Polyphonic Music Analysis. *Journal of New Music Research* 38, 1 (Mar. 2009), 3–18.
- [58] VIJAYAKUMAR, A. K., COGSWELL, M., SELVARAJU, R. R., SUN, Q., LEE, S., CRANDALL, D., AND BATRA, D. DIVERSE BEAM SEARCH: DECODING DIVERSE SOLUTIONS FROM NEURAL SEQUENCE MODELS.

- [59] VIJAYAKUMAR, A. K., COGSWELL, M., SELVARAJU, R. R., SUN, Q., LEE, S., CRANDALL, D., AND BATRA, D. DIVERSE BEAM SEARCH: DECODING DIVERSE SOLUTIONS FROM NEURAL SEQUENCE MODELS.
- [60] VOLK, A. Improving Audio Chord Estimation by Alignment and Integration of Crowd-Sourced Symbolic Music. 141–155.
- [61] WINOGRAD, T. Linguistics and the Computer Analysis of Tonal Harmony. *Journal of Music Theory* 12, 1 (1968), 2–49.
- [62] WU, Y., CARSAULT, T., AND YOSHII, K. Automatic Chord Estimation Based on a Frame-wise Convolutional Recurrent Neural Network with Non-Aligned Annotations. In *2019 27th European Signal Processing Conference (EUSIPCO)* (A Coruna, Spain, Sept. 2019), IEEE, pp. 1–5.

# Chapter A

## Additional Information

---

# Chapter B

## Project Proposal

---

# Inferring Harmony from Free Polyphony

Judah Daniels

May 8, 2023

DOS: Prof. Larry Paulson

Crsid: jasd6

College: Clare College

## B.1 Abstract

A piece of music can be described using a sequence of chords, representing a higher level harmonic structure of a piece. There is a small, finite set of chord types, but each chord can be realised on the musical surface in a practically infinite number of ways. Given a score, we wish to infer the underlying chord types.

The paper *Modeling and Inferring Proto-voice Structure in Free Polyphony* describes a generative model that encodes the recursive and hierarchical dependencies between notes, giving rise to a grammar-like hierarchical system [15]. This proto-voice model can be used to reduce a piece into a hierarchical structure which encodes an understanding of the tonal/harmonic relations of a piece.

Christoph Finkensiep suggests in his paper that the proto-voice model may be an effective way to infer higher level latent entities, such as harmonies or voice leading schemata. Thus in this project I will ask the question: is this parsing model an effective way to annotate harmonies? By ‘effective’ we are referring to two things:

- Accuracy: can the model successfully emulate how experts annotate harmonic progressions in musical passages?
- Practicality: can the model be used to do this within a reasonable time frame?

While the original model could in theory be used to generate harmonic annotations, its exhaustive search strategy would be prohibitively time-consuming in practice for any but the shortest musical extracts; one half measure can have over 100,000 valid derivations [15]. My approach will be to explore the use of heuristic search algorithms to solve this problem.

## B.2 Substance and Structure

### B.2.1 Core: Search

The core of this project is essentially a search problem characterised as follows:

- The state space  $S$  is the set of all possible partial reductions of a piece along with each reduction step that has been done so far.
- We have an initial state  $s_o \in S$ , which is the empty reduction, corresponding to the unreduced surface of the piece. The score is represented as a sequence of slices grouping notes that sound simultaneously. We are also given the segmentation of the original chord labels that we wish to retrieve.
- We have a set of actions,  $A$  modelled by a function  $action : A \times S \rightarrow S$ . These actions correspond to a single reduction step.
  - The reduction steps are the inverses of the operations defined by the generative proto-voice model.

- Finally we have a goal test,  $goal : S \rightarrow \{true, false\}$  which is true iff the partial reduction  $s$  has exactly one slice per segment of the input.
  - This means the partial reduction  $s$  contains a sequence of slices which start and end positions corresponding to the segmentation of the piece.
- At the first stage, this will be implemented using a random graph search algorithm, picking each action randomly, according to precomputed distributions.

### B.2.2 Core: Evaluation

The second core task is to create an evaluation module that iterates over the test dataset, and evaluates the partial reduction computed by the search algorithm above. This will be done by comparing the outputs to ground truth annotations from the Annotated Beethoven Corpus.

In order to do this I will make use of the statistical harmony model from Finkensiep’s thesis, *The Structure of Free Polyphony* [13]. This model provides a way of mapping between the slices that the algorithm generates and the chords in the ground truth. This can be used to empirically measure how closely the slices match the expert annotations.

### B.2.3 Extension

Once the base search implementation and evaluation module have been completed, the search problem will be tackled by heuristic search methods, with different heuristics to be trialled and evaluated against each other. The heuristics will make use of the chord profiles from Finkensiep’s statistical harmony model discussed above. These profiles relate note choices to the underlying harmony. Hence the heuristics may include:

- How the chord types relate to the pitches used.
- How the chord types relate which notes are used as ornamentation, and the degree of ornamentation.
- Contextual information about neighboring slices

### B.2.4 Overview

The main work packages are as follows:

**Preliminary Reading** – Familiarise myself with the proto-voice model, and read up on similar models and their implementations. Study heuristic search algorithms.

**Dataset Preparation** – Pre-process the Annotated Beethoven Corpus into a suitable representation for my algorithm.

**Basic Search** – Implement a basic random search algorithm that takes in surface and segmentations, and outputting the sequence of slices matching the segmentations.



**Evaluation Module** – Implement an evaluation module to evaluate the output from the search algorithm.

**End-to-end pipeline** – Implement a full pipeline from the data to the evaluation that can be used to compare different reductions.

**Heuristic Design** – Extension – Trial different heuristics and evaluate their performance against each other.

**Dissertation** – I intend to work on the dissertation throughout the duration of the project. I will then focus on completing and polishing the project upon completion.

## B.3 Starting Point

The following describes existing code and languages that will be used for this project:

**Haskell** – I will be using Haskell for this project as it is used in the proto-voice implementation. It must be noted that my experience with Haskell is limited, as I was first introduced to it via an internship this summer (July to August 2022).

**Python** – Python will be used for data handling. I have experience coding in Python.

**Prior Research** - Over the summer I have been reading the literature on computational models of music, as well as various parsing algorithms such as semi-ring parsing [17], and the CYK algorithm, which is used in the implementation of the proto-voice model.

**Protovoices-Haskell** – The paper *Modeling and Inferring Proto-Voice Structure in Free Polyphony* [15] includes an implementation of the proto-voice model in Haskell. A fork of this repository will form the basis of my project. This repository includes a parsing module which will be used to perform the actions in the search space of partial reductions. There is a module that can exhaustively enumerate reductions of a piece, but this is infeasible in practice due to the blowup of the derivation forest.

**MS3** – This is a library for parsing MuseScore Files and manipulating labels [23], which I will use as part of the data processing pipeline.

**ABC** – The *Annotated Beethoven Corpus* [38] contains analyses of all Beethoven string quartets composed between 1800 and 1826, encoded in a human and machine readable format. This will be used as a dataset for this project.

## B.4 Success Criteria

This project will be deemed a success if I complete the following tasks:

- Develop a baseline search algorithm that uses the proto-voice model to output a partial reduction of a piece of music up to the chord labels.
- Create an evaluation module that can take the output of the search algorithm and quantitatively evaluate its accuracy against the ground truth annotations by providing a score based on a statistical harmony model.
- Extension: Develop one or more search algorithms that use additional heuristics to inform the search, and compare the accuracy with the baseline algorithm.

## B.5 Timetable

Time frame	Work	Evidence
<b>Michaelmas (Oct 4 to Dec 2)</b>		
Oct 14 to Oct 24	<i>Oct 14</i> : Final proposal deadline. Preparation work: familiarise myself with the dataset and the proto-voice model implementation. Work on manipulating reductions using the proto-voice parser provided by the paper.	None
Oct 24 to Nov 7	Dataset preparation and handling.	Plot useful metrics about the dataset using Haskell
Nov 7 to Nov 21	Random Search implementation	None
Nov 21 to Dec 5	Evaluation Module. Continue with search implementation.	Evaluate a manually created derivation and plot results
<b>Vacation (Dec 3 to Jan 16)</b>		
Dec 5 to Dec 11	Evaluate performance of random search. Begin to work on extensions	Plot results
Dec 10 to Dec 21	Trial different heuristics. Implement an end-to-end pipeline from input to evaluation.	None
Dec 21 to Dec 27	None	None
Dec 27 to Jan 10	Continue trialing and evaluating heuristics	<i>Fulfill success criterion: At least one heuristic technique gives better performance than random search.</i>
<b>Lent (Jan 17 to Mar 17)</b>		
Jan 4 to Jan 20	Buffer Period to help keep on track	None
Jan 20 to Feb 3	<i>Feb 3</i> : Progress Report Deadline. Write progress report and prepare presentation. Write draft <i>Evaluation</i> chapter	Progress Report (approx. 1 page)
Feb 3 to Feb 17	Prepare presentation.	<i>Feb 8 – 15</i> : Progress Report presentation
Feb 17 to Mar 3	<i>Feb 17</i> : How to write a Dissertation briefing. Write draft Introduction and Preparation chapters. Incorporate feedback on Evaluation chapter.	Send draft Introduction and Preparation chapter to supervisor
Mar 3 to Mar 17	Write draft Implementation chapters. Incorporate feedback on Introduction and Preparation chapters.	Send draft Implementation chapters to Supervisor
<b>Vacation (Mar 18 to</b>		

## B.6 Resources

I plan to use my own laptop for development: MacBook Pro 16-inch, M1 Max, 32GB Ram, 1TB SSD, 24-core GPU.

All code will be stored on a GitHub repository, which will guarantee protection from data loss. I will easily be able to switch to using university provided computers upon hardware/software failure.

The project will be built upon work that has been done in the DCML (Digital cognitive musicology lab) based in EPFL. The files are in their Github repository, and I have been granted permission to access their in-house datasets of score annotations, as well as software packages which are used to handle the data.

## B.7 Supervisor Information

Peter Harrison, head of Centre for Music and Science at Cambridge, has agreed to supervise me for this. We have agreed on a timetable for supervisions for this year. I am also working with Christoph Finkensiep, a PHD student at the DCML, and originator of the proto-voice model. Professor Larry Paulson has agreed to be the representative university teaching officer.