UCLouvain

École polytechnique de Louvain

Making FuxCP generate harmonious counterpoints

Authors: **Bryce Burignat, Sacha de Menten**
Supervisor: **Peter Van Roy**
Readers: **Karim Haddad, Hélène Verhaeghe**
Academic year 2025–2026
Master [120] in Computer Science and Engineering

Abstract

FuxCP is a constraint-programming based generator for species counterpoint in the style of Johann Joseph Fux. It encodes Fux-inspired rules as explicit constraints over discrete musical variables (MIDI pitches on a rhythmic grid) and ranks feasible solutions using a structured cost model. Before this thesis, the project already contained an extensive rule base spanning multiple species and voice counts, but it was not yet robust as a generation system: optimization could crash after the first solution, several higher-species and multi-voice configurations were impractically slow or infeasible within reasonable time budgets, and recurring musical artefacts (e.g., repetition patterns and weak multi-voice harmonic control) appeared in generated outputs.

This thesis improves FuxCP along two complementary axes. First, it introduces structural improvements that make generation stable and controllable: a corrected optimization pipeline (including objective variable instantiation for Branch-and-Bound), refined branching and pruning choices, explicit solver controls (timeout and stagnation management), and a consolidated orchestration layer with a standalone CLI, a programmatic API returning a structured GenerationResult, configuration loading from CSV presets/cantus datasets, MIDI export, and cantus-firmus scale detection for direct inputs. Second, it improves the musical model through targeted constraint work: new voice-independence constraints, multi-voice harmonic completeness rules motivated by expert review, a Bitsch-inspired chord-membership rule for weak beats, and relaxed variants of selected constraints to preserve musical intent while keeping hard cases solvable.

Experiments on representative configurations show that previously problematic settings in fourth and fifth species and in multi-voice contexts become practically reachable, while the resulting counterpoints exhibit fewer systematic artefacts and improved stylistic coherence under fixed generation budgets.

Acknowledgments

This thesis was made possible thanks to the guidance and support of several people whom we would like to thank.

We would like to thank our supervisor, **Peter Van Roy**, who introduced us to the subject with such enthusiasm and answered all our questions when we were choosing our topic. His availability throughout the project and his in-depth knowledge of the topics covered were of great help.

Karim Haddad, professor of musicology at IRCAM in Paris, played a vital role in verifying the accuracy and musical correctness of our counterpoint. He also advised us on the theoretical decisions to be made regarding modifications or new constraints. His approachability and prompt responses via email were of great help to us during the second phase of the project.

Finally, we warmly thank the previous contributors to this project : **Tom Lai**, **Damien Sprockeels**, **Thibault Wafflard**, **Anton Lamotte**, **Luc Cleenewerk**, and **Diego de Patoul** for the quality and rigor of their work, which laid the foundations for the present thesis.

Note on AI utilization

During the preparation of this master's thesis, we utilized ChatGPT and Claude code for the following purpose:

1. Improve readability: The tool was used to improve fluency and readability by rephrasing redundant sentences.

2. Correcting grammar and spelling mistakes.

3. Help understanding and implementing functions for new constraints and documentation, and help creating scripts to facilitate the generation of counterpoints.

After using these tools, we diligently reviewed and edited the content produced by the tool. We take full responsibility for the final content presented in this thesis.

Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.1.1 Existing foundations and their limits. . . . 2

1.1.2 Problem statement . . . 2

1.1.3 Connection with other theses . . . 2

1.1.4 Dual scope. . . . 3

1.1.5 Contributions . . . 3

1.2 Research methodology . . . 3

1.2.1 Instrumentation and measurable signals . . . 4

1.2.2 Stabilisation-first approach and performance diagnosis . . . 4

1.2.3 Musical review and external feedback . . . 5

1.2.4 Procedure for implementing a change . . . 6

1.2.5 Configuration as a reproducible artifact . . . 7

1.2.6 Limits of the approach . . . 7

1.3 Roadmap . . . 7

2 Background **9**

2.1 Previous work . . . 9

2.1.1 FuxCP . . . 9

2.1.2 Project lineage and context . . . 10

2.1.3 State of the project before this thesis . . . 10

2.2 Species counterpoint and Fux terminology . . . 11

2.2.1 Global approach . . . 11

2.2.2 Main concepts[3] . . . 12

2.2.3 Other useful terminologies . . . 13

2.3 Representation choices in FuxCP . . . 14

2.3.1 Discrete time representation . . . 14

2.3.2 Musical notation and MIDI . . . 15

2.4 Constraint programming essentials . . . 15

2.4.1 Constraint satisfaction problems . . . 15

2.4.2 Constraint optimization and lexicographic objectives . . . 16

2.4.3 Propagation and search 16

3 FuxCP Architecture **18**

3.1 Overview 19
3.2 Resolution pipeline 21
3.2.1 Entry points 21
3.2.2 Call `create_problem(...)` 22
3.2.3 Create the search engine and run the solver 23
3.2.4 Export the final data 25
3.3 Costs priority ordering 26

4 Constraints system **28**

4.1 Role of the constraint model in FuxCP 29
4.2 Core musical representations used by constraints 29
4.2.1 Key variable arrays used by constraints 29
4.2.2 Decision variables and indexing 30
4.2.3 Other important derived variables 32
4.2.4 Notes on strata 32
4.3 Constraint posting pipeline and layering 33
4.3.1 Per-part constraints (species classes) 33
4.3.2 Cross-voice constraints (Two/Three/FourVoiceCounterpoint) . . 33
4.3.3 Stratum constraints 34
4.3.4 Conditional posting and experimentation. 34
4.4 Constraints organisation and naming convention 34
4.4.1 Rule families and theoretical numbering (G/H/M/P/R) 35
4.4.2 Scoping and context-specific variants 35
4.4.3 Code-level identifiers: encoding scope in names 36
4.4.4 Constraint identifiers vs implementation function names 37
4.5 Hard constraints, soft constraints, and relaxed constraints 37
4.5.1 Hard constraints: feasibility 37
4.5.2 Soft costs: preference modelling 38
4.5.3 Relaxed constraints: controlled weakening with penalties 39
4.6 Activation, ablation, and debugging workflow 39
4.6.1 Global activation masks 40
4.6.2 Ablation: locating feasibility blockers 40
4.6.3 Relaxation: recovering solutions while keeping musical intent . . 41
4.7 Worked examples: from theoretical rules to Gecode constraints 42
4.7.1 Example 1: consonance as a domain constraint 42
4.7.2 Example 2: dissonance implies passing motion (reified implication) 43
4.7.3 Example 3: a motion restriction into perfect consonances 44

5 Structural improvements **46**
5.1 Optimisation strategy . . . 47
5.1.1 Branching strategies . . . 47
5.1.2 Choice of search engine . . . 49
5.1.3 Cost function refinements . . . 50
5.2 Restructuration . . . 52
5.2.1 Goals of the restructuration . . . 52
5.2.2 Files moved, removed, and added . . . 52
5.2.3 Quick codebase documentation . . . 53
5.3 Other small changes . . . 54

6 Constraints improvements **56**
6.1 New constraints . . . 57
6.1.1 P8_noSuccessiveSamePerfectInterval . . . 57
6.1.2 P9_noSimultaneousRepetition . . . 58
6.1.3 H2_2_arsisHarmoniesCannotBeDissonant_multiVoice . . . 58
6.1.4 H2_3_dissonanceImpliesDiminution_multiVoice . . . 59
6.1.5 H2_chordMembershipOnDisjunctWeakBeats . . . 59
6.2 Relaxed versions . . . 61
6.2.1 G9_lastChordSameAsFundamental_soft . . . 61
6.2.2 M2_2_3v_twoConsecutiveNotesAreNotTheSame_soft . . . 62
6.2.3 H2_3_dissonanceImpliesDiminution_soft . . . 62
6.2.4 H2_2_arsisHarmoniesCannotBeDissonant_soft . . . 63
6.3 Revised constraints . . . 64
6.3.1 P4_successiveCost . . . 64
6.3.2 M2_1_varietyCost . . . 64
6.3.3 SP1_1M2_2V . . . 64
6.3.4 4.P5 . . . 65

7 Evaluation **66**
7.1 Experimental setup . . . 67
7.1.1 A single reference cantus firmus . . . 67
7.1.2 The three difficulty axes . . . 67
7.1.3 Objective mode and protocol . . . 68
7.1.4 Metrics . . . 68
7.2 Results . . . 69
7.2.1 Overall feasibility map . . . 69
7.2.2 Two-voice results . . . 69
7.2.3 Three-voice results . . . 70
7.2.4 Four-voice results . . . 71

7.2.5 Cross-voice synthesis . . . 71
7.3 Trade-offs and compromises . . . 72
7.3.1 Optimality versus time budget. . . . 72
7.3.2 Voice register control versus feasibility . . . 73

8 Conclusion **75**
8.1 Summary . . . 75
8.2 Limitations of the current model . . . 76
8.3 Future work . . . 77
8.3.1 Constraint-level efficiency and solver performance . . . 77
8.3.2 Musical validation and rule-base completion . . . 77
8.3.3 Robustness across cantus firmi, tonal contexts, and configuration space . . . 78

Bibliography **80**

A Full validated model of Fux counterpoint **82**
A.1 General Rules (G) . . . 82
A.2 Constraints of the First Species . . . 84
A.3 Constraints of the Second Species . . . 92
A.4 Constraints of the Third Species . . . 96
A.5 Constraints of the Fourth Species . . . 99
A.6 Constraints of the Fifth Species . . . 101
A.7 Tables of rules scope . . . 105

B User references **107**
B.1 Configuration and presets . . . 107
B.1.1 Command-line interface (main.cpp) . . . 107
B.1.2 Programmatic API (GenerationsAPI) . . . 109
B.1.3 Presets (`presets.csv`) . . . 109
B.1.4 GenerationCase struct . . . 110
B.1.5 GenerationResult struct . . . 111

C Generated counterpoints exemples **114**

A Complete FuxCP Code **121**
A.1 Counterpoint problem headers . . . 121
A.1.1 CounterpointProblem.hpp . . . 121
A.1.2 CounterpointUtils.hpp . . . 123
A.1.3 TwoVoiceCounterpoint.hpp . . . 124
A.1.4 ThreeVoiceCounterpoint.hpp . . . 125

A.1.5 FourVoiceCounterpoint.hpp . . . 125
A.2 Voice and part headers . . . 126
A.2.1 Voice.hpp . . . 126
A.2.2 Stratum.hpp . . . 127
A.2.3 Part.hpp . . . 127
A.2.4 CantusFirmus.hpp . . . 130
A.3 Species headers . . . 131
A.3.1 FirstSpeciesCounterpoint.hpp . . . 131
A.3.2 SecondSpeciesCounterpoint.hpp . . . 133
A.3.3 ThirdSpeciesCounterpoint.hpp . . . 134
A.3.4 FourthSpeciesCounterpoint.hpp . . . 134
A.3.5 FifthSpeciesCounterpoint.hpp . . . 136
A.4 Constraints and utility headers . . . 137
A.4.1 constraints.hpp . . . 137
A.4.2 Utilities.hpp . . . 140
A.4.3 Generations.hpp . . . 149
A.4.4 GenerationsAPI.hpp . . . 150
A.4.5 ConfigLoader.hpp . . . 151
A.4.6 Midi.hpp . . . 152
A.5 Test headers . . . 152
A.5.1 fuxTest.hpp . . . 152
A.5.2 figureTests.hpp . . . 156
A.6 Counterpoint problem sources . . . 157
A.6.1 CounterpointProblem.cpp . . . 157
A.6.2 CounterpointUtils.cpp . . . 174
A.6.3 TwoVoiceCounterpoint.cpp . . . 175
A.6.4 ThreeVoiceCounterpoint.cpp . . . 177
A.6.5 FourVoiceCounterpoint.cpp . . . 181
A.7 Voice and part sources . . . 185
A.7.1 Voice.cpp . . . 185
A.7.2 Stratum.cpp . . . 185
A.7.3 Part.cpp . . . 186
A.7.4 CantusFirmus.cpp . . . 191
A.8 Species sources . . . 194
A.8.1 FirstSpeciesCounterpoint.cpp . . . 194
A.8.2 SecondSpeciesCounterpoint.cpp . . . 199
A.8.3 ThirdSpeciesCounterpoint.cpp . . . 203
A.8.4 FourthSpeciesCounterpoint.cpp . . . 209
A.8.5 FifthSpeciesCounterpoint.cpp . . . 215

A.9 Constraints and utility sources 228
A.9.1 constraints.cpp 228
A.9.2 Utilities.cpp 241
A.9.3 Generations.cpp 248
A.9.4 GenerationsAPI.cpp 254
A.9.5 ConfigLoader.cpp 254
A.9.6 main.cpp 256
A.9.7 Midi.cpp 258
A.10 Test sources 260
A.10.1 fuxTest.cpp 260
A.10.2 figureTests.cpp 324
A.10.3 main.cpp 339
A.11 Build system and command-line interface 342
A.11.1 Makefile 342
A.11.2 fuxcp 344
A.12 Configuration data 347
A.12.1 cantus$_f irmus.csv$ 347
A.12.2 presets.csv 348

Chapter 1

Introduction

Chapter summary. This chapter outlines the starting point of this thesis, how the problems and their limitations were identified with a view to addressing them. It also discusses collaborations with other students on theses related to the same subject. Finally, the methodology employed during the research and a brief summary of our specific contributions round off the chapter.

Roadmap.

- Section 1.1 introduces the connection and interest generated between a treatise on musical counterpoint and computer science. The background to the problem and the division of research tasks among other students are also outlined.
- Section 1.2 explains how changes to an existing project were implemented. Understanding how to interpret musical results and which error signals to monitor allowed us to iterate effectively on fixes and new features.
- Section 1.3 shows the structure of the thesis with a quick summary of each chapter.

1.1 Motivation

The counterpoint treatise *Fux gradus ad Parnassum*[1] describes a conversation between himself and his student. Its rules, organized across five species of increasing rhythmic complexity and extended to multi-voice writing, define a well-scoped formal system in which musical quality and constraint satisfaction are inseparable concerns. This formal character makes it an attractive target for automated reasoning: the rules are enumerable, violations are measurable, and the esthetic outcome is audible.

1.1.1 Existing foundations and their limits.

Prior work on the FuxCP solver, developed at UCLouvain through several previous student's theses and research contributions, has made it possible to build a constraint solver capable of constructing counterpoints for up to four voices and five species. This thesis builds directly on that foundation, taking the system as it stood at the end of the previous development cycle and extending it along three axes: the resolution of computer bugs still present in the solver, the management of the numerous parameters through the creation of a centralized generation logic, and the incorporation of musical domain knowledge to guide both constraint design and parameter choices.

1.1.2 Problem statement

This thesis addresses the following problem: given a cantus firmus (the initial chosen melody) expressed as a sequence of MIDI pitches in a specified modal context, automatically generate one or more counterpoint voices conforming to the rules established in the *Fux gradus ad Parnassum* such that the result minimizes a hierarchically weighted cost function encoding both hard constraints and soft preferences.

More precisely, the system must handle five species of counterpoint (from the first, consisting of note-against-note writing, to the fifth,which combines the rhythmic patterns of all preceding species), across up to two from four simultaneous voices. A multitude of default parameters (voice spacing, cost mode, timeout, etc.) also have an impact on the output provided.

1.1.3 Connection with other theses

This thesis is part of a broader, ongoing development effort around FuxCP at UCLouvain, where several works focus on complementary aspects of the project.

First, the integration of FuxCP into composition workflows is being extended through a plug-in for Digital Audio Workstations (DAWs). This plug-in is being implemented in the master's thesis of Chris Bakashika[13], with an expected completion around August 2026. The objective is to make FuxCP directly usable inside common production environments, bridging the gap between a solver centered tool and composer side practice.

Second, the solver's behaviour is being studied from an experimental perspective through a parameter sensitivity analysis conducted by Dorian Genon[12]. This thesis extends the FuxCP constraint-based counterpoint solver by introducing a dynamic cost model and a cost group system, allowing composers to control how musical costs evolve positionally throughout a piece. It generalizes dynamic sliders to all cost types through a unified grouping mechanism, enabling control over harmonic, melodic and motion

behaviors. It is also expected around June 2026, and will complement this thesis by providing a more systematic view of the configuration space and its trade-offs.

Together, these parallel theses strengthen the FuxCP project along two directions: practical integration for end users (DAW plug-in) and empirical characterization of solver sensitivity (parameter analysis), while the present thesis focuses on improving FuxCP as a generation system through structural and constraint-level changes.

1.1.4 Dual scope.

A central challenge is that the later species and multi-voice configurations significantly increase search complexity: not all cantus firmi admit a solution under all configurations within practical time budgets, and the choice of solver parameters has a non-trivial impact on both solution cost and convergence speed. The thesis therefore has a dual scope: producing musically valid counterpoints across the full configuration matrix, and characterizing the solver's behavior through reproducible benchmarks so that future developers can make informed configuration choices.

1.1.5 Contributions

Here is a general list of all the final changes and additions to this project following the research.

- **Structural improvements (Chapter 5)**
  - Improved optimization strategy (section 5.1)
  - Restructuration (section 5.2)
- **Constraint-level improvements (chapter 6)** We fixed a batch of musical problems with generated counterpoints in collaboration with Karim Haddad,
- **Implementation** This is explained in detail throughout the thesis, the full source code is provided in the Appendix.
- **Evaluation (chapter 7)** Changes made to ensure solver efficiency and musical quality of the solutions.

1.2 Research methodology

This thesis contains two complementary improvement tracks: structural improvements (search strategy, orchestration, and codebase organization), and constraint-level improvements (new rules, revised rules, and relaxed variants). Both tracks follow the same principle: changes are introduced only when they are supported by concrete evidence,

either in solver behavior (feasibility and runtime), in musical artifacts (MIDI/score review), or in external expert feedback.

Overall, the research method is empirical and iterative: rather than deriving parameters and modeling choices only from musical arguments, we repeatedly generated counterpoints under a controlled configuration, evaluated the outcome (computationally and musically), identified systematic failure modes, and then corrected the relevant part of the system (constraints, costs, search settings, or orchestration). This method was applied from the simplest 2-voice cases up to multi-voice and higher-species configurations.

1.2.1 Instrumentation and measurable signals

To keep improvements focused on evidence, we relied on standardized generation outputs. Each run produces a structured result (search status, timings, costs, and solver statistics) and optionally detailed logs and exported artifacts. This made it possible to compare configurations and improvements consistently, even when the underlying musical content differed.

Concretely, the main signals used throughout the work were:

- **Feasibility signals:** whether a first solution is reached within budget, and the time-to-first-solution.
- **Optimisation signals:** best cost over time (number of improvements, time of last improvement), used to detect stagnation and to compare objective modes or search configurations.
- **Search effort signals:** solver statistics (nodes, failures, propagations, depth), used to distinguish "hard but progressing" from "thrashing" search behaviour.

These metrics were complemented by qualitative inspection (MIDI/score review) and by external feedback, but they provided the baseline needed to decide when a change measurably improves the generator.

1.2.2 Stabilisation-first approach and performance diagnosis

The first priority was to make generation stable enough to support systematic review. We progressively increased the difficulty of generated configurations (from simple 2-voice cases to multi-voice and higher species) and tuned the pipeline until it could reliably produce solutions within practical budgets. In particular, we observed that register variations (parameter `v_type`) strongly affect both feasibility and optimisation difficulty, so we treated register as a first-class axis during diagnosis.

What "tuning the pipeline" means. When a configuration was too slow or did not reach a first solution, we treated it first as a diagnosis problem (what prevents feasibility / what prevents progress) rather than as an immediate "musical" problem. In this context, tuning the pipeline consists of a small set of practical actions:

- testing several configurations of branching choices and pruning behaviour,
- for difficult or timed-out configurations, using a constraint-validation procedure based on ablation,
- introducing explicit solver-control parameters (we kept only timeout and stagnation stop) so that runs are bounded and comparable,
- and improving observability (structured logs, cost evolution, solver statistics, and exported MIDI scores), to be able to study as well feasibility and optimization

The ablation process consisted in rerunning the exact same generation case while deactivating one constraint at a time, and observe whether feasibility is recovered or runtime improves significantly. In practice, we considered ablation successful when it either restored feasibility (first solution found), produced a clear runtime reduction under the same budget, or significantly improved the optimization curve (lower cost sooner and/or lower final best cost). This isolates feasibility blockers and performance bottlenecks without requiring ad-hoc code edits.

When a rule was musically important but repeatedly appeared as a feasibility/performance blocker, we considered a **relaxed version** as an alternative: instead of forbidding violations, count them as penalties. This preserves the intent of the rule while allowing generation to continue on hard cases, and makes violations explicit and minimized rather than hidden.

1.2.3 Musical review and external feedback

Once generation became stable, we generated larger batches of counterpoints and performed a first-pass musical review at two levels:

- **MIDI listening** to detect obvious artifacts quickly (flat lines, repetitive patterns, jarring dissonances).
- **Score inspection** (e.g., via MuseScore) to confirm which structural pattern causes the perceived issue and whether it corresponds to a missing rule, a wrong scope, or a missing preference cost.

This review stage was not meant to replace a full musicological analysis, but to identify low-cost solver behaviours that remain feasible but are musically weak (especially

repetition patterns in multi-voice contexts), and issues such as unexpected chromaticism when borrow mode is enabled or when scale inference is ambiguous.

At the end of the semester, we benefited from several review sessions with **Karim Haddad** (professional musician at IRCAM). We sent him selected MIDI files and scores, including both typical generations and problematic examples. His feedback was especially useful to:

- detect musically unacceptable solver-produced patterns and explain the underlying flaw,
- prioritize which issues matter most perceptually (what should be fixed first),
- validate the musical relevance of newly added constraints and stricter multi-voice variants.

Where possible, rule additions were cross-checked against counterpoint treatises beyond Fux (e.g., Gallon–Bitsch[2]) to ensure changes are musically justified and not only ad-hoc fixes.

1.2.4 Procedure for implementing a change

When a new rule or modification seemed necessary, we followed a consistent procedure to avoid duplicating logic or introducing inconsistent modelling:

1. **Check whether the rule/mechanism already exists.** In some cases, the intended musical principle was already implemented but incorrectly applied, incorrectly scoped, or inactive in some configurations.
2. **Check whether it belongs to an existing mechanism.** If the behaviour was a natural extension of an existing rule family (or a missing case of an existing constraint), we preferred to integrate it there rather than creating a new independent function or variable.
3. **Otherwise, introduce the change with consistent structure.** When a new constraint was required, we implemented it in a style consistent with the existing constraint library: same naming convention, same scoping logic (species/voice-count conditions), and careful separation between feasibility rules (hard constraints), preferences (costs), and relaxed variants (penalized violations).

On the structural side (search and codebase organization), changes followed the same philosophy: prefer small, measurable modifications (branching heuristics, engine selection, stop criteria, output and logging) and keep the production pipeline predictable, while leaving more exploratory strategies to benchmarking tools.

1.2.5 Configuration as a reproducible artifact

A deliberate methodological choice is the externalization of tunable parameters into versioned CSV files (e.g., `presets.csv`, `cantus_firmus.csv`), loaded at runtime by the solver. This separates musical choices from solver code: a generation campaign can be described by a configuration manifest, reviewed independently of implementation details, and rerun consistently after code changes. It also makes comparisons meaningful, since differences in behaviour can be attributed either to code modifications or to explicit configuration edits.

1.2.6 Limits of the approach

This empirical methodology has limitations. First, parameter and rule adjustments are local decisions guided by specific observed failure modes; they do not guarantee any global optimum in the space of possible configurations, and they can introduce regressions on scenarios that were not re-tested after each change. Second, the solver does not guarantee a solution within a fixed time budget for all voice/species combinations; time limits were chosen pragmatically, and some configurations remain intractable within those bounds. Finally, although we improved parts of the search procedure, achieving consistently optimal convergence through principled restarts, deeper propagation strengthening, or fully systematic heuristic derivation remains a natural direction for future work.

1.3 Roadmap

This thesis is organized as follows:

- **Chapter 2 (Background)** introduces the musical and technical foundations required for the rest of the document: species counterpoint and Fux terminology, the representation choices used in FuxCP (discrete time grid, MIDI/interval encoding), and a short constraint-programming background.
- **Chapter 3 (FuxCP Architecture)** describes the project as a generation system: repository organization, resolution pipeline (entry points, problem creation, solver execution, export), and the cost priority ordering mechanism.
- **Chapter 4 (Constraint system)** explains how Fux rules are implemented as a structured constraint catalogue without listing every rule: decision/derived arrays used by constraints, posting layers (part / cross-voice / stratum), naming and scoping conventions, and the distinction between hard constraints, soft costs, and relaxed variants.

- **Chapter 5 (Structural improvements)** presents the contributions that make FuxCP robust and usable as a generator: solver control and optimisation strategy (branching, engine choice, pruning, objective handling), codebase restructuration (Generations engine, CLI/API, loaders, export), and smaller technical fixes/utilities.
- **Chapter 6 (Constraints improvements)** presents the model-level contributions: new constraints addressing recurrent musical flaws (including multi-voice strengthening motivated by expert review), relaxed versions of key constraints for hard cases, and small but important fixes to existing rules.
- **Chapter 7 (Evaluation)** evaluates the resulting system along three difficulty axes (voices, species, register), reports both computational effort and solution quality under fixed budgets, and discusses the main trade-offs and limitations of the current model.
- **Chapter 8 (Conclusion)** summarizes the contributions and results, then outlines the most promising directions for future work.

The appendices then provide additional material : interfaces and configuration formats, code excerpts, and generated examples.

Chapter 2

Background

Chapter summary. This chapter outlines the history of the FuxCP solver from its inception to the present day, detailing year by year all the improvements made by previous students. The basic theoretical concepts (musical and constraint-related) are reviewed here to provide a better understanding of the work carried out. Finally, the logic behind a traditional constraint solver system and its various components are discussed.

Roadmap.

- Section 2.1 explains why the FuxCP solver was created and traces its development over time, as well as the people involved.
- Section 2.2 defines the key musical concepts that Fux uses in his treatise and demonstrates how they are combined in order to outline his rules.
- Section 2.3 explores the key principles of constraint-based programming and provides an overview of how musical rules are translated into this field.

2.1 Previous work

2.1.1 FuxCP

FuxCP is a constraint-programming-based tool designed to generate (and validate) species counterpoints in the style of Johann Joseph Fux. The central idea is to encode Fux-inspired rules as constraints over discrete musical variables (pitches on a rhythmic grid, and to guide generation through a cost model that represents musical preferences among the many valid solutions.

The project was initiated as a collaboration between UCLouvain and IRCAM, with the goal of supporting computer-aided composition workflows. Historically, it was

conceived to assist composers (including users without programming expertise) by turning species-specific rules into explicit constraints that the computer can enforce and explore systematically. In this context, constraint programming is a particularly good fit: rules remain transparent and traceable, preferences can be represented as soft constraints or costs, and solutions can be inspected and steered instead of being produced by a black-box generator. FuxCP is implemented on top of the Gecode constraint solver [4].

2.1.2 Project lineage and context

Historically, FuxCP grew alongside IRCAM's OpenMusic (OM), a visual programming environment for computer-assisted composition [14]. The project evolved through several theses:

- In 2022, Sprockeels developed *Melodizer*, introducing the bridge between OpenMusic (Lisp) and Gecode (C++), and illustrating interactive constraint-based composition where mandatory rules can be combined with optional preferences [5].
- In 2023, Wafflard introduced FuxCP properly for two-voice species, mapping Fux's rules to constraints and exposing them through OM for composer-facing use [6].
- In 2024, Lamotte generalized the model to three voices, formalizing interactions with the lowest voice and introducing strata, while clarifying the role of soft constraints for preserving stylistic nuance [8].
- Later in 2025, Cleenewerk & de Patoul unified the accumulated rule sets, extended FuxCP to four voices, and migrated the implementation to an object-oriented C++ codebase on top of Gecode [7].
- Finally, in 2026, Tom Lai introduced a figure-driven validation methodology based on Minimal Unsatisfiable Subsets (MUS) to diagnose and resolve tensions between the formalism, Fux's examples, and auxiliary sources. It also extended and corrected the Gecode-based solver, adding a modular activation/deactivation mechanism that enables controlled experimentation with subsets of rules [9].

OpenMusic is no longer used in the current line of work. FuxCP is now maintained as a standalone C++ library with a stable API, which keeps the solver easier to test and integrate. As said in 1.1.3, a native graphical interface is being developed separately by Chris Bakashika [13].

2.1.3 State of the project before this thesis

Before our work, FuxCP already contained a large constraint catalog spanning multiple species and voice counts, and was able to validate many reference examples. The

codebase also already supported an optimization-oriented workflow (through an ordered cost representation used to compare solutions) and provided interfaces for running generations and exporting results.

However, the system had not yet been fully stabilized and tuned as a generator across the whole configuration space. In particular, some configurations were practically unusable due to optimization stability issues and search performance limitations (especially in higher species and multi-voice contexts). These limitations motivated the contributions of this thesis: improving FuxCP specifically as a *generation* system, while keeping the same overall modeling philosophy and rule base.

2.2 Species counterpoint and Fux terminology

This work is based on the especies counterpoint presented by Johann Joseph Fux in Gradus ad Parnassum (1725)[1]. The treatise is written as a dialogue between a master (Aloys) and a student (Joseph) and introduces counterpoint progressively, starting with two voices and then extending to three and four voices. Counterpoint construction starts from a given melody called the cantus firmus, and additional voices (counterpoints) are written according to a set of rules.

2.2.1 Global approach

Fux's approach is organized into five successive "species", which are mainly recognizable by their rhythmic structure[3] :

- First species: whole note against whole note (note-against-note).
- Second species: two half-notes against one cantus firmus whole note.
- Third species: four quarter-notes against one cantus firmus whole note.
- Fourth species: syncopation/ligature (conceptually close to first species, but shifted by half a measure).
- Fifth species: florid counterpoint, combining elements of the previous species.

Throughout the treatise, Fux uses terminology that is useful to define explicitly because it affects how rules are interpreted and implemented. In particular, he uses thesis (downbeat) and arsis (upbeat). In a 4/4 context, these correspond respectively to the first and third beat. He also uses diminution to describe the filling of a third by a diatonic passing note. Finally, he distinguishes perfect consonances (unisons, fifths, octaves) and imperfect consonances (thirds, sixths).

At a high level, Fux's rules can be read as a combination of[3] :

- harmonic constraints : which vertical intervals are allowed/preferred at given positions
- melodic constraints : how each voice moves
- motion constraints : how two voices move together

2.2.2 Main concepts[3]

Throughout this thesis, we use the following terms consistently.

- **Cantus firmus (CF):** the fixed reference melody (one note per measure in Fux's pedagogical setting), against which one or more counterpoint voices are written.
- **Counterpoint voice / part:** any generated melodic line written against the CF (and possibly against other counterpoint voices in multi-voice settings).
- **Measure positions:** because Fux's species are defined rhythmically, we refer to positions within a measure (e.g., strong/weak beats) rather than absolute durations.

Intervals and consonance vocabulary. Most rules are phrased in terms of *intervals*. We distinguish:

- **Harmonic (vertical) interval:** the interval between two notes sounding at the same time [11] (e.g., a counterpoint note against the CF note at that position).
- **Melodic (horizontal) interval:** the interval between consecutive notes inside the same voice.
- **Perfect vs imperfect consonances:** perfect consonances are unison, fifth, octave (and their compounds); imperfect consonances are thirds and sixths (and their compounds). These categories matter because many motion restrictions and cadential rules treat perfect consonances as structurally stronger and therefore more constrained.
- **Dissonance (in species contexts):** in Fux's approach, dissonances are tightly controlled and typically only allowed in specific rhythmic contexts (notably as passing events in 2nd/3rd species and as prepared/resolved events in 4th species).

Voice-leading and motion types. When describing the relation between two voices across time, we use the standard motion vocabulary:

- **Parallel motion:** both voices move in the same direction by the same interval class.

- **Similar (direct) motion:** both voices move in the same direction, but not necessarily by the same interval.
- **Contrary motion:** voices move in opposite directions.
- **Oblique motion:** one voice stays while the other moves.

This vocabulary is required for most rules, such as avoiding parallels (especially on perfect consonances) and controlling direct motion into perfect consonances.

2.2.3 Other useful terminologies

Many rule sets treat the end of the piece specially. We refer to:

- **Final:** the last note (often the tonic in the pedagogical setting).
- **Penultimate:** the note immediately before the final, where cadential preparation rules apply.

Species-specific terms used later. Some terms appear mainly in specific species and are referenced explicitly in the codebase and cost model:

- **Thesis / arsis:** strong vs weak beat positions (notably in 2nd species).
- **Syncopation / ligature (4th species):** a tied note structure where dissonance may appear on the strong beat provided it is prepared and resolved by step.
- **Passing motion (diminution):** a stepwise filling of a consonant span (typically a third) by an intermediate step, used to justify controlled dissonances on weak positions.
- **Cambiata (5th species):** a characteristic ornamental figure that introduces a controlled dissonance pattern; in FuxCP it is handled explicitly as a 5th-species feature.
- **Pitch class:** the octave-invariant representation of a pitch, i.e. a MIDI pitch modulo 12. It is used when reasoning about tonal membership and chord membership independently of octave placement[15].
- **Interval class:** the octave-invariant representation of an interval (typically absolute interval modulo 12). Consonance/dissonance sets in the model are usually expressed at the level of interval classes[15].
- **Borrowed note:** a pitch class that does not belong to the diatonic pitch-class set associated with the cantus firmus scale/mode label (`cf_scale`). Depending on configuration (`borrow_mode`), such notes may be forbidden or allowed but penalized through costs[7, 8].

Multi-voice terminology. In three- and four-voice counterpoint, we often need to refer to the vertical ordering of voices at a given time position. We therefore use:

- **Inner voices:** any voices between outer voices, i.e that is not the lowest or the highest voice.
- **Stratum:** an abstraction used in FuxCP to represent vertical layers of voices independently of which concrete part currently occupies them (explained in the Architecture chapter)[7, 8].
- **Pitch window / pitch band / register:** the allowed pitch interval (as a bound range) assigned to a generated voice in the solver. In FuxCP, this is controlled by the parameter `v_type`: it defines a fixed two-octave window centred on `cf[0] + 6·v_type`, where `cf[0]` is the first cantus firmus note in MIDI. This window is a modelling constraint (domain restriction), not a search heuristic.
- **Register overlap:** a situation where two voices are assigned pitch windows that are identical or very close, so that they compete in the same pitch band. In practice, this tends to increase search difficulty and can prevent the emergence of clear suspension chains in fourth and fifth species, which require the suspended voice to sit clearly above a moving bass.

For a full pedagogical discussion of these terms in Fux's original presentation, we refer the reader to [1, 3].

2.3 Representation choices in FuxCP

2.3.1 Discrete time representation

A core modeling choice in FuxCP is to represent counterpoint on a discrete time grid. The cantus firmus is defined as one note per measure, and species counterpoints are represented by placing notes on a finite set of positions per measure (up to four positions when quarter-note resolution is required). This matches the species progressions described by Fux and makes it possible to express rhythmic and harmonic rules on specific beats, like thesis vs arsis.

In the FuxCP implementation, note variables are stored in different arrays, depending on the context. Two of them are the main ones, used across all the codebase : the **notes** array, and the specific **SpeciesNotesCp** arrays[6, 7].

- The **notes** array is the one on which branching occurs. It contains 4 variables per measure, except for the last measure, which contains only one. Every other variables depend on this array.

- The **SpeciesNotesCp** arrays are reductions of the notes array, dependent on each species needs. For example, *secondSpeciesNotesCp* contains only the first and the third variables of each measures, as the second species only plays on thesis and arsis.

This time discretization is the basis for how later chapters define which constraints apply to which positions.

2.3.2 Musical notation and MIDI

To make the constraint model concrete and interoperable with common music tools, pitches are represented as MIDI note numbers, i.e., integers in the standard MIDI note range. In practice, FuxCP uses MIDI integers as the domain of pitch variables and exports solutions in a form that can be rendered and listened to. A complete mapping of note names to MIDI values is provided in the reference material used by the project.

Most rules in species counterpoint depend on intervals rather than absolute pitches. For this reason, FuxCP derives interval variables from pitch variables. A typical example is the harmonic interval between a counterpoint note and the cantus firmus note sounding at the same time. The model often uses:

- an absolute interval (in semitones, potentially larger than an octave), and
- an interval modulo 12 (to reason about interval classes within the octave).

Using modulo 12 matches the common "12 semitones per octave" representation and ensures that the same interval class is recognized across octaves (ex: C6–G6 and C6–G7).

These representation choices (discrete time grid, MIDI pitches and derived intervals) are the foundation that allows musical rules from Fux to be expressed later as constraints and evaluated through costs[5, 6].

2.4 Constraint programming essentials

2.4.1 Constraint satisfaction problems

Definition 2.1 (CSP)**.** A *constraint satisfaction problem* (CSP) is defined by a triple $\langle X, D, C \rangle$ where X is a set of decision variables, D associates to each variable $x \in X$ a finite domain $D(x)$ of possible values, and C is a set of constraints restricting the combinations of values that variables may take. A *solution* is an assignment of a value in $D(x)$ to every $x \in X$ such that all constraints in C are satisfied[10].

CSP view of counterpoint. In FuxCP, the variables X are mostly pitch variables (MIDI notes) arranged on a discrete rhythmic grid. Domains D_i are finite pitch ranges determined by the cantus firmus and the chosen register of each generated voice. Constraints C encode the musical rule set (allowed consonances, motion restrictions, species-specific rhythmic structure, etc.). A solution is therefore a complete assignment of all note variables that satisfies the active rule set.

2.4.2 Constraint optimization and lexicographic objectives

Definition 2.2 (COP)**.** A *constraint optimisation problem* (COP) is a CSP $\langle X, D, C \rangle$ together with an objective function f that assigns a value to each feasible assignment. The goal is to find a solution that satisfies all constraints and minimises (or maximises) f[10].

Why optimisation is needed. The rule set of species counterpoint typically admits many feasible solutions, even for a fixed cantus firmus. Moreover, a formally valid solution may still be musically weak (excessive repetitions, awkward register choices, poor harmonic variety). FuxCP therefore treats generation as optimisation: constraints define feasibility, while cost components encode preferences that allow the solver to rank valid counterpoints.

Lexicographic objectives. Rather than aggregating all preferences into a single scalar, FuxCP primarily uses a lexicographic cost vector. This matches the idea that some defects should never be traded for small gains elsewhere. Concretely, cost components are grouped and ordered by priority levels (the `importance` vector in presets), and the solver minimises the highest-priority group first; lower-priority costs are only minimised once higher priorities cannot be improved further.

2.4.3 Propagation and search

Constraint programming solvers alternate two mechanisms. *Propagation* repeatedly applies constraints to remove values from variable domains that cannot appear in any solution; when a domain becomes empty the current partial assignment fails. *Search* explores remaining alternatives by making decisions (branching), building a search tree whose nodes are partial assignments and whose leaves are complete feasible solutions[4].

The practical efficiency of CP comes from the interaction of both : stronger propagation reduces the size of the search tree, and a good branching order reaches feasible (and good) solutions sooner.

Branching A branching strategy in `Gecode` is declared through a call to `branch(*this, vars, varSel, valSel)`. The first parameter is the array of variables on which the solver must take decisions, the second specifies in which order to pick the next variable to instantiate (`INT_VAR_*`), and the third specifies which value to try first in that variable's domain (`INT_VAL_*`). A single problem may stack several successive calls to `branch`: each layer takes over once the previous one has finished assigning its own variables. The order of these layers is therefore as important as the choice of heuristics within each layer.

In FuxCP, branching layers typically commit pitch decisions first (the main musical degrees of freedom), then species/ornament structure when relevant, and finally objective-related variables to ensure a fully-defined cost for optimisation.

Search engines. Gecode provides multiple search engines. In this thesis, two are conceptually central: `DFS` (depth-first search), which enumerates feasible solutions without optimisation guarantees, and `BAB` (branch-and-bound), which searches for solutions of strictly improving cost and can be stopped at any time while still returning the best solution found so far. At the moment, FuxCP relies exclusively on `BAB` for generation, since the goal is not only feasibility but also solution quality.

Other gecode concepts. For further details on propagation, branching heuristics, and Gecode's search engines, we refer to the official Gecode manual[4].

Chapter 3

FuxCP Architecture

Chapter summary. This chapter presents the concrete software architecture of FuxCP as it is used in this thesis. It describes how the project is organized into a generation/orchestration layer, a constraint-programming model layer (problems, parts, strata), and supporting infrastructure (configuration loading and MIDI export). It also explains the end-to-end generation pipeline from entry points (CLI/API) to solver execution and exported outputs, and clarifies how configuration files and presets are resolved into a single `GenerationCase`. Finally, it introduces the cost priority ordering mechanism that shapes solution quality by defining how cost families are compared during optimisation.

Roadmap.

- Section 3.1 gives a high-level map of the repository, describing the role of the main modules (Generations, Utilities, ConfigLoader, Midi) and of the modelling core (CounterpointProblems and Parts).
- Section 3.2 then details the resolution pipeline: it covers the generation entry points, how a run is prepared and dispatched to `create_problem(...)`, how the search engine is created and controlled (timeout, stagnation, verbosity, output saving), and how results are returned through `GenerationResult` and optionally exported as MIDI and reports.
- Section B.1 explains how external configuration files (cantus firmi, presets, campaigns) and command-line/API overrides are merged into `GenerationCase`, and it documents the key configuration and result structures used throughout the codebase.
- Finally, Section 3.3 explains the semantics and implementation of cost priority ordering (importance vector), which defines the lexicographic structure used to compare solutions.

3.1 Overview

Global implementation files :

- **Generations** : The orchestration layer for actual generation. It handles generation cases, presets and parameter loading, benchmark control, timeout / stagnation stop logic, CSV output, MIDI export, and practical search launching.
- **Utilities** : The shared toolbox of the whole project. It defines global enums, constants, scales, interval/chord data, search-heuristic names, note / mode / voice naming helpers, scale-detection helpers for a cantus firmus, vector-set operations (union, intersection, difference), string/debug conversions for Gecode variables and arrays, MIDI/note naming helpers, and formatting helpers for solver statistics and logs.
- **Configloader** : The helper to load configurations of counterpoints presets, for keeping track of sensitivity testing of launching predetermined campaigns of generations.
- **Midi** : The helper to generated MIDI files from solver's outputs.

CounterpointProblems : Implementations of the high-level problem classes and construction helpers.

- **CounterpointProblem** : The root of the Constraints Programming Problem. It stores and coordinates the cantus firmus, the generated parts, the strata, local and global costs, final optimization arrays, string/debug output, and the global objective returned by cost().
- **Two-, Three-, and FourVoiceCounterpoint** : These classes decide how many generated counterpoints exist, which species are instantiated, how many strata are needed, which inter-voice rules are posted, and how local costs are unified into the final objective.
- **CounterpointUtils** : This is the main dispatch / factory layer. It is responsible for creating the correct species object from a species identifier, and the correct global problem class from the number of voices.

Parts : Implementations of the more precise problem classes.

- **Voice**: The most generic musical-line abstraction in the project. It owns the lowest-level variable arrays that are reused everywhere, such as *notes*, *h_intervals*, *m_intervals_brut* or *motions*

- **Part** : A musically meaningful generated line. Extends Voice and adds species identity, cost arrays, cost names and combined-cost names, helper arrays used by the various species, common getters for branching notes, melodic intervals, harmonic intervals, and costs.
- **CantusFirmus** : The fixed musical line given as input. Extends Part inside the problem, but it is not searched the same way as generated counterpoints.
- **Species classes** (FirstSpeciesCounterpoint to FifthSpeciesCounterpoint) : Responsible for defining species-specific arrays, rules, views of branching notes, and melodic and harmonic helper arrays. Extends Part, so the common voice/cost machinery is centralized while the rhythmic and rule-specific logic stays local to each species.
- **Stratum** : Vertical layer of the current piece. Extends Voice. In multi-voice counterpoint, the code distinguishes the lowest stratum, intermediate upper strata, and the highest stratum.
- **constraints** : The central reusable constraints library of the project. It contains all the harmonic, melodic, motion / progression, and stratum-level rules. This is the main file to edit when adding or correcting a rule.

Tests : Test entrypoint and test suites:

- **main** : The command-line test entrypoint. It checks the global constraint registry consistency, dispatches either to FigureTests (figs, filtered by number of voices / species / MUS / quick modes) or to the lower-level FuxTest routines, and also still contains a small manual experimentation block for directly constructing and solving a problem from hardcoded parameters.
- **fuxTest** : The low-level testing framework for individual constraints and targeted configurations. It initializes standard test parameters, builds problems from small hand-crafted setups, dispatches named tests such as 1H1, 2H2, etc., and is mainly used to check whether specific rules, species configurations, or debugging scenarios behave correctly.
- **figureTests**: The high-level regression / validation suite based on Fux's reference examples. It stores complete figure configurations (cantus firmus, expected counterpoint, species layout, parameters), rebuilds the corresponding problems, checks whether known examples are accepted, can enumerate solutions for a figure, and also includes MUS / unsat-diagnosis helpers to isolate conflicting constraints.

3.2 Resolution pipeline

This section describes the end-to-end execution flow used to generate counterpoints, from raw user inputs (CLI arguments or presets) to exported MIDI and benchmark reports. The pipeline is implemented around a single reusable entrypoint, `Generations::run_generation_case(...)`; both the CLI and the programmatic API build a `GenerationCase` and delegate to this same engine function.

3.2.1 Entry points

The generation pipeline can be triggered from two main entry points: the command-line interface (`main.cpp`) and the programmatic API (`GenerationsAPI`). In both cases, the first objective is to build an in-memory *generation case* structure (`GenerationCase`), which acts as the unified container for all parameters required by the shared generation engine.

The parsing phase performs the following operations:

- **CLI argument parsing.** The CLI parses positional arguments (number of voices and species list) and optional flags (cantus selection mode, preset name, time limits, output options, objective mode, optional voice-type overrides). In particular, it supports both cantus input modes: `-c CF_ID` (dataset cantus firmus) and `-cf-notes n1,n2,...` (direct MIDI cantus firmus).
- **Mode dispatching (list / campaign / single run).** Before constructing a generation case, the CLI can switch to non-generation modes: listing available cantus firmi, presets, or campaigns, and campaign execution (`-campaign <id>`). Campaign mode iterates over all configurations defined in `campaigns.csv` and constructs one `GenerationCase` per (campaign row × cantus id) combination.
- **API input mapping.** The programmatic API accepts a typed `GenerationInput` structure (cantus MIDI notes, species list, voice types, and optional overrides). It is converted into a `GenerationCase`, then delegated to the same resolution and execution pipeline as the CLI.

During configuration parsing, the pipeline imports preset defaults and computes the main derived constants, filling the remaining fields : cantus metadata and preset parameter arrays.

This preparation step performs three main kinds of work.

- **Cantus firmus and structural constants.** The cantus firmus is loaded as a vector of MIDI values and becomes the fixed reference voice of the problem. From its length, the pipeline derives the structural constants used everywhere in the

CSP (e.g., the number of measures, and therefore the size of note arrays for each species).

- **Optimisation setup (cost factors, ordering, and modes).** The cost-related arrays (weights, importance ranking, borrow mode, objective mode) are not applied as constraints yet, but they define how the future CSP will evaluate solutions:

  1. which families of costs are active and how they are weighted,
  2. how cost components should be grouped / prioritised,
  3. whether optimisation is purely lexicographic (`OBJECTIVE_LEX`) or uses an aggregated objective prepended to the returned cost vector.

  This ensures that when the problem is created, all voices can contribute to a single coherent global cost model (rather than independent per-voice scoring).

- **Run control and output preparation.** Finally, runtime limits (timeout and stagnation thresholds) are converted into the values expected by the search engine setup, and the output directory layout is resolved (root folder, optional subfolder, naming policy). This guarantees that a run is reproducible and that its artifacts (MIDI + logs) are grouped deterministically.

At the end of this stage, the pipeline has a complete *problem blueprint*: fixed cantus firmus data, a fully specified set of counterpoint voices (species and ranges), and an optimisation configuration that will be used when posting constraints and defining the objective in `create_problem(...)`.

3.2.2 Call `create_problem(...)`

The function `create_problem(...)` is the unified entry point that constructs the appropriate CSP class depending on the number of counterpoints requested. Since the cantus firmus is always present, this dispatch is driven by `spList.size()`:

- `spList.size() == 1` ⇒ `TwoVoiceCounterpoint`
- `spList.size() == 2` ⇒ `ThreeVoiceCounterpoint`
- `spList.size() == 3` ⇒ `FourVoiceCounterpoint`

Inside each constructor, the model is built in a mostly uniform way:

- **Instantiate parts.** The cantus firmus is instantiated as a dedicated `CantusFirmus` object. Each counterpoint voice is instantiated as a `Part` whose concrete class depends on its species (first to fifth species).

- **Create strata.** Strata objects (lowest stratum and upper strata depending on the number of voices) are created to represent the vertical ordering of simultaneously sounding notes. These strata are later used by harmonic rules defined relative to the lowest voice.
- **Post constraints.** The constructor posts:
  - general constraints (shared across configurations),
  - species-specific constraints (activated depending on the species),
  - inter-voice constraints (harmony and motion between voices),
  - optional constraints controlled by an activation mask (useful for ablation/debugging).
- **Unify and order costs.** Each part produces cost arrays; these are unified into a single cost vector for the problem. The `orderCosts()` procedure then groups/sorts cost factors according to the `importance` vector and prepares the objective variables (e.g. `globalCost`, `objectiveCostSum`, and the final vector returned by `cost()`).
- **Define branching.** The model constructor also defines the branching strategy: it typically branches on musical decision variables (notes, species arrays, syncopation-related booleans, etc.), and finally branches on the `cost()` vector to ensure all cost variables are fixed (this avoids unassigned-cost issues during BAB refinement).

At the end of `create_problem(...)`, the returned object is a complete `IntLexMinimizeSpace`: all variables exist, all constraints are posted, the objective is defined, and the branching strategy is registered.

3.2.3 Create the search engine and run the solver

After the problem is constructed, the pipeline instantiates a Gecode search engine and runs it through an iterative `next()` loop. Beyond raw execution, this stage is also where *solver control* happens: the user can decide how long the solver is allowed to run, how to stop when progress stalls, and what kind of solution stream is expected.

Search modes: next solution vs best solution. Two search behaviours are supported:

- **Enumeration ("next solution")**: a `DFS` engine returns feasible solutions one by one. This mode is useful for debugging, figure validation, or exploring the solution space without optimisation.

- **Optimisation ("best solution")**: a `BAB` engine returns a sequence of improving incumbents. Each call to `next()` produces either a strictly better solution (according to the objective vector) or `nullptr` when the search stops.

In the generation pipeline, `BAB` is the default because the goal is not only to find a valid counterpoint, but to refine it towards lower cost.

Solver control parameters exposed to the user. The pipeline exposes several user-facing controls through `GenerationCase` (CLI/API):

- **Timeout**: a hard maximum runtime that bounds the total search time.
- **Stagnation window**: a maximum duration without improvement of the best cost, after which the search is stopped.
- **Verbosity**: enables live progress logs (solutions found, improvements, costs, statistics).
- **Output saving**: enables/disables on-disk artefacts (MIDI, reports), while keeping the in-memory `GenerationResult` always available.

The engine is therefore controllable both for production usage (predictable bounded runs) and for diagnosis (more verbose runs, enumeration mode, etc.).

Stagnation: definition and implementation. Stagnation does *not* mean that no better solution exists: it only means that, under the current branching and propagation dynamics, the solver has not reached an improving incumbent for a while. In our pipeline, stagnation is implemented as a stop criterion that tracks the time since the last strict improvement of the best objective value. The timer is reset whenever an improvement is found; if no improvement occurs for longer than `stagnation_ms`, the stop condition triggers and the solver returns the best solution found so far.

Discussion about stagnation. Empirically, we observed two different behaviours:

- **Short stagnations** (brief plateaus) can occur before meaningful improvements: the solver may explore a difficult sub-tree and then suddenly discover a better region of the search space.
- **Long stagnations** (several minutes without improvement) were, in our experiments, a strong indicator of diminishing returns. In particular, on multiple configurations we compared the best solution obtained after a moderate budget (e.g., around 10–15 minutes) to the best solution obtained after letting the solver run for hours; once the search entered a long stagnating regime, further improvements were typically negligible relative to the additional runtime.

For that reason, stagnation is used as a practical stopping rule for generation: it avoids spending large amounts of time in regions where the solver is no longer producing meaningful progress, while still preserving the best incumbent found. Of course, it remains optional : setting it as 0 disables it.

Iterations, logging, and collected results. The run is performed as a loop over `next()`:

- each returned solution is evaluated (scalar cost and/or full lexicographic vector),
- improvements are detected and stored (best solution replacement),
- optional checkpoints can be recorded for reporting (time series of best cost),
- final `Search::Statistics` are collected (nodes, fails, propagations, depth, etc.).

This produces structured benchmark data: time to first solution, time to last improvement, number of solutions visited, number of improvements, final best cost, and termination reason (exhaustive, timeout, or stagnation).

Branching as an additional control leverage. Finally, solver behaviour depends strongly on branching choices: they define which variables are fixed early and therefore which propagation happens first. In the current codebase, branching is defined during problem construction (it is not a runtime flag exposed to end users), but it is a first-class part of the solver control story because it directly shapes stagnation and convergence speed. We therefore analyse and improve branching choices explicitly in Chapter 5.

3.2.4 Export the final data

Once the solver terminates, the pipeline returns a `GenerationResult` structure as its primary output. This object is always produced, whether the generation is triggered from the CLI or from the programmatic API, and it contains the search outcome and statistics, the best solution found (when any), and optional paths to exported artifacts when file export is enabled.

Two booleans control side effects:

- **`save_outputs`** decides whether files (MIDI, `.txt`, `.csv`, CF description files) are written to disk. When disabled, the generation remains purely in-memory and only returns `GenerationResult`.
- **`verbose`** controls live logging during the search and the final console summary, without affecting the returned `GenerationResult`.

Only errors are *always* logged and printed, no matter the value of those booleans.

In-memory output. Regardless of file export, `GenerationResult` contains the key information needed by downstream users : a `success` flag, a termination reason, timing metrics (time to first solution, last improvement, total time), Gecode search statistics, and the best solution in two forms: the raw concatenated solution vector (`full_solution`) and a voice-split representation (`generated_voices`). It also stores the resolved cantus firmus (notes and metadata), which is useful because the cantus can be provided either from the dataset or directly through the API.

MIDI export. When at least one solution is found, the solver's `full_solution` is a concatenation of branching notes for each counterpoint voice. The export step splits this vector back into per-voice note sequences using species-dependent sizes, stores them in `generated_voices`, and (when `save_outputs` is enabled) writes a multi-voice MIDI file containing:

- the cantus firmus (as reference),
- each generated counterpoint voice, tagged with its species.

The resulting MIDI path is recorded in `GenerationResult.midi_path`.

Text and CSV reports. When `save_outputs` is enabled, the pipeline also writes:

- a human-readable `.txt` report (configuration recap + search metrics + optional evolution of the cost vector),
- a `.csv` summary (single-line metrics + optional sections for improvements/checkpoints),
- a cantus firmus description file (MIDI values + note names + metadata).

The report paths are recorded in `GenerationResult.txt_path` and `GenerationResult.csv_path`.

Failure case. If no solution is found, `GenerationResult.success` is set to `false`. An error report is generated with the full configuration and termination reason, and its path is recorded in `GenerationResult.error_txt_path`. This keeps failed runs reproducible for debugging and ablation experiments.

3.3 Costs priority ordering

FuxCP uses a multi-criteria cost model: a generated counterpoint is evaluated by a set of cost components, each reflecting a musical preference or a penalised pattern. Rather

than reducing these components to a single scalar immediately, the solver can treat them as a *prioritised* vector. This priority structure is controlled by the `importance` vector, provided by presets and stored in `GenerationCase` as a list of 14 unique ranks (from 1 to 14) associated with fixed cost families (borrow, fifth, octave, successive perfect consonances, variety, triad, direct motion, motion, penult, cambiata, triad3, m2, syncopation, melodic).

Semantics. The importance rank defines a lexicographic priority level: rank 1 is the highest priority and is minimised first; rank 14 is only minimised once doing so does not degrade any higher-priority level. Conceptually, this matches the "Fux-like" idea that some musical defects are unacceptable compared to others, even if the total sum of penalties would be smaller. A practical consequence is that changing the importance ordering can significantly change the musical character of the generated counterpoints, even with identical constraint sets and identical cost weights.

How ordering is implemented. Internally, cost components are grouped by their importance level and then combined *within each level* (typically by summation) to form one aggregated value per priority level. The final objective returned to the solver is therefore an ordered list of these per-level aggregates (possibly with additional prepended objectives depending on `objectiveMode`, as described in Section *Cost Function Refinements*). Importantly, not all cost families exist in all configurations: some costs are species-specific (e.g., syncopation-related costs only make sense when at least one fourth-species voice is present). Missing costs are simply skipped when building the ordered objective vector.

Default ordering and musical rationale. In this thesis, the priority mechanism itself is not modified: we keep the existing design in which the cost families are ordered through the preset-provided importance ranking and then used as the primary lexicographic structure for optimisation. Since the complete default ordering and its detailed musical rationale have already been documented extensively in previous FuxCP theses, we do not duplicate that discussion here; the reader is referred in particular to the description of cost ordering and importance-based grouping in [8] and to the multi-voice formalisation context in [7]. The key point for our work is that this ordering remains the main lever that determines *which musical compromises are acceptable* during search: it defines what the solver is willing to sacrifice at low priority in order to satisfy higher-priority musical criteria.

Chapter 4

Constraints system

Chapter summary. This chapter explains how the FuxCP rule base is implemented as a constraint system. We first clarify the role of the constraint model, then define the core musical arrays and derived variables on which rules are expressed (notes grid, species views, intervals, motions, helper flags, and strata). We then describe where constraints are posted in the codebase (part-level, cross-voice, and stratum-level), how the catalogue is organized and named, and how the model distinguishes hard constraints, soft costs, and relaxed variants. Finally, we present the activation/ablation workflow used to debug infeasibility and performance issues, and we provide a few concrete examples showing how theoretical rules translate into actual Gecode constraints.

Roadmap.

- Section 4.1 introduces the purpose of the constraint model in FuxCP (feasibility vs. controlled experimentation) and explains why the model is organized modularly (voices/parts/strata + shared constraints library).
- Section 4.2 defines the main decision and derived arrays used by constraints (notes and indexing, species views, harmonic/melodic intervals, motion categories, structural booleans) and explains how strata provide stable multi-voice references.
- Section 4.4 describes how the rule base is organized and named (families G/H/M/P/R, scoping across species and voice counts, identifiers vs. implementation function names).
- Section 4.5 explains the distinction between hard constraints (feasibility), soft costs (preferences), and relaxed constraints (controlled weakening through penalties).
- Section 4.6 presents how the model can be controlled for diagnosis (activation masks, ablation, relaxation) without modifying constraint code.

- The final section provides worked examples linking a few representative theoretical rules to their concrete Gecode encodings and points to the appendices for the full catalogue.

4.1 Role of the constraint model in FuxCP

FuxCP models species counterpoint as a constraint satisfaction (and optimisation) problem: the unknown notes of the generated voices are represented as decision variables, and Fux-inspired rules are encoded as constraints over these variables. In that setting, the constraint system has two complementary responsibilities.

First, it defines **feasibility**: a solution must satisfy the musical rules that are considered mandatory for a given configuration (number of voices, chosen species, and register choices). This is what makes the generator a *rule-respecting* system rather than a free composition tool. Second, it induces the **quality** of the results : non-mandatory rules define costs for breaking them, sometimes depending on several contextual variables. These costs are the ones that are then minimized by the solver, deciding whether solutions are "good" or "bad".

Additionally, it supports **controlled experimentation**: because rule formalisation is complex and because some rules can be overly restrictive in practice, the system is designed so that constraints can be selectively activated/deactivated, which enables ablation and debugging workflows (e.g., identifying which rule blocks feasibility).

In practice, this constraint model is not written as one monolithic block. Instead, it is organised around reusable musical abstractions (voices, parts, strata) and a shared library of constraint functions. This modularity is essential to avoid duplicating rules across species and across voice counts, and to keep the codebase maintainable as new constraints or relaxed variants are introduced.

4.2 Core musical representations used by constraints

4.2.1 Key variable arrays used by constraints

Most constraints are expressed on a small set of shared arrays exposed by the `Voice`/`Part` layer (and, in multi-voice settings, by `Stratum`). Some of these arrays are *decision variables* (branched on), while others are *derived views* computed from decisions and used for propagation. The most important ones are:

- **`notes`** (`IntVarArray`): the core pitch decision grid for a voice.
- **`SpeciesNotesCp`** (`IntVarArgs` views): species-dependent projections of `notes` (e.g., thesis/arsis stream for second species).

- **`h_intervals`** (and a mod-12 view): harmonic interval descriptors used for consonance/dissonance membership checks.
- **`m_intervals_brut`**: raw melodic differences between consecutive notes, used to bound leap magnitudes and define stepwise predicates.
- **`motions`**: motion descriptors between two voices across a step (parallel/direct/-contrary/oblique), used by progression rules.
- **Species helper booleans** (e.g., `isDiminution`, `noSyncope`): structural flags frequently used in reified implications.
- **Strata** (`Stratum` objects, modelled as `Voice`): derived "vertical voices" used as stable references for multi-voice harmonic logic.

The rest of the rule base is largely expressed by relating these arrays through domain constraints (membership in consonance sets), arithmetic relations (interval magnitudes), and reified implications (structural flags controlling harmonic permission). The next two subsections define precisely how these arrays are indexed, derived, and used.

4.2.2 Decision variables and indexing

All constraints ultimately restrict the same underlying musical object: the sequence of pitches for each generated voice. Internally, FuxCP represents pitches as MIDI integers and discretises time on a fixed rhythmic grid aligned with measures. The modelling philosophy is to keep a small set of *decision variables* and derive from them a vocabulary of musically meaningful views (species streams, intervals, motion categories, structural flags). Most rules are expressed on these derived quantities, but because they are defined as functions of the decision variables, they propagate back to the note domains.

Primary decision array: `notes` and its indexing. For each generated `Part`, pitch decisions are stored in the `notes` array at quarter-note resolution. For each measure i (except the last), the array contains four cells:

$$i * 4 + 0 \text{ (downbeat/thesis)}, \quad i * 4 + 1, \quad i * 4 + 2 \text{ (third beat/arsis)}, \quad i * 4 + 3.$$

The last measure contains only one cell (the final note), so the total size is $4 \cdot (m-1)+1$ for a cantus firmus of m measures. This array is the main branching target: any melodic choice, harmonic choice, or ornament decision ultimately constrains these pitch variables.

A practical detail is that not all cells are meaningful for all species. In slower species, intermediate cells exist in `notes` but are not structurally used; they remain free variables

unless constrained through species-specific logic. This is why FuxCP relies heavily on species-dependent views rather than manipulating the raw `notes` grid everywhere.

Cantus firmus indexing and reference instants. The cantus firmus is stored as one note per measure (length m). Other voices store the full `notes` grid. When constraints compare a species voice against another voice in multi-voice settings, only some instants are guaranteed to be structurally meaningful across heterogeneous species. In particular:

- the **strong beat** (downbeat/thesis) is always a stable reference instant: it corresponds to index i in the cantus firmus and $i * 4$ for generated parts;
- intermediate cells $i * 4 + 1$, $i * 4 + 2$, $i * 4 + 3$ may be structurally meaningful (e.g., third species) or effectively free/unused (e.g., first species).

This distinction explains why many multi-voice checks compare weak-beat events of a fast voice against the other voices' strong-beat notes: those are the only positions that are always pinned and comparable across mixed species.

Species-dependent note views: `SpeciesNotesCp`. Because species are defined rhythmically, each species uses only a subset of the rhythmic grid. FuxCP defines species-specific projections of `notes`, referred to in the codebase as `SpeciesNotesCp` arrays. These views provide the musically correct stream of notes for each species:

- **First species:** one note per measure (thesis only).
- **Second species:** two notes per measure (thesis and arsis, typically $i * 4 + 0$ and $i * 4 + 2$).
- **Third species:** four notes per measure (all $i * 4 + 0..i * 4 + 3$ are meaningful).
- **Fourth species:** tied/suspension structure where strong-beat notes are often tied from the previous arsis and resolve mid-measure.
- **Fifth species:** mixed rhythmic patterns, supported by additional helper arrays for ornamental decisions.

Expressing constraints on these species views keeps rules close to their theoretical formulation (e.g., "arsis dissonance" or "weak-beat passing tone") while still constraining the underlying pitch decisions.

4.2.3 Other important derived variables

Derived interval arrays: harmonic and melodic. Most counterpoint rules are interval-based rather than pitch-based, so FuxCP derives interval arrays from `notes`:

- **Harmonic intervals** (`h_intervals`): vertical relations between a voice and its reference voice at a given instant. Many constraints operate on an interval-class representation $|a - b| \bmod 12$ because consonance/dissonance sets are octave-invariant. A key modelling point is that, in the baseline system, harmonic intervals stored on each `Part` are computed only against a reference layer (typically the lowest stratum). This is sufficient for two-voice counterpoint, but in 3+ voice settings it creates blind spots: dissonance against non-bass voices is not automatically captured unless additional multi-voice constraints are added (as done in the Improvements chapter).

- **Melodic intervals** (`m_intervals_brut`): raw differences between consecutive notes in the same voice. These values can be negative when moving downward, so constraints that bound leap magnitudes typically use absolute values (e.g., $|n_{t+1} - n_t| \leq k$). This raw-vs-absolute distinction matters because several "maximum melodic interval" rules are intended to be symmetric and must therefore be expressed on $|\cdot|$.

Motion categories and structural booleans. Voice-leading rules frequently depend on relative motion between voices. FuxCP therefore derives motion categories (parallel/direct/contrary/oblique) across steps, and also maintains small structural boolean arrays that are heavily used in reified implications. Two central examples are:

- `isDiminution` (or passing-tone eligibility flags), used to authorize controlled dissonances on weak positions;

- `noSyncope` (tie/syncopation structure), used notably for fourth/fifth species to represent whether a note is tied across the barline. This array is also an efficient branching target because it commits structural "tie or not" decisions early, which propagates strongly on admissible note positions.

These boolean views provide a natural encoding of "if ... then ..." musical rules (dissonance implies passing motion, perfect consonance implies restricted motion type, etc.).

4.2.4 Notes on strata

Two-voice counterpoint can be expressed directly with pairwise relations between cantus firmus and counterpoint. In three- and four-voice counterpoint, the vertical structure

becomes more complex: many rules depend on identifying *which voice is currently lowest*, or on controlling the global sonority at a given time position.

To support that, FuxCP introduces the **Stratum** abstraction: a stratum represents a vertical layer of the music at a given position and is itself modelled as a `Voice`. The code distinguishes a lowest stratum, potential intermediate strata, and a highest stratum depending on the number of voices. These strata provide a stable reference for constraints that are most naturally stated "relative to the bass" or "relative to the outer voices", and they help factor multi-voice harmonic logic without hardcoding voice indices everywhere.

4.3 Constraint posting pipeline and layering

Constraints are posted during problem construction, inside `create_problem(...)` and the constructors it dispatches to. A key architectural point is that constraints are not all posted in the same place: they are layered across objects that correspond to different musical scopes. This section describes that layering, which explains where rules belong and how they compose, without detailing each rule individually.

4.3.1 Per-part constraints (species classes)

Each generated musical line is represented as a `Part` (which extends the generic `Voice`) and is instantiated through a concrete species class (first to fifth species). These species classes are responsible for:

- creating the species-specific views of the rhythmic grid (e.g., thesis/arsis-only views for second species),
- defining helper arrays needed by the species (ornament decisions, syncopation structure, etc.),
- and posting rules that are local to one voice, such as melodic constraints and species-specific rhythmic constraints.

This organisation matches the musical reality: many Fux rules are stated per voice (allowed melodic intervals, required stepwise resolutions, syncopation structure), and species are the natural boundary for those rules.

4.3.2 Cross-voice constraints (Two/Three/FourVoiceCounterpoint)

Rules involving more than one voice are posted at the level of the global counterpoint problem classes (`TwoVoiceCounterpoint`, `ThreeVoiceCounterpoint`, `FourVoiceCoun-`

`terpoint`). These classes decide how many parts exist, instantiate them, and then post constraints that involve interactions between voices:

- harmonic constraints between a counterpoint voice and the cantus firmus,
- motion constraints between pairs of voices (e.g., controlling direct motion into perfect consonances),
- multi-voice restrictions that only make sense when several counterpoints coexist.

Placing these rules at the "global problem" level avoids duplicating cross-voice logic inside each species class and ensures that voice-count-specific rules are clearly scoped.

4.3.3 Stratum constraints

Finally, some constraints are most naturally expressed on the vertical sonority itself (e.g., properties of the chord implied by the set of simultaneous notes, or constraints that reference the lowest voice regardless of which part it is). They are always posted using the `Stratum` objects created during problem construction. Because strata are `Voice`-like objects, they provide the same derived-variable interface (notes, intervals, motions) while representing a vertical layer rather than an actual compositional line.

4.3.4 Conditional posting and experimentation.

Across all three layers, constraints are posted conditionally: the generation engine enables musical constraints by default (all active), but the implementation supports selectively disabling constraints for ablation experiments. Concretely, generation runs initialise the activation mask so that all constraints are active before creating the problem. This mechanism is what makes it possible to diagnose infeasible configurations (especially in higher species) without editing constraint code each time.

4.4 Constraints organisation and naming convention

The number of rules involved in species counterpoint is large, and many of them have multiple variants depending on the musical context (species, number of voices, and sometimes register roles such as lowest vs upper voice). To keep the codebase maintainable and to support debugging workflows (notably ablation through `activeConstraints[]`), FuxCP relies on a systematic naming convention that encodes both the **theoretical origin** of a rule and its **scope of application**.

4.4.1 Rule families and theoretical numbering (G/H/M/P/R)

The first level of organisation follows the theoretical presentation of the rules: each rule belongs to a family, and keeps the same numbering scheme as the written formalisation. Across previous FuxCP work, rules are typically grouped into:

- **G** (*General rules*): constraints that apply across multiple species or that enforce global musical invariants (e.g., tonal consistency, global restrictions).
- **H** (*Harmonic rules*): vertical constraints on simultaneous intervals (consonance/dissonance management, cadential requirements, etc.).
- **M** (*Melodic rules*): constraints within a single voice (allowed melodic intervals, stepwise resolutions, variety, etc.).
- **P** (*Progression / motion rules*): constraints describing how voices move together (direct/oblique/contrary motion restrictions, parallels, forbidden successions, etc.).
- **R** (*Rhythm / ornamentation rules*): constraints that are specific to rhythmic choices and ornament patterns (most notably in fifth species where rhythm is part of the decision space).

This classification is not only "documentation": it directly shapes how rules are implemented and tested, because it provides a stable vocabulary for describing what type of musical property a constraint is enforcing, without depending on a specific voice count or class name.

4.4.2 Scoping and context-specific variants

A crucial difficulty of the formalisation is that the *same theoretical rule* may require different implementations depending on the configuration. The previous four-voice formalisation explicitly highlights this issue by providing a scope table: certain rules appear multiple times because they have multiple formulations depending on whether the setting is 2 voices or 3–4 voices (and similarly for other context-dependent differences).

In FuxCP, this leads to the following general principle:

> A rule identifier refers to a *musical intent*, but the code may contain multiple *scoped variants* of that rule, each posted only when its context applies.

Typical sources of scoping differences include:

- **Voice count**: some rules are defined only for two voices, while others are meaningful only when a third or fourth voice exists (e.g., triad constraints in 3–4 voices, or constraints targeting the global sonority rather than a pairwise interval).

- **Species**: rhythmic structure changes which positions are considered "strong" or "weak", and whether dissonances can appear at all.
- **Role-based conditions**: in multi-voice settings, some rules depend on whether a voice (or stratum) is currently the lowest / highest, or whether it is an inner voice.

This is also why strata exist: they allow certain rules to be expressed relative to the sonority layers rather than hard-coding "voice 2/3/4" semantics (see Section 2.2).

4.4.3 Code-level identifiers: encoding scope in names

To operationalise this scoping, FuxCP uses identifiers that combine:

1. a **scope prefix** (where the rule is applied),
2. the **theoretical rule id** (species + family + number),
3. and sometimes a **context suffix** (most often the voice count).

This is the convention referred to throughout the codebase and in debugging notes, with examples such as `SP1_1H1` or `V2_G6`.

In practice, the most common prefixes are:

- `CF_...`: constraints attached to the cantus firmus context (or rules whose formulation is specific when the cantus firmus is one of the voices involved).
- `STRATUM_...`: constraints posted at stratum level (rules that refer to the vertical ordering / global sonority).
- `V2_/V3_/V4_...`: constraints posted in the global problem layer and whose existence depends on the number of voices.
- `SP1_... to SP5_...`: species-specific constraints; these represent the "bulk" of the catalogue.

When a rule exists in multiple scoped versions, a suffix is used to disambiguate, most commonly:

- `_2V, _3V, _4V` to indicate which voice-count context the formulation targets,
- or other short suffixes when a rule has multiple internal formulations within the same broad context.

For instance, the project explicitly references constraints such as `SP1_1M2_2V` (a first-species melodic rule scoped to 2 voices) and constraints `SP4_4P5_3V` / `SP4_4P5_4V` (a fourth-species progression rule with separate multi-voice formulations).

4.4.4 Constraint identifiers vs implementation function names

In the codebase, each rule exists at two distinct levels:

- The short **constraint identifier** we just talked about (e.g., `SP1_1M2_2V` or `SP4_4P5_3V`). These identifiers are intentionally compact and encode the musical context (species, rule family and number, voice-count variant).
- A concrete **implementation function name** in the constraints library (e.g., `M1_1_2v_melodicIntervalsNotExceedMinorSixth`). These names are longer and descriptive: they are meant to be readable by developers and to summarize the actual condition enforced by the code.

This separation is deliberate. The identifier captures the *theoretical rule reference and scope*, while the function name captures the *implementation intent*. It also prevents the documentation and experimental workflow from depending on internal refactorings: the underlying function can be renamed, split, or rewritten while keeping the same public identifier for traceability.

This naming convention provides immediate traceability between a theoretical rule, its implementation scope, and the configuration in which it applies, which is essential for keeping the rule catalogue readable and for locating the source of unexpected solver behaviour.

4.5 Hard constraints, soft constraints, and relaxed constraints

The constraint system in FuxCP distinguishes three different ways of enforcing musical knowledge: *hard constraints*, *soft constraints*, and *relaxed constraints*. This distinction is essential because Fux's treatise mixes strict rules (must always hold) with preferences (should hold when possible), and because practical generation sometimes requires weakening specific rules to recover feasibility without discarding the musical intent.

4.5.1 Hard constraints: feasibility

Hard constraints define the **feasible space** of the CSP. They restrict the domains of decision variables (ultimately the note variables) and guarantee that any solution returned by the solver respects the corresponding musical rule.

From a modelling standpoint, a hard constraint typically takes one of the following forms:

- a direct relation between derived musical variables (e.g., allowed harmonic intervals at a given position),

- a conditional rule implemented through reification (e.g., “if this is a perfect consonance, then motion cannot be direct”),
- a structural requirement imposed by the species representation (e.g., syncopation ties in fourth species).

The important point is that hard constraints are **not evaluated**: violating one hard rule does not increase a cost, it makes the assignment impossible. Therefore, if the set of active hard constraints is too restrictive or contradictory for a given configuration, the solver will return `UNSAT` (or, in practice, fail to find a solution within the time limits).

This is also why the codebase supports conditional posting and scoping: a hard rule must only be posted when its musical context applies (species, voice count, role in the sonority), otherwise it may accidentally eliminate valid solutions.

4.5.2 Soft costs: preference modelling

Soft costs encode **preferences** rather than feasibility. A preference does not forbid a solution: it assigns a *penalty* to musically undesirable patterns, and lets the optimisation process search for solutions with lower total penalty.

In FuxCP, soft costs are represented as non-negative integer variables (or arrays of such variables). Conceptually, each cost component corresponds to “how much” a certain preference is violated in the current candidate solution. Typical patterns are:

- a local penalty that is 0 when the preferred situation holds and positive otherwise,
- a count of occurrences of an undesirable pattern (e.g., repetitions, overuse of a motion type),
- a weighted penalty where the severity depends on the musical context (e.g., a stronger penalty on strong beats).

These costs are produced at two levels:

1. **locally** by each `Part` (and therefore each species class), which builds the species-specific helper arrays and the cost arrays relevant to that voice;
2. **globally** by the `Two/Three/FourVoiceCounterpoint` problem class, which unifies all per-part cost arrays into a single global cost vector used for optimisation.

The optimisation process itself is handled separately (in 5.1 last part). The important system-level property is that soft costs are always compatible with feasibility: they never invalidate a solution, they only contribute to decide which feasible solutions are considered best.

4.5.3 Relaxed constraints: controlled weakening with penalties

Relaxed constraints are a hybrid mechanism used when a rule is conceptually mandatory but needs to be **temporarily softened** to recover feasibility or to diagnose overly restrictive combinations.

The key idea is:

> A relaxed constraint keeps the *same musical intent* as its hard counterpart, but instead of forbidding violations, it counts them (or penalises them) and pushes the solver to avoid them through optimisation.

In the code, a relaxed constraint is usually built by conditionally converting the interdiction into a relaxation cost. In the global objective vector, relaxation penalties are considered a priority. When relaxation is enabled, the solver computes an aggregated `totalRelaxationCost` (the sum of the relaxed penalties) and puts it at the beginning of the returned cost vector. This enforces the desired behaviour: even when rules are relaxed, the solver first goal should be to minimise the amount of relaxation before improving other preferences.

Relaxation is therefore a controlled and traceable mechanism:

- it makes infeasible configurations solvable when the alternative would be "no result",
- it helps identify which theoretical rules are too restrictive in practice,
- and it provides a clean path to move a questionable hard rule into a preference (cost) without losing the ability to enforce it strictly when needed.

Summary. Hard constraints define what is musically acceptable, soft costs define what is musically preferred, and relaxed constraints provide an intermediate mechanism that preserves a rule's intent while shifting it from feasibility to optimisation. This separation is what allows FuxCP to remain faithful to Fux's rule-based nature while still being usable as a generator across a wide range of configurations.

4.6 Activation, ablation, and debugging workflow

Because the constraint catalogue is large and highly context-dependent (species, voice count, and multi-voice interactions), FuxCP includes a lightweight mechanism to control *which* rules are posted and *how* they are enforced. This mechanism serves two purposes:

- keeping the generation pipeline stable by clearly defining a default set of active rules,

- enabling debugging workflows such as ablation (disable rules) and relaxation (soften rules) without modifying constraint code.

4.6.1 Global activation masks

Constraint control is implemented through two boolean masks indexed by constraint identifiers:

- `activeConstraints[id]` decides whether a rule is posted at all.
- `softConstraints[id]` decides whether an active rule is posted as a strict feasibility constraint or in a relaxed/penalised form (when such a relaxed variant exists).

Both masks are global: they are configured before creating the CSP and then consulted during problem construction. This is consistent with the monotonic nature of constraint programming in Gecode: constraints are posted once during construction and cannot be removed afterwards. Therefore, rule activation must be decided *before* calling `create_problem(...)`.

During problem construction. Constraints are posted conditionally using a recurring pattern:

> if a rule is inactive, it is not posted; if it is active, the construction layer posts either the hard or relaxed variant depending on `softConstraints`.

This pattern is applied at multiple layers (species classes, global problem class, and stratum-level constraints). From a modelling perspective, the important point is that the activation masks control *posting*, not evaluation: if a rule is inactive, it does not exist in the CSP at all, which can drastically change feasibility and propagation.

In production generation. The default behaviour is deliberately simple: the generator starts from the most complete musical model possible, and only deviates from it when running explicit experiments.

Concretely, `Generations::run_generation_case` enables all rules by filling the activation mask with `true` just before calling `create_problem(...)`. As a consequence, the standard command-line generation and the programmatic API run the solver with a fully active rule set, ensuring that results are comparable across runs and across presets.

4.6.2 Ablation: locating feasibility blockers

When the solver fails to find a first solution (or systematically times out), a frequent cause is that the active rule set contains at least one constraint that is either incorrectly

scoped (posted in a context where it should not apply), too restrictive compared to the intended theoretical interpretation, or correct musically but creates an overly narrow feasible region in practice.

To diagnose such situations, FuxCP supports an ablation workflow:

1. Select a reference configuration (cantus firmus, species layout, voice types, preset, objective mode) that exhibits the problem (timeout or no solution).
2. Keep all settings fixed and run a series of executions where exactly one constraint identifier is deactivated (i.e., `activeConstraints[id] = false`).
3. Observe whether feasibility is recovered (first solution becomes reachable) and record the impact on runtime and solution quality.

This workflow is effective because it isolates the responsibility of a single rule without requiring any code changes. The cost of ablation is mainly computational (many runs), but it provides a clear signal: if disabling a specific identifier consistently restores feasibility, the rule is a strong candidate for re-scoping, correction, or conversion into a preference cost (as was done for some fourth-species multi-voice rules in this work).

4.6.3 Relaxation: recovering solutions while keeping musical intent

Ablation is diagnostic: it tells which rule blocks feasibility, but it removes the rule entirely. In contrast, the relaxation mechanism aims to preserve musical intent while still producing solutions.

When a rule has a relaxed form, setting `softConstraints[id] = true` keeps the rule conceptually present but shifts it from feasibility to optimisation:

- violations are counted or penalised,
- the solver is encouraged to avoid violations through the objective,
- but violations no longer make the problem unsatisfiable.

This is especially useful in two situations:

- **Discussed theoretical rules:** rules that are not always strictly enforced by Fux (or that have multiple accepted interpretations) can be treated as preferences rather than absolute restrictions.
- **Practical generation constraints:** in higher species and multi-voice settings, strict enforcement of every rule can collapse feasibility; relaxation allows generation to proceed, while still exposing which musical principles are being violated (through the relaxation cost components).

In other words, ablation answers "which rule prevents any solution ?", while relaxation answers "can we still generate, and how far are we from satisfying the strict rule set ?". This provides a clean bridge between feasibility debugging and the more musical question of how to trade strictness for generation quality.

4.7 Worked examples: from theoretical rules to Gecode constraints

This section illustrates, on a few representative examples, how a musical rule is expressed as a mathematical condition and then encoded as constraints in Gecode. We deliberately keep the examples small; the full catalogue is described in the "Full validated model of Fux counterpoint" Appendix, and its implementation is in the "Source code" appendix.

4.7.1 Example 1: consonance as a domain constraint

Rule. On strong beats in first species, the harmonic interval between counterpoint and cantus firmus must be consonant.

Formula. Let n_t^{cp} and n_t^{cf} be the MIDI pitches at time t, and let

$$h_t = (n_t^{cp} - n_t^{cf}) \bmod 12.$$

Then the rule is: $h_t \in \mathcal{C}$ where $\mathcal{C}$ is the set of consonant interval classes.

Gecode encoding. In the codebase, the first-species consonance rule is posted by restricting the domain of the harmonic-interval array to a set of allowed intervals:

```
// src/constrains.cpp
void H1_1_harmonicIntervalsAreConsonances(Home home, Part* part){

    //we also include negative values, since the fourth species could render the
    //harmonic intervals to be negative because of its deplacement
    dom(home, part->getFirstSpeciesHIntervals(), IntSet({UNISSON, MINOR_THIRD,
        MAJOR_THIRD, PERFECT_FIFTH, MINOR_SIXTH, MAJOR_SIXTH, PERFECT_OCTAVE,
        -MINOR_THIRD, -MAJOR_THIRD, -PERFECT_FIFTH, -MINOR_SIXTH, -MAJOR_SIXTH,
        -PERFECT_OCTAVE}));
}
```

This is a direct `dom(...)` constraint: it prunes all non-consonant values from the harmonic interval variables, and therefore propagates back to the pitch domains.

4.7.2 Example 2: dissonance implies passing motion (reified implication)

Rule. A dissonant weak-beat note is only allowed if it is a passing tone (approached and left by step).

Formula. Let h_t be the harmonic interval class at time t and define $d_t \equiv (h_t \notin \mathcal{C})$. Let n_{t-1}, n_t, n_{t+1} be consecutive pitches in the same voice. Define:

$$pass_t \equiv (|n_t - n_{t-1}| \in \{1, 2\}) \wedge (|n_{t+1} - n_t| \in \{1, 2\}).$$

Then the rule is the implication: $d_t \Rightarrow pass_t$.

Gecode encoding. In second species, the "passing-tone" (diminution) predicate is explicitly built as a boolean array `isDiminution`. It is derived from melodic-interval tests using reified constraints:

```
// src/Parts/SecondSpeciesCounterpoint.cpp
// create isDiminution Array
isDiminution = BoolVarArray(home, (nMeasures*notesPerMeasure.at(FIRST_SPECIES)-
1), 0, 1);

for(int i = 0; i < isDiminution.size(); i++){
    BoolVar btt3(home, 0, 1), btt4(home, 0, 1);
    BoolVar bta2nd(home, 0, 1), bat2nd(home, 0, 1);
    BoolVar btt3rd(home, 0, 1), band(home, 0, 1);

    rel(home, expr(home, abs(firstSpeciesMelodicIntervals[i])), IRT_EQ, 3, Reify(bt
    rel(home, expr(home, abs(firstSpeciesMelodicIntervals[i])), IRT_EQ, 4, Reify(bt
    rel(home, expr(home, abs(secondSpeciesMelodicIntervals[i*2])),   IRT_LQ, 2, Rei
    rel(home, expr(home, abs(secondSpeciesMelodicIntervals[i*2+1])), IRT_LQ, 2, Rei

    rel(home, btt3, BOT_OR, btt4, btt3rd);
    rel(home, bta2nd, BOT_AND, btt3rd, band);
    rel(home, band, BOT_AND, bat2nd, isDiminution[i]);
}
```

Then, the arsis dissonance rule is enforced by linking dissonant harmonic intervals to this boolean predicate (if dissonant, then diminution must hold):

```
// src/constrains.cpp
```

```
void H2_2_arsisHarmoniesCannotBeDisonnant(Home home, Part* part){
    for(int i = 0; i < part->getNMeasures()-1; i++){
        if(i != part->getNMeasures()-2){ // not the penultimate measure
            for(int d : DISONANCE){
                rel(home, part->getHIntervals()[(i*4)+2], IRT_EQ, d,
                    Reify(part->getIsDiminution()[i], RM_PMI));
            }
        }
    }
}
```

This is the standard Gecode "reified implication" pattern: for each dissonant class d, the equality `hInterval == d` implies `isDiminution[i] == 1` (mode `RM_PMI`).

4.7.3 Example 3: a motion restriction into perfect consonances

Rule. Direct motion into a perfect consonance is restricted: if the target harmony is perfect, the preceding motion should not be direct.

Formula. Let p_{t+1} be a boolean predicate "the harmonic interval at $t+1$ is perfect", and let dir_t be a boolean predicate "motion at step $t \to t+1$ is direct". Then the rule can be written: $p_{t+1} \Rightarrow \neg dir_t$.

Gecode encoding. In multi-voice settings, the project penalizes direct motion into perfect consonances by assigning a non-zero value to `directCostArray[j]` when the pattern occurs:

```
// src/constrains.cpp
void P1_1_4v_noDirectMotionFromPerfectConsonance(Home home, Part* part){
    for(int j = 0; j < part->getFirstSpeciesMotions().size()-1; j++){

        // if the counterpoint is the lowest part -> the cost is 0
        rel(home, (!part->getIsNotLowest()[j]) >> (part->getDirectCostArray()[j]==0

        // if the counterpoint is neither the lowest nor the highest part:
        // (motion == 2) and (next harmony is unison or fifth) => cost = 2
        rel(home, (part->getIsNotLowest()[j] && !part->getIsHighest()[j] &&
            (part->getFirstSpeciesMotions()[j]==2 &&
             (part->getFirstSpeciesHIntervals()[j+1]==0 ||
              expr(home, abs(part->getFirstSpeciesHIntervals()[j+1]))==7)))
            >> (part->getDirectCostArray()[j]==2));
```

```
        // if the counterpoint is the highest part and the condition is true,
        // then add a direct move cost
        rel(home, (part->getIsHighest()[j] &&
            (part->getFirstSpeciesMotions()[j]==2 &&
             (part->getFirstSpeciesHIntervals()[j+1]==0 ||
              expr(home, abs(part->getFirstSpeciesHIntervals()[j+1]))==7)))
            >> (part->getDirectCostArray()[j]==part->getDirectMoveCost()));
    }
}
```

This shows the typical encoding style in the codebase: the musical condition is written as a boolean expression on derived arrays (motion category + target harmony class) and is used in a reified implication to set a cost variable.

Chapter 5

Structural improvements

Chapter summary. This chapter presents the main contributions that improve FuxCP as a *generation system*. It still obviously has an impact on the musical output, but without any modification to the musical model. We first document how the optimisation process was made robust and practically usable (branching refinements, search-engine choice, pruning behaviour, and objective handling). We then describe the codebase restructuration that consolidates the generation pipeline into a reusable engine (CLI and API), clarifies module responsibilities, and improves maintainability. Finally, we list smaller but impactful technical fixes and utilities (such as cantus scale detection and cleanups in species modelling and cost wiring) that remove instability sources and improve efficiency.

Roadmap.

- Section 5.1 explains the optimisation strategy as implemented in the production code: how branching was revised (including the final cost-branching layer and boolean branching for syncopation), how pruning was strengthened through the `constrain` method, why `BAB` is used as the default search engine (and how timeout/stagnation make it practical), and how objective modes shape solution comparison.
- Section 5.2 describes the codebase restructuration: the replacement of the previous wrapper by the `Generations` engine, the separation of tests from the production pipeline, and the introduction of dedicated modules (CLI, programmatic API, MIDI export, and CSV configuration loading), together with the rationale behind each move.
- Section 5.3 concludes with smaller technical changes that improve robustness and usability without changing the rule base: scale inference for direct cantus inputs, simplifications in fourth-species helper variables, fixes to default cost wiring, and initialization issues that could otherwise pollute the objective vector.

5.1 Optimisation strategy

The FuxCP solver is built on top of the `Gecode` constraint programming library. Solving a counterpoint problem amounts to exploring a search tree whose nodes represent partial note assignments; each leaf corresponds to a complete score that satisfies every harmonic and melodic (hard) constraint.

Three decisions shape this exploration and directly govern how quickly the solver converges towards a musically interesting solution : the order in which variables are enumerated (the *branching*), the algorithmic engine that traverses the tree (the *search engine*), and how the solutions are compared to each other (the *cost function*). We describe here the choices kept in the final version of the code, then the avenues that were explored but not retained.

5.1.1 Branching strategies

Inherited starting point. When the project was taken over, every variant (2, 3 and 4 voices) applied the same branching skeleton: first the notes of the lowest voice with the heuristic `INT_VAR_DEGREE_MAX + INT_VAL_SPLIT_MIN` (the former selects the most constrained variable in the sense of the number of constraints touching it, the latter splits the domain in two and tries the lower half first), then the arrays linked to species and ornaments (`speciesArray`, `cambiataCostArray`, `syncopeCostArray`) with combinations of `DEGREE_MAX` and minimum or random values, and finally the global `solutionArray` with `INT_VAR_SIZE_MIN + INT_VAL_MIN` (variable whose remaining domain is smallest, smallest value first).

Contributions retained in the current version. Three modifications to the branching have been integrated into the production code and concern primarily two-voice counterpoint.

Firstly, the terminal heuristic on `solutionArray` has been replaced by `INT_VAR_AFC_MAX + INT_VAL_RND(1U)`. The selector `AFC_MAX` (*Accumulated Failure Count*) consults the solver's history and prioritises variables that have been involved in the largest number of past failures, on the assumption that these variables most heavily constrain the rest of the tree and should therefore be fixed early. The value selector `VAL_RND(1U)` introduces a reproducible random draw (the seed `1U` guarantees that two identical runs produce the same sequence), which breaks the natural symmetries that the previous `VAL_MIN` could not avoid. This combination was retained after empirical comparison with several alternatives, some of which still appear in the code as commented-out lines that document the experimentation.

Then, an additional terminal branching has been added on the cost variable itself: `branch(*this, cost(), INT_VAR_NONE(), INT_VAL_MIN())`. This line is not intended to steer the search towards a good solution; its purpose is to guarantee that the `cost` variable is fully instantiated by the time we read its value from the solution. Without this layer, most configurations triggered `ValOfUnassignedVar` errors because the solver considered a score valid even though its cost variable still had a non-singleton domain; the addition fixes this problem cleanly.

Finally, for the fourth and fifth species, the branching applied to syncopations has been redirected to a boolean array. Instead of branching directly on `getSyncopeCostArray()` as in the original version, we retrieve via `getNoSyncope()` a `BoolVarArray` that indicates for each measure whether the note is tied or not, and we branch on that array with `BOOL_VAR_AFC_MAX + BOOL_VAL_MIN`. Branching on a binary domain is intrinsically faster than branching on integers, and the "syncopation or not" decision immediately propagates more constraints on related variables than a decision on the associated cost would.

Complementary pruning improvement. In addition to branching choices, we also revised the optimisation plumbing to benefit from stronger default pruning. Concretely, the `CounterpointProblem::constrain` method was adjusted to delegate to `IntLexMinimizeSpace::constrain(best)`. This ensures that Gecode's standard branch-and-bound pruning is applied consistently when a new incumbent solution is found, and it significantly improved runtime on several configurations where the search otherwise stagnates despite reasonable branching heuristics.

Avenues explored and set aside. Several branching modifications have been tested without entering the final version, generally because they turned out to be less effective than the version kept.

The most structural avenue was to apply to three- and four-voice counterpoint the same reordering that had been adopted for two voices. Concretely, this meant branching on `syncopeCostArray` before `speciesArray` (whereas the original order branches on species first and on ornaments afterwards), on the grounds that the "syncopation or not" decision sharply reduces the domain of the species array and should therefore be resolved first. Jointly, the heuristic on `speciesArray` was tested with `INT_VAL_SPLIT_MIN()` value instead of `INT_VAL_RND(3U)`, judged more deterministic and propagating constraints better.

Several heuristic combinations on `solutionArray` were also compared before settling on `AFC_MAX + VAL_RND(1U)`, namely combinations between the variable decisions `SIZE_MIN` and `AFC_MAX`, and the value decisions `INT_VAL_MIN`, `INT_VAL_MAX` and `VAL_-`

RND(1U).

Another set of discarded workarounds relates to the historical ValOfUnassignedVar failures observed during optimisation. Before adopting the final "branch on cost()" solution, we tested more invasive approaches such as branching directly on large internal note arrays (e.g., part-level note grids) and introducing additional deterministic tie-breakers in stratum ordering. While these approaches could reduce the occurrence of unassigned objective variables in some cases, they also made the search dramatically slower and were finally replaced by the simpler and more robust terminal branching on cost().

Finally, the adaptive pipeline implemented in intermetiate testing files was chaining several engines within a single resolution (a short DFS to quickly secure a first feasible solution, then BAB for the main optimisation, then RBS<BAB> in case of stagnation). This strategy remained an evaluation tool and has not been integrated into the production binary.

Summary. The final version therefore allows any counterpoint generation configuration, while keeping a homogeneous core for production (BAB engine across all configurations, AFC heuristic as the terminal layer, boolean branching on syncopations for 4th and 5th species) and a separate experimentation environment that makes it possible to evaluate alternative engines and combinations.

5.1.2 Choice of search engine

About search engines. Relevant engines families for our problems are: enumeration engines, which return feasible solutions without optimality guarantees (e.g., DFS), and optimisation engines, which search for solutions of decreasing cost (e.g., BAB). LDS can be seen as a compromise when a good baseline heuristic exists, while restart-based schemes such as RBS<BAB> attempt to escape stagnating regions of the search tree by periodically restarting the search with controlled discrepancies.

The BAB solver. The BAB solver is kept as the default engine because the end user is not only interested in "a valid counterpoint" but in "a good counterpoint". In practice, the first feasible solution is often musically acceptable but not the most convincing one; BAB provides a consistent workflow where the solver progressively refines the best solution found so far, and can be stopped at any time while still returning the best solution encountered.

Timeouts. Timeouts have been added to make BAB usable in practice. Generation runs attach a custom stop condition to the search options: a hard timeout bounds the total runtime, and a stagnation threshold stops the search when progress becomes

negligible. In the current implementation, the stagnation timer is updated during the search loop, so the stopping rule behaves as a practical "no progress in the search process" criterion, rather than requiring the solver to fully prove optimality.

Evaluation and diagnosis. We also run the same models under `DFS`, `LDS` and restart-based variants of `BAB`. These engines are useful to measure time-to-first-solution, or to evaluate whether `BAB` stagnates on some configurations, but they are not exposed as the default generation strategy: the production goal is a simple, predictable pipeline that always returns the best solution found under a clear cost definition.

5.1.3 Cost function refinements

In the FuxCP Codebase, each part (species instance) produces several cost components that correspond to musical preferences or penalized patterns. These local costs are then unified at the `CounterpointProblem` level into a single vector (`finalCosts`), which is reordered according to the importance ranking. Finally, the `cost()` method returns the lexicographic objective vector that is consumed by `BAB` to compare candidate solutions.

The **objectiveMode** variable decides the return value of the **cost()** function, used by Gecode to compare two solutions. Since the project relies on a lexicographic minimization interface (`IntVarArgs`), the order and content of this returned cost vector directly defines the optimisation behaviour.

Let $\texttt{finalCosts} = (c_0, c_1, \ldots, c_{n-1})$ be the list of cost components, and let

$$\texttt{globalCost} = \sum_{i=0}^{n-1} c_i$$

as enforced by `rel(*this, globalCost, IRT_EQ, expr(*this, sum(finalCosts)))`.

In the cost() function, two values can be prepended to the detailed vector :

- First, if relaxation is enabled (`hasRelaxation`), the additional cost is `totalRelaxationCost`, the sum of most important relaxed costs.
- Then, in all modes except **OBJECTIVE_LEX**, the solver minimises an additionnal scalar objective `objectiveCostSum`

OBJECTIVE_LEX In this mode, `cost()` returns only the detailed cost vector (possibly starting with `totalRelaxationCost` if relaxation is enabled). Gecode then applies a pure **lexicographic** optimisation: the solver first minimises the first component of the returned vector; among all solutions that are optimal for this component, it minimises the second one; and so on. This behaves like a true multi-objective optimisation where the priority order is given by the order of the elements in `finalCosts` (and `totalRelaxationCost` is first if present).

OBJECTIVE_TOTAL This mode introduces a single primary objective:

$$\texttt{objectiveCostSum} = \texttt{globalCost}.$$

Concretely, `cost()` returns a vector starting with `objectiveCostSum` and followed by the detailed costs. The optimisation therefore works in two layers:

1. Minimise the total aggregated cost `globalCost`.
2. Break ties lexicographically using the detailed cost components (and `totalRelaxationCost` if enabled).

This keeps the global criterion dominant while still making the individual cost factors meaningful as tie-breakers.

OBJECTIVE_MIXED This mode mixes the total cost with a proxy for lexicographic priorities. First, a weighted score `lexScore` is built from the ordered factors (`orderedFactors`) using decreasing weights:

$$\texttt{lexScore} = \sum_{i=0}^{k-1} w_i \cdot f_i, \quad \text{with} \quad w_i = k - i,$$

where $k = \texttt{n_unique_costs}$ and $f_i = \texttt{orderedFactors}[i]$. Then the primary objective is defined as:

$$\texttt{objectiveCostSum} = \texttt{globalCost} + \texttt{lexScore}.$$

So the solver first minimises a **single scalar** that balances (with equal coefficient in the current implementation) the total cost and this weighted priority score, and then uses the detailed cost vector as lexicographic tie-breakers. Intuitively, this is a compromise between "minimise the sum of everything" and "respect the intended priority ordering".

OBJECTIVE_PONDERED This mode defines `objectiveCostSum` as a **weighted linear combination** of the individual cost factors:

$$\texttt{objectiveCostSum} = \sum_{i=0}^{n-1} \alpha_i \cdot c_i,$$

with coefficients α_i decreasing with i:

$$\alpha_i = 100 - \left\lfloor \frac{80\, i}{n} \right\rfloor,$$

so the first cost component has weight 100 and the last one is around 20. As in the other non-lex modes, the solver first minimises this weighted sum, then breaks ties lexicographically using the detailed cost vector (and `totalRelaxationCost` if enabled).

This behaves like a “soft priority ordering”: earlier costs matter more, but later costs still contribute directly to the main objective. This is meant to avoid last costs values 'explosions', that we can observe otherwise.

5.2 Restructuration

5.2.1 Goals of the restructuration

Beyond musical and solver-level improvements, the project required a structural cleanup to make the codebase easier to maintain, test, and reuse from multiple entry points. The main goals were:

- **separate concerns**: generation engine vs. tests vs. utilities,
- **reduce duplication**: centralize the generation pipeline in a single reusable layer,
- **enable reuse**: expose a stable programmatic API in addition to the CLI,
- **support reproducibility**: configuration files and standardized outputs (CSV/MIDI/logs).

5.2.2 Files moved, removed, and added

This section summarizes the main file-level changes and the rationale behind them.

Replaced `problem_wrapper` by `Generations`. The former `problem_wrapper` files were replaced by `Generations` to become a complete orchestration engine for generation runs. The new engine is more than a small wrapper around solver calls: it centralizes the full pipeline (case construction, cantus/preset resolution, solver execution, stopping criteria, logging, and artifact export). This reduces scattered logic across multiple files and ensures that the CLI and programmatic entrypoints both rely on the same implementation.

Tests moved into a dedicated Tests/ folder. The test-related sources (`figureTests`, `fuxTest`, and the old `main` file) were moved into a dedicated folder to clearly separate the production engine from validation and experimentation utilities. This reflects the fact that the former `main` file was already primarily a testing entry point and helps keep the build and repository structure readable: generation features are not interleaved with debugging helpers and figure validation logic.

Added reusable infrastructure: API, CLI, MIDI export, configuration loader. Several new modules were introduced to make FuxCP usable as a standalone tool and as a reusable library :

- **`GenerationsAPI`** : a programmatic entrypoint for external use (and for non-CLI integration). It converts a clean `GenerationInput` into a `GenerationCase` and delegates to the shared generation engine.
- **New `main`** : a CLI-focused entry point that parses user arguments (or campaign modes) and launches generations through the same engine layer, rather than duplicating solver logic.
- **`Midi`** : a dedicated export module to write multi-voice MIDI files from the solver's solution vector, making results directly inspectable by listening, or importing through specialized application[16].
- **`ConfigLoader`** : a dedicated loader for CSV-based configuration (presets, cantus firmi, campaigns). This replaces hardcoded configurations and makes experiments reproducible and easy to modify without recompilation.

`CounterpointUtils` moved into `CounterpointProblems/`. `CounterpointUtils` acts as a shared dispatch/factory layer : it creates the correct species part from a species identifier and the correct global problem class from the number of voices. Since it is conceptually part of the problem-construction layer and is used across all `CounterpointProblems` classes, moving it next to them improves locality and reduces "miscellaneous root folder" clutter.

`constraints` moved into `Parts/`. The constraints library is tightly coupled to the part-level modelling: it is included directly by `Part` and, in return, mostly manipulates `Part`-level arrays (notes, intervals, motions, species views). For that reason, placing it next to `Part` and the species classes makes the dependency structure clearer: *the constraints operate on the musical parts and their derived variables.*

That said, the constraints are not "part logic" in the musical sense: they describe global modelling rules. An alternative would be to keep them in a top-level `Constraints/` folder to emphasize their conceptual role, but it would be the only file on this folder. The current placement is therefore a pragmatic choice to reflect actual include-dependencies and developer workflow (editing constraints typically requires being next to the part/species definitions they target).

5.2.3 Quick codebase documentation

As the project grows, lightweight internal documentation becomes necessary for two reasons.

First, FuxCP is meant to be extended by future students and contributors. A short, up-to-date "map" of the repository (main folders, responsibilities, and entry

points) reduces onboarding time and helps new developers locate the correct place to implement a change (constraint, cost, branching, export, configuration) without having to re-discover the architecture from scratch.

Second, during this thesis we repeatedly faced the practical cost of missing documentation. In order to implement or debug a new feature, we often had to reverse-engineer a significant portion of the repository: tracing the resolution pipeline, understanding how parts and strata interact, and identifying where a given concept is actually enforced (hard constraint, cost, relaxed constraint, or search heuristic). This led to several iterations where changes were first attempted in the wrong layer (e.g., adding an evaluation cost where feasibility was expected, modifying a species class when the issue was actually in the global problem layer, or tuning constraint propagation where the bottleneck came from branching) before finding the appropriate location. Providing a compact documentation section is therefore a concrete productivity improvement: it prevents repeated unnecessary exploration work, encourages cleaner separation of concerns, and makes future development cycles more efficient and less error-prone.

5.3 Other small changes

Cantus firmus scale detection. We added helper functions to infer an indicative scale name from a cantus firmus given as raw MIDI notes. This is used in the cantus-resolution step: when a cantus is provided directly (instead of being loaded from `cantus_firmus.csv`), the system now detects a likely scale and uses it to fill missing metadata such as `cf_scale` (and the default `cf_name`). This improves usability and consistency of the pipeline: direct cantus inputs behave similarly to dataset cantus inputs (same metadata fields, clearer logs), and the verbose output explicitly reports the detected scale versus the one specified by the user when both are available.

extended_domain. The previous codebase used an `extended_domain` concept that carried extra computations related to scale handling, but these computations were not actually used in a meaningful way by the solver. In practice, `extended_domain` always behaved like a standard pitch-domain restriction and only appeared in a fixed set of contexts. We therefore replaced it by a single **`domain`** variable, which is simpler to understand and to use.

is_off variable. In the 4th species model, the `is_off` variable (used to encode whether a note belongs to the "off-beat / shifted" structure implied by syncopation) was originally implemented through two explicit loops and multiple intermediate constraints. We rewrote this mechanism using reified domain membership: instead of manually iterating and constraining cases, we express the membership logic directly and let

Gecode propagate it. This reduces boilerplate code and improves efficiency in 4th species, where performance was already a bottleneck.

unitedCosts. In the 3-voice and 4-voice pipelines, the `unitedCosts` aggregation was sometimes left uninitialized when no voice contributed to that specific cost family. This is problematic because an uninitialized cost variable can later pollute the global objective vector and lead to unstable behaviour (random values, inconsistent comparisons, or misleading cost logs). We fixed this by explicitly setting `unitedCosts` to 0 when it is not used, ensuring that the cost pipeline remains well-defined in every configuration.

varietyCostArray. This array was not initialized in the 2-voice SecondSpeciesCounterpoint implementation, which prevented 2-voice configurations from reliably carrying variety-related penalties into the unified objective vector. We added proper initialization so that variety-based preferences can be applied consistently across voice counts, and in particular so that `M2_1_varietyCost` can be used in 2-voice second-species generations instead of being limited to the 3-voice and 4-voice pipelines.

Chapter 6

Constraints improvements

Chapter summary. This chapter presents the constraint-level contributions of the thesis: the changes that directly modify the musical model. We first introduce several new constraints that were required to remove recurrent musical flaws observed in generated counterpoints (notably in multi-voice contexts), including stricter voice-independence rules and multi-voice harmonic completeness rules motivated by expert review. We then present relaxed (soft) variants of important constraints, used to preserve musical intent while keeping generation feasible and efficient on difficult configurations. Finally, we document a set of revised constraints: small but important fixes and integration corrections (bug fixes, activation fixes, and scoping adjustments) that improve correctness, robustness, and baseline musical quality without changing the overall modelling philosophy.

Roadmap.

- Section 6.1 introduces newly added constraints. For each rule, we explain the musical flaw it targets, its origin when relevant (notably expert feedback and treatise references), and its technical encoding at the Gecode level.
- Section 6.2 presents relaxed versions of selected constraints. These variants preserve the rule's intent but replace strict infeasibility by explicit violation penalties, enabling generation to proceed on hard cases while still prioritising "closest to the strict model" solutions.
- Section 6.3 concludes with revised constraints: implementation fixes and integration adjustments (e.g., correcting interval handling, enabling an existing cost consistently across voice counts, and deactivating an overly restrictive multi-voice rule in fourth species), together with their practical impact.

6.1 New constraints

6.1.1 P8_noSuccessiveSamePerfectInterval

This constraint forbids *successive identical perfect consonances* between any two voices (successive fifths and successive octaves/unisons). It was introduced after review sessions with Karim Haddad (IRCAM), who repeatedly observed this pattern in generated counterpoints. He took time to explain why it is perceived as a strong voice-leading flaw (it destroys independence between voices), and pointed us to the corresponding discussion in the Gallon–Bitsch counterpoint treatise as a practical reference for strict writing([2], p.24-25, rules 42-43).

Musical intent. The goal is to prevent the solver from exploiting a common "cheap" pattern: keeping two voices locked in the same perfect interval across several structural instants. Even when the rest of the score is valid, repeated perfect consonances strongly reduce the perception of independent melodic lines.

Encoding. The implementation checks the harmonic interval class (modulo 12) between pairs of voices on several index sets, depending on the rhythmic structure:

- successive thesis (strong beats),
- thesis → arsis (where applicable),
- and, when at least one third species is present and no fourth species is present, successive notes on the finer grid.

For each of these index sets, the common logic is factored in an intermediate helper *noSuccessiveSamePerfectIntervalOnIndices*. It identifies perfect consonance classes (fifth and octave/unison) and forbids the case where the interval at time t is perfect and equal to the interval at time $t + 1$.

(a) Successive unison (b) Successive fifths

Figure 6.1: Two counterpoints rejected by the constraint

6.1.2 P9_noSimultaneousRepetition

This constraint forbids *simultaneous repetition* between two voices, i.e., situations where two voices repeat their pitch at the same time step. It was introduced after review sessions with Karim Haddad (IRCAM), who identified it as a recurring weakness in generated material: two voices "freeze" together, which produces a static texture and makes the counterpoint sound mechanically constrained. Although this is authorized by basic Fux counterpoints ([3]), the Gallon–Bitsch treatise supports a stricter version of this voice-independence discipline ([2], p.20, rule 31).

Musical intent. Repetition in a single voice can be acceptable. The problematic case is when *two voices repeat simultaneously*, which tends to erase motion and creates a rigid block-like effect. The constraint therefore targets the "both repeat together" pattern rather than banning repetition entirely.

Encoding. The constraint is applied on the same structural index sets as P8 (successive thesis, thesis→arsis, and in some mixed cases successive notes). The common logic is factored into *noSimultaneousRepetitionOnIndices*. For each checked step $t \to t+1$, it detects whether voice A repeats (same pitch at both instants) and whether voice B repeats at the same time; the simultaneous repetition case is forbidden.

(a) Only one voice repeats : valid (b) Both voices repeat : invalid

Figure 6.2: Valid and invalid repetition

6.1.3 H2_2_arsisHarmoniesCannotBeDissonant_multiVoice

In the original rule set, the second-species arsis constraint was checked only against the lowest stratum. In multi-voice counterpoint (3+ voices), this leaves a blind spot: the second-species weak beat may be consonant with the bass but dissonant with another sounding voice, without being detected. This constraint fills that gap by checking the second-species arsis against *every other voice*.

This multi-voice variant was introduced after review sessions with Karim Haddad (IRCAM). He pointed out a recurring flaw in generated 3+ voice counterpoints: a second-species arsis note could be "acceptable" with respect to the bass, yet produce a clear dissonance against another sounding voice. He explained that, in strict contrapuntal writing, weak-beat events must remain controlled *with respect to all voices* (not only the lowest one), and referenced the corresponding discussion in the Gallon–Bitsch treatise.

Musical intent. In second species, weak-beat events (arsis) may only introduce dissonance when justified as a passing event (diminution). In multi-voice writing, this must hold with respect to all sounding voices, not only with respect to the lowest voice.

Encoding. For each measure i (excluding the penultimate, as in the base rule), the implementation compares the arsis note of the second-species part (index $i * 4 + 2$) with the *structural note* of every other voice at the same measure. Since only strong-beat notes are guaranteed pinned for slower voices, the other voice is sampled at its strong beat (cantus firmus uses index i, other parts use $i * 4$). The harmonic interval class (modulo 12) is computed, and each dissonant class is allowed only if `isDiminution[i]` holds (reified implication).

6.1.4 H2_3_dissonanceImpliesDiminution_multiVoice

Similarly, the original third-species dissonance rule was primarily expressed against the bass. In multi-voice settings, dissonances between the third-species voice and an upper voice could therefore slip through. This constraint generalizes the check by evaluating weak-beat dissonances against every other voice.

Musical intent. In third species, dissonances on weak positions are acceptable only as joint notes : they must be approached and left by step. In multi-voice counterpoint, this condition must be satisfied with respect to all voices that contribute structural harmony.

Encoding. For each measure i and for each weak-beat cell $k \in \{1, 2, 3\}$ (index $j = i * 4 + k$), the implementation builds a local passing-tone predicate:

$$isPassing \equiv (|n_j - n_{j-1}| \in [1, 2]) \wedge (|n_{j+1} - n_j| \in [1, 2]),$$

in other words, approached and left by step (1–2 semitones). Then, for every other voice, it compares the third-species weak-beat note at j against that voice's structural strong beat for measure i, computes the interval class modulo 12, and enforces: if the interval is dissonant then `isPassing` must hold (reified implication).

6.1.5 H2_chordMembershipOnDisjunctWeakBeats

This constraint was introduced after Karim Haddad's reviews, and it corresponds to a harmonic rule that is especially explicit in the Gallon–Bitsch treatise ([2], p.32, rule 59). Karim identified a recurring multi-voice flaw: some weak-beat notes were not "wrong enough" to be filtered by the usual consonance/dissonance checks against the bass, yet

they clearly did not belong to the harmony implied by the tonality, producing unstable or stylistically inconsistent chords.

Musical intent. In Fux's original pedagogical setting, many dissonance rules can be formulated as "dissonant with the cantus firmus is forbidden unless it is a passing event", because the cantus firmus effectively defines the harmonic reference. This logic is also what the baseline FuxCP rules implement: harmonic intervals are often computed relative to the lowest stratum, and dissonance checks are expressed against that reference voice. In contrast, Bitsch's approach makes explicit a stronger multi-voice requirement: on weak beats, if a note is not a passing tone, it should belong to the *current harmony* (pitch-class set of the sonority), not only be "non-dissonant" with one reference voice. The goal of this constraint is therefore to extend the strictness of the multi-voice harmonic language beyond pairwise bass checks.

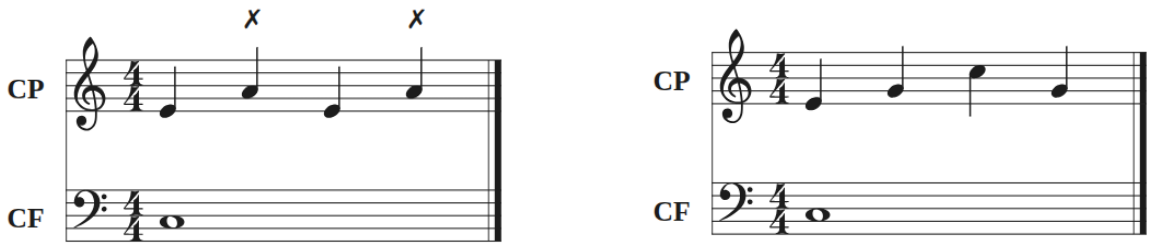
Encoding. The constraint applies only to second and third species voices. For each measure i (excluding the penultimate measure) and each relevant weak-beat offset (second species: $+2$; third species: $+1, +2, +3$), it computes:

- a passing-tone predicate `isPassing` (approached and left by step), with species-adapted neighbourhoods,
- a chord-membership predicate `anyInChord` testing whether the weak-beat pitch class matches at least one structural pitch class of the other voices.

The chord itself is defined as the set of pitch classes of all voices' structural notes in the current half-measure. A key detail is the handling of fourth species: a fourth-species voice changes harmony mid-measure (its suspension resolves on the third beat), so the second-half harmony is computed using its resolution note (index $i * 4 + 2$) while other voices keep their strong beat. Finally, the rule enforces that every checked weak-beat note satisfies:

$$isPassing \ \vee \ anyInChord,$$

In other words, it must be a passing tone or belong to the current half-measure chord.



(a) $A = M_6$ consonant but $A \notin \{C, E, G\}$ (b) G, C, $G \in \{C, E, G\}$

Figure 6.3: Valid and invalid counterpoints according to the harmony of the measure

6.2 Relaxed versions

The constraints in this section are *relaxed* (soft) versions of rules that normally exist as hard feasibility constraints. They follow the relaxation mechanism described earlier: instead of forbidding violations, the relaxed variants introduce boolean violation indicators (typically 0/1) and accumulate them into a relaxation cost array. This allows the solver to keep generating solutions on difficult configurations while still preferring solutions that satisfy the original strict rule, since relaxation penalties are prioritised in the objective vector.

6.2.1 G9_lastChordSameAsFundamental_soft

This relaxed constraint corresponds to the cadential rule that the global sonority should be anchored on the same fundamental pitch class as the cantus firmus. In practice, this is enforced by comparing pitch classes (modulo 12) rather than absolute octaves.

Hard version (intent). The strict rule enforces that the lowest stratum matches the cantus firmus in pitch class at structurally important points (in particular at the end, and also at the beginning to stabilise the overall tonal frame). This prevents solutions that technically satisfy many local rules but end with an unstable or stylistically inconsistent final sonority.

Relaxed version (encoding). The relaxed variant replaces the hard equalities by two reified match tests (start and end). Each mismatch becomes a binary penalty:

- cost = 1 if the pitch class of the lowest stratum does not match the cantus firmus at the final,
- cost = 1 if the pitch class of the lowest stratum does not match the cantus firmus at the initial.

This turns the rule into a "nearly mandatory" preference: the solver can violate it if needed to preserve feasibility, but violations are explicit and strongly discouraged by the relaxation cost.

6.2.2 M2_2_3v_twoConsecutiveNotesAreNotTheSame_soft

This relaxed constraint corresponds to a simple but important melodic rule in multi-voice contexts: a voice should avoid repeating the exact same pitch on two consecutive note positions. In practice, repeated notes are a very common "cheap" pattern for the solver: they preserve feasibility easily while other voices handle the harder harmonic structure, but they often produce flat and unconvincing melodic lines.

Note on renaming. Earlier versions of the codebase carried an inconsistent historical name for the 3 voices version of this rule in specific. In practice, the rule has always targeted a much simpler phenomenon: **two consecutive notes being identical**. The constraint (and its relaxed counterpart `_soft`) has been renamed accordingly to the corresponding 2 voices version `M2_2_2v_twoConsecutiveNotesAreNotTheSame`. This change does not introduce a new musical idea; it clarifies the actual behaviour of the constraint and fixes the name inconsistency.

Hard version (intent). The strict rule enforces that two consecutive notes in the targeted voice are not equal. In other words, the melodic interval between consecutive positions must not be 0. This is especially relevant in three-voice settings (and above), where one voice can otherwise become a static filler line while still satisfying most harmonic constraints.

Relaxed version (encoding). The relaxed variant replaces the hard inequality by a binary violation indicator:

- for each consecutive pair of notes under the scope of the rule, set cost $= 1$ if the two pitches are equal, else 0,
- accumulate these penalties into the relaxation cost array for the voice.

This makes repeated notes possible when needed to recover feasibility, but keeps them explicit and minimised through the relaxation-priority mechanism.

6.2.3 H2_3_dissonanceImpliesDiminution_soft

This relaxed constraint targets a central third-species principle: dissonances are acceptable only when they are explained by a passing motion (diminution), rather than appearing as unsupported vertical clashes.

Hard version (intent). In third species, weak positions may contain dissonances, but only if they correspond to a stepwise passing structure (diminution). The strict rule therefore enforces that each checked position is either consonant, or explicitly marked as part of a diminution pattern.

Relaxed version (encoding). The relaxed version introduces a boolean "ok" condition per checked position:

$$ok \equiv (\text{consonant}) \vee (\text{isDiminution})$$

and assigns a penalty of 1 when *ok* is false (i.e., dissonant without diminution), and 0 otherwise. The penalties are stored in the part's relaxation cost array. This keeps the musical intent explicit: even when a solution violates the strict rule, the violation is counted and can be prioritised in optimisation.

6.2.4 H2_2_arsisHarmoniesCannotBeDissonant_soft

This relaxed constraint targets the second-species rule that weak-beat (arsis) harmonies should not be dissonant, except when justified by a diminution pattern.

Hard version (intent). Second species is organised around thesis (strong beat) and arsis (weak beat). The thesis must remain structurally stable, while the arsis may only introduce dissonance under tightly controlled passing conditions. The strict rule therefore enforces that, at each arsis position, the harmonic interval belongs to the consonant set, unless the measure is explicitly identified as a diminution case. The implementation typically excludes special cadential positions (e.g., near the penultimate) where separate cadence rules apply.

Relaxed version (encoding). The relaxed variant reifies the same logic as a binary violation:

- detect whether the arsis harmonic interval is consonant (membership in the consonance interval set),
- combine it with the diminution flag: $ok \equiv (\text{arsis consonant}) \vee (\text{isDiminution})$,
- set relaxation cost to 1 if *ok* is false, else 0.

As a result, dissonant arsis events without diminution do not make the problem infeasible, but they become explicit, minimised compromises during optimisation.

6.3 Revised constraints

This section groups smaller but important changes made to already existing constraints and cost mechanisms. Unlike the previous sections (new constraints and relaxed variants), these revisions do not introduce new musical principles: they improve correctness, scoping, and maintainability of the existing rule base.

6.3.1 P4_successiveCost

This function had a lot of repetitions, and a few unnecessary steps in some computations. This made it difficult to understand, so we made it more readable by :

- Setting up simple booleans that could be used for conditions, as *p1Second* for *p1->getSpecies() == SECOND_SPECIES*
- Simplifying conditions regarding 4th species by defining some values with the ternary operator *'?'*
- Creating a function to avoid repetition of large code block
- Expressing a bit more effectively some of its constraints

6.3.2 M2_1_varietyCost

`M2_1_varietyCost` was already implemented as a soft cost intended to discourage local melodic repetition, but it was effectively only active in 3-voice and 4-voice configurations. In the 2-voice pipeline, the corresponding cost array was not consistently initialized/propagated into the unified cost vector, which made repeated-note patterns appear frequently in generated counterpoints.

We revised the integration so that `varietyCostArray` is also included in 2-voice generation. The cost definition itself is unchanged: it penalizes repeated pitches within a short look-ahead window on the branching grid (each equality contributes a fixed penalty, each inequality contributes 0). This keeps the rule purely preferential (never affecting feasibility), but improves baseline musical quality in 2 voices, where repetition is particularly audible.

6.3.3 SP1_1M2_2V

In the second-species implementation of `SP1_1M2_2V`, the constraint was intended to limit the magnitude of melodic leaps by comparing values in `secondSpeciesMelodicIntervals` against thresholds (e.g., 8 and 12 semitones). However, this interval array is computed as a raw difference between consecutive MIDI pitches, so it can

be negative when the melody moves downward. As a result, the original comparison only constrained ascending intervals correctly and could unintentionally accept large descending leaps.

We fixed this by simply applying an absolute value to the melodic interval before comparison, so the bound is enforced symmetrically in both directions. This change does not modify the musical intent of the rule, but it makes its implementation consistent with the standard interpretation of "maximum melodic interval" constraints.

6.3.4 4.P5

While investigating feasibility and performance issues in 4th species, we noticed that the multi-voice variants of rule `4.P5` (`SP4_4P5_3V` and `SP4_4P5_4V`) were still activated, despite existing comments indicating that they were not meant to be enforced in the current baseline. In practice, keeping these constraints active made many 4th-species configurations extremely difficult to solve (often preventing the solver from reaching a first solution within reasonable time).

We therefore deactivated these two constraints (by commenting out where they were posted) and added an explicit warning in the code to prevent accidental reactivation. The rationale is that the musical principle behind `4.P5` can still be relevant, but enforcing it as a hard constraint in multi-voice 4th species is overly restrictive for generation; reintroducing it later as a soft preference (cost) is a more robust way to keep its intent without collapsing feasibility.

Chapter 7

Evaluation

Chapter summary. This chapter evaluates the counterpoint solver along three difficulty axes : number of voices, species, and register. They report both the quality of the counterpoints produced (cost) and the computational effort required (time and search statistics). The measurements distinguish carefully between proven optima and best-effort solutions returned at timeout. The closing discussion makes the deliberate compromises of the experimental design explicit.

Roadmap.

- Section 7.1 This section discusses the choice of input data for the generations, such as the cantus firmus, the number of voices, the vocal range, etc. The widest yet still reasonable range of data has been selected in order to provide a comprehensive and relevant overview of the possibilities for counterpoint.
- Section 7.1.4 There are several ways to assess the quality of a configuration for a counterpoint. A balance must be struck between quality, measured in terms of cost, and feasibility in terms of computation time and solution convergence.
- Section 7.2 This section presents the benchmark results based on the benchmark configurations. The feasibility and quality of counterpoints, ranging from the simplest to the most complex, are discussed in detail.
- Section 7.3 Various compromises had to be made on several points: convergence speed, feasibility and cost-effectiveness. This ensures that the solutions are suitable for real-world applications.

7.1 Experimental setup

7.1.1 A single reference cantus firmus

All controlled experiments are run on one cantus firmus, `CF` composed by the nine-note C-major line `C3 D3 F3 E3 G3 F3 E3 D3 C3`. Fixing the melody allows us to isolate the variables of interest (number of voices, mode, and register) from the confounding effect associated with a variable input.

For simplicity and on the advice of Karim Haddad, we lowered the cantus firmus by one octave, leaving ample room above it for the counterpoint voices. This is important since register is considered an experimental axis. The input melody is deliberately short: the computational cost increases sharply with its length, so that a short reference allows the easy cases to be solved optimally while letting the difficult cases reveal the solver's limitations within a reasonable time frame.

7.1.2 The three difficulty axes

A counterpoint generation involves, among other things, the specification of a triplet

$$(n, m, r)$$

where

- $n \in \{2, 3, 4\}$ is the number of voices (one of which is always the *cantus firmus* (CF)),
- $m \in \{1, 2, 3, 4, 5\}$ is the species of the *target* counterpoint voice (from first to fifth species, the fifth species offering the greatest rhythmic freedom),
- $r = (r_1, \ldots, r_n)$ encodes the register of each of the n voices.

We distinguish two types of setups.

Classical counterpoint. All voices, including the CF, are in first species, except for a single voice that takes on the role of the target species m.

Example. In a 4-voice piece in the third species (target species $m = 3$), a possible assignment is

$$\mathrm{CF} + 1^{\mathrm{st}} + 1^{\mathrm{st}} + 3^{\mathrm{rd}}.$$

Grand mélanges. In a "grand mélange", we relax the classical constraint and allow each non-CF voice to adopt an arbitrary species, while still imposing first species on the CF. Thus, any combination such as

$$\mathrm{CF} + 2^{\mathrm{nd}} + 3^{\mathrm{st}} + 4^{\mathrm{th}}.$$

is admissible, provided the CF remains in first species.

Registers are encoded as integers `v_type` per voice and are treated as first-class axis. A voice of register `v_type` is centered around $6 \cdot \texttt{v_type} + \mathbf{cf}[0]$, and confined to the two-octave window

$$[\, 6 \cdot \texttt{v_type} + \mathrm{cf}[0] - \mathbf{12},\ 6 \cdot \texttt{v_type} + \mathrm{cf}[0] + \mathbf{12} \,]$$

where cf[0] is the first cantus-firmus note, so a higher `v_type` places the voice higher above the cantus firmus. Different `v_type` values are therefore genuinely different sub-problems with different note domains, not mere reorderings of one search which is why register affects feasibility and cost as much as the species does.

7.1.3 Objective mode and protocol

The generated reference set consistently uses the `pond` cost objective : a single weighted lexicographic score across the fourteen cost categories ranked in the order of the predefined set.

All executions are single-threaded and sequential, so the time metrics are free of CPU contention. The StagnationStop termination criterion combines an absolute timeout with a stagnation window and returns EXHAUSTIVE, TIMEOUT_MAX, or STAGNATION. Stagnation is disabled here, so NO_SOLUTION means a true lack of result within the given time limit. Each generation was given an eight minute timeout, this allows "simple" problems to converge to a cost-optimal solution and more complex problems to find solutions (sometimes taking several minutes for the first iteration).

7.1.4 Metrics

The primary quality measure is `best_cost`, the scalar `weighted cost` of the best counterpoint found (lower is better). A run that produces no counterpoint has no cost. Two metrics of times characterise convergence: `ms_first`, the time to the first feasible solution and `ms_total`: the total runtime. The second one is bounded by the eight minutes limit unless the search exhausts. A small `ms_first` with `ms_total` pinned at the budget marks a solver that finds a solution quickly then shaves cost until stopped. On the other hand, a large `ms_first` marks a case where reaching feasibility

is itself the bottleneck. The categorical termination status is decisive: `EXHAUSTIVE` means the cost is a *proven optimum*, `TIMEOUT_MAX` means it is an *upper bound*, and NO SOLUTION means no counterpoint was found within budget. Gecode's node, failure and propagation counts complete the picture by showing whether a configuration converges or merely thrashes.

7.2 Results

7.2.1 Overall feasibility map

On our reference `CF`, all so-called standard cases were tested for every possible combination of voices and species. Since the choice of register for each voice significantly affects the solution both musically and in terms of feasibility, we had to test all possible register configurations for each case. For the simplest forms (first, second and third species) the impact on feasibility is minimal. For the fourth and fifth species, the choice of register can have a significant impact and requires careful consideration.

7.2.2 Two-voice results

Species / register	`best_cost`	termination	`ms_total`	`ms_first`	`n_sol`
sp1, vt1	696	EXHAUSTIVE	0.15 s	1.8 ms	16
sp1, vt2	656	EXHAUSTIVE	0.16 s	2.8 ms	8
sp1, vt3	656	EXHAUSTIVE	0.15 s	3.8 ms	15
sp2, vt1	776	EXHAUSTIVE	44.9 s	6.4 s	47
sp2, vt2	776	EXHAUSTIVE	18.9 s	5.7 ms	64
sp2, vt3	776	EXHAUSTIVE	8.8 s	6.4 ms	40
sp3, vt1	4323	TIMEOUT	180 s	8.0 ms	90
sp3, vt2	3515	TIMEOUT	180 s	10.2 ms	86
sp3, vt3	1925	TIMEOUT	180 s	9.6 ms	112
sp4, vt1	1759	TIMEOUT	180 s	2.2 s	7
sp4, vt2	—	no solution	—	—	0
sp4, vt3	1109	EXHAUSTIVE	3.1 s	2.4 s	11
sp5, vt1	—	no solution	—	—	0
sp5, vt2	—	no solution	—	—	0
sp5, vt3	2331	TIMEOUT	180 s	0.16 s	74
sp5, vt4	10107	TIMEOUT	180 s	72 ms	85

Table 7.1: Two-voice results on `CF8` (`default` preset, 480 s budget). `n_sol` counts the intermediate solutions emitted before the budget expired.

At two voices the model is comfortably tractable for the lower species: first and second species exhaust their search space within a minute. Beyond that the budget bites in two distinct ways. Cost-wise, third species rises sharply (1925–4323 against a 776 ceiling for second species) because four notes per measure multiply the soft-penalty surface. Fifth species adds another factor and reaches 10107 at `vt4`. Feasibility-wise, the

three `no_solution` cells (sp4 `vt2`, sp5 `vt1`–`vt2`) flag register pairs where the target voice sits inside the cantus firmus band and the harmonic constraints become unsatisfiable. Two patterns will recur as voices are added: *cost grows with species density* fourth species the lone exception (because it contains only two notes per measure and one of them a forced tie) and exhaustiveness collapses past second species, so it is importan to realize that the benchmarks below report anytime costs rather than proven optima.

7.2.3 Three-voice results

Species	feasible / run	`best_cost`	median	`worst_cost`	exhausted
sp1	9 / 9	1071	1431	2037	1
sp2	9 / 9	3225	3768	4322	0
sp3	7 / 9	4267	8054	13395	0
sp4	7 / 12	1720	2638	2638	0
sp5	7 / 12	13816	22681	47254	0

Table 7.2: Three-voice results on `CF8`: cantus firmus + one first-species voice + target voice, 480 s budget. The *feasible / run* column counts register pairs that returned at least one solution among those launched.

Adding a voice has two compounding effects. First, the cost scale climbs across the board: first species rises from 656 to 1071 (≈1.6×), second from 776 to 3225 (≈4×), fifth from 2331 to 13816 (≈6×).

Second, the `EXHAUSTIVE` regime essentially disappears and only one first-species register (`vt3-1`, cost 1071) is proven optimal. The species-feasibility ranking from two voices is sharpened: fourth and fifth species each solve the seven of twelve register pairs where the target voice sits high enough to clear the auxiliary voice, the same band-overlap signature as sp4 `vt2` in two voices. Note that fourth species still produces a *lower* median than third species (2638 vs 8054): the suspension regime controls only one half of each measure, so it accumulates fewer soft penalties even when the absolute search problem is harder.

7.2.4 Four-voice results

Species	feasible / run	best_cost	median	worst_cost	max_first
sp1	17 / 17	8304	11637	15864	34 ms
sp2	12 / 12	11710	15671	43768	35 ms
sp3	11 / 11	18239	22934	53528	106 s
sp4	9 / 9	11396	14760	16835	364 s
sp5	8 / 8	27556	33845	57130	391 s

Table 7.3: Four-voice results on CF8: cantus firmus + two first-species voices + target voice, 480 s budget. Every run hit TIMEOUT_MAX; no register pair returned NO_OUTPUT. max_first is the worst time to first solution across registers.

With the four-voice grid, every configuration of every target species returns a counterpoint within the 480 s budget, including fourth and fifth species where two and three voices left several NO_OUTPUT cells. The cost scale lifts again (5−10× over two voices) and the species ordering by best cost settles as sp1 < sp4 < sp2 < sp3 < sp5, confirming the two-voice pattern: fourth species is hard to satisfy but cheap to optimise, third species is easy to enter but accumulates more penalties than fourth species ever can. The cost of entering in the optimisation phase also separates species clearly: first and second species hit a first solution in tens of milliseconds, third species takes seconds, and fourth species can wait six minutes before the first admissible assignment is found, leaving very little budget for improvement.

7.2.5 Cross-voice synthesis

Species	2v best	3v best	4v best	2v exh.	3v exh.	4v exh.
sp1	656	1071	8304	3/3	1/9	0/17
sp2	776	3225	11710	3/3	0/9	0/12
sp3	1925	4267	18239	0/3	0/7	0/11
sp4	1109	1720	11396	1/3	0/7	0/9
sp5	2331	13816	27556	0/2	0/7	0/8

Table 7.4: Best cost and exhaustiveness rate across voice counts. Exhaustiveness is reported over *feasible* register pairs (NO_OUTPUT cells excluded from the denominator).

Two trends emerge from Table 7.4. (i) Adding a voice multiplies the best achievable cost by ≈1.6× to ≈6× depending on the species, and (ii) exhaustiveness drops from 7/12 feasible cells at two voices to a single proven-optimum cell at three voices and none at four. The species effect is more subtle than "higher species number = higher cost": by cost, only fifth species behaves that way at every voice count; fourth species sits between first and second on the cost axis but is the hardest to render feasible at low registers (three NO_OUTPUT/NO_SOLUTION cells at two voices, five at three voices). Third species is uniformly feasible but pays for its four notes per measure with a steeper

penalty growth than its number alone would suggest. In short, voice count drives both axes (cost and provability) monotonically, whereas species splits the two axes: *higher species ⇒ harder to satisfy*, but *cost ranks by note density, not by species number.*

7.3 Trade-offs and compromises

The evaluation is in large part a catalog of deliberate compromises.

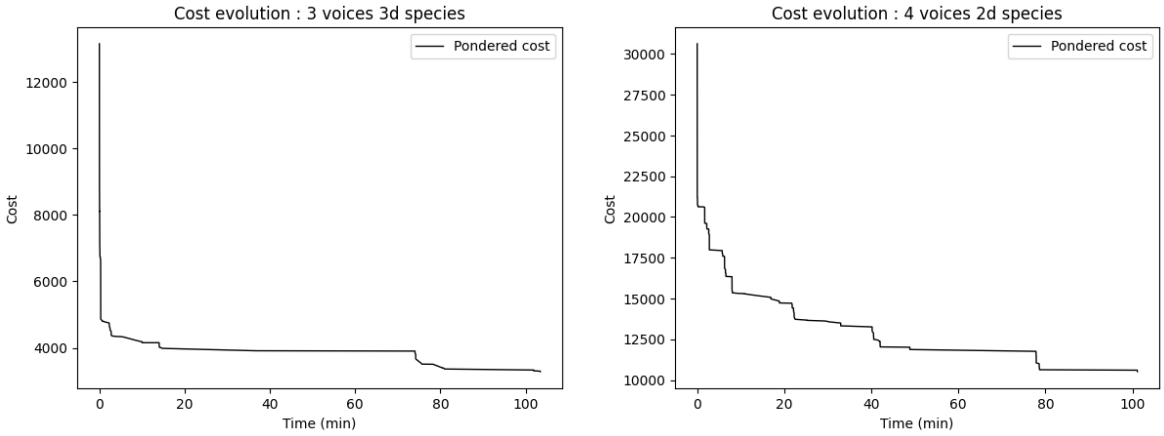
7.3.1 Optimality versus time budget.

Outside the trivial cases, most reported costs are upper bounds obtained at timeout. If the search did not yield an optimal solution every "best" must be read as "best found within budget". However, after a while, the search is not exhaustive, but the cost savings become minimal and the time gap between each solution found grows ever wider.

In practice, this behaviour can be visualised through the evolution of the best objective value during a `BAB` run: the curve is a *staircase*. This is expected, because `BAB` maintains an incumbent (best solution so far) and only updates it when a strictly better solution is discovered. The search can therefore spend a long time exploring subtrees that eventually yield no improvement, until it reaches a region where a new incumbent is found and the objective drops suddenly.

Figure 7.1a illustrates a typical behaviour on a 3-voice 3rd-species configuration. Most of the improvement happens very early: the cost drops sharply in the first minutes, then reaches a long plateau where improvements become rare and marginal. In this run, after roughly 10–20 minutes the solver already reached a stable region (around the first plateau), and the remaining hour mostly produced small refinements, with one late but moderate improvement after a long stagnant period.

Figure 7.1b shows a more difficult configuration (4 voices, 2nd species), representative of worst cases. Here, improvements are spread over a longer horizon: after the initial sharp drop, the solver keeps making occasional progress, but long stagnation periods appear and the next improvement may only happen after tens of minutes. This confirms that stagnation does not imply that no better solutions exist: significant improvements can still appear unexpectedly after long plateaus. However, the practical cost of waiting can become disproportionate: in our experiments, once stagnation lasts several minutes, the additional improvements obtained by letting the solver run for hours were typically small compared to the extra runtime. Therefore, reported results should be interpreted as a trade-off: longer budgets may yield better solutions on hard instances, but the marginal gain decreases quickly and becomes unpredictable.



(a) 3 voices, 3rd species. (b) 4 voices, 2nd species.

Figure 7.1: Evolution of the best objective value over time during 2-hour BAB runs (staircase behaviour).

7.3.2 Voice register control versus feasibility

Each counterpoint voice in the solver is assigned an integer parameter `v_type` that determines the pitch window within which that voice is allowed to move. Understanding how this parameter behaves and what the trade-off imposed on higher species is necessary to produce both feasible and musically meaningful solutions.

Definition and effect. The parameter `v_type` does not act as a search heuristic; it directly sets the register of the voice by defining a fixed two-octave window centred on $6 \cdot \texttt{v_type} + \texttt{cf}[0]$, where $\texttt{cf}[0]$ is the first note of the cantus firmus expressed as a MIDI pitch. Concretely, the lower and upper bounds passed to the counterpoint constructor are

$$\ell = 6 \cdot v - 12 + \texttt{cf}[0], \qquad u = 6 \cdot v + 12 + \texttt{cf}[0],$$

so a higher value of v places the entire voice higher in the pitch space relative to the cantus firmus, while a value of zero or one allows the voice to descend below it. This calculation is applied identically in `TwoVoiceCounterpoint`, `ThreeVoiceCounterpoint`, and `FourVoiceCounterpoint`: in the multi-voice case each voice receives its own `v_type` entry from the input vector, allowing independent register assignment per voice.

Practical range and aesthetic choice for species 1–3. We chose to limit the range of `v_type` values between 1 and 3 for several reasons. First, to ensure that the other voices don't pass through notes much lower than the cantus firmus, for the sake of consonance analysis. Similarly, limiting the interval to a maximum value of 3 prevents voices from soaring too high. A value of 1 or 2 keeps the counterpoint voice in roughly the same register as the cantus firmus or just above it, while a value of 3 places it a fifth or so higher. This range was adopted as a convention rather than a hard rule, nothing

prevents using null or negatives `v_type` to write a counterpoint that descends below the cantus firmus, nor does a higher value violate any musical constraint enforced by the solver.

For first, second, and third species, the `v_type` parameter is a purely aesthetic choice. These species do not rely on tied notes, and their consonance requirements can be satisfied in any register, so the solver will find valid counterpoints across the full range of usable values.

Constraints for fourth and fifth species. Fourth and fifth species introduce syncopated suspensions notes held across the bar line that create a dissonance resolved by stepwise descent and this mechanism requires the suspended voice to sit clearly above a moving bass. When the `v_type` of a suspension voice is set too low (typically $v \leq 2$), the solver's search exhausts its time budget without finding a solution: the pitch window forces the voice into competition with the cantus firmus register, preventing the formation of genuine suspension chains. This is not a theoretical impossibility but a practical consequence of register overlap. Empirically, for voices in the 4th and 5th species, a value of $v = 3$ provides the minimum separation needed for reliable convergence; $v = 4$ restores feasibility when $v = 3$ still fails. For multi-voice configurations, the separation between voice registers also matters: assigning two voices the same or adjacent `v_type` concentrates them in the same pitch band and tends to prevent the solver from finding solutions, regardless of species.

Interaction with the parallel-unisons constraint. An additional issue affected fourth and fifth species independently of the register choice. The constraint forbidding two voices from repeating the same pitch on consecutive beats (intended to prevent hidden parallel unisons or octaves) was incorrectly applied to tied notes. In fourth species, the mandatory suspension tie causes the suspended pitch to appear on both the weak and the following strong beat; the constraint interpreted this repetition as a violation and declared the problem infeasible at every register. Fifth species suffered a weaker version of the same problem, with every syncopation effectively banned. The fix was to exempt fourth and fifth species entirely from this constraint, on the grounds that held notes across the bar are a defining feature of these species, not a prohibited gesture. Once this exemption was in place, the register guidance above ($v \geq 3$ for suspension voices) became the sole remaining practical limitation.

Chapter 8

Conclusion

8.1 Summary

This thesis aimed to improve FuxCP as a practical generator of species counterpoint: not only a formally specified rule base, but a system that can reliably produce musically convincing solutions within reasonable time budgets across multiple species and voice counts.

We improved the project along two complementary axes. On the **structural side**, we made the optimisation pipeline robust and controllable: the solver execution was stabilized, search was better guided through revised branching and pruning behaviour, and generation runs were bounded and comparable through explicit solver-control parameters (timeout and stagnation management). We also consolidated the orchestration of generations into a reusable engine with a standalone CLI and a programmatic API returning a consistent structure through `GenerationResult`. Configuration loading (cantus firmi, presets, campaigns) and MIDI export were integrated as first-class modules, and small utilities such as cantus-firmus scale detection were added to make direct inputs behave consistently.

On the **constraint-model side**, we introduced targeted changes that directly address recurring musical artefacts observed in generated output. This includes new constraints improving voice independence (preventing solver-produced "locking" patterns) and strengthened multi-voice harmonic checks motivated by expert review and treatise references. We also implemented relaxed (soft) variants of selected constraints: these preserve musical intent while allowing generation to proceed on difficult configurations where strict enforcement would collapse feasibility or lead to extreme runtimes. Finally, several revised constraints fixed implementation issues (e.g., symmetric handling of melodic interval magnitude, activation/scoping fixes, and removal of overly restrictive fourth-species variants), improving correctness and baseline quality without changing the overall modelling philosophy.

Finally, our evaluation reports feasibility and performance over representative grids of configurations (number of voices, target species, and register choices). The results show that configurations that were previously impractical—especially in fourth and fifth species and in multi-voice settings—become reachable under fixed budgets, while the musical output exhibits fewer systematic weaknesses and improved stylistic coherence. At the same time, the evaluation highlights the remaining limitations: outside small cases, the reported best costs remain "best found within budget" rather than proven optima, and some configurations still display strong sensitivity to register choices and time budgets. These observations directly motivate the future work directions listed next.

8.2 Limitations of the current model

Even after the improvements presented in this thesis, the system still exhibits limitations that are important to keep in mind when interpreting generation results.

Valid but musically unpleasant solutions. The solver optimises the objective defined by the current constraint/cost model, and can therefore produce solutions that are formally valid yet musically unconvincing. Typical cases include excessive static behaviour in one voice (acting as a filler line), harmonies that remain technically consonant with a reference voice but feel unstable in the full sonority, and rhythmic/melodic patterns that satisfy local rules but weaken the overall musical line. Several annotated examples collected during expert review illustrate this category of issues, such as passing-note behaviour occurring on strong beats, non-permitted seventh-chord sonorities, questionable initial sonorities, and cadential leading-tone resolution requirements.

Interactions and redundancies checks in the rules. As the rule base grew over multiple theses, some mechanisms can partially overlap, which may introduce redundancy in the model (and sometimes unnecessary propagation overhead). This kind of redundancy matters because removing always-zero cost components reduces the size of the cost vector and the number of reified constraints, which can improve propagation and runtime on large configurations.

Long-range musical structure and position effects. Many constraints and costs are local (beat-to-beat or measure-to-measure). This is appropriate for species counterpoint, but it can create uneven musical quality across the piece: the model may strongly constrain openings and cadences while leaving more freedom in the middle measures, where the solver can exploit low-cost patterns. Capturing longer-horizon

musical structure (phrasing, directional contour, or mid-piece harmonic planning) remains limited in the current model.

Budget and input sensitivity. Finally, the best solutions reported by the generator are "best found within the allocated budget" rather than proven global optima for most non-trivial configurations. Runtime and output quality remain sensitive to configuration choices such as register windows and the selected cantus firmus. This motivates both a broader evaluation across inputs and further work on search and propagation efficiency.

8.3 Future work

Future work naturally follows from the current limitations. We group the most relevant directions into three axes: improving solver efficiency, completing musical validation and extending the rule base, and studying robustness across inputs and configurations.

8.3.1 Constraint-level efficiency and solver performance

Even if the model already accounts for several CP considerations, most constraints were primarily written with a clear musical intent. A careful CP-focused revision of the constraints library could therefore produce major speed-ups, especially for higher species and multi-voice settings. Promising directions include:

- **Propagation strengthening:** rewrite "check-like" constraints into tighter domain restrictions when possible, and reduce reification overhead when equivalent stronger formulations exist.
- **Redundancy reduction:** identify overlapping mechanisms and simplify the model when a hard rule already guarantees a property that other rules/costs also enforce indirectly.
- **Search efficiency:** explore more systematic restart policies and adaptive heuristics (beyond the current pragmatic combination of branching, timeout and stagnation) to reduce stagnation and improve convergence reliability.

This work is mostly orthogonal to musical decisions: the goal is to reach feasible solutions faster and reduce the cost of optimisation, without changing the intended style.

8.3.2 Musical validation and rule-base completion

Although the generator is now stable and usable across a wider range of configurations, the musical model is not fully validated yet. During review sessions with Karim Haddad

(IRCAM), several remaining issues were identified but could not all be addressed within the time frame of this thesis (technical stability and feasibility recovery had priority). A structured continuation would be to maintain a validation backlog where each issue becomes either a scope/encoding correction of an existing rule, a new hard constraint when the behaviour is unacceptable in strict writing, or a soft cost when the behaviour should be discouraged rather than forbidden.

Several remaining issues identified during musical review (not yet fully handled by the current model) are worth turning into a structured validation backlog, for example:

- **Direct perfect intervals caused by voice crossing**, leading to an audible loss of voice independence. This can be fixed by re-implementing P8 with strata.
- **Cadential leading-tone resolution**, which is not always enforced consistently across voices in end-of-piece contexts (e.g., B→C in C major).
- **Overly permissive initial sonorities**, such as openings that are technically consonant but stylistically discouraged (e.g., starting on an inversion).
- **Unacceptable dissonance patterns in the full sonority**, where local dissonance rules are satisfied relative to a reference voice but the resulting vertical sonority resembles a non-permitted seventh-chord-like structure in strict writing.

Finally, further melodic exploration would likely improve musicality. Beyond Fux, other treatises (e.g., the Gallon–Bitsch tradition) contain additional melodic and multi-voice harmonic rules that could be integrated. In complex configurations, well-chosen *hard* rules can also reduce the feasible space enough to improve time-to-first-solution, provided they are scoped carefully and formulated with propagation in mind.

8.3.3 Robustness across cantus firmi, tonal contexts, and configuration space

Most evaluation runs in this thesis were performed on a short reference cantus firmus in C major to keep benchmarks comparable. While the system supports other cantus firmi and scales (including direct cantus inputs with scale detection), we did not perform a broad study of how cantus choice affects feasibility, runtime, and musical quality. A larger evaluation campaign across diverse cantus firmi (lengths, melodic profiles, tonal contexts) would help:

- quantify variability of performance and quality across inputs,
- identify input-dependent failure modes (e.g., narrow ambitus, repeated notes, atypical leaps),

- determine whether additional cantus-aware heuristics or constraints are needed.

An additional direction that would further increase practical value would be explicit diversity objectives to avoid repeatedly converging to similar outputs.

Bibliography

[1] Johann Joseph Fux. *Gradus ad Parnassum, sive Manuductio ad Compositionem Musicae regularem*. Vienna: Joannis Petri van Ghelen, 1725.

[2] Noël Gallon, and Marcel Bitsch. *Traité de contrepoint*. Paris: Durand, 1964.

[3] Johann Joseph Fux. *The Study of Counterpoint. From Johann Joseph Fux's* Gradus ad Parnassum. Trans. by Alfred Mann. Revised edition. The Norton Library N277. With the collaboration of John Edmunds. New York: W. W. Norton & Company, 1971.

[4] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. *Modeling and Programming with Gecode*. Published May 28, 2019. Corresponds to Gecode 6.2.0. Accessed 2026-05-22. May 28, 2019. URL: https://www.gecode.dev/doc-latest/MPG.pdf (visited on 05/22/2026).

[5] Damien Sprockeels. "Melodizer: A Constraint Programming Tool for Computer-Aided Musical Composition". Master's thesis. Louvain-la-Neuve, Belgium: École polytechnique de Louvain, Université catholique de Louvain, 2022. URL: https://hdl.handle.net/2078.2/26512.

[6] Thibault Wafflard. "FuxCP: A Constraint Programming Based Tool Formalizing Fux's Musical Theory of Counterpoint". Master's thesis. Louvain-la-Neuve, Belgium: École polytechnique de Louvain, Université catholique de Louvain, 2023. URL: https://hdl.handle.net/2078.2/33371.

[7] Luc Cleenewerk, and Diego de Patoul. "FuxCP: A Constraint Programming Formalization of Fux's Musical Theory of Four-Voice Counterpoint". Master's thesis. Louvain-la-Neuve, Belgium: École polytechnique de Louvain, Université catholique de Louvain, 2024. URL: https://hdl.handle.net/2078.2/41477.

[8] Anton Lamotte. "FuxCP: Constraint Programming Formalisation of Three-Voice Counterpoint According to Fux". Master's thesis. Louvain-la-Neuve, Belgium: École polytechnique de Louvain, Université catholique de Louvain, 2024. URL: http://hdl.handle.net/2078.1/thesis:43960.

[9] Tom Lai. “FuxCP : A Validated Constraint Programming Formalization of Fux Four-Voice Counterpoint”. Master’s thesis. Louvain-la-Neuve, Belgium: École polytechnique de Louvain, Université catholique de Louvain, 2025. URL: https://webperso.info.ucl.ac.be/~pvr/LAI_08721900_2025.pdf.

[10] Pierre Schauss, and Yves Deville. *Constraints programming*. Corresponds to the "LINFO2365 - Constraints Programming" course of UCLouvain for year 2025-2026. 2025. URL: https://learning.edx.org/course/course-v1:LouvainX+Louv31x+3T2025/home.

[11] Jean Louis Fabre. *Gradus ad Parnassum*. Educative channel of videos on the Fux Gradus Ad Parnassum treatease. 2026. URL: https://www.youtube.com/@gradusadparnassumfr.

[12] Dorian Genon. “Improving Composer Control of the FuxCP Solver through Cost Aggregation and Dynamic Sliders”. Master’s thesis. Louvain-la-Neuve, Belgium: École polytechnique de Louvain, Université catholique de Louvain, 2026.

[13] Chris Bakashika. “A DAW plug-in for the FuxCP constraint formalization of Fux counterpoint”. Master’s thesis. Louvain-la-Neuve, Belgium: École polytechnique de Louvain, Université catholique de Louvain.

[14] IRCAM. *OpenMusic — IRCAM Forum*. Accessed 2026-02-22. URL: https://forum.ircam.fr/projects/detail/openmusic/ (visited on 02/22/2026).

[15] Mark Gotham. *OpenMusic theory*. Accessed 2026-05-20. URL: https://viva.pressbooks.pub/openmusictheory/front-matter/introduction/ (visited on 05/20/2026).

[16] Muse Group. *MuseScore Studio*. Accessed 2026-05-22. URL: https://www.musehub.com/app/musescore-studio (visited on 05/22/2026).

Appendix A

Full validated model of Fux counterpoint

This chapter presents the complete set of contrapuntal rules implemented in the solver. The content is based on the work of Lamotte (2024), Wafflard (2023), and Sprockeels (2022), revised and expanded by Cleenewerk & de Patoul (2024), Tom Lai (2025), and this thesis. The goal here is to provide a unified and consistent reference: rules have been reorganised, reformulated for clarity, and their scopes made explicit. In some cases, redundant or outdated rules from previous versions were removed, while others were refined to align more closely with the intended Fuxian style.The new and modified constraints are clearly indicated.

For a detailed explanation of these rules, their underlying logic, and their origin in *Gradus ad Parnassum*, the reader may refer to the corresponding theses: T. Wafflard for two-voice rules, A. Lamotte for three-voice rules and L. Cleenewerk & D. de Patoul for four-voice rules.

A.1 General Rules (G)

The rules **G1** to **G3** do not have explicit mathematical formalizations, since they are not implemented as separate constraints but are instead embedded directly into the structure of the program itself. They apply universally, regardless of the number of voices or the species considered.

G1

Harmonic intervals are calculated from the lowest note.

G2

The number of measures of the counterpoints must be the same as the number of measures of the cantus firmus.

$$m_{\text{cp}} = m_{\text{cf}} \tag{A.1}$$

This guarantees structural alignment between the cantus firmus and the counterpoint.

G3

The counterpoints must have the same time signature and the same tempo as the cantus firmus.

G4

The counterpoints must be in the same key as the cantus firmus.

G5

The range of the counterpoints must be consistent with the instruments used.

This prevents writing notes that would be impractical or impossible to perform.

G6

Chromatic melodies are forbidden for two and three voice composition, and to be avoided for four voice composition.

`2V & 3V:`

$$\forall \rho \in positions(m-2) \quad \begin{cases} (M_{brut}[\rho] = 1 \wedge M_{brut}[\rho+1] = 1) \iff \bot \\ (M_{brut}[\rho] = -1 \wedge M_{brut}[\rho+1] = -1) \iff \bot \end{cases} \tag{A.2}$$

For four voices, the preference of not having chromatic melodies is already taken into account by the cost of using borrowed notes.

G7

Melodic intervals should be small.

$$\forall \rho \in positions(m-1)$$
$$M_{deg_costs}[\rho] = \begin{cases} cost_{secondMdeg} & \text{if } M[\rho] \in \{0,1,2\} \\ cost_{thirdMdeg} & \text{if } M[\rho] \in \{3,4\} \\ cost_{fourthMdeg} & \text{if } M[\rho] = 5 \\ cost_{tritoneMdeg} & \text{if } M[\rho] = 6 \\ cost_{fifthMdeg} & \text{if } M[\rho] = 7 \\ cost_{sixthMdeg} & \text{if } M[\rho] \in \{8,9\} \\ cost_{seventhMdeg} & \text{if } M[\rho] \in \{10,11\} \\ cost_{octaveMdeg} & \text{if } M[\rho] = 12 \end{cases} \tag{A.3}$$

G9 (new soft version)

The first and last notes of the lowest stratum must share their pitch class with the first note of the cantus firmus (i.e. the tonal centre is anchored at both ends of the piece).
This rule applies in all configurations. It is available both as a hard equality and as a soft (penalty) variant.

$$\begin{aligned} N(low)[0,0] \mod 12 &= N(cf)[0,0] \mod 12 \\ N(low)[0,m-1] \mod 12 &= N(cf)[0,0] \mod 12 \end{aligned} \tag{A.4}$$

Implemented through `G9_lastChordSameAsFundamental` (hard) and `G9_lastChordSameAsFundamental_soft` (soft). The soft variant emits two violation indicators (one per equation) and accumulates them into the part's `relaxationCostArray`, so the constraint can be temporarily downgraded to a preference during the relaxation pipeline.

A.2 Constraints of the First Species

Harmonic Rules

1.H1

All notes on the downbeat are consonant with the notes (on the downbeat) of the lowest stratum.
This rule applies in all configurations.

$$\begin{aligned} &\forall j \in [0, m-1] \\ &H[0,j] \in Cons \end{aligned} \tag{A.5}$$

1.H2

In two voices, the first harmony must be a perfect consonance.
This rule applies to all species in two voices.

$$H[0,0] \in Consp \tag{A.6}$$

1.H3

In two voice composition, the last harmonic interval must be a perfect consonance. When composing for three or four voices, the last chord should be composed only of notes of the harmonic triad.
This rule applies in all configurations.

$$\begin{aligned} \texttt{2V:} \quad & H[0,m-1] \in Consp \\ \texttt{3V \& 4V:} \quad & \forall s \in \{b,c,d\}, \quad H[s,m-1] \in Consh_{\text{triad}} \end{aligned} \tag{A.7}$$

1.H4

The key tone is tuned according to the first note of the cantus firmus.
This rule applies in all configurations.

$$\begin{aligned} N(a)[0,0] \mod 12 = N(cf)[0,0] \mod 12 \\ N(a)[0,m-1] \mod 12 = N(cf)[0,0] \mod 12 \end{aligned} \tag{A.8}$$

1.H5

The voices cannot play the same note at the same time except in the first and last measure
This rule applies to thesis notes in all species with two and three voices.

$$\begin{aligned} \forall p_1, p_2 \in \{cf, cp_1, cp_2\} \text{ with } p_1 \neq p_2 \quad \forall i \in \{0,1,2,3\} \quad \forall j \in [1, m-2] \\ N(p_1)[i,j] \neq N(p_2)[i,j] \end{aligned} \tag{A.9}$$

1.H6

On thesis notes, imperfect consonances are preferred to perfect ones; fifths preferred to octaves.
This rule applies in all configurations.

$$\forall j \in [0, m-1]$$
$$Pconscosts[j] = \begin{cases} cost_{Pcons_oct} & \text{if } H[0,j] = 0, \\ cost_{Pcons_fifth} & \text{if } H[0,j] = 7, \\ 0 & \text{otherwise.} \end{cases} \tag{A.10}$$

1.H7

In two voice composition, the harmonic interval of the penultimate note must be a major sixth or a third (minor or major) depending on whether or not the cantus firmus is the lowest stratum. In three voice composition, that harmonic interval must be either a minor third, a major third, a perfect fifth, a major sixth or an octave.
This rule applies to the penultimate *note* for all species with two and three voices

`2V:`

$$\rho := \max(positions(m)) - 1$$
$$H[\rho] = \begin{cases} 9 & if\ A(cf)[0, m-2] \\ 3 \vee 4 & \text{otherwise} \end{cases} \tag{A.11}$$

`3V:`

$$H[p] \in \{0, 3, 4, 7, 9\} \tag{A.12}$$

1.H8

The harmonic triad should be used as much as possible, and in four voices it should be used in its natural order with the octave if possible.
This rule applies to thesis chords for three and four voices composition.

`3V:`

$$\forall j \in [0, m-1]$$
$$(H(b)[0,j] \notin 3, 4) \vee (H(c)[0,j] \neq 7) \iff C_{prefer_harmonic_triad}[j] = 1 \tag{A.13}$$

`4V:`

`Worst case:` Not using the harmonic triad : either there is no third or fifth, or there is a note that does not belong to the harmonic triad.

$$
\begin{aligned}
&\forall j \in [0, m-1] \\
&\big(\forall h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\} \quad (h \notin \{3,4\})\big) \\
&\vee \ \big(\forall h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\} \quad (h \notin \{7\})\big) \\
&\vee \ \Big(\exists h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\} \quad (h \notin \mathrm{Cons}_{\text{h-triad}})\Big) \\
&\implies C_{prefer_harmonic_triad}[j] = not_harmonic_triad_cost
\end{aligned}
\tag{A.14}
$$

`Better:` no octave, doubling the fifth.

$$
\begin{aligned}
&\forall j \in [0, m-1] \\
&\left(\begin{array}{l}
\left(\displaystyle\sum_{h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\}} \mathbf{1}_{h=7} = 2\right) \\
\wedge \neg\big(\forall h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\} \quad (h \notin \{3,4\})\big) \\
\wedge \neg\Big(\exists h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\} \quad (h \notin \mathrm{Cons}_{\text{h-triad}})\Big)
\end{array}\right) \\
&\implies C_{prefer_harmonic_triad}[j] = double_fifth_cost
\end{aligned}
\tag{A.15}
$$

`Better yet:` no octave, doubling the third. In this case, we must also check that the two thirds are of the same nature (minor or major).

$$
\begin{aligned}
&\forall j \in [0, m-1] \\
&\left(\begin{array}{l}
\left(\displaystyle\sum_{h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\}} \mathbf{1}_{h \in \{3,4\}} = 2\right) \\
\wedge \neg\big(\forall h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\} \quad (h \notin \{7\})\big) \\
\wedge \neg\Big(\exists h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\} \quad (h \notin \mathrm{Cons}_{\text{h-triad}})\Big)
\end{array}\right) \\
&\implies (C_{prefer_harmonic_triad}[j] = double_thirds_cost) \\
&(H_b[0,j] \in \{3,4\} \wedge H_c[0,j] \in \{3,4\}) \implies (H_b[0,j] = H_c[0,j]) \\
&(H_b[0,j] \in \{3,4\} \wedge H_d[0,j] \in \{3,4\}) \implies (H_b[0,j] = H_d[0,j]) \\
&(H_c[0,j] \in \{3,4\} \wedge H_d[0,j] \in \{3,4\}) \implies (H_c[0,j] = H_d[0,j])
\end{aligned}
\tag{A.16}
$$

`Even better:` octave used, but not in the natural order.

$$
\begin{aligned}
&\forall j \in [0, m-1] \\
&(H_b[0,j] = 7) \wedge (H_c[0,j] \in \{3,4\}) \wedge (H_d[0,j] = 0) \\
&\qquad \Longleftrightarrow (C_{prefer_harmonic_triad}[j] = triad_with_octave_cost \\
&(H_b[0,j] \in \{3,4\}) \wedge (H_c[0,j] = 0) \wedge (H_d[0,j] = 7) \\
&\qquad \Longleftrightarrow (C_{prefer_harmonic_triad}[j] = triad_with_octave_cost \\
&(H_b[0,j] \in \{3,4\}) \wedge (H_c[0,j] = 7) \wedge (H_d[0,j] = 0) \\
&\qquad \Longleftrightarrow (C_{prefer_harmonic_triad}[j] = triad_with_octave_cost \\
&(H_b[0,j] = 0) \wedge (H_c[0,j] \in \{3,4\}) \wedge (H_d[0,j] = 7) \\
&\qquad \Longleftrightarrow (C_{prefer_harmonic_triad}[j] = triad_with_octave_cost \\
&(H_b[0,j] = 0) \wedge (H_c[0,j] = 7) \wedge (H_d[0,j] \in \{3,4\}) \\
&\qquad \Longleftrightarrow (C_{prefer_harmonic_triad}[j] = triad_with_octave_cost
\end{aligned}
\tag{A.17}
$$

`Best:` Harmonic triad with the octave, in natural order.

$$
\begin{aligned}
&\forall j \in [0, m-1] \\
&(H_b[j] = 7) \wedge (H_c[j] = 0) \wedge (H_d[j] \in \{3,4\}) \Longleftrightarrow C_{\text{prefer-harmonic-triad}}[j] = 0
\end{aligned}
\tag{A.18}
$$

1.H11

Octaves should be preferred over unisons.
This rule applies to all configurations.

This rule is taken into account by **1.H5** for two and three voices, and by **1.H8** for four voices.

1.H12

In three- and four-voice counterpoint, the final chord must not contain a minor third.
This rule applies to any species in three and four voice counterpoints.

$$
H[0, m-1] \notin 3 \tag{A.19}
$$

1.H13

For every pair of upper voices, the thesis note of each measure must never form a minor second interval.
This rule is applied to every species of counterpoint in three and four voices configuration.

$$\begin{gathered}\forall j \in [0, m-1], \quad s_1, s_2 \in \{b, c, d\} \\ H(s_1, s_2)[0, j] \neq 1\end{gathered} \tag{A.20}$$

Melodic Rules

1.M2

Intervals cannot exceed a minor sixth interval, but octave leaps are allowed in three and four voices
This rule applies to all first-species counterpoints.

`2V:`

$$\begin{gathered}\forall j \in [0, m-2] \\ M[0, j] \leq 8\end{gathered} \tag{A.21}$$

`3V & 4V:`

$$\begin{gathered}\forall j \in [0, m-2] \\ \Big(M[0, j] \leq 8 \ \vee \ M[0, j] = 12\Big).\end{gathered} \tag{A.22}$$

1.M4

The notes of each part should be as diverse as possible.
This rule applies to all configurations in three and four voices.

$$\begin{gathered}\forall p \in \{cp1, cp2, cp3\}, \ \forall j \in [0, m-1], \ \forall k \in [j+1, \min(j+3, m-1)] : \\ \Big(N(p)[0, j] = N(p)[0, j * k]\Big) \iff C_{variety}[j + m \cdot k] = 2.\end{gathered} \tag{A.23}$$

Motion Rules

1.P1 & 3.P2

Reaching a perfect consonance by direct motion is forbidden in two voices, and should be avoided when composing for three and four voices. In four voice composition, it is to be tolerated between inner voices rather than outer voices.
This rule applies to all configurations.

`2V:`

$$\begin{gathered}\forall j \in [0, m-2] \\ H[0, j+1] \in Consp \ \Rightarrow \ P[0, j] \neq 2\end{gathered} \tag{A.24}$$

`3V:`

$$\begin{gathered}\forall j \in [0, m-2] \\ \Big(P[0,j] = 2 \wedge H[0,j+1] \in Cons_p\Big) \iff C_{direct_move_to_p_cons}[j] = 8\end{gathered} \tag{A.25}$$

`4V:`

$$\begin{aligned}&P(b)[0,j] = 2 \wedge H(b)[0,j] \in Cons_p \wedge H(b)[0,j+1] \in Cons_p \iff C_{direct_move_to_p_cons}[j] = 2,\\ &P(c)[0,j] = 2 \wedge H(c)[0,j] \in Cons_p \wedge H(c)[0,j+1] \in Cons_p \iff C_{direct_move_to_p_cons}[j] = 2,\\ &P(d)[0,j] = 2 \wedge H(d)[0,j] \in Cons_p \wedge H(d)[0,j+1] \in Cons_p \iff C_{direct_move_to_p_cons}[j] = 8.\end{aligned} \tag{A.26}$$

1.P2

Contrary motions are preferred to oblique motions which are preferred to direct motions.
This rule applies in all configurations.

$$\begin{gathered}\text{With } cost_{con} = 0 \quad \text{and} \quad cost_{con} < cost_{obl} < cost_{dir} \\ \forall j \in [0, m-1] \\ P_{costs}[j] = \begin{cases} cost_{con} & \text{if } P[0,j] = 0, \\ cost_{obl} & \text{if } P[0,j] = 1, \\ cost_{dir} & \text{if } P[0,j] = 2. \end{cases}\end{gathered} \tag{A.27}$$

1.P3

At the start of any measure, an octave cannot be reached by the lower voice going up and the upper voice going down more than a third skip.
This rule applies in all configurations.

$$\begin{gathered}i := \max(B),\ \forall j \in [0, m-1],\ \forall p \in \{cf, cp1, cp2, cp3\} \\ H(p)[0,j+1] = 0 \wedge P(p)[i,j] = 0 \wedge M_{brut}(p)[i,j] < -4 \wedge \neg A(p) \iff \bot,\end{gathered} \tag{A.28}$$

1.P4 (modif)

Successive perfect consonances should be avoided.
This rule applies in all configurations, except second species in three and four voices (see 2.P3).
The cost wiring has been redone (`P4_successiveCost`) and the rule has been split into two complementary hard variants 1.P8 and 1.P9 below, which target the two patterns previously left under-constrained: the literal repetition of the same perfect interval and the simultaneous repetition of pitches by two voices.

$$\begin{gathered}\forall p_1 \neq p_2 \in \{cf, cp1, cp2, cp3\},\ \forall j \in [0, m-1] \\ \Big(H(p_1,p_2)[0,j] \in Cons_p \wedge H(p_1,p_2)[0,j+1] \in Cons_p\Big)\ \Rightarrow\ C_{succ_p_cons} = 2.\end{gathered} \tag{A.29}$$

1.P5

Each part starts distant from the lowest stratum.

This is a general indication for the composer (to make it easier for parts to move towards each other in contrary motion), but no explicit constraint is implemented.

1.P6

It is prohibited that all parts move in the same direction.
This rule applies in three and four voices configurations.

$$\begin{gathered}\forall j \in [0, m-1] \\ \bigvee_{p \in \{cf, cp1, cp2, cp3\}} P(p)[0,j] \in \{0,1\}.\end{gathered} \tag{A.30}$$

1.P7

It is prohibited to use successive ascending sixths on a direct upwards motion.
This rule applies to all configurations in three or four voices.

$$\begin{gathered}\forall j \in [1, m-1],\ \forall p_1 \neq p_2,\ \text{with } sixth := \{8, 9\} \\ H(p_1,p_2)[0,j-1] \notin sixth \\ \vee\ H(p_1,p_2)[0,j] \notin sixth \\ \vee\ M_{brut}(p_1)[0,j] \leq 0 \\ \vee\ M_{brut}(p_2)[0,j] \leq 0\end{gathered} \tag{A.31}$$

1.P8 (new)

Two voices cannot keep the same perfect consonance on successive beats (or notes when at least one of the two voices is in third species).
This rule applies in all configurations and is implemented as a hard reformulation of 1.P4 but narrowed to the literal repetition of the same perfect interval (unison, fifth, octave) rather than the broader notion of two successive perfect consonances.

$$\begin{gathered}\forall p_1 \neq p_2 \in \{cf, cp_1, cp_2, cp_3\},\ \forall (t, t') \in \text{successiveBeats}(p_1, p_2) \\ \neg\Big(H(p_1,p_2)[t] = H(p_1,p_2)[t']\ \wedge\ H(p_1,p_2)[t] \in Cons_p\Big)\end{gathered} \tag{A.32}$$

When one of the two voices belongs to the third species, the check is also performed on every pair of successive note positions and not only on the strong/weak-beat grid. Implemented as `P8_noSuccessiveSamePerfectInterval` with per-configuration switches `V2_1P8`, `V3_1P8`, `V4_1P8`.

1.P9 (new)

Two voices cannot simultaneously repeat their pitches on successive beats: at any pair of consecutive instants, at least one voice must change pitch.
This rule applies to first, second and third species in all configurations. It is intentionally relaxed for fourth and fifth species where the suspension tie of fourth species ($arsis_{\mathrm{m}} = thesis_{\mathrm{m+1}}$) and the held cantus firmus would make the constraint vacuously infeasible. The exemption is enforced at the call site (one of the two voices in fourth or fifth species disables the check on that pair).

$$\begin{gathered}\forall p_1 \neq p_2 \in \{cf, cp_1, cp_2, cp_3\} \text{ s.t. } species(p_i) \notin \{4, 5\}, \\ \forall (t, t') \in \text{successiveBeats}(p_1, p_2) \\ \neg\Big(N(p_1)[t] = N(p_1)[t'] \ \wedge \ N(p_2)[t] = N(p_2)[t']\Big)\end{gathered} \tag{A.33}$$

When one of the voices is in third species, the check additionally covers every pair of consecutive note positions. Implemented as `P9_noSimultaneousRepetition` with per-configuration switches `V2_1P9`, `V3_1P9`, `V4_1P9`.

A.3 Constraints of the Second Species

Harmonic Rules

2.H2 (modified)

Arsis harmonies cannot be dissonant except if there is a diminution.
This rule applies to all numbers of voices, on second species counterpoints.
The rule is now split into the two complementary fragments `H2_2_arsisHarmoniesCannotBeDisonnant` (the "no dissonance" side) and `H2_3_disonanceImpliesDiminution` (the "if dissonance then passing tone" side). Both have a `_multiVoice` extension that generalises the original two-voice formulation to three and four voices (every other voice acts as the harmonic reference, not only the lowest stratum).
Soft variants `H2_2_..._soft` and `H2_3_..._soft` count violations into the relaxation cost array. In two voices, the multi-voice form is now also active so that disjunct weak-beat fourths against an upper voice are caught (see 2.H7 below).

$$\forall j \in [0, m-2],\ \forall p :\ species(p) = 2$$

$$IsDim(p)[j] = \begin{cases} \top & \text{if } M^2(p)[0,j] \in \{3,4\} \wedge M^1(p)[0,j] \in \{1,2\} \wedge M^1(p)[2,j] \in \{1,2\} \\ \bot & \text{otherwise} \end{cases}$$

$$\forall j \in [0, m-2] \quad \neg IsCons(p)[2,j] \ \Rightarrow\ IsDim(p)[j]$$

$$\textbf{Multi-voice extension:} \quad \forall p' :\ p' \neq p,\ \forall j \in [0, m-2]$$

$$\neg\Big(N(p)[2,j] - N(p')[\text{ref}(p', i)] \in Cons \pmod{12}\Big) \Rightarrow IsDim(p)[j] \tag{A.34}$$

2.H3 & 2.H4

When composing in two voices, in the penultimate measure the harmonic interval of perfect fifth must be used for the thesis note if possible. Otherwise, a sixth interval should be used instead.
This rule applies to all two voice configurations in which the counterpoint is of the second species.

$$H[0, m-2] \in \{7, 8, 9\}$$

$$penulthesiscost = \begin{cases} cost_{penulthesis} & \text{if } H[0, m-2] \neq 7 \\ 0 & \text{otherwise} \end{cases} \tag{A.35}$$

2.H6 & 3.H5

The half notes (or quarter notes in the third species) must be coherent ("concordant") with respect to the whole notes.

This is not really a constraint, but more of an indication. It was decided to leave it in this complete set of rules because it is a redundant concept in *Gradus ad Parnassum*.

2.H7 (new) — chord membership on disjunct weak beats

A disjunct weak-beat note (i.e. a weak beat that is not reached and left by step) must belong to the harmony of its half-measure: it must be a chord tone of an implied diatonic triad whose other members are the structural pitches sounded by the other voices on the downbeat (or on the resolution beat, for a fourth-species suspension).
This rule applies to second species (the arsis of every measure), third species (every

weak beat) and fifth species (only on measurements flagged as belonging to the third species via textttgetIsThirdSpeciesArray). The penultimate measure is exempt to leave the cadence free. Genuine passing tones (approached *and* left by step) are exempt. The rule was motivated by Bitsch §59.

$$\forall p \text{ with } \mathit{species}(p) \in \{2,3,5\},\ \forall i \in [0, m-2), \tag{A.36}$$

$$\forall k \in \text{weakOffsets}(\mathit{species}(p))j := 4i + k \tag{A.37}$$

$$\text{isPassing}(j) := \Big(|M_{brut}(p)[j{-}1]| \in \{1,2\}\Big) \wedge \Big(|M_{brut}(p)[j{+}1]| \in \{1,2\}\Big) \tag{A.38}$$

$$\text{harmonyPCs}(i,k) := \Big\{\, N(p')[\text{ref}(p',i,k)] \bmod 12 \,\Big|\, p' \in \text{allVoices} \Big\} \tag{A.39}$$

$$\neg\text{isPassing}(j) \implies \text{anyInChord}(j) \vee \Big(\exists T \in \text{DiatonicTriads(key)} :\ \text{harmonyPCs}(i,k) \subseteq T \wedge N(p) \tag{A.40}$$

with `weakOffsets` $= \{2\}$ for second species and $\{1,2,3\}$ for third and fifth species. `ref` reads the downbeat of the corresponding half-measure, except for a fourth-species voice which contributes its resolution beat $(4i+2)$ for weak offsets $\{2,3\}$, following the suspension semantics of Bitsch §59. `anyInChord`(j) is the fallback predicate "the weak-beat pitch class equals at least one structural pitch class of another voice"; it ensures the constraint never becomes stricter than before on harmonies that fit no diatonic triad. The two conditions are combined disjunctively: anyInChord $\vee$ inTriad, so a note satisfies the rule if either condition holds. Implemented as `H2_4_chordMembershipOnDisjunctWeakBeats`.

Melodic Rules

2.M1

If the two voices are getting so close that there is no contrary motion possible without crossing each other, then the melodic interval of the counterpoint can be an octave leap. This rule applies to all two voice configurations in which the counterpoint is of the second species.

$$\begin{gathered}\forall j \in [0, m-2],\ \forall M_{cf}[j] \neq 0 \\ M[0,j] = 12 \ \Rightarrow\ (H[0,j] \leq 4) \ \wedge\ (\neg A(cp)[j] \ \Leftrightarrow\ M_{cf}[j] > 0)\end{gathered} \tag{A.41}$$

2.M2 (modif)

In two-voice composition, two consecutive notes cannot be the same. When writing a three or four voice composition, the 4th-to-last, the 3rd-to-last and the 2nd-to-last may

be the same.
This rule applies to all configurations, to second species counterpoints.
The multi-voice handler has been renamed (and refocused) from `M2_2_3v_melodicIntervalsNotExceedMinorSixth` to `M2_2_3v_twoConsecutiveNotesAreNotTheSame`: the legacy name was misleading, because the body actually forbids the simultaneous repetition of two consecutive notes, not a melodic-interval ceiling. A soft variant `M2_2_3v_twoConsecutiveNotesAreNotTheSame_soft` is now also available for the AdaptiveBench relaxation stage.

`2V:`

$$\begin{gathered}\forall \rho \in positions(m) \\ N[\rho] \neq N[\rho+1]\end{gathered} \tag{A.42}$$

`3V & 4V:`

$$\begin{gathered}\forall p:\ species(p)=2,\ \forall j \in [1, m-3] \\ \Big((N[2,j-1] \neq N[0,j]) \wedge (N[0,j] \neq N[2,j])\Big) \wedge \\ \Big((N[2,m-3] \neq N[0,m-2]) \vee (N[0,m-2] \neq N[2,m-2])\Big)\end{gathered} \tag{A.43}$$

Motion Rules

2.P1 & 1.P2

If the melodic interval of the counterpoint between the thesis and the arsis is larger than a third, then the motion is perceived based on the arsis note.
This rule applies to all counterpoints of the second species.

$$\begin{gathered}\forall j \in [0, m-2] \\ Preal[j] = \begin{cases} P[2,j] & \text{if } M[0,j] > 4 \\ P[0,j] & \text{otherwise} \end{cases}\end{gathered} \tag{A.44}$$

Note: The rule concerning the battuta octave (1.P3) always perceives motion from the last note of the previous measure, but this is already explicitly formulated in the formalization of the rule, so it does not impact this rule or need a new formulation.

2.P3

For three and four part composition, successive fifths on the downbeat are only allowed when they are separated by a third on the upbeat.
This rule applies to all configurations in three and four voice composition in which

second species counterpoints are involved.

$$\forall p_1, p_2 \in \{cf, cp1, cp2, cp3\} \text{ with } p_1 \neq p_2 \text{ and } (species(p_1) = 2 \vee species(p_2) = 2), \quad \forall j \in [0, m-3]$$

$$Cost_{succ_p_cons} = \begin{cases} 0 & \text{if } (H(p_1,p_2)[0,j] \notin Cons_p) \ \vee \ (H(p_1,p_2)[0,j+1] \notin Cons_p) \\ 0 & \text{if } (H(p_1,p_2)[0,j] = 5) \ \wedge \ (H(p_1,p_2)[0,j+1] = 5) \\ & \quad \wedge \Big(H(p_1,p_2)[2,j] \in \{3,4\}\Big) \\ 2 & \text{otherwise} \end{cases} \tag{A.45}$$

A.4 Constraints of the Third Species

Harmonic Rules

3.H1

If five notes follow each other by joint degrees in the same direction, then the harmonic interval of the third note must be consonant.
This rule applies in all configurations, to third species counterpoints.

As discussed in **??**, this rule is currently disabled and must be modified in a future work.

3.H2

If the third harmonic interval of a measure is dissonant then the second and the fourth interval must be consonant and the third note must be a diminution.

$$\forall j \in [0, m-3]$$
$$IsCons[2,j] \vee (IsCons[1,j] \wedge IsCons[3,j] \wedge IsDim[j]) \tag{A.46}$$

where $IsDim[j] = \top$ when the 3$^{\text{rd}}$ note of the measure j is a dimension.

3.H3

It is best to avoid the second and third harmonies of a measure to be consonant with a one-degree melodic interval between them.

This rule applies in all configurations, to third species counterpoints.

$$\forall j \in [0, m-2]$$
$$Cambiatacosts[j] = \begin{cases} cost_{Cambiata} & \text{if } IsCons[1,j] \wedge IsCons[2,j] \wedge M[1,j] \leq 2 \\ 0 & \text{otherwise} \end{cases} \tag{A.47}$$

3.H4

In two voice composition, in the penultimate measure, if the counterpoint is the lowest stratum, then the harmonic interval of the first note should be a minor third.

$$A(cp)[m-2] \implies H[0, m-2] = 3 \tag{A.48}$$

3.H6

If the harmonic triad could not be used on the downbeat, it should be used on the second or third beat.
This rule applies in all three and four voice configurations in which there is a third species counterpoint.

$$\forall j \in [0, m-2]$$
$$(H[0,j] \notin Cons_{h_triad}) \wedge (H[1,j] \notin Cons_{h_triad}) \wedge (H[2,j] \notin Cons_{h_triad}) \tag{A.49}$$
$$\iff cost_{harmonic_triad_3rd_species}[j] = 1$$

3.H7

On the downbeat of a four-part composition, it is now preferred to double the third or the fifth, rather than to use a unison.
This rule applies only in four voice composition, when a third species counterpoint is involved.

This rule is handled by 1.H8 by setting *triad_with_unison_cost* superior to the *double_thirds_cost* and the $double_fifths_cost$.

Melodic Rules

3.M1

Each note and its two beats further peer are preferred to be different.
This rule applies in all configurations, to third species counterpoints.

$$\forall \rho \in positions(m-2)$$
$$MtwoSame_{costs}[i,j] = \begin{cases} cost_{MtwoSame} & \text{if } M^2[\rho] = 0 \\ 0 & \text{otherwise} \end{cases} \tag{A.50}$$

3.M2

No melodic interval between 9 and 11 semitones.
This rule applies to all third-species counterpoints.

$$\begin{gathered} \forall i \in \mathcal{B}, \quad \forall j \in [0, m-2], \\ M[i,j] \notin \{9, 10, 11\} \end{gathered} \tag{A.51}$$

3.M3

The second note of the penultimate measure must be more than a semitone away from the fourth note.
This rule applies to all third-species counterpoints.

$$M_{brut}[1, m-2] + M_{brut}[2, m-2] > 1 \tag{A.52}$$

Motion Rules

3.P1

The motion is perceived based on the fourth note.
This rule applies in all configurations, to third species counterpoints.

$$\begin{gathered} \forall j \in [0, m-2] \\ P_{real}[j] = P[3, j] \end{gathered} \tag{A.53}$$

A.5 Constraints of the Fourth Species

Harmonic Rules

4.H1

Arsis harmonies must be consonant.
This rule applies in all configurations, to fourth species counterpoints.

$$\begin{aligned} &\forall j \in [0, m-2] \\ &H[2, j] \in Cons \end{aligned} \tag{A.54}$$

4.H2 (modif)

In two part composition, if the fourth species counterpoint is the lowest stratum, then no harmonic seventh interval can occur. The same rule is now also enforced, with the same semantics, in three- and four-voice configurations whenever a fourth-species counterpoint is involved (switches `V3_4H2`, `V4_4H2`).
The original wording covered only the two-voice case; the multi-voice extension applies the same arsis-thesis constraint between the fourth-species voice and the new lowest stratum of the surrounding texture.

$$\begin{aligned} &\forall j \in [1, m-2] \\ &A(cp)[j] \;\Rightarrow\; H[0, j] \notin \{10, 11\} \end{aligned} \tag{A.55}$$

4.H3

In the penultimate measure, the harmonic interval of the thesis note must be a major sixth or a minor or major third depending on the cantus firmus pitch.
This rule applies to two voice compositions in which the counterpoint is of the fourth species.

$$H[0, m-2] = \begin{cases} 9 & \text{if } A(cf)[m-2] \\ 3 \vee 4 & \text{otherwise} \end{cases} \tag{A.56}$$

Melodic Rules

4.M1

Arsis half notes should be the same as their next halves in thesis.
This rule applies in all configurations, to fourth species counterpoints.

$$\forall j \in [0, m-2]$$
$$NoSync_{costs} = \begin{cases} cost_{NoSync} & \text{if } M[2,j] \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{A.57}$$

The cost should be set high, because this syncopation is the very nature of the species.

4.M2

Each arsis note and its two measures further peer are preferred to be different.
This rule applies in all configurations, to fourth species counterpoints.

$$\forall j \in [0, m-2]$$
$$MtwomSame_{costs} = \begin{cases} cost_{MtwomSame} & \text{if } N[2,j] = N[2,j+2] \\ 0 & \text{otherwise} \end{cases} \tag{A.58}$$

Motion Rules

4.P1

Dissonant harmonies must be followed by the next lower consonant harmony.
This rule applies in all configurations, to fourth and fifth species counterpoints.

`4sp:`

$$\forall j \in [1, m-1]$$
$$\neg isCons[0,j] \implies M_{brut}[0,j] \in \{1,2\} \tag{A.59}$$

`5sp:`

$$\forall j \in [1, m-1]$$
$$\neg isCons[0,j] \wedge S[0,j] = 4 \implies M^2_{brut}[0,j] \in \{1,2\} \tag{A.60}$$

The use of M^2 is justified because, in the fifth species, every beat can potentially contain a note. In this case, it is necessary to check the melodic interval specifically between

the thesis and the arsis note.

4.P2

If the fourth species counterpoint is not the lowest stratum then no second harmony can be preceded by a unison/octave harmony.
This rule applies in all configurations, to fourth species counterpoints.

$$\begin{gathered}\forall j \in [1, m-2] \\ \neg A[j+1] \;\Rightarrow\; H[2,j] \neq 0 \;\wedge\; H[0,j+1] \notin \{1,2\}\end{gathered} \tag{A.61}$$

A.6 Constraints of the Fifth Species

Harmonic Rules

5.H6 (new) — consonance of third-species weak-beat cells

When a fifth-species voice realises a third-species cell (four quarter notes in a single measure), each of the three weak-beat positions of that cell must be consonant with the lowest stratum, modulo legitimate passing tones.
This rule is implemented directly inside `FifthSpeciesCounterpoint.cpp` under the switch `SP5_H6` (an existing enum entry). It iterates over the three weak beats (+1, +2, +3) of every third-species cell and enforces `isThirdSpeciesArray[j] ⇒ (isConsonance[j] ∨ isPassing(j))`. Phantom continuation cells, second-species cells and fourth-species cells are untouched (the `isThirdSpeciesArray` gate ensures this). The tonal (chord-membership) layer for fifth species cells is handled separately via `H2_4_chordMembershipOnDisjunctWeakBeats` when the species is fifth species.

$$\begin{gathered}\forall p \text{ with } species(p) = 5,\ \forall i \in [0, m-2),\ \forall k \in \{1,2,3\} \text{ s.t. } \mathrm{isThirdSp}(p)[4i+k] \\ j := 4i + k \\ \mathrm{isPassing}(j) := \Big(|N(p)[j] - N(p)[j{-}1]| \in \{1,2\}\Big) \wedge \Big(|N(p)[j{+}1] - N(p)[j]| \in \{1,2\}\Big) \\ \neg\mathrm{isPassing}(j) \implies H(p)[j] \in Cons\end{gathered} \tag{A.62}$$

Gated by `activeConstraints[SP5_H6]`. Unlike the multi-voice rules 2.H2/2.H3, this check measures the interval only against the lowest stratum (the cantus firmus in two voices); for multi-voice configurations the `H2_4_chordMembershipOnDisjunctWeakBeats` call ensures membership in the implied triad.

Melodic Rules

5.M1

No unison between two consecutive notes.
This rule applies to all fifth species counterpoints.

$$\begin{gathered}\forall j \in [0, m-2], \\ N[0,j] \neq N[1,j], \\ N[1,j] \neq N[2,j], \\ N[2,j] \neq N[3,j] \\ N[3,j] \neq N[0,j+1]\end{gathered} \tag{A.63}$$

5.M2

For each measure from the second to the second-to-last, if a note is part of the fourth species and the corresponding subsequent note is constrained, then these two notes must not be the same.
This rule applies to all fifth species counterpoints.

$$\begin{gathered}\forall j \in [1, m-2] \\ (S[0,j] = 4 \wedge S[2,j] = 4) \implies N[0,j] \neq N[2,j]\end{gathered} \tag{A.64}$$

Rhythmic Rules

The fifth species does not operate in the same manner as the previous four. In this case, the S array is used to indicate the species assigned to each note, describing both its rhythmic value and the rules that govern it. The rhythmic constraints outlined below are defined on this S array, and from it one can deduce, via the equations of this section, which set of rules applies to every note.

To make the formalization more readable, we define $IsSx$:

$$\begin{gathered}\forall x \in \{0,1,2,3,4\},\ \forall \rho \in positions(m) \\ IsSx[\rho] = \begin{cases} \top & \text{if } S[\rho] = x \\ \bot & \text{otherwise} \end{cases}\end{gathered} \tag{A.65}$$

5.R1

There must always be a note in thesis and in arsis, except the very first thesis and the very last arsis.
This rule applies in all configurations, to fifth species counterpoints.

$$\begin{gathered}\forall j \in [0, m-1] \\ \neg IsS_0[0, j] \quad \text{where } j \neq 0\end{gathered} \tag{A.66}$$

$$\forall j \in [0, m-1]: \quad \neg IsS_0[2, j] \quad \text{where } j \neq m-1 \tag{A.67}$$

5.R2

The 4th species can only exist in first and third beat.
This rule applies in all configurations, to fifth species counterpoints.

$$\begin{gathered}\forall i \in \{1, 3\},\ \forall j \in [0, m-1] \\ \neg IsS_4[i, j]\end{gathered} \tag{A.68}$$

5.R3

A 4th species in the third beat necessarily implies a 4th species in the first beat of the following measure and vice versa. The fourth beat should then have no note.
This rule applies in all configurations, to fifth species counterpoints.

$$\begin{gathered}\forall j \in [0, m-2] \\ IsS_4[2, j] \iff IsS_4[0, j+1]\end{gathered} \tag{A.69}$$

$$\begin{gathered}\forall j \in [0, m-2] \\ IsS_4[2, j] \implies IsS_0[3, j]\end{gathered} \tag{A.70}$$

5.R4

A 3rd species cannot be followed by no note.
This rule applies in all configurations, to fifth species counterpoints.

$$\begin{gathered}\forall \rho \in positions(m-1) \\ IsS_3[\rho] \implies \neg IsS_0[\rho+1]\end{gathered} \tag{A.71}$$

5.R5

Only 3rd species and 4th species are used.
This rule applies in all configurations, to fifth species counterpoints.

$$\begin{gathered}\forall \rho \in positions(m) \\ \neg IsS_1[\rho] \ \wedge \ \neg IsS_2[\rho]\end{gathered} \tag{A.72}$$

5.R6

The first and penultimate measures are linked to the 4th species.
This rule applies in all configurations, to fifth species counterpoints.

$$IsS_0[0,0] \wedge IsS_0[1,0] \wedge IsS_4[2,0] \tag{A.73}$$

$$IsS_4[0,m-2] \wedge IsS_0[1,m-2] \wedge IsS_4[2,m-2] \tag{A.74}$$

5.R7

An arsis note, regardless of its species, must be the consonance just below the thesis note if the latter belongs to the fourth species.
This rule applies in all configurations, to fifth species counterpoints.

$$\begin{gathered}\forall j \in [1,m-2] \\ \neg IsCons[0,j] \wedge IsS_4[0,j] \implies M^2_{brut}[0,j] \in \{-1,-2\} \wedge IsCons[2,j]\end{gathered} \tag{A.75}$$

5.R8

To avoid multiple same final solutions, the non-displayed notes (whose values in S are 0) must be of the same value as the note on the next beat.
This rule applies in all configurations, to fifth species counterpoints.

$$\begin{gathered}\forall \rho \in positions(m-1) \\ IsS_0[\rho] \implies (N[\rho] = N[\rho+1])\end{gathered} \tag{A.76}$$

5.R9

When composing two counterpoints of the fifth species, the counterpoints must use different species for at least half of the beats. If there are three counterpoints of the fifth species, this rule applies between the fifth species counterpoint with the minimum range and the one with the maximum range. The middle one remains unconstrained by this rule.

This rule applies when there are multiple counterpoints of the fifth species.

$$species(cp_1) = species(cp_2) = 5 \iff \sum_{i=0}^{3} \sum_{j=0}^{m-1} (S(cp_1)[i,j] = S(cp_2)[i,j]) < \frac{sm}{2} \quad \text{(A.77)}$$

Determining the Constraints that Apply to Each Note

$$\begin{gathered} \forall x \in \{3,4\},\ \forall cst_x \in Constraints(x),\ \forall V \in Variables(cst_x) \\ \Big(\bigwedge_{v \in V} IsS_x[v_{pos}] \Big) \implies cst_x(V) \end{gathered} \quad \text{(A.78)}$$

where $Constraints(x)$ is the set of constraints of the species x, and $Variables(cst_x)$ is the set of variables concerned by the constraint cst_x, and v_{pos} is the position of the v related note in the array N.

Special case for 1.H1 and 4.H1

$$\forall j \in [0, m-1]: \quad IsS_3[0,j] \implies (H[0,j] \in Cons) \quad \text{(A.79)}$$

$$\begin{gathered} \forall j \in [0, m-2] \\ IsS_4[2,j] \implies (H[2,j] \in Cons) \end{gathered} \quad \text{(A.80)}$$

A.7 Tables of rules scope

	All species	**First sp.**	**Second sp.**	**Third sp.**	**Fourth sp.**	**Fifth sp.**
nV	G1–5, G7, G9 **(new)**, 1H1, 1H4, 1H6, 1P2, 1P4 **(modif)**, 1P8 **(new)**, 1P9 **(new, sp1–3 only)**		2H2 **(modif)**, 2H7 **(new)**, 2P1	3H1, 3H2, 3H3, 3M1, 3M2, 3M3, 3P1, 2H7 **(new)**	4H1, 4H2 **(modif, now nV)**, 4M1, 4M2, 4P1, 4P2	5.R1–R8, 5M1, 5M2, 5H6 **(new)**, 2H7 **(new, on 3sp cells)**
2V	1H2, 1H3, 1H7, 1P1	1M2	2H3, 2M1, 2M2 **(modif)**	3H4	4H2, 4H3	
2V3V	G6, 1H5					
3V	1H7, 1H8, 1H10, 1P1					
3V4V	1H3, 1H12, 1H13, 1M4, 1P6, 1P7	1M2	2M2 **(modif)**, 2P3	3H6	4P3, 4P5, 4H2 **(new for 3v/4v)**	5.R9
4V	1H8, 1P1			3H7		

Table A.1: Table of Scopes. Annotations: **(new)** = constraint introduced post-fork, **(modif)** = constraint whose body or scope was modified post-fork. The fork point is commit `a2613d9` (Tom Lai's last commit, May 2025).

Appendix B

User references

Here you will find a detailed overview of the practical tools available to help you use the solver and its settings in as many different configurations as possible.

B.1 Configuration and presets

The generation pipeline is parameterized through external CSV configuration files and lightweight command-line overrides. At runtime, all configuration data is consolidated into the single structure `GenerationCase`. Concretely, the selected cantus firmus is resolved first (either from the cantus dataset when `-c CF_ID` is provided, or directly from `-cf-notes`), and then the preset is applied by copying its parameter arrays into `GenerationCase`.

This design makes generations reproducible: a run is fully determined by the cantus input, the preset name, and the few explicit overrides passed through the CLI or the API. This section develops the technical details of this initialization phase.

B.1.1 Command-line interface (main.cpp)

The CLI is designed to make single experiments and batch runs easy.

Core arguments. A generation run starts with:

```
./GenerateCounterpoint <nb_voix> <especes...> [options]
```

where the number of species arguments must be `nb_voix - 1` (the cantus firmus is always implicit as the first voice). The CLI checks that `nb_voix` is in the supported range and that each species argument is valid.

Main options. The most important options are:

- `-cf-notes n1,n2,...` (direct cantus firmus notes),
- `-c CF_ID` (select cantus firmus from `cantus_firmus.csv`),
- `-preset NAME` (select preset from `presets.csv`),
- `-v vt1,vt2,...` (override voice types),
- `-m lex|total|mixed|pond` (objective mode),
- `-t timeout_ms` and `-s stagnation_ms`,
- `-o output_root` and `-subdir NAME`.

These options only build the configuration; the CSP itself is constructed later via `create_problem(...)` once `resolve_cf(...)` and `apply_preset(...)` have succeeded:

- resolve_cf either loads the cantus firmus data from file if `-c CF_ID` has been specified, or computes them from direct value if not.
- apply_preset is the function responsible for loading the presets from files and applying them to the `GenerationCase` struct

A cantus firmus must be provided either by `-c CF_ID` (dataset mode) or by `-cf-notes`; otherwise `resolve_cf` fails and the run stops early.

Listing and campaign modes. For usability, the CLI also supports non-generation commands:

- `-list-cf`, `-list-presets`, `-list-campaigns`,
- `-campaign <id>` to run a batch defined in `campaigns.csv`.

In campaign mode, each row is converted into a `GenerationCase`, then executed exactly like a normal run.

Precedence rules. CLI defaults are applied first, then explicit CLI overrides, and finally the configuration is completed by resolving external data (cantus firmus dataset and presets). Cantus selection is resolved via `resolve_cf`: when `-c` is used, `use_preset_cf` is enabled and the cantus is loaded from `cantus_firmus.csv`; otherwise the cantus is taken from `-cf-notes`.

B.1.2 Programmatic API (GenerationsAPI)

In addition to the CLI, the project exposes a programmatic entrypoint to generate counterpoint from other modules without relying on command-line parsing. The API is defined in `GenerationsAPI.hpp/cpp` as:

```
GenerationResult generate_counterpoint(const GenerationInput& input)
```

where `GenerationInput` is a lightweight structure containing the cantus firmus as MIDI notes, the species list and the voice types, plus optional overrides (preset name, timeouts, objective mode, output options, and verbosity). Internally, the API converts `GenerationInput` into a full `GenerationCase`, calling the same resolution helpers as the CLI (`resolve_cf` and `apply_preset`), then delegates execution to the shared engine function `Generations::run_generation_case(gc, save_outputs, verbose)`.

The programmatic API follows the same pattern as the CLI regarding precedence rules : `GenerationInput` is converted into `GenerationCase`, then `resolve_cf` and `apply_preset` are called before delegating to `run_generation_case`.

Unlike the CLI, the API currently accepts the cantus firmus only in direct form (MIDI notes), and always uses `use_preset_cf = false`.

For examples of use, see "`QUICK TEST on API`" part in Tests/main.cpp

B.1.3 Presets (`presets.csv`)

Presets define the default behaviour of the generator in terms of *cost weights* and *priority ordering*. They do not contain musical input: the cantus firmus is now provided independently (by ID or directly as MIDI notes), while presets focus on how solutions are evaluated.

The `presets.csv` file follows a simple schema:

- **name**: preset identifier (used via `-preset NAME`)
- **melodic_params**: 8 integers (melodic costs)
- **general_params**: 8 integers (general/harmonic costs)
- **specific_params**: 7 integers (species-specific costs)
- **importance**: 14 integers (lexicographic importance ranking of cost families)
- **borrow_mode**: integer describing the borrowing policy (e.g., forbidden/allowed)
- **description**: human-readable label (no commas, display-only)

A key practical detail is that list-valued columns (the parameter arrays and importance vector) are stored as a single CSV cell using **space-separated integers**. This makes presets easy to edit manually while keeping them compact.

At startup, the preset file is loaded into a map `name` $\rightarrow$ `PresetEntry`. When a run selects a preset, it is applied by copying its arrays into the current `GenerationCase`:

`melodic_params`, `general_params`, `specific_params`, `importance`, `borrow_mode`.

If the preset name is unknown, generation stops early with an explicit error.

Example presets. The repository currently provides presets such as `default`, `strict`, `musique`, `variete_haute`, etc. Each preset corresponds to a coherent "profile" (e.g., stricter melodic penalties, increased weight on variety, or disabling borrow). The CLI can list available presets and their descriptions.

B.1.4 GenerationCase struct

Here is an explaination of the relevant parameters given through "GenerationCase" to the generations process:

- **cf** : the cantus firmus, in midi values. It is either (i) resolved from `cf_id` when `use_preset_cf` is enabled, or (ii) provided directly through `-cf-notes`. If neither mode provides notes, the run stops early.
- **cf_id** : identifier of the cantus firmus when using cantus firmus presets (`-c CF_ID`). It is used to resolve the cantus from `cantus_firmus.csv`, and enables `use_preset_cf`.
- **use_preset_cf** : boolean that selects the cantus input mode. If `true`, the cantus is loaded from `cantus_firmus.csv` using `cf_id`; otherwise the solver expects `cf` to be provided directly (via `--cf-notes`).
- **cf_name** : human-readable cantus name. It is loaded from `cantus_firmus.csv`, or defaults to `"direct_cf_`$< cf_scale >$`"` followed by the corresponding scale.
- **cf_scale** : indicative scale label. It is loaded from `cantus_firmus.csv`, or computed for direct cantus firmus.
- **n_voices** : the total number of voices desired (cantus firmus included)
- **spList** : the list of the species of the counterpoints to generate

- **v_type** : the relative height of the counterpoint, compared to the first not of the cantus firmus. Each voice has a 2-octave ranged, centered on **cf[0] + v_type*6**, in MIDI value
- **melodic_params** : array of factors for the corresponding melodic costs (unison, m2, M2, sauts, m3, M3, P4, sup4)
- **general_params** : array of factors for the corresponding other costs (octave, P5, m6, M6, P8, mvtCommun, variete, triade)
- **specific_params** : array of factors for the corresponding specific costs (parfaite, faiblet, mvtDirect, syncope, cambiata, melodique, fux2)
- **importance** : array of the **unique** ranking of the corresponding cost categories, from 1 to 14 (borrow, fifth, octave, succ, variety, triad, direct, motion, penult, cambiata, triad3, m2, syncopation, melodic).
- **borrow_mode** : if activated, notes that are not on the cantus firmus scale are authorized, in exchange for a bigger **borrow** cost
- **obj_mode** : decides the general cost objective by defining the **objectiveMode** variable. With OBJECTIVE_LEX, the **cost()**‘ function returns the original cost array. If not, it builds an aggregated sum and adds it at the beginning of the array.
- **timeout_ms** : the maximal duration of the total BAB search (in ms)
- **stagnation_ms** : the maximal duration of one iteration of the BAB search (in ms)
- **output_root** : the path to the global output directory (generally the ’result’ folder)
- **output_subdir** : the name of an eventual subfolder

Most runtime-related fields have default values in `GenerationCase` (e.g., preset name, timeout/stagnation, objective mode, output root), and are overridden only when explicitly set by the CLI/API.

B.1.5 GenerationResult struct

As stated before, the generation process returns a `GenerationResult` structure containing the search outcome, the generated solution, and optional output file paths. Here is an explaination of its fields :

- **success** : boolean indicating whether at least one solution was found. It is set to `true` when the BAB search returns a best solution, and `false` otherwise.
- **termination** : textual description of the stopping reason of the search. Typical values are `EXHAUSTIVE`, `TIMEOUT_MAX`, or `STAGNATION`.
- **nb_solutions** : the total number of solutions found during the BAB search.
- **nb_improvements** : the number of times a newly found solution strictly improved the current best cost.
- **best_cost** : the scalar cost value of the best solution found, according to the current objective mode.
- **ms_first_solution** : the elapsed time (in ms) before the first valid solution was found.
- **ms_last_improve** : the elapsed time (in ms) at which the last improvement of the best solution was found.
- **ms_total** : the total runtime of the search (in ms), until completion or forced stop.
- **stats** : the raw `Gecode::Search::Statistics` object returned by the solver. It contains low-level information such as the number of nodes explored, failures, propagations, restarts, and maximal depth.
- **full_solution** : the full solution vector returned by the solver, as MIDI values. It contains all generated notes, concatenated voice after voice, in the internal order used by the problem.
- **generated_voices** : the same result, but split voice by voice. Each element is a pair `(notes, species)` containing the note vector of one generated counterpoint and its corresponding species. This representation is easier to reuse for export or downstream processing.
- **improvements** : list of all best-solution updates during the search, stored as tuples `(solution_number, elapsed_ms, cost)`.
- **checkpoints** : periodic snapshots of the search, stored as tuples `(solution_number, elapsed_ms, best_cost)`. They are mainly used for reporting and benchmark analysis.
- **solutions_log** : full chronological log of all solutions found during the BAB search, stored as tuples `(solution_number, elapsed_ms, scalar_cost, lex_vector)`. This is the most detailed record of the optimization process.

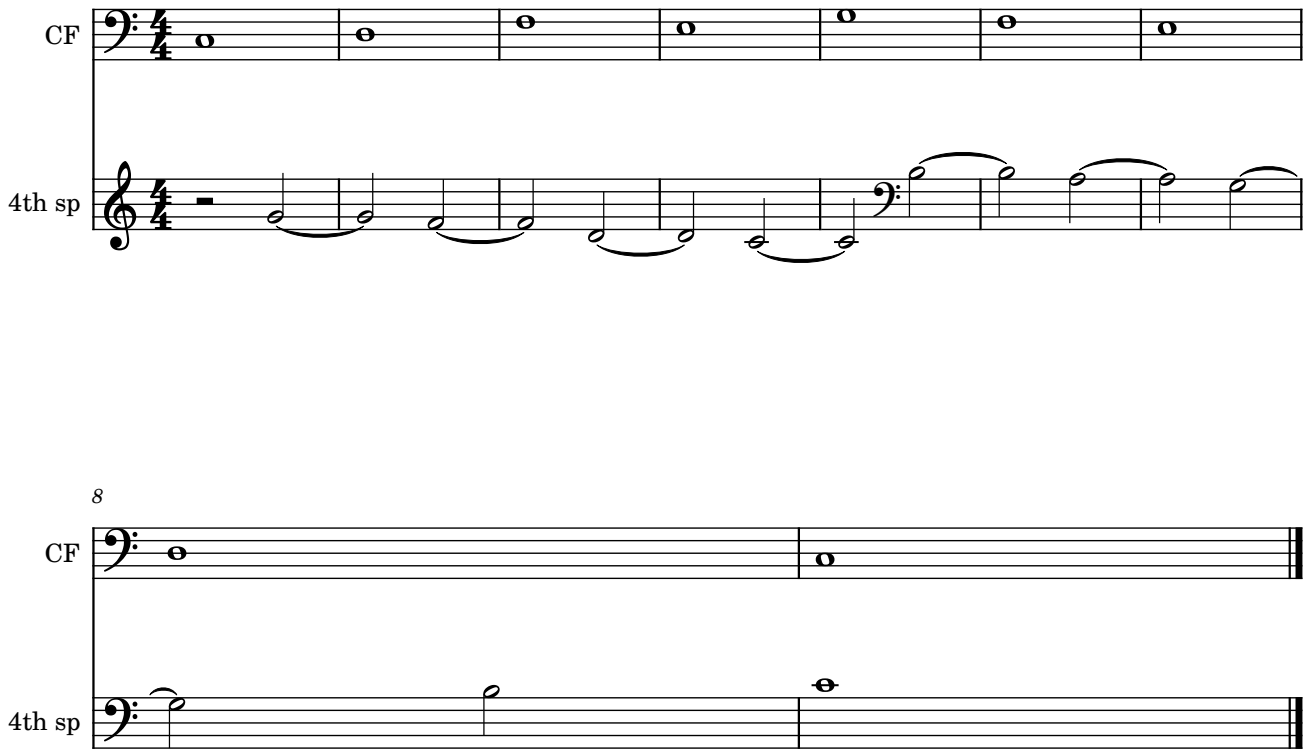
- **midi_path** : path of the generated MIDI file, when `save_outputs` is enabled and at least one solution was found.
- **txt_path** : path of the generated text report, when `save_outputs` is enabled and at least one solution was found.
- **csv_path** : path of the generated CSV report, when `save_outputs` is enabled and at least one solution was found.
- **error_txt_path** : path of the generated error report, when `save_outputs` is enabled but no solution was found.
- **resolved_cf** : the actual cantus firmus used during generation, in MIDI values. This is useful because the cantus may come either from presets or from direct API input.
- **resolved_cf_name** : the resolved human-readable name of the cantus firmus.
- **resolved_cf_scale** : the resolved scale label associated with the cantus firmus.

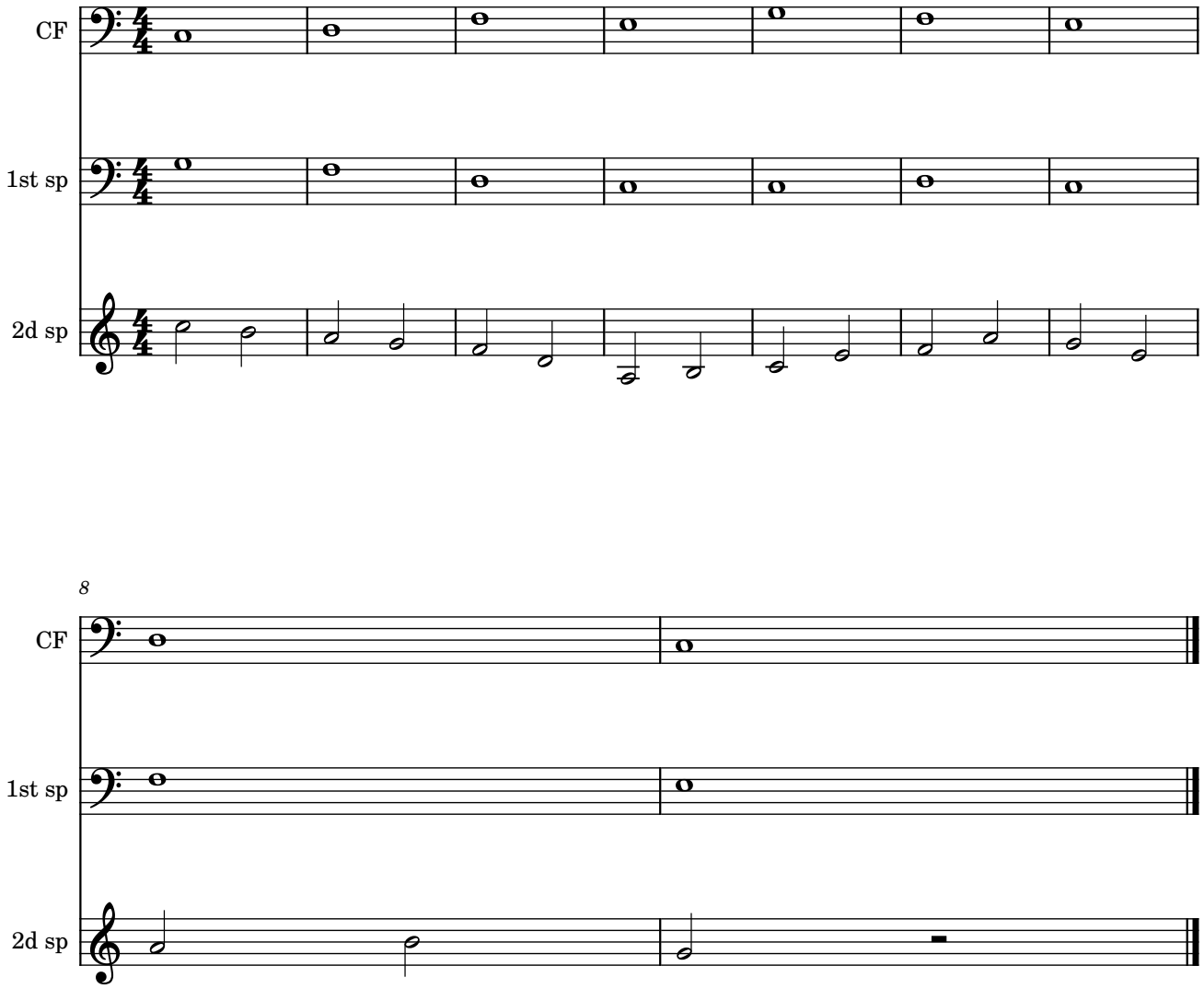
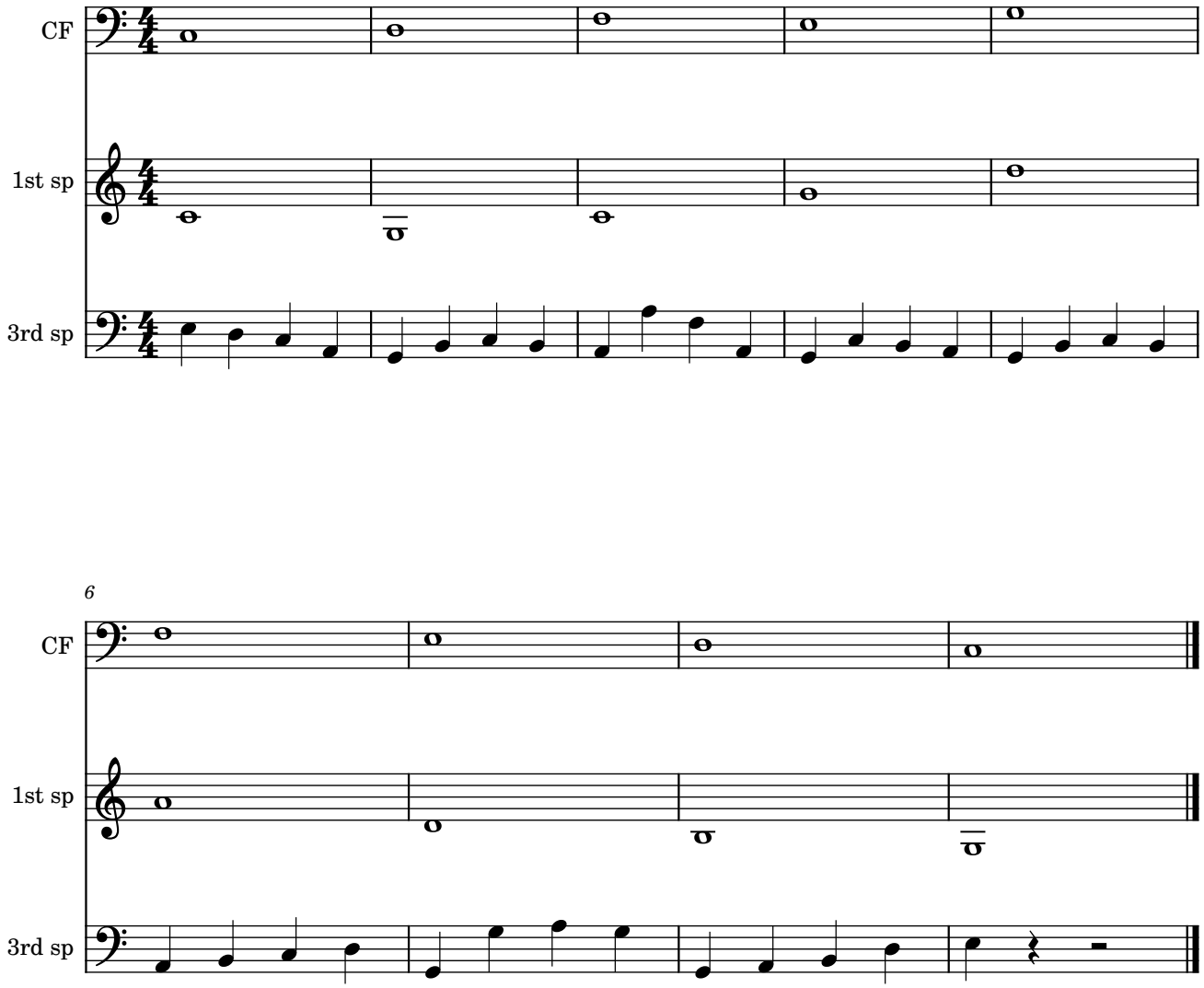
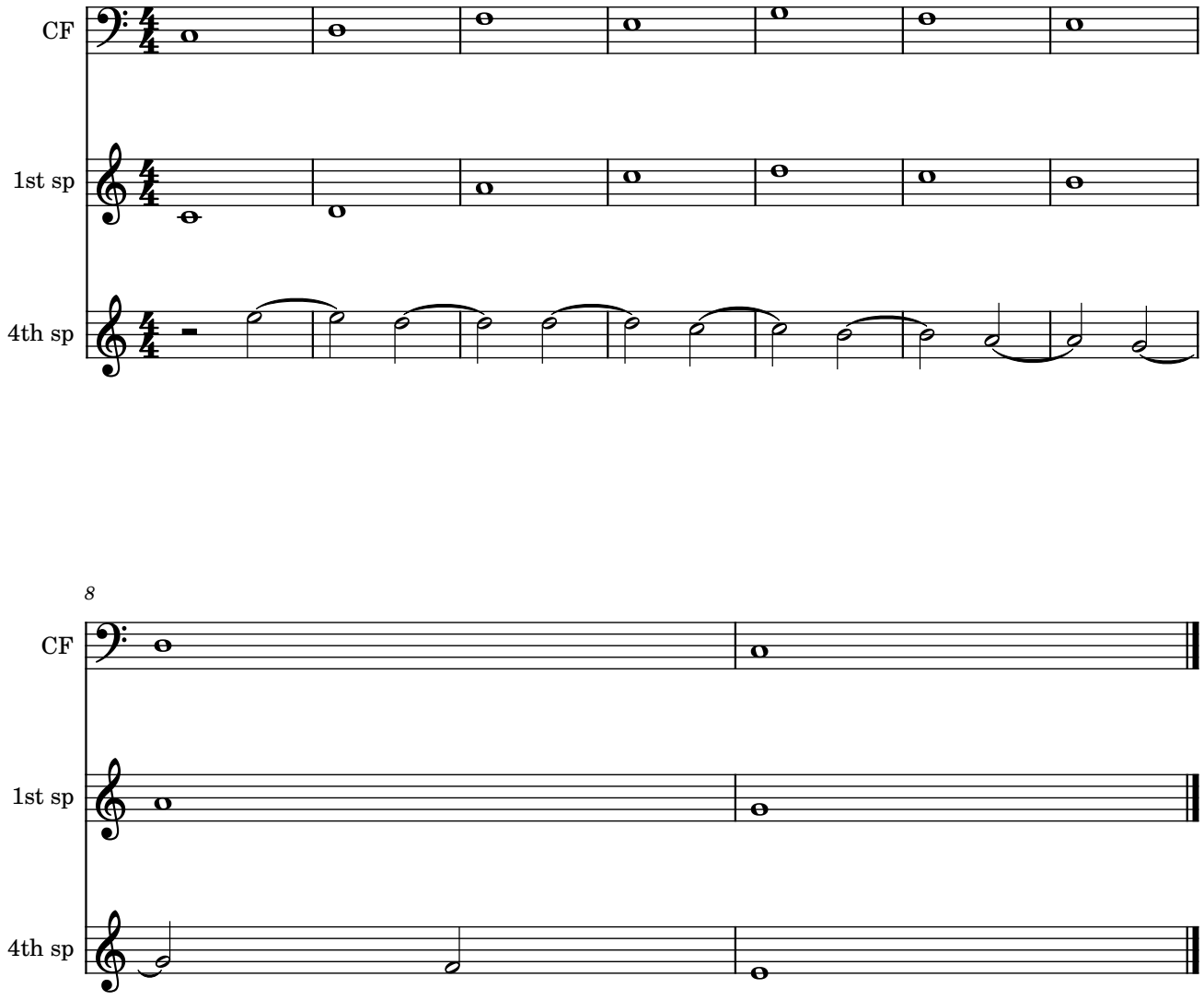
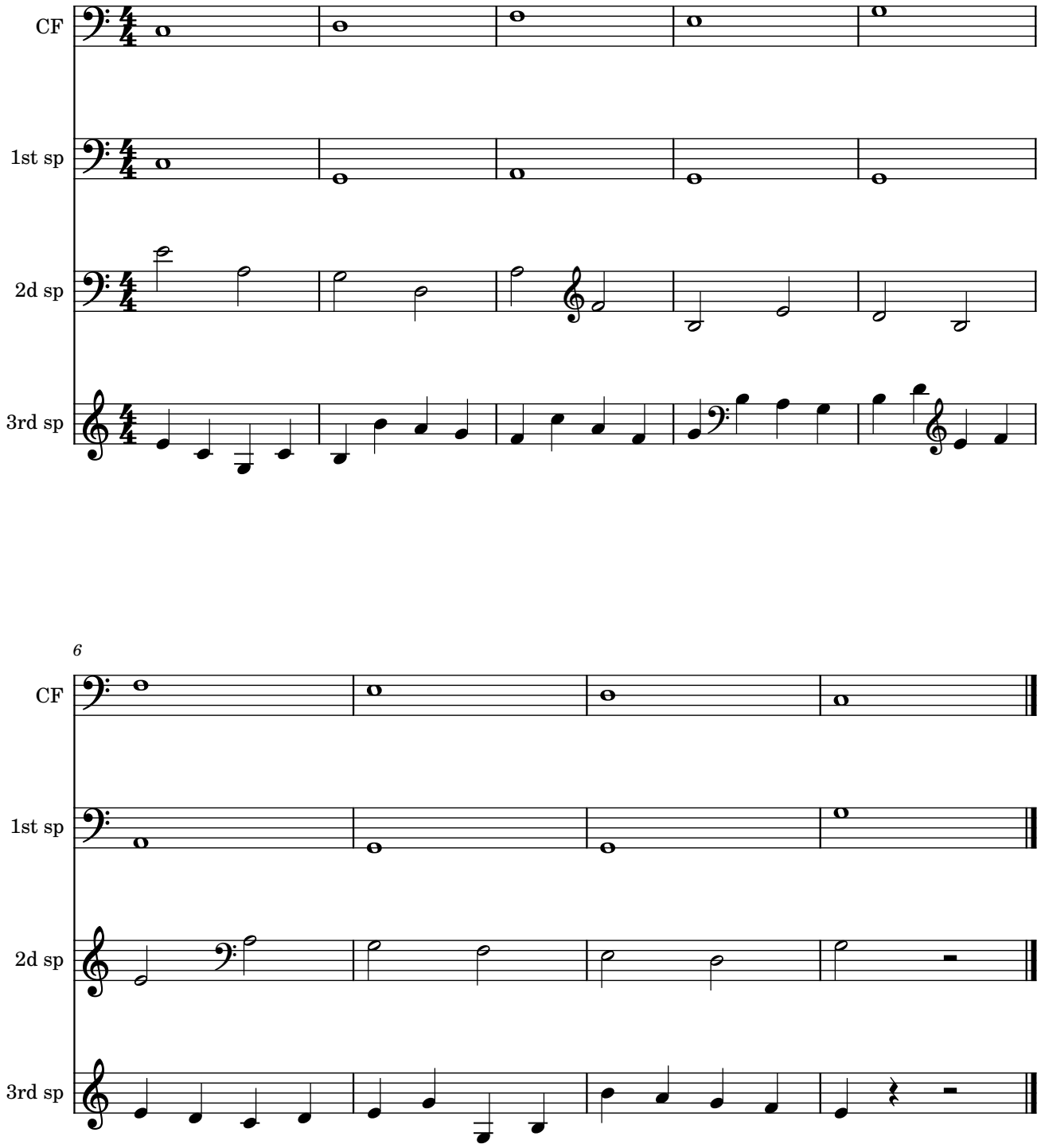
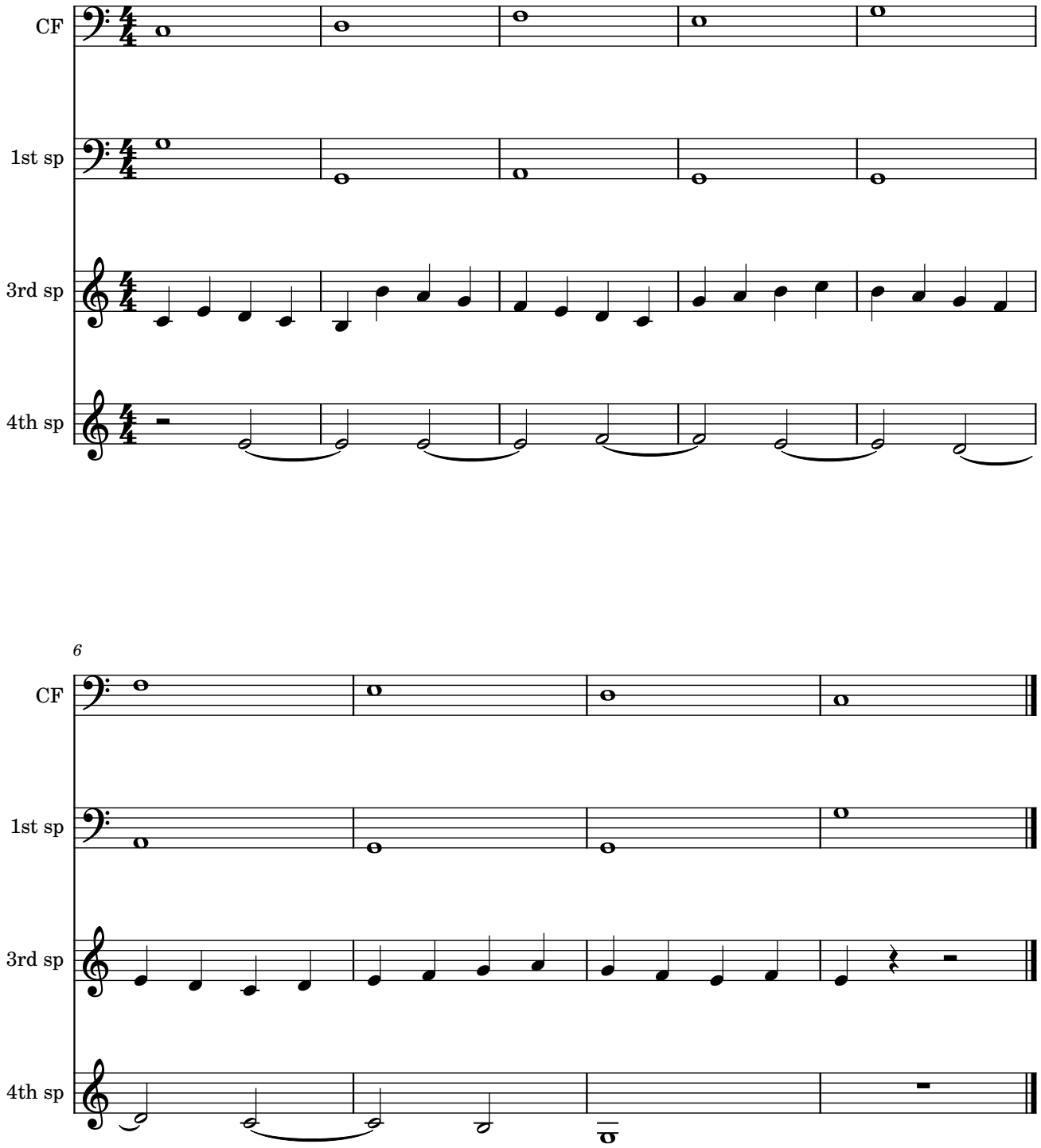
In practice, `GenerationResult` acts both as a search report and as a reusable output container: it stores the final musical result, the main performance indicators of the search, and the paths of the exported files when generation is run with output saving enabled.

Appendix C

Generated counterpoints exemples

Here are several examples of counterpoints generated using the FuxCP solver over the reference CF. A folder containing all the generated counterpoints is available on the project's GitHub repository.



Two-voice counterpoint, fourth target species above the reference cantus firmus in C major.

Three-voice counterpoint, second target species above the reference cantus firmus.

Three-voice counterpoint, third target species above the reference cantus firmus.

Three-voice counterpoint, fourth target species above the reference cantus firmus.

Grand mélange, four voices: first, second and third species simultaneously above the reference cantus firmus

Grand mélange, four voices: first, third and fourth species simultaneously above the reference cantus firmus

Appendix A

Complete FuxCP Code

The full FuxCP source tree is included below for archival reference. The repository can also be browsed online at https://github.com/AarnorDeDardaliel/FuxCP5. Files are grouped by role: counterpoint problem, voices/parts, species, shared utilities, tests, and build/configuration. Blocks marked with `// [Claude Code]` indicate the constraints, the bash CLI wrapper, the CSV loader and the generation engine that were drafted with the assistance of Claude Code.

A.1 Counterpoint problem headers

A.1.1 CounterpointProblem.hpp

```
//
// Created by Luc Cleenewerk and Diego de Patoul.
//

#ifndef MYPROJECT_COUNTERPOINTPROBLEM_HPP
#define MYPROJECT_COUNTERPOINTPROBLEM_HPP

#include "../Utilities.hpp"
#include "../Parts/FirstSpeciesCounterpoint.hpp"
#include "../Parts/SecondSpeciesCounterpoint.hpp"
#include "../Parts/CantusFirmus.hpp"

/** Types of search engines */
enum {
    dfs_solver, //0
    bab_solver, //1
};

/** Modes d'optimisation des co ts */
enum ObjectiveMode {
    OBJECTIVE_LEX,   // ordre lexicographique (actuel)
    OBJECTIVE_TOTAL, // somme globale des co ts
    OBJECTIVE_MIXED,  // mix somme globale + priorit  lexicographique pond r e
    OBJECTIVE_PONDERED // somme lexicographique pond r e
};


static const char* obj_mode_label(ObjectiveMode m) {
    switch (m) {
        case OBJECTIVE_TOTAL: return "total (somme globale)";
        case OBJECTIVE_MIXED: return "mixed (somme + score lex pond\u00e9r\u00e9)";
        case OBJECTIVE_PONDERED: return "pond (score lex pond\u00e9r\u00e9)";
        default:              return "lex (lexicographique par priorit\u00e9)";
    }
}
static const char* obj_mode_short(ObjectiveMode m) {
    switch (m) {
        case OBJECTIVE_TOTAL: return "total";
        case OBJECTIVE_MIXED: return "mixed";
        case OBJECTIVE_PONDERED: return "pond";
        default:              return "lex";
    }
}
static const char* cost_label(ObjectiveMode m) {
    switch (m) {
        case OBJECTIVE_TOTAL: return "Co\u00fbt total";
```

```
47          case OBJECTIVE_MIXED: return "Score mixte";
48          case OBJECTIVE_PONDERED: return "Score pond r  ";
49          default:              return "Somme lex (indic.)";
50      }
51  }
52
53  static ObjectiveMode parse_obj_mode(const string& s) {
54      if (s == "total") return OBJECTIVE_TOTAL;
55      if (s == "mixed") return OBJECTIVE_MIXED;
56      if (s == "pond") return OBJECTIVE_PONDERED;
57      return OBJECTIVE_LEX;
58  }
59
60
61  /**
62   * This (abstract) class gives a general model for a counterpoint problem.
63   */
64  class CounterpointProblem : public IntLexMinimizeSpace{
65  protected:
66      CantusFirmus* cantusFirmus;
67
68      int nMeasures;      /// the number of measures in the score to generate
69      int n_costs;
70      int n_unique_costs;
71      Stratum* lowest;
72      IntVarArray successiveCostArray;
73      IntVarArray triadCostArray;
74      Part* counterpoint_1;
75      Part* counterpoint_2;
76      Part* counterpoint_3;
77      Stratum* upper_1;
78      Stratum* upper_2;
79      Stratum* upper_3;
80      IntVarArray unitedCosts;
81      IntVarArray sortedCosts;
82      IntVarArray solutionArray;
83      vector<int> importance;
84      vector<string> importanceNames;
85      vector<string> unitedCostNames;
86      IntVarArray orderedFactors;
87      IntVarArray finalCosts;
88      vector<IntVarArray> sorted_voices;
89      vector<IntVarArray> measures_order;
90      unordered_map<string, int> prefs;
91      vector<vector<string>> costLevels;
92
93      IntVar globalCost;
94      ObjectiveMode objectiveMode = OBJECTIVE_LEX;
95      IntVar objectiveCostSum;
96      // vector<int> species;        /// the species of the counterpoint to generate
97
98      IntVarArray combinedCosts;
99
100     IntVarArray problemRelaxationCosts;
101     IntVar totalRelaxationCost;
102     bool hasRelaxation = false;
103
104 public:
105     /**
106      * Constructor of the class.
107      * @param cf a vector<int> representing the cantus firmus.
108      * @param k the key of the score. it takes values from the notes in headers/Utilities.hpp
109      * @param lb the lowest note possible for the counterpoints in MIDI
110      * @param ub the highest note possible for the counterpoints in MIDI
111      */
112     CounterpointProblem(vector<int> cf, int v_type, vector<int> m_costs, vector<int> g_costs,
        vector<int> s_costs, vector<int> imp, int nV,
113         ObjectiveMode objMode = OBJECTIVE_LEX);
114
115     CounterpointProblem(CounterpointProblem& s);
116     virtual IntLexMinimizeSpace* copy();
117
118     virtual string to_string() const;
119
120     double getCost() const;
121
122
123     /**
124      * Constrain method for bab search
125      * @todo modify this function if you want to use branch and bound
126      * @param _b a space to constrain the current instance of the Problem class with upon
        finding a solution
127      */
128     void constrain(const IntLexMinimizeSpace& _b);
129
130     LinIntExpr getEnhancedMeanCostsSum(IntVarArray costsArray);
131     LinIntExpr getEnhancedPonderedCostsSum(IntVarArray costsArray);
132     LinIntExpr getPonderedCostsSum(IntVarArray costsArray);
133
134     IntVarArgs cost() const;
```

```
135
136     /// Getters
137     //Part* getCounterpoint(){ return counterpoint_1; }
138     ///destructor
139     virtual ~CounterpointProblem(){ delete cantusFirmus; delete lowest; delete counterpoint_1;
        delete counterpoint_2; delete counterpoint_3; delete upper_1;
140         delete upper_2, delete upper_3;}
141
142     Home getHome();
143
144     void setPreferenceMap(vector<string> importance_names);
145
146     void orderCosts();
147
148     int getSize();
149
150     int* return_solution();
151
152     IntVarArray getSolutionArray();
153
154     Stratum* getLowest();
155
156     // --- Added getters for counterpoint_2 and counterpoint_3 (Modified by Tom Lai)
157     Part* getCantusFirmus();
158     Part* getCounterpoint_1();
159     Part* getCounterpoint_2();
160     Part* getCounterpoint_3();
161     // ---
162
163     Stratum* getUpper_1(){ return upper_1; }
164     Stratum* getUpper_2(){ return upper_2; }
165     Stratum* getUpper_3(){ return upper_3; }
166
167     int* get_species_array_5sp(int ctp_index);
168     int* get_extended_cp_domain(int ctp_index);
169     int  get_ext_cp_domain_size(int ctp_index);
170
171     vector<IntVarArray> getMeasuresOrder() {
172         return measures_order;
173     }
174     // Helper functions for setStrata
175     void setVoiceNote(IntVarArray& voices, int voiceIndex, Part* part, int measureIndex);
176     void setStrataAtPosition(int measureIndex, int position, int nVoices, IntVarArray& );
177     void setVoiceLowestFlags(int measureIndex, int nVoices, int size);
178     void setMelodicIntervalConstraints(int measureIndex, int nVoices);
179
180     void setStrata();
181
182     IntVarArray get_combinedCosts(){ return combinedCosts; }
183     void computeCombinedCosts();
184
185     void uniteRelaxationCosts();
186     IntVarArray getProblemRelaxationCosts(){ return problemRelaxationCosts; }
187 };
188
189
190 /**
191  * Creates a search engine for the given problem
192  * Should only be used when using OM, otherwise you can create the solver etc in the main file
193  * @todo Modify this function to add search options etc
194  * @param pb an instance of the Problem class representing a given problem
195  * @param type the type of search engine to create (see enumeration in headers/gecode_problem.
        hpp)
196  * @return a search engine for the given problem
197  */
198 Search::Base<CounterpointProblem>* make_solver(CounterpointProblem* pb, int type);
199
200
201 /**
202  * Returns the next solution space for the problem
203  * Should only be used when using OM
204  * @param solver a solver for the problem
205  * @return an instance of the Problem class representing the next solution to the problem
206  */
207 CounterpointProblem* get_next_solution_space(Search::Base<CounterpointProblem>* solver);
208
209 #endif //MYPROJECT_COUNTERPOINTPROBLEM_HPP
```

A.1.2 CounterpointUtils.hpp

```
1 //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 // This file contains the declarations of the main functions to create problems and
      counterpoints.
4 //
5
6 #ifndef COUNTERPOINTUTILS_HPP
```

```
#define COUNTERPOINTUTILS_HPP

#include "../Utilities.hpp"
#include "TwoVoiceCounterpoint.hpp"
#include "ThreeVoiceCounterpoint.hpp"
#include "FourVoiceCounterpoint.hpp"
#include "../Parts/FirstSpeciesCounterpoint.hpp"
#include "../Parts/SecondSpeciesCounterpoint.hpp"
#include "../Parts/ThirdSpeciesCounterpoint.hpp"
#include "../Parts/FourthSpeciesCounterpoint.hpp"
#include "../Parts/FifthSpeciesCounterpoint.hpp"

using namespace std;
using namespace Gecode;


/**
 * This function creates the appropriate Part given the species of the counterpoint requested.
 * @return a pointer to a Part object
*/
Part* create_counterpoint(Home home, int species, int nMeasures, vector<int> cantusFirmus, int
    lowerBound, int upperBound, Stratum* low,
    CantusFirmus* c, int v_type, vector<int> m_costs, vector<int> g_costs, vector<int> s_costs,
    int bm, int nV);

/**
 * This function creates the appropriate counterpoint problem given the number of counterpoints
    (size of the species list) requested.
 * @return a pointer to a counterpoint problem object
*/
CounterpointProblem* create_problem(vector<int> cf, vector<Species> sp, vector<int> v_type,
    vector<int> m_costs, vector<int> g_costs,
    vector<int> s_costs, vector<int> imp, int bm, ObjectiveMode objMode = OBJECTIVE_LEX);

bool notInt(char* argv);

bool has_solution(CounterpointProblem* problem);



#endif
```

A.1.3 TwoVoiceCounterpoint.hpp

```
//
// Created by Damien Sprockeels on 13/06/2024.
// Extended and developed by Luc Cleenewerk and Diego de Patoul up to August 2024.
//

#ifndef MYPROJECT_TWOVOICECOUNTERPOINT_HPP
#define MYPROJECT_TWOVOICECOUNTERPOINT_HPP

#include "../Utilities.hpp"
#include "CounterpointProblem.hpp"
#include "../Parts/FirstSpeciesCounterpoint.hpp"
#include "../Parts/SecondSpeciesCounterpoint.hpp"
#include "../Parts/CantusFirmus.hpp"

/**
 * This class models a counterpoint problem with 2 voices.
 */
class TwoVoiceCounterpoint : public CounterpointProblem{
protected:
    Species species;        /// the species of the counterpoint to generate

public:
    /**
     * Constructor of the class.
     * @param cf a vector<int> representing the cantus firmus.
     * @param sp the species of the counterpoint. it takes values from the enum "species" in
    headers/Utilities.hpp
     * @param k the key of the score. it takes values from the notes in headers/Utilities.hpp
     * @param lb the lowest note possible for the counterpoint in MIDI
     * @param ub the highest note possible for the counterpoint in MIDI
     */
    TwoVoiceCounterpoint(vector<int> cf, Species sp, int v_type, vector<int> m_costs, vector<int
    > g_costs, vector<int> s_costs, vector<int> imp,
        int bm, ObjectiveMode objMode = OBJECTIVE_LEX);

    TwoVoiceCounterpoint(TwoVoiceCounterpoint& s);
    IntLexMinimizeSpace* copy() override;

    string to_string() const override;



    ///destructor : mother class destructor is sufficient, nothing to add
```

```
42
43 };
44
45 #endif //MYPROJECT_TWOVOICECOUNTERPOINT_HPP
```

A.1.4 ThreeVoiceCounterpoint.hpp

```
1 //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 //
4
5 #ifndef MYPROJECT_THREEVOICECOUNTERPOINT_HPP
6 #define MYPROJECT_THREEVOICECOUNTERPOINT_HPP
7
8 #include "../Utilities.hpp"
9 #include "CounterpointProblem.hpp"
10 #include "../Parts/FirstSpeciesCounterpoint.hpp"
11 #include "../Parts/SecondSpeciesCounterpoint.hpp"
12 #include "../Parts/ThirdSpeciesCounterpoint.hpp"
13 #include "../Parts/CantusFirmus.hpp"
14
15 /**
16  * This class models a counterpoint problem with 3 voices.
17  */
18 class ThreeVoiceCounterpoint : public CounterpointProblem{
19 protected:
20
21     vector<Species> species;        /// the species of the counterpoints to generate
22
23 public:
24     /**
25      * Constructor of the class.
26      * @param cf a vector<int> representing the cantus firmus.
27      * @param sp the species of the counterpoint. it takes values from the enum "species" in
    headers/Utilities.hpp
28      * @param k the key of the score. it takes values from the notes in headers/Utilities.hpp
29      * @param lb the lowest note possible for the counterpoint in MIDI
30      * @param ub the highest note possible for the counterpoint in MIDI
31      */
32     ThreeVoiceCounterpoint(vector<int> cf, vector<Species> sp, vector<int> v_type, vector<int>
    m_costs, vector<int> g_costs,
33         vector<int> s_costs, vector<int> imp, int bm, ObjectiveMode objMode = OBJECTIVE_LEX);
34
35     ThreeVoiceCounterpoint(ThreeVoiceCounterpoint& s);
36     IntLexMinimizeSpace* copy() override;
37
38     string to_string() const override;
39
40     /// Getters
41     // Part* getCounterpoint(){ return counterpoint; }
42
43     ///destructor : mother class destructor is sufficient, nothing to add
44
45     void uniteCounterpoints();
46
47     void uniteCosts();
48 };
49
50 #endif //MYPROJECT_THREEVOICECOUNTERPOINT_HPP
```

A.1.5 FourVoiceCounterpoint.hpp

```
1 //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 //
4
5 #ifndef MYPROJECT_FOURVOICECOUNTERPOINT_HPP
6 #define MYPROJECT_FOURVOICECOUNTERPOINT_HPP
7
8 #include "../Utilities.hpp"
9 #include "CounterpointProblem.hpp"
10 #include "../Parts/FirstSpeciesCounterpoint.hpp"
11 #include "../Parts/SecondSpeciesCounterpoint.hpp"
12 #include "../Parts/ThirdSpeciesCounterpoint.hpp"
13 #include "../Parts/CantusFirmus.hpp"
14
15 /**
16  * This class models a counterpoint problem with 4 voices.
17  */
18 class FourVoiceCounterpoint : public CounterpointProblem{
19 protected:
20
21     vector<Species> species;        /// the species of the counterpoints to generate
22
```

```
public:
    /**
     * Constructor of the class.
     * @param cf a vector<int> representing the cantus firmus.
     * @param sp the species of the counterpoint. it takes values from the enum "species" in headers/Utilities.hpp
     * @param k the key of the score. it takes values from the notes in headers/Utilities.hpp
     * @param lb the lowest note possible for the counterpoint in MIDI
     * @param ub the highest note possible for the counterpoint in MIDI
     */
    FourVoiceCounterpoint(vector<int> cf, vector<Species> sp, vector<int> v_type, vector<int> m_costs, vector<int> g_costs,
        vector<int> s_costs, vector<int> imp, int bm, ObjectiveMode objMode = OBJECTIVE_LEX);

    FourVoiceCounterpoint(FourVoiceCounterpoint& s);
    IntLexMinimizeSpace* copy() override;

    string to_string() const override;

    /// Getters
    // Part* getCounterpoint(){ return counterpoint; }

    /// destructor : mother class destructor is sufficient, nothing to add

    void uniteCounterpoints();

    void uniteCosts();
};

#endif
```

A.2 Voice and part headers

A.2.1 Voice.hpp

```
//
// Created by Luc Cleenewerk and Diego de Patoul.
//

#ifndef FUXCP_BASE_VOICE_HPP
#define FUXCP_BASE_VOICE_HPP

#include "../Utilities.hpp"

#include "gecode/kernel.hh"
#include "gecode/int.hh"
#include "gecode/search.hh"
#include "gecode/minimodel.hh"
#include "gecode/set.hh"

using namespace Gecode;
using namespace Gecode::Search;
using namespace std;

/// abstract class!!! Should not be instanciated
/// This class represents a Voice, so it creates all the variables associated to that part and posts the constraints that are species independent
class Voice{
    protected:
        int nMeasures;
        int size;

        int lowerBound;
        int upperBound;

        IntVarArray notes;
        IntVarArray h_intervals;  // with respect to lowest stratum. TODO set to 0 if isLowest
        IntVarArray m_intervals_brut;
        IntVarArray motions;


    public:
        Voice(Home home, int nMes, int lb, int ub);

        Voice(Home home, Voice& s);  // clone constructor

        int getNMeasures() { return nMeasures; }
        int getSize() { return size; }

        int getLowerBound() { return lowerBound; }
        int getUpperBound() { return upperBound; }

        IntVarArray getNotes() { return notes; }
        IntVarArray getMelodicIntervals();
```

```
49
50          /// must be implemented in the child classes, returns the variables to branch on
51          // virtual IntVarArray getBranchingNotes();
52
53          virtual Voice* clone(Home home);
54
55          virtual string to_string() const;
56
57          virtual IntVarArgs getFirstNotes();
58
59          IntVarArray getHIntervals();
60
61          IntVarArgs getSecondHInterval();
62
63          IntVarArray getMotions();
64 };
65
66
67 #endif //FUXCP_BASE_VOICE_HPP
```

A.2.2 Stratum.hpp

```
1 //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 //
4
5 #ifndef STRATUM_HPP
6 #define STRATUM_HPP
7
8 #include "../Utilities.hpp"
9 #include "Voice.hpp"
10
11 #include "gecode/kernel.hh"
12 #include "gecode/int.hh"
13 #include "gecode/search.hh"
14 #include "gecode/minimodel.hh"
15 #include "gecode/set.hh"
16
17 using namespace Gecode;
18 using namespace Gecode::Search;
19 using namespace std;
20
21 /// This class represents a Stratum
22 class Stratum : public Voice{
23     protected:
24         // intvararray with species of each measure?
25
26     public:
27         Stratum(Home home, int nMes, int lb, int ub);
28
29         Stratum(Home home, int nMes, int lb, int ub, IntVarArray lowestNotes);
30
31         Stratum(Home home, int nMes, int lb, int ub, IntVarArray lowestNotes, int nV);
32
33         Stratum(Home home, int nMes, int lb, int ub, IntVarArray lowestNotes, int nV1, int nV2);
34
35         Stratum(Home home, Stratum& s);  // clone constructor
36
37         Stratum* clone(Home home) override;
38
39         string to_string() const override;
40
41         void setNote(Home home, int index, IntVar note);
42
43 };
44
45
46 #endif //STRATUM_HPP
```

A.2.3 Part.hpp

```
1 //
2 // Created by Damien Sprockeels on 11/06/2024.
3 // Extended and developed by Luc Cleenewerk and Diego de Patoul up to August 2024.
4 //
5
6 #ifndef FUXCP_BASE_COUNTERPOINT_HPP
7 #define FUXCP_BASE_COUNTERPOINT_HPP
8
9 #include "../Utilities.hpp"
10 #include "Voice.hpp"
11 #include "Stratum.hpp"
12 #include "constraints.hpp"
13
```

```
14 #include "gecode/kernel.hh"
15 #include "gecode/int.hh"
16 #include "gecode/search.hh"
17 #include "gecode/minimodel.hh"
18 #include "gecode/set.hh"
19
20 using namespace Gecode;
21 using namespace Gecode::Search;
22 using namespace std;
23
24 /// abstract class!!! Should not be instanciated
25 /// This class represents a part, so it creates all the variables associated to that part and
       posts the constraints that are species independent
26 class Part : public Voice {
27     protected:
28         Species species;
29         int nVoices;
30         int borrowMode;
31         int voice_type;
32         //Stratum* lowest;
33         IntVarArray melodicDegreeCost;
34         IntVarArray fifthCostArray;
35         IntVarArray octaveCostArray;
36         BoolVarArray is_off;
37         IntVarArray offCostArray;
38         IntVarArray costs;
39         IntVarArray varietyCostArray;
40         IntVarArray directCostArray;
41         BoolVarArray isConsonance;
42         IntVarArray speciesArray;
43         BoolVarArray isNotLowest;
44         BoolVarArray isHighest;
45
46         vector<int> borrowed_scale;
47         vector<int> scale;
48         vector<int> chromatic_scale;
49
50         vector<int> cp_range;
51
52         vector<int> domain;
53         vector<int> off_domain;
54
55         vector<string> cost_names;
56
57         int secondCost;
58         int thirdCost;
59         int fourthCost;
60         int tritoneCost;
61         int fifthCost;
62         int sixthCost;
63         int seventhCost;
64         int octaveCost;
65
66         int borrowCost;
67         int h_fifthCost;
68         int h_octaveCost;
69         int succCost;
70         int varietyCost;
71         int triadCost;
72         int directMoveCost;
73         int penultCost;
74
75         int penultSixthCost;
76         int cambiataCost;
77         int mSkipCost;
78         int triad3rdCost;
79         int m2ZeroCost;
80         int syncopationCost;
81         int prefSlider;
82
83         IntVarArray relaxationCostArray;
84
85         int directCost;
86         int obliqueCost;
87         int contraryCost;
88
89         IntVarArray toCombineCosts;
90         vector<string> toCombineCostNames;
91
92         //First Species specific variables
93         IntVarArray firstSpeciesNotesCp;
94         IntVarArray firstSpeciesHarmonicIntervals;
95         IntVarArray firstSpeciesMelodicIntervals;
96         IntVarArray firstSpeciesMotions;
97         IntVarArray firstSpeciesMotionCosts;
98
99         //Second species specific variables
100        BoolVarArray isDiminution;
101        IntVarArray penultCostArray;
102        IntVarArray secondSpeciesMotions;
```

```
103         IntVarArray secondSpeciesMelodicIntervals;
104         IntVarArray secondSpeciesRealMotions;
105 
106         //Third species specific variables
107         IntVarArray thirdSpeciesHarmonicIntervals;
108         IntVarArray thirdSpeciesMelodicIntervals;
109         BoolVarArray is5QNArray;
110         IntVarArray cambiataCostArray;
111 
112         //Fourth species specific variables
113         BoolVarArray isNoSyncopeArray;
114         IntVarArray snycopeCostArray;
115         IntVarArray fourthSpeciesMelodicIntervals;
116 
117     public:
118         Part(Home home, int nMes, Species sp, vector<int> cf, int lb, int ub, int v_type, vector
    <int> m_costs, vector<int> g_costs,
119             vector<int> s_costs, int nV, int bm);
120 
121         // Part(Part& s); (no longer copy constructor since not a space anymore. Now just a
    clone constructor to deep copy the object (called by the Space's copy constructor))
122         Part(Home home, Part& s);  // clone constructor
123 
124         /// must be implemented in the child classes, returns the variables to branch on
125         // virtual IntVarArray getBranchingNotes();
126 
127         virtual Part* clone(Home home) override;
128 
129         virtual string to_string() const override;
130 
131         // Returns a "0/1" dump of the per-cell 3rd-species classification, or "" if the
132         // species has no such notion. Only 5th species overrides this (see header). Used
133         // by Generations to record which flat-MIDI cells are genuine 3rd-species attacks.
134         virtual string getSolutionRhythm() const { return ""; }
135 
136         virtual IntVarArray getBranchingNotes();
137 
138         virtual IntVarArray getFirstHInterval();
139 
140         virtual IntVarArray getFirstMInterval();
141 
142         virtual IntVarArray getMotions();
143 
144         int getSpecies() { return species; }
145 
146         IntVarArray getPartNotes();
147 
148         IntVarArray getCosts();
149 
150         BoolVarArray getIsHighest();
151 
152         int getSuccCost();
153 
154         int getTriadCost();
155 
156         int getVarietyCost();
157 
158         IntVar getVarietyArray(int idx);
159 
160         virtual int getHIntervalSize();
161 
162         BoolVarArray getConsonance();
163 
164         BoolVarArray getIsNotLowest();
165 
166         int getSecondCost();
167         int getThirdCost();
168         int getFourthCost();
169         int getTritoneCost();
170         int getFifthCost();
171         int getSixthCost();
172         int getSeventhCost();
173         int getOctaveCost();
174         int getHFifthCost();
175         int getHOctaveCost();
176         int getDirectCost();
177         int getBorrowCost();
178         int getPenultCost();
179         int getDirectMoveCost();
180         int getCambiataCost();
181 
182         IntVarArray getMelodicDegreeCost();
183 
184         IntVarArray getFirstSpeciesHIntervals();
185 
186         IntVarArray getFirstSpeciesNotes();
187 
188         IntVarArray getFifthCostArray();
189 
190         IntVarArray getOctaveCostArray();
```

```
191
192         IntVarArray getFirstSpeciesMIntervals();
193
194         IntVarArray getFirstSpeciesMotions();
195
196         IntVarArray getDirectCostArray();
197
198         BoolVarArray getIsOffArray();
199
200         vector<int> getOffDomain();
201
202         vector<int> getDomain();
203
204         /// Diatonic scale of the key (all MIDI notes of the detected scale). Used by
205         /// H2_4_chordMembershipOnDisjunctWeakBeats to build the implied diatonic triads.
206         vector<int> getScale() { return scale; }
207
208         IntVarArray getOffCostArray();
209
210         BoolVarArray getIsDiminution();
211
212         IntVarArray getPenultCostArray();
213
214         IntVarArray getSecondSpeciesMotions();
215
216         IntVarArray getSecondSpeciesMIntervals();
217
218         IntVarArray getSecondSpeciesRealMotions();
219
220         BoolVarArray getIs5QNArray();
221
222         IntVarArray getThirdSpeciesHIntervals();
223
224         IntVarArray getThirdSpeciesMIntervals();
225
226         IntVarArray getCambiataCostArray();
227
228         BoolVarArray getNoSyncope();
229
230         IntVarArray getSyncopeCostArray();
231
232         IntVarArray getFourthSpeciesMIntervals();
233
234         IntVarArray getSpeciesArray();
235
236         void add_cost(Home home, int idx, IntVarArray to_be_added, IntVarArray costs);
237
238         vector<string> getCostNames();
239
240         IntVarArray getToCombineCosts();
241
242         vector<string> getToCombineCostNames();
243
244         void add_toCombineCost(Home home, int idx, IntVarArray to_be_added, IntVarArray costs);
245
246         IntVarArray getRelaxationCostArray();
247         void initRelaxationCostArray(Home home, int size);
248 };
249
250
251 #endif //FUXCP_BASE_COUNTERPOINT_HPP
```

A.2.4 CantusFirmus.hpp

```
1 //
2 // Created by Damien Sprockeels on 12/06/2024.
3 // Extended and developed by Luc Cleenewerk and Diego de Patoul up to August 2024.
4 //
5
6 #ifndef MYPROJECT_CANTUSFIRMUS_HPP
7 #define MYPROJECT_CANTUSFIRMUS_HPP
8
9 #include "../Utilities.hpp"
10 #include "Part.hpp"
11 class CantusFirmus : public Part {
12 private:
13     vector<int> cf_vector;
14     IntVarArray disArray; // Array of dissonances
15
16 public:
17     CantusFirmus(Home home, int size, vector<int> cf, Stratum* low, int v_type, vector<int>
      m_costs, vector<int> g_costs, vector<int> s_costs,
18         int nV);
19
20     string to_string() const override;
21
22     CantusFirmus(Home home, CantusFirmus& s); // clone constructor
```

```
23      CantusFirmus* clone(Home home) override;
24
25      IntVarArray getFirstHInterval() override;
26
27      IntVarArray getFirstMInterval() override;
28
29      IntVarArray getMotions() override;
30
31      IntVarArgs getFirstNotes() override;
32
33      int getHIntervalSize() override;
34
35  };
36
37
38  #endif //MYPROJECT_CANTUSFIRMUS_HPP
```

A.3 Species headers

A.3.1 FirstSpeciesCounterpoint.hpp

```
1   //
2   // Created by Damien Sprockeels on 11/06/2024.
3   // Extended and developed by Luc Cleenewerk and Diego de Patoul up to August 2024.
4   //
5
6   #ifndef FUXCP_BASE_FIRSTSPECIESCOUNTERPOINT_HPP
7   #define FUXCP_BASE_FIRSTSPECIESCOUNTERPOINT_HPP
8
9   #include "Part.hpp"
10  #include "CantusFirmus.hpp"
11
12  /**
13   * This class represents a counterpoint of the first species. It inherits from the Part class.
14   * todo modify it so it also works for 3 and 4 voices. Add the appropriate constraints by making
       a constructor that takes the number of voices as a parameter
15   * todo maybe it should take a Stratum (object or just IntVarArray) for the lowest voice or
       something like that depending on the formalization
16   */
17  class FirstSpeciesCounterpoint : public Part
18  {
19  protected:
20      Species motherSpecies;                          /// The species from which this is called.
21      CantusFirmus* cantus;
22      IntVarArray disArray; // Array of dissonances
23
24  public:
25      /**
26       * General constructor. It takes the mother species as an argument and calls the super
         constructor from the part class.
27       * Additionally, it sets all the 1st species specific variables as well as the general rules
         that have to be applied
28       * regardless of the species. It does not apply 1st species specific rules and does not post
         branching.
29       * todo maybe add a Stratum object or IntVarArray for the lowest voice
30       * @param nMes the number of measures in the composition
31       * @param cf the cantus firmus todo maybe it should be a CantusFirmusObject
32       * @param lb the lower bound for the counterpoint
33       * @param ub the upper bound for the counterpoint
34       * @param k the key of the composition
35       * @param mSpecies the species from which this is called.
36       * @param low the lowest stratum
37       * @param c the cantus firmus
38       * @param v_type the voice type of the counterpoint
39       * @param m_costs the user-defined melodic costs
40       * @param g_costs the user-defined general costs
41       * @param s_costs the user-defined specific costs
42       * @param bm parameter specifying if borrow Mode is enabled or not
43       * @param nV the number of voices - as it is a 2 voice constructor, this parameter contains
         the number of voices
44       */
45      FirstSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Species
         mSpecies, Stratum* low, CantusFirmus* c,  int v_type
46      , vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV);
47
48      /**
49       * 2 VOICES CONSTRUCTOR. This constructor is only used when creating a counterpoint of the
         first species. It calls the other constructor with
50       * FIRST_SPECIES as the mother species. Additionally, it posts 1st species specific
         constraints.
51       * @param nMes the number of measures in the composition
52       * @param cf the cantus firmus todo maybe it should be a CantusFirmusObject
53       * @param lb the lower bound for the counterpoint
54       * @param ub the upper bound for the counterpoint
```

```
55      * @param k the key of the composition
56      * @param mSpecies the species from which this is called.
57      * @param low the lowest stratum
58      * @param c the cantus firmus
59      * @param v_type the voice type of the counterpoint
60      * @param m_costs the user-defined melodic costs
61      * @param g_costs the user-defined general costs
62      * @param s_costs the user-defined specific costs
63      * @param bm parameter specifying if borrow Mode is enabled or not
64      * @param nV the number of voices - as it is a 2 voice constructor, this parameter is passed
        to the general constructor
65      */
66     FirstSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum* low,
        CantusFirmus* c,  int v_type, vector<int> m_costs
67     , vector<int> g_costs, vector<int> s_costs, int bm, int nV);
68
69     /**
70      * 3 VOICES CONSTRUCTOR. This constructor is only used when creating a counterpoint of the
        first species. It calls the other constructor with
71      * FIRST_SPECIES as the mother species. Additionally, it posts 1st species specific
        constraints.
72      * @param nMes the number of measures in the composition
73      * @param cf the cantus firmus todo maybe it should be a CantusFirmusObject
74      * @param lb the lower bound for the counterpoint
75      * @param ub the upper bound for the counterpoint
76      * @param k the key of the composition
77      * @param mSpecies the species from which this is called.
78      * @param low the lowest stratum
79      * @param c the cantus firmus
80      * @param v_type the voice type of the counterpoint
81      * @param m_costs the user-defined melodic costs
82      * @param g_costs the user-defined general costs
83      * @param s_costs the user-defined specific costs
84      * @param bm parameter specifying if borrow Mode is enabled or not
85      * @param nV1 the number of voices - this parameter is there to differenciate it from other
        number of voices
86      * @param nV2 the number of voices - as it is a 3 voice constructor, this parameter is
        passed to the general constructor
87      */
88     FirstSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum* low,
        CantusFirmus* c,  int v_type, vector<int> m_costs
89     , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2);
90
91     /**
92      * 4 VOICES CONSTRUCTOR. This constructor is only used when creating a counterpoint of the
        first species. It calls the other constructor with
93      * FIRST_SPECIES as the mother species. Additionally, it posts 1st species specific
        constraints.
94      * @param nMes the number of measures in the composition
95      * @param cf the cantus firmus todo maybe it should be a CantusFirmusObject
96      * @param lb the lower bound for the counterpoint
97      * @param ub the upper bound for the counterpoint
98      * @param k the key of the composition
99      * @param mSpecies the species from which this is called.
100     * @param low the lowest stratum
101     * @param c the cantus firmus
102     * @param v_type the voice type of the counterpoint
103     * @param m_costs the user-defined melodic costs
104     * @param g_costs the user-defined general costs
105     * @param s_costs the user-defined specific costs
106     * @param bm parameter specifying if borrow Mode is enabled or not
107     * @param nV1 the number of voices - this parameter is there to differenciate it from other
        number of voices
108     * @param nV2 the number of voices - this parameter is there to differenciate it from other
        number of voices
109     * @param nV3 the number of voices - as it is a 4 voice constructor, this parameter is
        passed to the general constructor
110     */
111    FirstSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum* low,
        CantusFirmus* c,  int v_type, vector<int> m_costs
112    , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2, int nV3);
113
114    /**
115     * This function returns a string with the characteristics of the counterpoint. It calls the
        to_string() method from
116     * the Part class and adds 1st species specific characteristics.
117     * @return a string representation of the current instance of the FirstSpeciesCounterpoint
        class.
118     */
119    string to_string() const override;
120
121    /// Copy constructor. This needs to copy all useful attributes and update variables. Must
        call the super copy constructor       NO LONGER NEEDED
122    // FirstSpeciesCounterpoint(FirstSpeciesCounterpoint &s);
123    /// Copy function
124    // virtual Space *copy() override;
125
126    FirstSpeciesCounterpoint(Home home, FirstSpeciesCounterpoint& s); // clone constructor
127    FirstSpeciesCounterpoint* clone(Home home) override;
128
```

```
    virtual IntVarArray getBranchingNotes() override;

    IntVarArray getFirstHInterval() override;

    IntVarArray getMotions() override;

    IntVarArray getFirstMInterval() override;

    int getHIntervalSize() override;

};

#endif //FUXCP_BASE_FIRSTSPECIESCOUNTERPOINT_HPP
```

A.3.2 SecondSpeciesCounterpoint.hpp

```
//
// Created by Damien Sprockeels on 12/06/2024.
// Extended and developed by Luc Cleenewerk and Diego de Patoul up to August 2024.
//

#ifndef MYPROJECT_SECONDSPECIESCOUNTERPOINT_HPP
#define MYPROJECT_SECONDSPECIESCOUNTERPOINT_HPP


#include "FirstSpeciesCounterpoint.hpp"
#include "../Utilities.hpp"

///This class extends the first species class because most rules of the first species are also valid for the second species
/**
 * This class represents a counterpoint of the second species. It inherits from the FirstSpeciesCounterpoint class.
 */
class SecondSpeciesCounterpoint : public FirstSpeciesCounterpoint {
protected:
    IntVarArray secondSpeciesNotesCp;               /// The notes of the counterpoint that have to follow the rules for the 2nd species
    IntVarArray secondSpeciesHarmonicIntervals;     /// The harmonic intervals between the notes that have to follow the 2nd species rules and the cantus firmus
    IntVarArray secondSpeciesMotionCosts;
    IntVarArray secondSpeciesRealMotionCosts;
    IntVarArray secondSpeciesArsisArray;

public:
    /**
     *
     * @param size
     * @param cf
     * @param lb
     * @param ub
     * @param k
     */

    SecondSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, int mSpec, Stratum* low, CantusFirmus* c, int v_type, vector<int> m_costs
    , vector<int> g_costs, vector<int> s_costs, int bm, int nV);

    SecondSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, Stratum* low, CantusFirmus* c, int v_type, vector<int> m_costs
    , vector<int> g_costs, vector<int> s_costs, int bm, int nV);

    SecondSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, Stratum* low, CantusFirmus* c, int v_type, vector<int> m_costs
    , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2);

    SecondSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, Stratum* low, CantusFirmus* c, int v_type, vector<int> m_costs
    , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2, int nV3);

    string to_string() const override;

    // SecondSpeciesCounterpoint(SecondSpeciesCounterpoint &s);    No longer a copy, now a clone (since it is not a space, and copy constructor are for spaces, while cloning functions are just used to create a deep copy of an object)

    // virtual Space *copy() override;

    SecondSpeciesCounterpoint(Home home, SecondSpeciesCounterpoint &s); // clone constructor
    SecondSpeciesCounterpoint* clone(Home home) override;

    IntVarArray getFirstHInterval() override;

    IntVarArray getMotions() override;

    IntVarArray getFirstMInterval() override;

```

```
62      IntVarArray getBranchingNotes() override;
63
64      int getHIntervalSize() override;
65
66  };
67
68
69  #endif //MYPROJECT_SECONDSPECIESCOUNTERPOINT_HPP
```

A.3.3 ThirdSpeciesCounterpoint.hpp

```
1   //
2   // Created by Luc Cleenewerk and Diego de Patoul.
3   //
4
5   #ifndef MYPROJECT_THIRDSPECIESCOUNTERPOINT_HPP
6   #define MYPROJECT_THIRDSPECIESCOUNTERPOINT_HPP
7
8
9   #include "FirstSpeciesCounterpoint.hpp"
10  #include "../Utilities.hpp"
11
12  class ThirdSpeciesCounterpoint : public FirstSpeciesCounterpoint {
13  protected :
14      IntVarArray thirdSpeciesNotesCp;                /// The notes of the counterpoint that have
        to follow the rules for the 2nd species
15      IntVarArray thirdSpeciesMotions;                /// This array is the array of REAL motions
16      IntVarArray thirdSpeciesMotionCosts;
17      IntVarArray m2IntervalsArray;
18      IntVarArray m2ZeroArray;
19      IntVarArray thirdHTriadArray;
20      IntVarArray thirdSpeciesAbsMelodic;
21
22  public:
23
24      ThirdSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, int mSpec,
        Stratum* low, CantusFirmus* c, int v_type, vector<int> m_costs
25      , vector<int> g_costs, vector<int> s_costs, int bm, int nV);
26
27      ThirdSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, Stratum* low,
        CantusFirmus* c, int v_type, vector<int> m_costs
28      , vector<int> g_costs, vector<int> s_costs, int bm, int nV);
29
30      ThirdSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, Stratum* low,
        CantusFirmus* c, int v_type, vector<int> m_costs
31      , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2);
32
33      ThirdSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, Stratum* low,
        CantusFirmus* c, int v_type, vector<int> m_costs
34      , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2, int nV3);
35
36      string to_string() const override;
37
38      ThirdSpeciesCounterpoint(Home home, ThirdSpeciesCounterpoint &s); // clone constructor
39
40      ThirdSpeciesCounterpoint* clone(Home home) override;
41
42      IntVarArray getFirstHInterval() override;
43
44      IntVarArray getMotions() override;
45
46      IntVarArray getFirstMInterval() override;
47
48      IntVarArray getBranchingNotes() override;
49
50      int getHIntervalSize() override;
51  };
52
53  #endif
```

A.3.4 FourthSpeciesCounterpoint.hpp

```
1   //
2   // Created by Luc Cleenewerk and Diego de Patoul.
3   //
4
5   #ifndef FUXCP_BASE_FOURTHSPECIESCOUNTERPOINT_HPP
6   #define FUXCP_BASE_FOURTHSPECIESCOUNTERPOINT_HPP
7
8   #include "Part.hpp"
9   #include "CantusFirmus.hpp"
10
11  /**
12   * This class represents a counterpoint of the first species. It inherits from the Part class.
```

```
13  * todo modify it so it also works for 3 and 4 voices. Add the appropriate constraints by making
        a constructor that takes the number of voices as a parameter
14  * todo maybe it should take a Stratum (object or just IntVarArray) for the lowest voice or
       something like that depending on the formalization
15  */
16 class FourthSpeciesCounterpoint : public Part{
17 protected:
18
19     CantusFirmus* cantus;
20     IntVarArray fourthSpeciesNotesCp;
21     IntVarArray fourthSpeciesHIntervals;
22     IntVarArray m2IntervalsArray;
23     IntVarArray firstHInterval;
24     IntVarArray m2ZeroArray;
25     IntVarArray sol;
26
27 public:
28     /**
29      * General constructor. It takes the mother species as an argument and calls the super
       constructor from the part class.
30      * Additionally, it sets all the 1st species specific variables as well as the general rules
        that have to be applied
31      * regardless of the species. It does not apply 1st species specific rules and does not post
        branching.
32      * todo maybe add a Stratum object or IntVarArray for the lowest voice
33      * @param nMes the number of measures in the composition
34      * @param cf the cantus firmus todo maybe it should be a CantusFirmusObject
35      * @param lb the lower bound for the counterpoint
36      * @param ub the upper bound for the counterpoint
37      * @param k the key of the composition
38      * @param mSpecies the species from which this is called.
39      * @param low the lowest stratum
40      * @param c the cantus firmus
41      * @param v_type the voice type of the counterpoint
42      * @param m_costs the user-defined melodic costs
43      * @param g_costs the user-defined general costs
44      * @param s_costs the user-defined specific costs
45      * @param bm parameter specifying if borrow Mode is enabled or not
46      * @param nV the number of voices - as it is a 2 voice constructor, this parameter contains
       the number of voices
47      */
48     FourthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Species
       mSpecies, Stratum* low, CantusFirmus* c,  int v_type
49     , vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV);
50
51     FourthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum* low,
        CantusFirmus* c,  int v_type, vector<int> m_costs
52     , vector<int> g_costs, vector<int> s_costs, int bm, int nV);
53
54     FourthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum* low,
        CantusFirmus* c,  int v_type, vector<int> m_costs
55     , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2);
56
57     FourthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum* low,
        CantusFirmus* c,  int v_type, vector<int> m_costs
58     , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2, int nV3);
59
60     /**
61      * This function returns a string with the characteristics of the counterpoint. It calls the
        to_string() method from
62      * the Part class and adds 1st species specific characteristics.
63      * @return a string representation of the current instance of the FirstSpeciesCounterpoint
       class.
64      */
65     string to_string() const override;
66
67     /// Copy constructor. This needs to copy all useful attributes and update variables. Must
       call the super copy constructor       NO LONGER NEEDED
68     // FirstSpeciesCounterpoint(FirstSpeciesCounterpoint &s);
69     /// Copy function
70     // virtual Space *copy() override;
71
72     FourthSpeciesCounterpoint(Home home, FourthSpeciesCounterpoint& s); // clone constructor
73     FourthSpeciesCounterpoint* clone(Home home) override;
74
75     virtual IntVarArray getBranchingNotes() override;
76
77     IntVarArray getFirstHInterval() override;
78
79     IntVarArray getMotions() override;
80
81     IntVarArray getFirstMInterval() override;
82
83     int getHIntervalSize() override;
84
85 };
86
87 #endif //FUXCP_BASE_FIRSTSPECIESCOUNTERPOINT_HPP
```

A.3.5 FifthSpeciesCounterpoint.hpp

```
//
// Created by Luc Cleenewerk and Diego de Patoul.
//

#ifndef FUXCP_BASE_FIFTHSPECIESCOUNTERPOINT_HPP
#define FUXCP_BASE_FIFTHSPECIESCOUNTERPOINT_HPP

#include "Part.hpp"
#include "CantusFirmus.hpp"

class FifthSpeciesCounterpoint : public Part{

protected:

    int solutionLength;
    int m2Len;
    CantusFirmus* cantus;
    IntVarArray fifthSpeciesNotesCp;
    IntVarArray fifthSpeciesHIntervals;
    IntVarArray firstHInterval;
    IntVarArray fifthSpeciesSuccMIntervals;
    IntVarArray fifthSpeciesMIntervals;
    IntVarArray fifthSpeciesMTAIntervals;
    IntVarArray fifthSpeciesM2Intervals;
    IntVarArray fifthSpeciesMAllIntervals;
    IntVarArray fifthSpeciesMotions;
    IntVarArray fifthSpeciesMotionsCosts;
    BoolVarArray isNthSpeciesArray;
    BoolVarArray isConstrainedArray;
    BoolVarArray isMostlyThirdArray;
    BoolVarArray isThirdSpeciesArray;
    BoolVarArray isFourthSpeciesArray;
    BoolVarArray isNotCambiata;
    IntVarArray m2ZeroCostArray;
    IntVarArray thirdHTriadArray;

public:

    /**
     * General constructor. It takes the mother species as an argument and calls the super constructor from the part class.
     * Additionally, it sets all the 1st species specific variables as well as the general rules that have to be applied
     * regardless of the species. It does not apply 1st species specific rules and does not post branching.
     * todo maybe add a Stratum object or IntVarArray for the lowest voice
     * @param nMes the number of measures in the composition
     * @param cf the cantus firmus todo maybe it should be a CantusFirmusObject
     * @param lb the lower bound for the counterpoint
     * @param ub the upper bound for the counterpoint
     * @param k the key of the composition
     * @param mSpecies the species from which this is called.
     * @param low the lowest stratum
     * @param c the cantus firmus
     * @param v_type the voice type of the counterpoint
     * @param m_costs the user-defined melodic costs
     * @param g_costs the user-defined general costs
     * @param s_costs the user-defined specific costs
     * @param bm parameter specifying if borrow Mode is enabled or not
     * @param nV the number of voices - as it is a 2 voice constructor, this parameter contains the number of voices
     */
    FifthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Species mSpecies, Stratum* low, CantusFirmus* c,  int v_type
    , vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV);

    FifthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum* low, CantusFirmus* c,  int v_type, vector<int> m_costs
    , vector<int> g_costs, vector<int> s_costs, int bm, int nV);

    FifthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum* low, CantusFirmus* c,  int v_type, vector<int> m_costs
    , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2);

    FifthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum* low, CantusFirmus* c,  int v_type, vector<int> m_costs
    , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2, int nV3);

    /**
     * This function returns a string with the characteristics of the counterpoint. It calls the to_string() method from
     * the Part class and adds 1st species specific characteristics.
     * @return a string representation of the current instance of the FirstSpeciesCounterpoint class.
     */
    string to_string() const override;

```

```
78      // Dump of isThirdSpeciesArray as "0/1 0/1 ..." (one entry per flat-MIDI cell).
79      // Lets an external audit separate real 3rd-species attacks (governed by SP5_H6)
80      // from phantom continuation/syncope cells that are rendered as quarters anyway.
81      string getSolutionRhythm() const override;
82
83      /// Copy constructor. This needs to copy all useful attributes and update variables. Must
        call the super copy constructor       NO LONGER NEEDED
84      // FirstSpeciesCounterpoint(FirstSpeciesCounterpoint &s);
85      /// Copy function
86      // virtual Space *copy() override;
87
88      FifthSpeciesCounterpoint(Home home, FifthSpeciesCounterpoint& s); // clone constructor
89      FifthSpeciesCounterpoint* clone(Home home) override;
90
91      virtual IntVarArray getBranchingNotes() override;
92
93      IntVarArray getFirstHInterval() override;
94
95      IntVarArray getMotions() override;
96
97      IntVarArray getFirstMInterval() override;
98
99      int getHIntervalSize() override;
100
101     void createSpeciesArrays(Home home);
102
103 };
104
105 #endif
```

A.4 Constraints and utility headers

A.4.1 constraints.hpp

```
1  //
2  // Created by Luc Cleenewerk and Diego de Patoul.
3  // This file contains the declarations of the functions that post the constraints.
4  //
5
6  #ifndef CONSTRAINTS_HPP
7  #define CONSTRAINTS_HPP
8
9  class Part;
10
11 #include "Part.hpp"
12
13 /* ================================================
14  *              GENERAL CONSTRAINTS
15  * ================================================
16  */
17
18 /**
19  * G4 : if a note is borrowed, then a cost must be set
20  */
21 void G4_counterpointMustBeInTheSameKey(Home home, Part* part);
22
23 /**
24  * G6 : chromatic melodies are prohibited
25  */
26 void G6_noChromaticMelodies(Home home, Part* part, int mSpec);
27
28 /**
29  * G7 : prefer smaller melodic intervals (and set tritones to be last resort)
30  */
31 void G7_melodicIntervalsShouldBeSmall(Home home, Part* part, int mSpec);
32
33 /**
34  * G9 : the last chord must be in the scale of the composition
35  */
36 void G9_lastChordSameAsFundamental(Home home, Stratum* lowest, Part* cantusFirmus);
37
38 /* ================================================
39  *              HARMONIC CONSTRAINTS
40  * ================================================
41  */
42
43 /**
44  * 1.H1 : the harmonic intervals of the first note of each measure must be consonances
45  */
46 void H1_1_harmonicIntervalsAreConsonances(Home home, Part* part);
47
48 void H1_3_fiveConsecutiveNotesByJointDegree(Home home, Part* part);
49
50 /**
```

```
51  * 1.H2 : The first harmonic interval must be a perfect consonance in 2 voice compositions
52  */
53 void H2_1_startWithPerfectConsonance(Home home, Part* part);
54
55 void H2_2_arsisHarmoniesCannotBeDissonant(Home home, Part* part);
56
57 void H2_3_dissonanceImpliesDiminution(Home home, Part* part);
58
59 // [Claude Code]
60 /**
61  * Extension of H2_2 for compositions with 3 or more voices.
62  * The base H2_2 only checks the arsis against the lowest stratum, so dissonances
63  * against upper voices (e.g. the cantus firmus when it is not the lowest) are not
64  * caught. This variant checks the arsis pitch-class interval of a 2nd-species voice
65  * against every other voice and ties the dissonance escape to isDiminution[i].
66  */
67 void H2_2_arsisHarmoniesCannotBeDissonant_multiVoice(Home home, Part* secondSpPart, vector<Part
        *> otherVoices);
68
69 // [Claude Code]
70 /**
71  * Extension of H2_3 for 3+ voices, 3rd species. Checks all three weak beats (+1/+2/+3)
72  * of each measure against every other voice's strong beat. A weak beat may be dissonant
73  * only if it is a genuine passing tone (approached AND left by step).
74  */
75 void H2_3_dissonanceImpliesDiminution_multiVoice(Home home, Part* thirdSpPart, vector<Part*>
        otherVoices);
76
77 // [Claude Code]
78 /**
79  * MANDATORY tonal rule (Bitsch  59 ). A disjunct weak-beat note of a 2nd/3rd-species voice
80  * must belong to the harmony of its half-measure (pitch classes of the voices' structural
81  * notes). One harmony per measure normally; a 4th-species voice resolves on the 3rd beat
82  * and yields a second harmony for the 2nd half (beats +2/+3). Passing tones stay exempt;
83  * the penultimate measure (cadence) is exempt.
84  */
85 void H2_4_chordMembershipOnDisjunctWeakBeats(Home home, Part* sp, vector<Part*> allVoices);
86
87 /**
88  * 1.H3 : The last chord must be a perfect consonance. This applies to 2 voices, the 3 and 4
        voice version is applied in the stratum constructors
89  */
90 void H3_1_endWithPerfectConsonance(Home home, Part* part);
91
92 void H3_2_penultimateNoteDomain(Home home, Part* part);
93
94 void H3_3_cambiataCost(Home home, Part* part);
95
96 //note : H5 was divided into 2 different ocnstraints since the cantusFirmus has a smaller notes
        array than the other parts
97
98 /**
99  * 1.H5 : Voices cannot play the same notes (except for the first and last note) (cantusFirmus
        version)
100  */
101 void H5_1_cpAndCfDifferentNotes(Home home, Part* part, Part* cf);
102
103 /**
104  * 1.H5 : Voices cannot play the same notes (except for the first and last note) (inter-voices
        version)
105  */
106 void H5_1_differentNotes(Home home, vector<Part*> part);
107
108 /**
109  * 1.H6 : Imperfect consonances are preferred (sets costs for fifth and octave)
110  */
111 void H6_1_preferImperfectConsonances(Home home, Part* part);
112
113 /**
114  * 1.H7 : the penultimate note must be a major sixth or a minor third depending on if it is the
        lowest stratum or not (counterpoint version)
115  */
116 void H7_1_2v_penultimateSixthOrThird(Home home, Part* part);
117
118 /**
119  * 1.H8 : the harmonic triad should be used as much as possible (3 voices)
120  */
121 void H8_3v_preferHarmonicTriad(Home home, Part* part, IntVarArray triadCostArray, Stratum*
        upper1, Stratum* upper2);
122
123 /**
124  * 1.H8 : the harmonic triad should be used as much as possible (4 voices)
125  */
126 void H8_4v_preferHarmonicTriad(Home home, IntVarArray triadCostArray, Stratum* upper1, Stratum*
        upper2, Stratum* upper3);
127
128 /* ================================================
129  *              MELODIC CONSTRAINTS
130  * ================================================
131  */
```

```
132
133
134 /**
135  * 1.M1 : melodic intervals cannot exceed a minor sixth
136  */
137 void M1_1_2v_melodicIntervalsNotExceedMinorSixth(Home home, Part* part);
138
139 /**
140  * 1.M1 : melodic intervals cannot exceed a minor sixth but may include an octave
141  */
142 void M1_1_3v_melodicIntervalsNotExceedMinorSixth(Home home, Part* part);
143
144 /**
145  * 2.M1 : If the two voices are getting so close that there is no contrary motion possible
        without crossing each other, then the melodic interval of
146  * the counterpoint can be an octave leap.
147  */
148 void M1_2_octaveLeap(Home home, Part* part, Stratum* low);
149
150 void M2_2_2v_twoConsecutiveNotesAreNotTheSame(Home home, Part* part);
151
152 void M2_2_3v_twoConsecutiveNotesAreNotTheSame(Home home, vector<Part*> parts, bool
        containsThirdSpecies);
153
154 /**
155  * 1.M2 : The notes of each part should be as diverse as possible
156  */
157 void M2_1_varietyCost(Home home, vector<Part*> parts);
158
159 void P1_1_2v_noDirectMotionFromPerfectConsonance(Home home, Part* part);
160
161 void P1_1_4v_noDirectMotionFromPerfectConsonance(Home home, Part* part);
162
163 void P1_2_2v_noDirectMotionFromPerfectConsonance(Home home, Part* part);
164
165 void P1_2_3v_noDirectMotionFromPerfectConsonance(Home home, Part* part);
166
167 void P1_2_4v_noDirectMotionFromPerfectConsonance(Home home, Part* part);
168
169 void P3_0_noBattuta(Home home, Part* part);
170
171 void P3_1_noBattuta(Home home, Part* part);
172
173 void P3_2_noBattuta(Home home, Part* part);
174
175 void P4_successiveCost(Home home, vector<Part*> parts, IntVarArray successiveCostArray);
176
177 void P8_noSuccessiveSamePerfectInterval(Home home, std::vector<Part*> parts);
178
179 void P9_noSimultaneousRepetition(Home home, std::vector<Part*> parts);
180
181 void P6_3v_noMoveInSameDirection(Home home, vector<Part*> parts);
182
183 void P6_4v_noMoveInSameDirection(Home home, vector<Part*> parts);
184
185 void P7_noSuccessiveAscendingSixths(Home home, vector<Part*> parts);
186
187 void P1_1_3v_noDirectMotionFromPerfectConsonance(Home home, Part* part);
188
189 void H7_1_3v_penultimateSixthOrThird(Home home, Part* part);
190
191 /* ================================================
192  *                OTHER CONSTRAINTS
193  * ================================================
194  */
195
196 /**
197  * This function creates the isOff array to be able to know if a note is borrowed or not
198  */
199 void initializeIsOffArray(Home home, Part* part);
200
201 /**
202  * Two fifth species rhythms should be as diverse as possible
203  */
204 void R9_5_twoFifthSpeciesDiversity_3v(Home home, Part* cp1, Part* cp2);
205
206 /**
207  * For the upper strata, there shouldn't be a minor second interval between the thesis notes
208  */
209 void noMinorSecondBetweenUpper(Home home, vector<Stratum*> strata);
210
211 /* ================================================
212  *          SOFT (RELAXED) CONSTRAINT VERSIONS
213  * ================================================
214  */
215
216 /**
217  * Soft version of G9: last chord same as CF fundamental.
218  * Violations counted in problemRelaxCosts (IntVarArray with 0/1 entries).
219  */
```

```
220 void G9_lastChordSameAsFundamental_soft(Home home, Stratum* lowest, Part* cantusFirmus,
        IntVarArray relaxCosts, int startIdx);
221
222 /**
223  * Soft version of M2_2_3v: melodic intervals of 2sp should not have unisons.
224  * Violations counted in the Part's relaxationCostArray.
225  */
226 void M2_2_3v_twoConsecutiveNotesAreNotTheSame_soft(Home home, vector<Part*> parts, bool
        containsThirdSpecies);
227
228 /**
229  * Soft version of H2_3: dissonance implies diminution for 3rd species.
230  * Violations counted in the Part's relaxationCostArray.
231  */
232 void H2_3_dissonanceImpliesDiminution_soft(Home home, Part* part);
233
234 /**
235  * Soft version of H2_2: arsis harmonies dissonance for 2nd species.
236  * Violations counted in the Part's relaxationCostArray.
237  */
238 void H2_2_arsisHarmoniesCannotBeDissonant_soft(Home home, Part* part);
239
240 #endif
```

A.4.2 Utilities.hpp

```
1  //
2  // Created by Luc Cleenewerk and Diego de Patoul.
3  // This file contains the declarations of general utility functions.
4  //
5
6  #ifndef UTILITIES
7  #define UTILITIES
8
9  #include "gecode/kernel.hh"
10 #include "gecode/int.hh"
11 #include "gecode/search.hh"
12 #include "gecode/minimodel.hh"
13 #include "gecode/set.hh"
14
15 #include <iostream>
16 #include <vector>
17 #include <string>
18 #include <map>
19 #include <iostream>
20 #include <fstream>
21 #include <set>
22
23 using namespace std;
24 using namespace Gecode;
25
26 /*****************************************************
27  *                                                   *
28  *                  Useful constants                 *
29  *                                                   *
30  *****************************************************/
31
32 /** Files */
33 const string LOG_FILE = "log.txt";
34 const string STATISTICS_FILE = "statistics.txt";
35 const string STATISTICS_CSV = "statistics";
36
37 /** Types of search engines */
38 enum solver_types{
39     DFS_SOLVER, //0
40     BAB_SOLVER, //1
41     LDS_SOLVER  //2
42 };
43
44 // /** Branching strategies */
45 // enum variable_selection{
46 //     DEGREE_MAX,                     //0
47 //     DOM_SIZE_MIN,                   //1
48 //     RIGHT_TO_LEFT,                  //2
49 //     LEFT_TO_RIGHT_SOPRANO_TO_BASS,  //3
50 //     AFC_MAX,                        //4
51 // };
52
53 // /// go <-- soprano->bass: 4-3-2-1-8-7-6-5 etc
54 // auto right_to_left = [](const Space& home, const IntVar& x, int i) {
55 //     return i;
56 // };
57
58 // /// go --> soprano->bass
59 // auto left_to_right_soprano_to_bass = [](const Space& home, const IntVar& x, int i) {
60 //     return (i/4) * 4 + (4 - i%4);
61 // };
```

```
62
63 // const vector<IntVarBranch> variable_selection_heuristics = {INT_VAR_DEGREE_MAX(),
64 //                                                             INT_VAR_SIZE_MIN(),
65 //                                                             INT_VAR_MERIT_MAX(right_to_left),
66 //                                                             INT_VAR_MERIT_MIN(
    left_to_right_soprano_to_bass)};
67
68 const vector<string> variable_selection_heuristics_names = {"Degree max", "Domain size min", "
    Left to right",
69                                                              "Right to left", "AFC max"};
70
71 enum value_selection{
72     VAL_MIN,            //0
73     VAL_MAX,            //1
74     VAL_MED,            //2
75     VAL_RND,            //3
76 };
77
78 // /// value selection heuristic
79 // auto branchVal = [](const Space& home, IntVar x, int i) {
80 //     return x.min();
81 // };
82
83 // /// commit function (EQ and DIFF)
84 // auto branchCommit = [](Space& home, unsigned int a, IntVar x, int i, int n){
85 //     if (a == 0U){
86 //         rel(home, x, IRT_EQ, n);
87 //     } else {
88 //         rel(home, x, IRT_NQ, n);
89 //     }
90 // };
91
92 const vector<IntValBranch> value_selection_heuristics = {INT_VAL_MIN(), INT_VAL_MAX(),
    INT_VAL_MED(), INT_VAL_RND(1U)};
93
94 const vector<string> value_selection_heuristics_names = {"Value min", "Value max", "Median value
    ", "Value random"};
95
96 /** Voice ranges */
97 const int BASS_MIN = 40;
98 const int BASS_MAX = 60;
99 const int TENOR_MIN = 48;
100 const int TENOR_MAX = 69;
101 const int ALTO_MIN = 55;
102 const int ALTO_MAX = 75;
103 const int SOPRANO_MIN = 60;
104 const int SOPRANO_MAX = 84;
105
106 /** Melodic costs */
107 const int UNISON_COST = 0;
108 const int SECOND_COST = 1;
109 const int THIRD_COST = 3;
110 const int FOURTH_COST = 6;
111 const int FIFTH_COST = 6;
112 const int SIXTH_COST = 12;
113 const int SEVENTH_COST = 18;
114 const int OCTAVE_COST = 6;
115
116 const int MAX_MELODIC_COST = SEVENTH_COST;
117
118
119 /** Notes */
120 const int B_SHARP = 0;
121 const int C = 0;
122 const int C_SHARP = 1;
123 const int D_FLAT = 1;
124 const int D = 2;
125 const int D_SHARP = 3;
126 const int E_FLAT = 3;
127 const int E = 4;
128 const int F_FLAT = 4;
129 const int E_SHARP = 5;
130 const int F = 5;
131 const int F_SHARP = 6;
132 const int G_FLAT = 6;
133 const int G = 7;
134 const int G_SHARP = 8;
135 const int A_FLAT = 8;
136 const int A = 9;
137 const int A_SHARP = 10;
138 const int B_FLAT = 10;
139 const int B = 11;
140 const int C_FLAT = 11;
141
142 const vector<std::string> noteNames = {"C", "Csharp", "D", "Eb", "E", "F", "Fsharp", "G", "Ab",
    "A", "Bb", "B"};
143
144 /** Voice positions */
145 enum voices{
146     BASS,       //0
```

```
147     TENOR,      //1
148     ALTO,       //2
149     SOPRANO     //3
150 };
151
152 const vector<std::string> voiceNames = {"Bass", "Tenor", "Alto", "Soprano"};
153
154 /** scale degrees */
155 enum degrees{
156     FIRST_DEGREE,       //0
157     SECOND_DEGREE,      //1
158     THIRD_DEGREE,       //2
159     FOURTH_DEGREE,      //3
160     FIFTH_DEGREE,       //4
161     SIXTH_DEGREE,       //5
162     SEVENTH_DEGREE      //6
163 };
164
165 const vector<std::string> degreeNames = {"First degree", "Second degree", "Third degree", "
        Fourth degree", "Fifth degree",
166                                          "Sixth degree", "Seventh degree"};
167
168 /** Intervals */
169 // "classic" intervals
170 enum intervals{
171     UNISSON,            //0
172     MINOR_SECOND,       //1
173     MAJOR_SECOND,       //2
174     MINOR_THIRD,        //3
175     MAJOR_THIRD,        //4
176     PERFECT_FOURTH,     //5
177     TRITONE,            //6
178     PERFECT_FIFTH,      //7
179     MINOR_SIXTH,        //8
180     MAJOR_SIXTH,        //9
181     MINOR_SEVENTH,      //10
182     MAJOR_SEVENTH,      //11
183     PERFECT_OCTAVE      //12
184 };
185
186 // augmented/diminished intervals
187 const int AUGMENTED_SECOND = 3;
188 const int AUGMENTED_FOURTH = TRITONE;
189 const int DIMINISHED_FIFTH = TRITONE;
190
191 /** Chords */
192 enum chordTypes{
193     MAJOR_CHORD,                //0
194     MINOR_CHORD,                //1
195     DIMINISHED_CHORD,           //2
196     AUGMENTED_CHORD,            //3
197     DOMINANT_SEVENTH_CHORD,     //4
198     MAJOR_SEVENTH_CHORD,        //5
199     MINOR_SEVENTH_CHORD         //6
200 };
201
202 /// Types of chords represented by the intervals between their notes in root position up to an
        octave
203 const vector<int> MAJOR_CHORD_INTERVALS = {MAJOR_THIRD, MINOR_THIRD, PERFECT_FOURTH};
204 const vector<int> MINOR_CHORD_INTERVALS = {MINOR_THIRD, MAJOR_THIRD, PERFECT_FOURTH};
205 const vector<int> DIMINISHED_CHORD_INTERVALS = {MINOR_THIRD, MINOR_THIRD, TRITONE};
206 const vector<int> AUGMENTED_CHORD_INTERVALS = {MAJOR_THIRD, MAJOR_THIRD, MAJOR_THIRD};
207 const vector<int> DOMINANT_SEVENTH_CHORD_INTERVALS = {MAJOR_THIRD, MINOR_THIRD, MINOR_THIRD,
        MAJOR_SECOND};
208 const vector<int> MAJOR_SEVENTH_CHORD_INTERVALS = {MAJOR_THIRD, MINOR_THIRD, MAJOR_THIRD,
        MINOR_SECOND};
209 const vector<int> MINOR_SEVENTH_CHORD_INTERVALS = {MINOR_THIRD, MAJOR_THIRD, MINOR_THIRD,
        MAJOR_SECOND};
210
211 const vector<vector<int>> chordQualitiesIntervals = {MAJOR_CHORD_INTERVALS,
        MINOR_CHORD_INTERVALS,
212                                                      DIMINISHED_CHORD_INTERVALS,
        AUGMENTED_CHORD_INTERVALS,
213                                                      DOMINANT_SEVENTH_CHORD_INTERVALS,
        MAJOR_SEVENTH_CHORD_INTERVALS,
214                                                      MINOR_SEVENTH_CHORD_INTERVALS};
215
216 const vector<std::string> chordQualityNames = {"Major", "Minor", "Diminished", "Augmented", "
        Dominant seventh",
217                                                "Major seventh", "Minor seventh"};
218
219 // Chord states
220 enum chordStates{
221     FUNDAMENTAL_STATE,  //0
222     FIRST_INVERSION,    //1
223     SECOND_INVERSION,   //2
224     THIRD_INVERSION     //3
225 };
226
227 const vector<std::string> stateNames = {"Fundamental state", "First inversion", "Second
```

```
    inversion", "Third inversion"};
228
229 /** Modes */
230 enum Mode {
231     IONIAN,      //0 , major mode
232     DORIAN,      //1
233     PHRYGIAN,    //2
234     LYDIAN,      //3
235     MIXOLYDIAN, //4
236     AEOLIAN,     //5 , natural minor mode
237     LOCRIAN      //6
238 };
239
240 // syntactic sugar for more commonly used modes
241 const int MAJOR_MODE = IONIAN;
242 const int MINOR_MODE = AEOLIAN;
243
244 const vector<std::string> modeNames = {"Major", "Dorian", "Phrygian", "Lydian", "Mixolydian", "
    Minor", "Locrian"};
245
246 /** Scales */
247 /// defined by the intervals between their notes
248
249 //@todo turn this into a dictionary with the name of the scale as key and the vector of
    intervals as value
250
251 const vector<int> MAJOR_SCALE = {MAJOR_SECOND, MAJOR_SECOND, MINOR_SECOND, MAJOR_SECOND,
    MAJOR_SECOND, MAJOR_SECOND, MINOR_SECOND};
252 const vector<int> NATURAL_MINOR_SCALE = {MAJOR_SECOND, MINOR_SECOND, MAJOR_SECOND, MAJOR_SECOND,
     MINOR_SECOND, MAJOR_SECOND, MAJOR_SECOND};
253 const vector<int> HARMONIC_MINOR_SCALE = {MAJOR_SECOND, MINOR_SECOND, MAJOR_SECOND, MAJOR_SECOND
    , MINOR_SECOND, AUGMENTED_SECOND, MINOR_SECOND};
254 const vector<int> MELODIC_MINOR_SCALE = {MAJOR_SECOND, MINOR_SECOND, MAJOR_SECOND, MAJOR_SECOND,
     MAJOR_SECOND, MAJOR_SECOND, MINOR_SECOND};
255 const vector<int> BORROWED_SCALE = {PERFECT_FOURTH, MAJOR_THIRD, MAJOR_SECOND, MINOR_SECOND};
256 const vector<int> CHROMATIC_SCALE = {MINOR_SECOND,MINOR_SECOND,MINOR_SECOND,MINOR_SECOND,
    MINOR_SECOND,MINOR_SECOND,MINOR_SECOND,MINOR_SECOND,MINOR_SECOND,
257     MINOR_SECOND,MINOR_SECOND,MINOR_SECOND};
258
259 /// Modes eccl siastiques (gammes diatoniques). IONIAN_SCALE est identique    MAJOR_SCALE
260 /// et AEOLIAN_SCALE est identique    NATURAL_MINOR_SCALE ; ils sont red finis ici pour
261 /// permettre la d tection automatique de la gamme    partir du cantus firmus.
262 const vector<int> IONIAN_SCALE     = {MAJOR_SECOND, MAJOR_SECOND, MINOR_SECOND, MAJOR_SECOND,
    MAJOR_SECOND, MAJOR_SECOND, MINOR_SECOND};
263 const vector<int> DORIAN_SCALE     = {MAJOR_SECOND, MINOR_SECOND, MAJOR_SECOND, MAJOR_SECOND,
    MAJOR_SECOND, MINOR_SECOND, MAJOR_SECOND};
264 const vector<int> PHRYGIAN_SCALE   = {MINOR_SECOND, MAJOR_SECOND, MAJOR_SECOND, MAJOR_SECOND,
    MINOR_SECOND, MAJOR_SECOND, MAJOR_SECOND};
265 const vector<int> LYDIAN_SCALE     = {MAJOR_SECOND, MAJOR_SECOND, MAJOR_SECOND, MINOR_SECOND,
    MAJOR_SECOND, MAJOR_SECOND, MINOR_SECOND};
266 const vector<int> MIXOLYDIAN_SCALE = {MAJOR_SECOND, MAJOR_SECOND, MINOR_SECOND, MAJOR_SECOND,
    MAJOR_SECOND, MINOR_SECOND, MAJOR_SECOND};
267 const vector<int> AEOLIAN_SCALE    = {MAJOR_SECOND, MINOR_SECOND, MAJOR_SECOND, MAJOR_SECOND,
    MINOR_SECOND, MAJOR_SECOND, MAJOR_SECOND};
268 const vector<int> LOCRIAN_SCALE    = {MINOR_SECOND, MAJOR_SECOND, MAJOR_SECOND, MINOR_SECOND,
    MAJOR_SECOND, MAJOR_SECOND, MAJOR_SECOND};
269
270 // Gammes pentatoniques (5 notes) et blues (6 notes), utilis es pour la d tection auto.
271 const vector<int> PENTATONIC_MAJOR_SCALE = {MAJOR_SECOND, MAJOR_SECOND, MINOR_THIRD,
    MAJOR_SECOND, MINOR_THIRD};
272 const vector<int> PENTATONIC_MINOR_SCALE = {MINOR_THIRD, MAJOR_SECOND, MAJOR_SECOND, MINOR_THIRD
    , MAJOR_SECOND};
273 const vector<int> BLUES_MAJOR_SCALE      = {MAJOR_SECOND, MINOR_SECOND, MINOR_SECOND,
    MINOR_THIRD, MAJOR_SECOND, MINOR_THIRD};
274 const vector<int> BLUES_MINOR_SCALE      = {MINOR_THIRD, MAJOR_SECOND, MINOR_SECOND,
    MINOR_SECOND, MINOR_THIRD, MAJOR_SECOND};
275
276 /** Part related */
277 enum Species{
278     FIRST_SPECIES,  //0
279     SECOND_SPECIES, //1
280     THIRD_SPECIES,  //2
281     FOURTH_SPECIES, //3
282     FIFTH_SPECIES,  //4
283     CANTUS_FIRMUS   //5
284 };
285
286 const map<int,int> notesPerMeasure = {{FIRST_SPECIES, 1}, {SECOND_SPECIES, 2},
287                                       {THIRD_SPECIES, 4}, {FOURTH_SPECIES, 2},
288                                       {FIFTH_SPECIES, 4}, {CANTUS_FIRMUS, 1}};
289
290 enum nVoices{
291     TWO_VOICES,     //0
292     THREE_VOICES,   //1
293     FOUR_VOICES     //2
294 };
295
296 enum motions{
297     CONTRARY_MOTION,       //0
298     OBLIQUE_MOTION,        //1
```

```
    PARALLEL_MOTION,       //2
};

const vector<int> PERFECT_CONSONANCES = {UNISSON, PERFECT_FIFTH, PERFECT_OCTAVE};
const vector<int> IMPERFECT_CONSONANCES = {MAJOR_THIRD, MINOR_THIRD, MAJOR_SIXTH, MINOR_SIXTH};
const vector<int> CONSONANCES = {UNISSON, MINOR_THIRD, MAJOR_THIRD, PERFECT_FIFTH, MINOR_SIXTH,
    MAJOR_SIXTH, PERFECT_OCTAVE};
const vector<int> TRIAD = {UNISSON, MINOR_THIRD, MAJOR_THIRD, PERFECT_FIFTH};
const vector<int> DISONANCE = {MINOR_SECOND, MAJOR_SECOND, PERFECT_FOURTH, TRITONE,
    MINOR_SEVENTH, MAJOR_SEVENTH};

const int not_harmonic_triad_cost  = 16;
const int double_fifths_cost       = 8;
const int double_thirds_cost       = 6;
const int triad_with_octave_cost   = 2;

enum costValues{
    NO_COST,        //0
    LOW_COST,       //1
    MEDIUM_COST,    //2
    HIGH_COST = 4,
    LAST_RESORT = 8,
    PROPORTIONAL_TO_SIZE = 2,   /// To multiply by size
    FORBIDDEN = 64              /// To multiply by size
};

//** Max Step between two notes */
const int MAX_STEP = 19; // Tom Lai

//** Constraints */
// constraints enum
enum constraints{
    CF_1H1,
    CF_1H2_2V,
    CF_1H3_2V,
    CF_1H7_2V,
    CF_1P1,
    CF_1H7_3V,
    CF_1P3,
    STRATUM_UPPER_1H3,
    STRATUM_UPPER_1H10,
    STRATUM_UPPER_1H12,
    STRATUM_1H3,
    STRATUM_1H12,
    V2_G6,
    V2_G9,
    V2_1H2,
    V2_1H3,
    V2_1H5,
    V2_1M4,
    V2_1P8,
    V2_1P9,
    V3_G6,
    V3_1H4,
    V3_1H5,
    V3_1H8,
    V3_4H2,
    V3_1M4,
    V3_1P4,
    V3_1P6,
    V3_1P7,
    V3_1P8,
    V3_1P9,
    V3_2M2,
    V3_5R9,
    V4_G6,
    V4_1H4,
    V4_1H8,
    V4_4H2,
    V4_1M4,
    V4_1P4,
    V4_1P6,
    V4_1P7,
    V4_1P8,
    V4_1P9,
    V4_2M2,
    V4_5R9,
    V4_U2,
    SP1_G4,
    SP1_G7,
    SP1_1H1,
    SP1_1H6,
    SP1_1H7_2V,
    SP1_1M2_2V,
    SP1_1P1_2V,
    SP1_1P3_2V,
    SP1_1H7_3V,
    SP1_1M2_3V,
    SP1_1P1_3V,
    SP1_1P3_3V,
```

```
387     SP1_1M2_4V,
388     SP1_1P1_4V,
389     SP1_1P3_4V,
390     SP2_2H2,
391     SP2_2M1,
392     SP2_2P3,
393     SP2_2H3_2V,
394     SP2_2M2_2V,
395     SP2_2P1_2V,
396     SP2_2H3_3V,
397     SP2_1P1_3V,
398     SP2_2H3_4V,
399     SP2_1P1_4V,
400     SP3_3H1,
401     SP3_3H2,
402     SP3_3H3,
403     SP3_3M1,
404     SP3_1P3,
405     SP3_U1,
406     SP3_U2,
407     SP3_U3,
408     SP3_3H4_2V,
409     SP3_1H7_3V,
410     SP3_3H6_3V,
411     SP3_1P1_3V,
412     SP3_1H7_4V,
413     SP3_3H6_4V,
414     SP3_1P1_4V,
415     SP4_G6,
416     SP4_G7,
417     SP4_1H6,
418     SP4_4H1,
419     SP4_4M1,
420     SP4_4M2,
421     SP4_4P1,
422     SP4_4P2,
423     SP4_1M2,
424     SP4_U2,
425     SP4_4H2_2V,
426     SP4_4H3_2V,
427     SP4_1H2_2V,
428     SP4_4P5_3V,
429     SP4_4P5_4V,
430     SP5_3M3,
431     SP5_3M4,
432     SP5_H3,
433     SP5_H4,
434     SP5_H5,
435     SP5_H6,
436     SP5_M1,
437     SP5_M2,
438     SP5_M3,
439     SP5_M4,
440     SP5_P1,
441     SP5_4P1,
442     SP5_P3,
443     SP5_P4,
444     SP5_2V_1,
445     SP5_2V_2,
446     SP5_2V_3,
447     SP5_2V_4,
448     SP5_3V_1,
449     SP5_3V_2,
450     SP5_4V_1,
451     SP5_4V_2
452 };
453
454 const int consSize = static_cast<int>(SP5_4V_2+1); // Number of constraints
455 extern vector<bool> activeConstraints;
456 extern vector<bool> softConstraints;
457
458 enum toCombineConstraints{
459     H1_1,
460 };
461
462 const vector<string> constraintNames = {
463     "CF_1H1", "CF_1H2_2V", "CF_1H3_2V", "CF_1H7_2V", "CF_1P1", "CF_1H7_3V", "CF_1P3",
464     "STRATUM_UPPER_1H3", "STRATUM_UPPER_1H10", "STRATUM_UPPER_1H12", "STRATUM_1H3", "
        STRATUM_1H12",
465     "V2_G6", "V2_G9", "V2_1H2", "V2_1H3", "V2_1H5", "V2_1M4", "V2_1P8", "V2_1P9",
466     "V3_G6", "V3_1H4", "V3_1H5", "V3_1H8", "V3_4H2", "V3_1M4", "V3_1P4", "V3_1P6", "V3_1P7", "
        V3_1P8", "V3_1P9", "V3_2M2", "V3_5R9",
467     "V4_G6", "V4_1H4", "V4_1H8", "V4_4H2", "V4_1M4", "V4_1P4", "V4_1P6", "V4_1P7", "V4_1P8", "
        V4_1P9", "V4_2M2",  "V4_5R9","V4_U2",
468     "SP1_G4", "SP1_G7", "SP1_1H1", "SP1_1H6", "SP1_1H7_2V", "SP1_1M2_2V", "SP1_1P1_2V", "
        SP1_1P3_2V", "SP1_1H7_3V", "SP1_1M2_3V", "SP1_1P1_3V", "SP1_1P3_3V", "SP1_1M2_4V", "
        SP1_1P1_4V", "SP1_1P3_4V",
469     "SP2_2H2", "SP2_2M1", "SP2_2P3", "SP2_2H3_2V", "SP2_2M2_2V", "SP2_2P1_2V", "SP2_2H3_3V", "
        SP2_1P1_3V", "SP2_2H3_4V", "SP2_1P1_4V",
470     "SP3_3H1", "SP3_3H2", "SP3_3H3", "SP3_3M1", "SP3_1P3", "SP3_U1", "SP3_U2", "SP3_U3", "
```

```
    SP3_3H4_2V", "SP3_1H7_3V", "SP3_3H6_3V", "SP3_1P1_3V", "SP3_1H7_4V", "SP3_3H6_4V", "
    SP3_1P1_4V",
471     "SP4_G6", "SP4_G7", "SP4_1H6", "SP4_4H1", "SP4_4M1", "SP4_4M2", "SP4_4P1", "SP4_4P2", "
    SP4_1M2", "SP4_U2", "SP4_4H2_2V", "SP4_4H3_2V", "SP4_1H2_2V", "SP4_4P5_3V", "SP4_4P5_4V",
472     "SP5_3M3", "SP5_3M4", "SP5_H3", "SP5_H4", "SP5_H5", "SP5_H6", "SP5_M1", "SP5_M2", "SP5_M3",
    "SP5_M4", "SP5_P1", "SP5_4P1", "SP5_P3", "SP5_P4", "SP5_2V_1", "SP5_2V_2", "SP5_2V_3", "
    SP5_2V_4", "SP5_3V_1", "SP5_3V_2", "SP5_4V_1", "SP5_4V_2"
473 };
474
475 const vector<string> combinedCostNames = {"1H1"};
476
477 /*******************************************
478  *                                         *
479  *                Functions                *
480  *                                         *
481  *******************************************/
482
483 /**
484  * Returns the name of a constraint based on its integer value
485  * @param constraint an integer representing the constraint
486  * @return string the name of the constraint
487  */
488 string get_constraint_name(int constraint);
489
490 /**
491  * For a given set of intervals between notes that loops and a starting note, returns all the
      possible notes
492  * @param note the starting note
493  * @param intervals the set of intervals between notes. It must make a loop. For example, to get
       all notes from a major
494  * scale from note, use {2, 2, 1, 2, 2, 2, 1}. To get all notes from a minor chord, use {3, 4,
      5}.
495  * @return vector<int> all the notes
496  */
497 vector<int> get_all_notes_from_interval_loop(int n, vector<int> intervals);
498
499 /**
500  * For a given tonality (root + mode), returns all the possible notes
501  * @param root the root of the tonality (in [0,11])
502  * @param scale the set of tones and semitones that define the scale
503  * @return vector<int> all the possible notes from that tonality
504  */
505 vector<int> get_all_notes_from_scale(int root, vector<int> scale);
506
507 /**
508  * D tecte automatiquement la gamme musicale d'un cantus firmus.
509  *
510  * On suppose que la premi re note du cantus firmus donne la tonique. On teste alors,
511  * dans un ordre de priorit  (du plus g n ral au plus sp cifique : Ionien, Aeolien,
512  * Dorien, Mixolydien, Lydien, Phrygien, Locrien, mineur harmonique, mineur m lodique),
513  * la premi re gamme dont l'ensemble des classes de hauteurs (mod 12) contient toutes
514  * les notes du cantus firmus. Si plusieurs gammes conviennent, la plus g n rale est
515  * retenue. Si aucune ne correspond exactement, on retombe sur la gamme majeure.
516  *
517  * @param cf le cantus firmus (notes MIDI)
518  * @return les intervalles de la gamme d tect e (utilisable avec get_all_notes_from_scale)
519  */
520 vector<int> detect_scale_for_cf(const vector<int>& cf);
521
522 /**
523  * Renvoie le nom de la gamme d tect e pour un cantus firmus (   des fins de log).
524  */
525 string detect_scale_name_for_cf(const vector<int>& cf);
526
527 /**
528  * For a given chord (root + mode), returns all the possible notes
529  * @param root the root of the chord
530  * @param quality the set of tones and semitones that define the chord
531  * @return vector<int> all the possible notes from that chord
532  */
533 vector<int> get_all_notes_from_chord(int root, vector<int> quality);
534
535 /**
536  * Get all values for a given note
537  * @param note a note
538  * @return vector<int> a vector containing all the given notes
539  */
540 vector<int> get_all_given_note(int note);
541
542 /**
543  * Transforms an int* into a vector<int>
544  * @param ptr an int* pointer
545  * @param size the size of the array
546  * @return a vector<int> containing the same values as the array
547  */
548 vector<int> int_pointer_to_vector(int* ptr, int size);
549
550 /**
551  * Union algorithm found and adapted from GeeksForGeeks.com
552  */
```

```
553 vector<int> vector_union(vector<int> v1, vector<int> v2);
554
555 /**
556  * Intersection algorithm found and adapted from GeeksForGeeks.com
557  */
558 vector<int> vector_intersection(vector<int> v1, vector<int> v2);
559
560 /**
561  * Difference algorithm found and adapted from GeeksForGeeks.com
562  */
563 vector<int> vector_difference(vector<int> v1, int lb, int ub);
564
565 /**
566  * Transforms a vector of integers into a string
567  * @param vector a vector of integers
568  * @return string the string representation of the vector
569  */
570 string int_vector_to_string(vector<int> vector);
571
572 /**
573  * Transforms an IntVarArray into a string, with values separated by a space (for csv
        utilisation)
574  * @param array IntVarArray
575  * @return string the string representation of the vector
576  */
577 string int_var_array_to_string(IntVarArray array);
578
579 /**
580  * Prints the Search::Statistics object into a readable format
581  * @param stats a Search::Statistics object representing the statistics of a search
582  * @return The string representation of the statistics object
583  */
584 string statistics_to_string(Search::Statistics stats);
585
586 /**
587  * Prints the Search::Statistics object into a csv format (coma separated)
588  * @param stats a Search::Statistics object representing the statistics of a search
589  * @return The string representation of the statistics object
590  */
591 string statistics_to_csv_string(Search::Statistics stats);
592
593 /**
594  * Returns the value of a variable as a string
595  * @param var an integer variable
596  * @param absolute a boolean indicating if the value should be returned as an absolute value (
        default is false)
597  * @return a string representing the value of the variable
598  */
599 string intVar_to_string(const IntVar &var, bool absolute = false);
600
601 /**
602  * Returns the values of an array of variables as a string. Calls the intVar_to_string function
603  * @param vars an array of integer variables
604  * @return a string representing the values of the variables
605  */
606 string intVarArray_to_string(IntVarArray vars);
607
608 /**
609  * Returns the value of a variable as a string
610  * @param var an integer variable
611  * @param absolute a boolean indicating if the value should be returned as an absolute value (
        default is false)
612  * @return a string representing the value of the variable
613  */
614 string boolVar_to_string(const BoolVar &var, bool absolute = false);
615
616 /**
617  * Returns the values of an array of variables as a string
618  * @param vars an array of integer variables
619  * @return a string representing the values of the variables
620  */
621 string boolVarArray_to_string(BoolVarArray vars);
622
623 /**
624  * Returns the values of an array of variables as a string
625  * @param vars an array of integer variables
626  * @return a string representing the values of the variables, ignoring unassigned variables
627  */
628 string cleanIntVarArray_to_string(IntVarArray vars);
629
630 /**
631  * Returns the values of an IntVarArgs as a string
632  * @param args an IntVarArgs
633  * @return a string representing the values
634  */
635 string intVarArgs_to_string(IntVarArgs args);
636
637 /**
638  * Returns the Species object corresponding to an integer
639  * @param sp integer representing the species (1 for 1sp, 2 for 2sp, ...)
```

```
 * @return a Species object (basically an int too)
 */
Species int_to_species(int sp);

/**
 * Returns the name of the note given through MIDI value
 * @param note the MIDI value of the note
 * @return its name
 */
string midi_to_french(int note);

/**
 * Returns the name of a note based on its MIDI value
 * @param note an integer
 */
string midi_to_letter(int note);

/**
 * Returns the name of a mode based on its integer value
 * @param mode an integer
 * @return a string representing the name of the mode
 */
string mode_int_to_name(int mode);

/**
 * returns a string with the time
 * @return a string with the time
 */
string time();

/**
 * Write a text into a log file
 * @param message the text to write
 */
void write_to_log_file(const char *message, const string& filename);

void writeToLogFile(const char* message);


/* ================================================
 *         CONSTRAINT HELPERS
 * ================================================
 */

IntVarArray expandCantusNotes(Home home, IntVarArray cantus);
vector<int> createRangeVector(int from, int to, int multiplier);

/* ================================================
 *         DYNAMIC BRANCHING SELECTORS
 * ================================================
 * The solutionArray branching heuristic in TwoVoiceCounterpoint reads
 * these globals at construction time, so callers can swap them without
 * recompiling. Defaults reproduce the production setting
 * (AFC_MAX + VAL_RND(1U) + seed 1) so any other binary keeps its current
 * behaviour as long as it does not touch these.
 */

enum BranchVarSel {
    BR_VAR_SIZE_MIN = 0,
    BR_VAR_SIZE_MAX,
    BR_VAR_DEGREE_MAX,
    BR_VAR_AFC_MAX,
    BR_VAR_ACTION_MAX,
    BR_VAR_NONE
};

enum BranchValSel {
    BR_VAL_MIN = 0,
    BR_VAL_MAX,
    BR_VAL_MED,
    BR_VAL_RND,
    BR_VAL_SPLIT_MIN,
    BR_VAL_SPLIT_MAX
};

extern BranchVarSel g_solution_var_sel;
extern BranchValSel g_solution_val_sel;
extern unsigned int g_solution_val_rnd_seed;

const char* branch_var_sel_name(BranchVarSel s);
const char* branch_val_sel_name(BranchValSel s);
BranchVarSel parse_branch_var_sel(const string& s);
BranchValSel parse_branch_val_sel(const string& s);

// Applies branch(home, vars, ...) using the current global selectors.
// The helper is intentionally limited to the solutionArray case so that
// touching the globals never changes the other branching layers.
void branch_solution_array_dynamic(Home home, const IntVarArgs& vars);

#endif
```

A.4.3 Generations.hpp

```cpp
#pragma once


#include <iostream>
#include <fstream>
#include <sstream>
#include <chrono>
#include <iomanip>
#include <vector>
#include <map>
#include <string>
#include <utility>
#include <tuple>
#include <algorithm>
#include <gecode/int.hh>
#include <gecode/search.hh>

#include "Utilities.hpp"
#include "Midi.hpp"
#include "CounterpointProblems/CounterpointUtils.hpp"
#include "CounterpointProblems/CounterpointProblem.hpp"
#include "CounterpointProblems/TwoVoiceCounterpoint.hpp"
#include "CounterpointProblems/ThreeVoiceCounterpoint.hpp"
#include "CounterpointProblems/FourVoiceCounterpoint.hpp"
#include "ConfigLoader.hpp"

// ============================================================
// Internal generation case
// Used by the engine and by the API / CLI.
// ============================================================

struct GenerationCase {
    int cf_id = 1;
    vector<int> cf;
    string cf_name;
    string cf_scale;
    bool use_preset_cf = false;

    int n_voices = 0;
    vector<Species> spList;
    vector<int> v_type;

    string preset_name = "default";
    vector<int> melodic_params;
    vector<int> general_params;
    vector<int> specific_params;
    vector<int> importance;
    int borrow_mode = 0;
    ObjectiveMode obj_mode = OBJECTIVE_PONDERED;

    int timeout_ms = 300000;
    int stagnation_ms = 120000;

    string output_root = "../../results";
    string output_subdir;
};

// ============================================================
// Public result returned by reusable generation entrypoints
// ============================================================

struct GenerationResult {
    // ----- Bench Results -----
    bool success = false;
    string termination = "NONE";

    int nb_solutions = 0;
    int nb_improvements = 0;
    double best_cost = 1e18; // must start above any possible cost, else no solution is ever kept
    double ms_first_solution = 0;
    double ms_last_improve = 0;
    double ms_total = 0;

    Gecode::Search::Statistics stats;

    vector<int> full_solution;
    vector<pair<vector<int>, Species>> generated_voices;
    // Per-counterpoint rhythm classification (only non-empty for 5th species):
    // a "0/1" string of isThirdSpeciesArray, used to audit which flat-MIDI cells
    // are genuine 3rd-species attacks vs phantom (continuation/syncope) cells.
    vector<string> rhythm_dump;

    vector<tuple<int,double,double>> improvements;
    vector<tuple<int,double,double>> checkpoints;
    vector<tuple<int,double,double,string>> solutions_log;

```

```
    // ----- General informations -----
    string midi_path;
    string txt_path;
    string csv_path;
    string error_txt_path;

    vector<int> resolved_cf;
    string resolved_cf_name;
    string resolved_cf_scale;
};

// ============================================================
// Generation engine facade
// ============================================================

class Generations {
public:
    Generations() = default;

    // Main reusable engine entrypoint.
    // This is the one both CLI-side wrappers and API-side wrappers should call.
    static GenerationResult run_generation_case(GenerationCase& gc, bool save_outputs = true,
    bool verbose = true);

};

// Optional convenience helpers if you want them callable outside.
bool resolve_cf(GenerationCase& gc, bool verbose=true);
bool apply_preset(GenerationCase& gc);
```

A.4.4 GenerationsAPI.hpp

```
#pragma once

#include "Generations.hpp"

// ====================================================================
// GenerationAPI.hpp     simplified public entrypoint for generation.
//
// This API is meant for external callers that want to generate a
// counterpoint without depending on the CLI or on the internal workflow
// of the generation engine.
//
// This interface is intentionally lighter than the CLI: the cantus firmus
// is always provided directly, no campaign logic, no list commands, no
// testing-specific options
//
// Typical usage:
//   1. Fill a GenerationInput structure
//   2. Call generate_counterpoint(input)
//   3. Read the returned GenerationResult
//
// For custom metaparameters (melodic_params, general_params, specific_params and importance) :
//   - Add a line in config > presets.csv
//   - set 'preset_name' to the name of your new preset
//
// For example of uses, see "QUICK TEST on API" part in Tests/main.cpp
//
// ====================================================================


// Input of the public generation API.
struct GenerationInput {
    // ----- Required -----

    // MIDI notes of the cantus firmus. The first note also acts as the tonal reference for
    scale detection.
    // Example: {60, 62, 65, 64, 67, 65, 64, 62, 60}
    vector<int> cf_notes;

    // Species, of the generated counterpoint voices only.
    // Example: {FIRST_SPECIES, THIRD_SPECIES}
    // This means: 1 cantus firmus, 1 first-species generated voice, 1 third-species generated
    voice
    vector<Species> species;

    // Voice-position / register types of the generated counterpoint voices.
    // Possible value of counterpoint notes are (cf_notes[0] + v_type * 6) +/- PERFECT_OCTAVE,
    with PERFECT_OCTAVE = 12
    // Example: {0, 2}
    // This means : first counterpoint centered around cf_notes[0], second counterpoint around
    cf_notes[0] + 2 * 6
    vector<int> v_type;


    // ----- Optionnal -----
    // If not specified, the engine should fall back to its default preset behavior
```

```
52
53      // Name of the preset to load from presets.csv.
54      string preset_name = "";
55
56      // Maximum total runtime of the search, in milliseconds.
57      int timeout_ms = -1;
58
59      // Maximum allowed stagnation time, in milliseconds. Search stops if no improvement is found
         during this delay.
60      int stagnation_ms = -1;
61
62      // Objective mode used by the solver.
63      //
64      // Possible values:
65      //   OBJECTIVE_LEX
66      //   OBJECTIVE_TOTAL
67      //   OBJECTIVE_MIXED
68      //   OBJECTIVE_PONDERED
69      ObjectiveMode obj_mode = OBJECTIVE_PONDERED;
70
71
72      // Root directory where result files should be written.
73      string output_root = "";
74
75      // Optional subdirectory added under output_root. Useful for grouping several generations in
         the same campaign or test.
76      string output_subdir = "";
77
78      // If true, the engine writes output files (txt / csv / midi when relevant). If false,
        generation is run without saving any files.
79      bool save_outputs = true;
80
81      // If true, the engine log  (txt / csv / midi when relevant). If false, generation will only
         print errors and checkpoints.
82      bool verbose = false;
83  };
84
85  // Launches one generation from a fully specified GenerationInput.
86  //
87  // The returned GenerationResult contains:
88  //   - search statistics
89  //   - termination status
90  //   - the full solution
91  //   - generated voices split by species
92  //   - resolved cantus firmus metadata
93  //   - output file paths when saving is enabled
94  GenerationResult generate_counterpoint(const GenerationInput& input);
95
96  GenerationResult generate_counterpoint(vector<int> cf_notes, vector<Species> species, vector<int
        > v_type);
```

A.4.5 ConfigLoader.hpp

```
1  // [Claude Code]
2  // ConfigLoader.hpp
3  // Chargement des configurations CSV (cantus_firmus, presets).
4  // Toutes les valeurs autrefois hardcod es dans Generations.cpp sont d sormais
5  // externalis es dans FuxCP/config/*.csv.
6
7  #ifndef FUXCP_CONFIG_LOADER_HPP
8  #define FUXCP_CONFIG_LOADER_HPP
9
10 #include <string>
11 #include <vector>
12 #include <map>
13
14 struct CFEntry {
15      int id;
16      std::vector<int> notes;  // notes MIDI du cantus firmus
17      std::string name;        // libell  court (ex "Do Majeur (court)")
18      std::string scale;       // gamme indicative (informatif uniquement)
19      std::string description;
20 };
21
22 struct PresetEntry {
23      std::string name;
24      std::vector<int> melodic_params;   // 8 entiers
25      std::vector<int> general_params;   // 8 entiers
26      std::vector<int> specific_params;  // 7 entiers
27      std::vector<int> importance;       // 14 entiers
28      int borrow_mode;
29      std::string description;
30 };
31
32 class ConfigLoader {
33 public:
34      // Localise le dossier config/ (cherche ../config/ depuis le CWD courant).
```

```
    static std::string find_config_dir();

    // Lecture des deux CSV (chemin = dossier config).
    static std::map<int, CFEntry>     load_cantus_firmus(const std::string& config_dir);
    static std::map<std::string, PresetEntry> load_presets(const std::string& config_dir);

    // Helpers
    static std::vector<int> parse_int_list(const std::string& s, char sep = ' ');
    static std::vector<std::string> split(const std::string& s, char sep);
    static std::string trim(const std::string& s);
};

#endif
```

A.4.6 Midi.hpp

```
#ifndef MIDI_HPP
#define MIDI_HPP

#include <vector>
#include <string>
#include <utility>
#include "Utilities.hpp"
#include "CounterpointProblems/CounterpointProblem.hpp"

// Utils
std::vector<int> extract_notes(CounterpointProblem* best,
        vector<Species> spList,
        size_t cfSize);

/**
 * Calcule la taille du tableau de branching notes pour une esp ce et une taille de CF donn es
    .
 */
int branchingNotesSize(Species sp, int cfSize);

/**
 * G  n  re un fichier MIDI    partir d'un Cantus Firmus et d'une solution (2 voix).
 * @param filename Nom du fichier de sortie (ex: "resultat.mid")
 * @param cantusFirmus Vecteur de notes MIDI
 * @param solution Vecteur de notes MIDI (contrepoint)
 * @param species L'esp ce utilis e pour d terminer le rythme
 */
void saveMidi(const std::string& filename,
              const std::vector<int>& cantusFirmus,
              const std::vector<int>& solution,
              Species species);

/**
 * G  n  re un fichier MIDI multi-voix Format 1 (une piste par voix).
 * Piste 0 = Cantus Firmus, pistes suivantes = voix de contrepoint.
 * @param filename Nom du fichier de sortie
 * @param cantusFirmus Vecteur de notes MIDI du cantus firmus
 * @param voices Vecteur de paires (notes, esp ce) pour chaque voix de contrepoint
 */
void saveMidiMultiVoice(const std::string& filename,
                        const std::vector<int>& cantusFirmus,
                        const std::vector<std::pair<std::vector<int>, Species>>& voices);


#endif
```

A.5 Test headers

A.5.1 fuxTest.hpp

```
//
// Created by Luc Cleenewerk and Diego de Patoul.
// This file is the header file of the testing framework implementation.
//

#ifndef FUX_TESTS_HPP
#define FUX_TESTS_HPP

#include "../Utilities.hpp"
#include "../Midi.hpp"
#include "../Parts/Part.hpp"
#include "../CounterpointProblems/CounterpointUtils.hpp"
#include "../CounterpointProblems/CounterpointProblem.hpp"
#include "../Parts/CantusFirmus.hpp"

```

```
16 using namespace Gecode;
17 using namespace std;
18
19 class FuxTest{
20
21 protected:
22     vector<Species> spList;
23     vector<int> cantusFirmus;
24     vector<int> v_type;
25     int borrowMode;
26     vector<int> cp;
27     int idx;
28     int cfSize;
29     vector<int> melodic_params;
30     vector<int> general_params;
31     vector<int> specific_params;
32     vector<int> importance;
33
34 public:
35
36 // ========== Modified by Tom ====================
37
38     void test_G6();
39     void test_G6_2v_1sp();
40
41     void test_1H1();
42     void test_1H1_2v_1sp();
43     void test_1H1_3v_1sp();
44     void test_1H1_4v_1sp();
45     void test_1H1_2v_2sp();
46     void test_1H1_3v_2sp();
47     void test_1H1_4v_2sp();
48     void test_1H1_2v_3sp();
49     void test_1H1_3v_3sp();
50     void test_1H1_4v_3sp();
51     void test_1H1_2v_4sp();
52     void test_1H1_3v_4sp();
53     void test_1H1_4v_4sp();
54     void test_1H1_2v_5sp();
55     void test_1H1_3v_5sp();
56     void test_1H1_4v_5sp();
57
58     void test_1H2();
59     void test_1H2_2v_1sp();
60     void test_1H2_2v_2sp();
61     void test_1H2_2v_3sp();
62     void test_1H2_2v_4sp();
63
64     void test_1H3();
65     void test_1H3_2v_1sp();
66     void test_1H3_3v_1sp();
67     void test_1H3_4v_1sp();
68     void test_1H3_2v_2sp();
69     void test_1H3_3v_2sp();
70     void test_1H3_4v_2sp();
71     void test_1H3_2v_3sp();
72     void test_1H3_3v_3sp();
73     void test_1H3_4v_3sp();
74     void test_1H3_2v_4sp();
75     void test_1H3_3v_4sp();
76     void test_1H3_4v_4sp();
77
78     void test_1H4();
79     void test_1H4_2v_1sp();
80     void test_1H4_3v_1sp();
81     void test_1H4_4v_1sp();
82     void test_1H4_2v_2sp();
83     void test_1H4_3v_2sp();
84     void test_1H4_4v_2sp();
85     void test_1H4_2v_3sp();
86     void test_1H4_3v_3sp();
87     void test_1H4_4v_3sp();
88     void test_1H4_2v_4sp();
89     void test_1H4_3v_4sp();
90     void test_1H4_4v_4sp();
91
92     void test_1H5();
93     void test_1H5_2v_1sp();
94     void test_1H5_3v_1sp();
95     void test_1H5_2v_2sp();
96     void test_1H5_3v_2sp();
97     void test_1H5_2v_3sp();
98     void test_1H5_3v_3sp();
99     void test_1H5_2v_4sp();
100    void test_1H5_3v_4sp();
101
102    void test_1H6();
103    void test_1H6_2v_1sp();
104
105    void test_1H7();
```

```
106     void test_1H7_2v_1sp();
107     void test_1H7_3v_1sp();
108     void test_1H7_2v_2sp();
109     void test_1H7_3v_2sp();
110     void test_1H7_2v_3sp();
111     void test_1H7_3v_3sp();
112     void test_1H7_2v_4sp();
113     void test_1H7_3v_4sp();
114
115     void test_1H10();
116     void test_1H10_3v_1sp();
117     void test_1H10_3v_2sp();
118     void test_1H10_3v_3sp();
119     void test_1H10_3v_4sp();
120
121     void test_1M2();
122     void test_1M2_2v_1sp();
123     void test_1M2_3v_1sp();
124     void test_1M2_4v_1sp();
125
126     void test_1P1_2v_1sp();
127
128     void test_2H2();
129     void test_2H2_2v_2sp();
130     void test_2H2_3v_2sp();
131     void test_2H2_4v_2sp();
132
133     void test_2M2();
134     void test_2M2_2v_2sp();
135
136     void test_2v_1sp_fig22();
137     void test_2v_1sp_fig23();
138     void test_2v_2sp_fig38();
139     void test_2v_2sp_fig39();
140     void test_2v_2sp_fig40();
141     void test_2v_2sp_fig41();
142     void test_2v_2sp_fig42();
143     void test_2v_2sp_fig43();
144     void test_2v_2sp_fig44();
145     void test_2v_2sp_fig45();
146     void test_2v_3sp_fig55();
147     void test_2v_3sp_fig56();
148     void test_2v_3sp_fig57();
149     void test_2v_3sp_fig58();
150     void test_2v_3sp_fig59();
151     void test_2v_3sp_fig60();
152     void test_2v_4sp_fig74();
153     void test_2v_4sp_fig75();
154     void test_2v_4sp_fig76();
155     void test_2v_4sp_fig77();
156     void test_2v_4sp_fig78();
157     void test_3v_1sp_fig108();
158     void test_3v_1sp_fig109();
159     void test_3v_1sp_fig110();
160     void test_3v_1sp_fig111();
161     void test_3v_1sp_fig112();
162     void test_3v_1sp_fig113();
163     void test_3v_1sp_fig114();
164     void test_3v_1sp_fig115();
165     void test_3v_1sp_fig116();
166     void test_3v_1sp_fig117();
167     void test_3v_1sp_fig118();
168     void test_3v_1sp_fig119();
169     void test_3v_2sp_fig125();
170     void test_3v_2sp_fig128();
171     void test_3v_2sp_fig129();
172     void test_3v_3sp_fig132();
173     void test_3v_3sp_fig133();
174     void test_3v_4sp_fig150();
175     void test_3v_4sp_fig151();
176     void test_4v_1sp_fig171();
177     void test_4v_1sp_fig172();
178     void test_4v_2sp_fig175();
179     void test_4v_2sp_fig176();
180     void test_4v_3sp_fig184();
181     void test_4v_3sp_fig186();
182     void test_4v_4sp_fig193();
183     void test_4v_4sp_fig196();
184
185
186     int get_motions(CounterpointProblem* problem, int i);
187
188 // ================================================
189
190     FuxTest(char* test);
191
192     FuxTest(int testNumber);
193
194     FuxTest(int testNumber, int i);
195
```

```
196     vector<Species> getSpList();
197     vector<int> getCf();
198     vector<int> getVType();
199     int getBMode();
200     vector<int> getCp();
201     int getIdx();
202 
203     CounterpointProblem* test_1sp_H1();
204     CounterpointProblem* test_1sp_H2();
205     CounterpointProblem* test_1sp_H3();
206     CounterpointProblem* test_1sp_H3_2();
207     CounterpointProblem* test_1sp_H4_1();
208     CounterpointProblem* test_1sp_H4_2();
209     CounterpointProblem* test_1sp_H5();
210     CounterpointProblem* test_1sp_H7();
211     CounterpointProblem* test_1sp_H7_2();
212     CounterpointProblem* test_2sp_H2();
213     CounterpointProblem* test_3sp_H1();
214     CounterpointProblem* test_4sp_H2();
215 
216     CounterpointProblem* dispatcher(char* test);
217 
218     void test_2v_1sp_fig22_setter(int i);
219     void test_2v_1sp_fig23_setter(int i);
220 
221     void test_2v_2sp_fig38_setter(int i);
222     void test_2v_2sp_fig39_setter(int i);
223     void test_2v_2sp_fig40_setter(int i);
224 
225     void test_2v_3sp_fig55_setter(int i);
226 
227     void test_2v_4sp_fig74_setter(int i);
228 
229     void test_2v_5sp_fig82_setter(int i);
230 
231     void test_3v_1sp_fig108_setter(int i);
232     void test_3v_1sp_fig109_setter(int i);
233     void test_3v_1sp_fig110_setter(int i);
234     void test_3v_1sp_fig111_setter(int i);
235 
236     void test_3v_2sp_fig125_setter(int i);
237 
238     void test_3v_4sp_fig146_setter(int i);
239 
240     void test_4v_1sp_fig166_setter(int i);
241     void test_4v_1sp_fig167_setter(int i);
242 
243     void test_4v_2sp_fig176_setter(int i);
244 
245     void test_configuration();
246 
247     void test_gen_2v_1sp();
248     void test_gen_2v_2sp();
249     void test_gen_2v_3sp();
250     void test_gen_2v_4sp();
251     void test_gen_3v_1sp();
252     void test_gen_3v_2sp();
253     void test_gen_3v_2sp_2sp();
254     void test_gen_3v_3sp();
255     void test_gen_3v_3sp_3sp();
256     void test_gen_3v_4sp();
257     void test_gen_4v_1sp();
258     void test_gen_4v_2sp();
259     void test_gen_4v_2sp_2sp_2sp();
260     void test_sacha_4v_3sp();
261     void test_gen_4v_3sp();
262     void test_gen_4v_4sp();
263 
264     void test_gen_custom();
265     void test_gen_constrains_check(Species species, int n_voices, vector<int> v_cases, vector<
        int> v_cases_1sp);
266 
267     void test_gen_classic();
268     void test_gen_multi();
269     void test_gen_all();
270     void test_sacha();
271 };
272 
273 void printVector(const std::vector<int>& array);
274 
275 void printIntVarArray(const IntVarArray& array);
276 
277 std::vector<CounterpointProblem*> get_all_solutions(CounterpointProblem* problem);
278 
279 
280 #endif
```

A.5.2 figureTests.hpp

```
#ifndef FIGURE_TESTS_HPP
#define FIGURE_TESTS_HPP

#include "fuxTest.hpp"

class FigureTests {
private:
    std::vector<Species> spList;
    std::vector<int> cantusFirmus;
    std::vector<int> cp;
    std::vector<int> v_type;
    std::vector<int> melodic_params;
    std::vector<int> general_params;
    std::vector<int> specific_params;
    std::vector<int> importance;
    std::vector<int> notesSpeciesFor5sp; // species for each notes in the 5th species
    int borrowMode;
    void test_configuration();
    CounterpointProblem* set_configuration();
    std::vector<CounterpointProblem*> get_all_solutions();
    bool is_unsat(const set<int>& cons_set);
    set<int> minimize (const std::set<int>& activeCons);
    void findAllMUSes();
    void find_unsat_constraints();

public:
    FigureTests();

    // Two voice first species figures
    void test_2v_1sp_fig22();
    void test_2v_1sp_fig23();

    // Two voice second species figures
    void test_2v_2sp_fig38();
    void test_2v_2sp_fig39();
    void test_2v_2sp_fig40();
    void test_2v_2sp_fig41();
    void test_2v_2sp_fig42();
    void test_2v_2sp_fig43();
    void test_2v_2sp_fig44();
    void test_2v_2sp_fig45();

    // Two voice third species figures
    void test_2v_3sp_fig55();
    void test_2v_3sp_fig56();
    void test_2v_3sp_fig57();
    void test_2v_3sp_fig58();
    void test_2v_3sp_fig59();
    void test_2v_3sp_fig60();

    // Two voice fourth species figures
    void test_2v_4sp_fig74();
    void test_2v_4sp_fig75();
    void test_2v_4sp_fig76();
    void test_2v_4sp_fig77();
    void test_2v_4sp_fig78();

    // Two voice fifth species figures
    void test_2v_5sp_fig82();
    void test_2v_5sp_fig83();
    void test_2v_5sp_fig87_1();

    // Three voice first species figures
    void test_3v_1sp_fig108();
    void test_3v_1sp_fig109();
    void test_3v_1sp_fig110();
    void test_3v_1sp_fig111();
    void test_3v_1sp_fig112();
    void test_3v_1sp_fig113();
    void test_3v_1sp_fig114();
    void test_3v_1sp_fig115();
    void test_3v_1sp_fig116();
    void test_3v_1sp_fig117();
    void test_3v_1sp_fig118();
    void test_3v_1sp_fig119();

    // Three voice second species figures
    void test_3v_2sp_fig125();
    void test_3v_2sp_fig126();
    void test_3v_2sp_fig127();
    void test_3v_2sp_fig128();
    void test_3v_2sp_fig129();

    // Three voice third species figures
    void test_3v_3sp_fig130();
    void test_3v_3sp_fig131();
    void test_3v_3sp_fig132();
```

```
    void test_3v_3sp_fig133();

    // Three voice fourth species figures
    void test_3v_4sp_fig146();
    void test_3v_4sp_fig147();
    void test_3v_4sp_fig148();
    void test_3v_4sp_fig149();
    void test_3v_4sp_fig150();
    void test_3v_4sp_fig151();

    // Three voice fifth species figures
    void test_3v_5sp_fig154();
    void test_3v_5sp_fig155();
    void test_3v_5sp_fig156();
    void test_3v_5sp_fig157();

    // Four voice first species figures
    void test_4v_1sp_fig166();
    void test_4v_1sp_fig167();
    void test_4v_1sp_fig168();
    void test_4v_1sp_fig169();
    void test_4v_1sp_fig170();
    void test_4v_1sp_fig171();
    void test_4v_1sp_fig172();

    // Four voice second species figures
    void test_4v_2sp_fig173();
    void test_4v_2sp_fig174();
    void test_4v_2sp_fig175();
    void test_4v_2sp_fig176();

    // Four voice third species figures
    void test_4v_3sp_fig183();
    void test_4v_3sp_fig184();
    void test_4v_3sp_fig185();
    void test_4v_3sp_fig186();

    // Four voice fourth species figures
    void test_4v_4sp_fig196();

    // Four voice fifth species figures
    void test_4v_5sp_fig200();
    void test_4v_5sp_fig201();

    // Multispecies figures
    void test_4v_Xsp_fig204();

    // Run all figure tests
    void run_all_tests();

    void run_twoVoice_tests();
    void run_threeVoice_tests();
    void run_fourVoice_tests();

    void run_thirdSpecies_tests();
    void run_fourthSpecies_tests();
    void run_fifthSpecies_tests();

    void MUSTest();

    void quickTest();

    // Add new method for testing cost summing
    void test_cost_summing();
};

#endif // FIGURE_TESTS_HPP
```

A.6 Counterpoint problem sources

A.6.1 CounterpointProblem.cpp

```
//
// Created by Luc Cleenewerk and Diego de Patoul.
//

#include "../../headers/CounterpointProblems/CounterpointProblem.hpp"
#include "../../headers/CounterpointProblems/CounterpointUtils.hpp"

/**
 * Constructor of the class.
 * @param cf a vector<int> representing the cantus firmus.
 * @param k the key of the score. it takes values from the notes in headers/Utilities.hpp
 * @param lb the lowest note possible for the counterpoint in MIDI
```

```
 * @param ub the highest note possible for the counterpoint in MIDI
 */
CounterpointProblem::CounterpointProblem(vector<int> cf, int v_type, vector<int> m_costs, vector<int> g_costs, vector<int> s_costs,
    vector<int> imp, int nV, ObjectiveMode objMode){
    nMeasures = cf.size();
    lowest = new Stratum(*this, nMeasures, 0, 127);
    cantusFirmus = new CantusFirmus(*this, nMeasures, cf, lowest, v_type, m_costs, g_costs, s_costs, nV);
    importance = imp;
    objectiveMode = objMode;
    n_unique_costs = 0;
    importanceNames = {"borrow", "fifth", "octave", "succ", "variety", "triad", "direct", "motion", "penult", "cambiata", "triad3" ,"m2", "syncopation", "melodic"};

    //creating the map with the names of the costs and their importance

    prefs = {};

    setPreferenceMap(importanceNames);

    orderedFactors = IntVarArray(*this, 14, 0, 1000);   //orderedFactors will contain the costs in order of their importance
    for(int i = 0; i < 14; i++){
        vector<string> tmp = {};
        costLevels.push_back(tmp);                      //costLevels will contain all the costs at a specific level of importance (1 to 14)
    }

    for(const auto& entry : prefs){
        int val = entry.second-1;
        costLevels[val].push_back(entry.first);         //initialize the costLevels
    }

    globalCost = IntVar(*this, 0, 2000000);             //contains the global cost
    objectiveCostSum = IntVar(*this, 0, 2000000);          //contains the costs sum to use with the objectiveMode options

    writeToLogFile("counterpointproblem constructor");

    // Constraints Cost
    combinedCosts = IntVarArray(*this, combinedCostNames.size(), 0, 10000);         //combinedCosts will contain the costs of the different counterpoints

    // 1H1
    // combined constraints has to be respect user preferences
    int respect_percent_1H1 = 50; //50 allowed every Fux figures tested (figures 60 is the worst)
    //TODO: should be replaced by the user preference

    int nCP = 1; // number of counterpoints
    if (counterpoint_2 != nullptr) {
        nCP ++;
    }
    if (counterpoint_3 != nullptr) {
        nCP ++;
    }
    int n_thesis_notes = this->nMeasures * nCP;
    int threshold = n_thesis_notes - (respect_percent_1H1 * n_thesis_notes) / 100;
    rel(*this, combinedCosts[H1_1], IRT_LQ, threshold); // 1H1
}

// COPY CONSTRUCTOR
CounterpointProblem::CounterpointProblem(CounterpointProblem& s) : IntLexMinimizeSpace(s){
    if (s.cantusFirmus) {
        cantusFirmus = s.cantusFirmus->clone(*this);
    } else {
        cantusFirmus = nullptr;
    }
    if (s.lowest) {
        lowest = s.lowest->clone(*this);
    } else {
        lowest = nullptr;
    }
    if (s.counterpoint_1) {
        counterpoint_1 = s.counterpoint_1->clone(*this);
    } else {
        counterpoint_1 = nullptr;
    }
    if (s.counterpoint_2) {
        counterpoint_2 = s.counterpoint_2->clone(*this);
    } else {
        counterpoint_2 = nullptr;
    }
    if (s.counterpoint_3) {
        counterpoint_3 = s.counterpoint_3->clone(*this);
    } else {
        counterpoint_3 = nullptr;
    }
    if(s.upper_1){
```

```
        upper_1 = s.upper_1->clone(*this);
    } else {
        upper_1 = nullptr;
    }
    if(s.upper_2){
        upper_2 = s.upper_2->clone(*this);
    } else {
        upper_2 = nullptr;
    }
    if(s.upper_3){
        upper_3 = s.upper_3->clone(*this);
    } else {
        upper_3 = nullptr;
    }
    nMeasures = s.nMeasures;
    importance = s.importance;
    prefs = s.prefs;
    sorted_voices = s.sorted_voices;
    combinedCosts = s.combinedCosts;
    unitedCostNames = s.unitedCostNames;
    costLevels = s.costLevels;
    n_unique_costs = s.n_unique_costs;
    importanceNames = s.importanceNames;
    combinedCosts.update(*this, s.combinedCosts);
    successiveCostArray.update(*this, s.successiveCostArray);
    triadCostArray.update(*this, s.triadCostArray);
    unitedCosts.update(*this, s.unitedCosts);
    solutionArray.update(*this, s.solutionArray);
    sortedCosts.update(*this, s.sortedCosts);
    orderedFactors.update(*this, s.orderedFactors);
    finalCosts.update(*this, s.finalCosts);
    for(int i = 0; i < sorted_voices.size(); i++){
        sorted_voices[i].update(*this, s.sorted_voices[i]);
    }
    globalCost.update(*this, s.globalCost);
    objectiveMode = s.objectiveMode;
    objectiveCostSum.update(*this, s.objectiveCostSum);

    hasRelaxation = s.hasRelaxation;
    problemRelaxationCosts.update(*this, s.problemRelaxationCosts);
    if(hasRelaxation){
        totalRelaxationCost.update(*this, s.totalRelaxationCost);
    }
}

IntLexMinimizeSpace* CounterpointProblem::copy(){
    return new CounterpointProblem(*this);
}

void CounterpointProblem::constrain(const IntLexMinimizeSpace& _b){

    //const CounterpointProblem &b = dynamic_cast<const CounterpointProblem &>(_b);
    IntLexMinimizeSpace::constrain(_b); // Faster, default one (the previous one was empty ?!?)

}

IntVarArgs CounterpointProblem::cost() const{
    IntVarArgs baseCosts = IntVarArgs(finalCosts);

    // If relaxation costs exist, prepend totalRelaxationCost as highest priority
    if(hasRelaxation){
        baseCosts = totalRelaxationCost + finalCosts;
    }

    if (objectiveMode != OBJECTIVE_LEX) {
        return IntVarArgs(objectiveCostSum + baseCosts);
    }
    return baseCosts;
}

string CounterpointProblem::to_string() const {
    string text = "Counterpoint problem : \n";
    text += cantusFirmus->to_string();
    text += "\n";
    text += "All costs : \n";
    text += intVarArray_to_string(unitedCosts);
    text += "\nLowest : \n";
    text += lowest->to_string();
    text += "\n";
    text += "Upper : \n";
    text += upper_1->to_string();
    text += "\n";
    text += "Final Costs : \n";
    text += intVarArray_to_string(finalCosts);
    text += "\n";
    return text;
}

Home CounterpointProblem::getHome(){
    return *this;
```

```
}

void CounterpointProblem::setPreferenceMap(vector<string> importance_names){
    for(int i = 0; i < importance.size(); i++){
        prefs.insert({importance_names[i], importance[i]});
    }
}

double CounterpointProblem::getCost() const {
    double total = 0;
    IntVarArgs costs = cost();
    for (int i = 0; i < costs.size(); ++i) {
        total += costs[i].val();
    }
    return total;
}

void CounterpointProblem::orderCosts(){
    for(int i = 0; i < 14; i++){
        if(!costLevels[i].empty()){
            int sm_size = 0;
            for(int k = 0; k < costLevels[i].size(); k++){
                //goes through every cost at the cost level
                for(int t = 0; t < unitedCostNames.size(); t++){
                    if(unitedCostNames[t]==costLevels[i][k]){
                        //adjusts the amount of costs at this level
                        sm_size++;
                    }
                }
            }
            //now cst_sm_size contains the size of all the cost array on this level combined
            IntVarArgs sm(sm_size); //sum of all the costs on this level
            int idx = 0;
            for(int k = 0; k < costLevels[i].size(); k++){
                //goes through every cost at the cost level
                for(int t = 0; t < unitedCostNames.size(); t++){
                    if(unitedCostNames[t]==costLevels[i][k]){
                        //add the cost to the intvarargs
                        sm[idx] = unitedCosts[t];
                        idx++;
                    }
                }
            }
            //if there is at least one cost at this level
            if(idx>0){
                //then sum all the intvarargs of this level together and add them to the
    orderedFactors
                rel(*this, orderedFactors[n_unique_costs], IRT_EQ, expr(*this, sum(sm)));
                //increase the amount of unique costs
                n_unique_costs++;
            }
        }
    }

    finalCosts = IntVarArray(*this, n_unique_costs, 0, 1000000);
    for(int i = 0; i < n_unique_costs; i++){
        //adds the cost/costs to the finalCosts list, which will be the one that will be
    minimized
        //(this eliminates all the <not assigned> values of the intvararray for costs which are
    not set)
        rel(*this, finalCosts[i], IRT_EQ, orderedFactors[i]);
    }

    // Adapte l'objectif (la cible de minimisation BAB) selon le mode demand .
    // objectiveCostSum est automatiquement ajout  en premier   lment   si objectiveMode != 0
    if (objectiveMode == OBJECTIVE_PONDERED) {
        int nCosts = finalCosts.size();
        IntArgs coeffs(nCosts);
        for (int i = 0; i < nCosts; ++i) coeffs[i] = 100 - i*80/nCosts; // First has weight 100,
     last has weight ~20

        linear(*this, coeffs, finalCosts, IRT_EQ, objectiveCostSum);
    }
    else if (objectiveMode == OBJECTIVE_MIXED) {
        IntVarArgs lexArgs(n_unique_costs);
        IntArgs weights(n_unique_costs);
        for(int i = 0; i < n_unique_costs; i++){
            lexArgs[i] = orderedFactors[i];
            weights[i] = n_unique_costs - i;
        }
        IntVar lexScore(*this, 0, 2000000);
        linear(*this, weights, lexArgs, IRT_EQ, lexScore);

        IntVar mixCost(*this, 0, 4000000);
        IntArgs mixWeights(2);
        mixWeights[0] = 1;
        mixWeights[1] = 1;
        IntVarArgs mixVars(2);
        mixVars[0] = globalCost;
        mixVars[1] = lexScore;
```

```
271         linear(*this, mixWeights, mixVars, IRT_EQ, objectiveCostSum);
272     }
273     else { // objectiveMode == OBJECTIVE_TOTAL (or OBJECTIVE_LEX, for consistency)
274         rel(*this, objectiveCostSum, IRT_EQ, globalCost);
275     }
276 
277     rel(*this, globalCost, IRT_EQ, expr(*this, sum(finalCosts)));
278 }
279 
280 
281 void CounterpointProblem::setStrata(){
282     // Determine number of voices
283     int nVoices = 2;
284     if(upper_2 != nullptr) nVoices++;
285     if(upper_3 != nullptr) nVoices++;
286 
287     int size = counterpoint_1->getNMeasures();
288 
289     // Initialize measure_orders for each measure
290     sorted_voices = {};
291     measures_order = {};
292 
293     // Process each measure
294     for(int i = 0; i < size; i++){
295         // For each measure, create voices array with the actual notes at that measure
296         IntVarArray voices = IntVarArray(*this, nVoices, 0, 127);
297 
298         // Set cantus firmus note
299         rel(*this, voices[0], IRT_EQ, cantusFirmus->getNotes()[i]);
300 
301         // Set counterpoint notes based on species-specific rules for this measure
302         setVoiceNote(voices, 1, counterpoint_1, i);
303         if(nVoices >= 3) setVoiceNote(voices, 2, counterpoint_2, i);
304         if(nVoices >= 4) setVoiceNote(voices, 3, counterpoint_3, i);
305 
306         // Sort voices and create order array for this specific measure
307         measures_order.push_back(IntVarArray(*this, nVoices, 0, nVoices-1));
308         sorted_voices.push_back(IntVarArray(*this, nVoices, 0, 127));
309         sorted(*this, voices, sorted_voices[i], measures_order[i]);
310 
311         // ---- Tie-breaker: if two voices have same pitch, enforce deterministic order
312         for (int a = 0; a < nVoices; ++a) {
313             for (int b = a + 1; b < nVoices; ++b) {
314                 BoolVar eq(*this, 0, 1);
315                 rel(*this, voices[a], IRT_EQ, voices[b], Reify(eq, RM_EQV));
316                 // If equal pitch, force order[a] < order[b] (priority: smaller index wins)
317                 rel(*this, eq >> (measures_order[i][a] < measures_order[i][b]));
318             }
319         }
320 
321         // Set stratum constraints for each position in the measure
322         int maxPos = (i == size-1) ? 1 : 4;  // Only set first position for last measure
323         for(int pos = 0; pos < maxPos; pos++){
324             // Set stratum constraints using the position-specific ordering
325             setStrataAtPosition(i, pos, nVoices, measures_order[i]);
326         }
327 
328         // Set voice flags and melodic interval constraints (only once per measure)
329         setVoiceLowestFlags(i, nVoices, size);
330         if(i > 0) setMelodicIntervalConstraints(i, nVoices);
331     }
332 }
333 
334 // Helper function implementations for setStrata
335 
336 void CounterpointProblem::setVoiceNote(IntVarArray& voices, int voiceIndex, Part* part, int
    measureIndex) {
337     // Set the appropriate note for a voice at a specific position within the measure
338     if ((part->getSpecies() == FOURTH_SPECIES || part->getSpecies() == FIFTH_SPECIES) &&
    measureIndex == 0) {
339         // First measure: use resolution note (index 2) for fourth/fifth species
340         rel(*this, voices[voiceIndex], IRT_EQ, part->getNotes()[(measureIndex*4)+2]);
341     } else if (part->getSpecies() == FOURTH_SPECIES && measureIndex != 0 && measureIndex !=
    nMeasures-1) {
342         // Middle measures of fourth species: choose between thesis and arsis based on joined
    degree
343         BoolVar isJoinedDegree(*this, 0, 1);
344         rel(*this, expr(*this, abs(part->getNotes()[measureIndex*4+2] - part->getNotes()[
    measureIndex*4])), IRT_LQ, 2, Reify(isJoinedDegree));
345         rel(*this, (isJoinedDegree == 0) >> (voices[voiceIndex] == part->getNotes()[measureIndex
    *4]));      // thesis
346         rel(*this, (isJoinedDegree == 1) >> (voices[voiceIndex] == part->getNotes()[(
    measureIndex*4)+2])); // arsis
347     } else if (part->getSpecies() == FIFTH_SPECIES && measureIndex != 0 && measureIndex !=
    nMeasures-1) {
348         // Fifth species: use species at this specific position
349         IntVar currentSpecies(*this, -1, 4);
350         rel(*this, currentSpecies, IRT_EQ, part->getSpeciesArray()[measureIndex*4]);
351         // Fourth species logic for fifth species
352         BoolVar isJoinedDegree(*this, 0, 1);
```

```
353         rel(*this, expr(*this, abs(part->getNotes()[measureIndex*4+2] - part->getNotes()[
    measureIndex*4])), IRT_LQ, 2, Reify(isJoinedDegree));
354         rel(*this, (currentSpecies == FOURTH_SPECIES && isJoinedDegree == 0) >> (voices[
    voiceIndex] == part->getNotes()[measureIndex*4]));
355         rel(*this, (currentSpecies == FOURTH_SPECIES && isJoinedDegree == 1) >> (voices[
    voiceIndex] == part->getNotes()[measureIndex*4+2]));
356         // Third species logic for fifth species
357         rel(*this, (currentSpecies != FOURTH_SPECIES) >> (voices[voiceIndex] == part->getNotes()
    [measureIndex*4]));
358     } else {
359         // Default case: use first note of the measure
360         rel(*this, voices[voiceIndex], IRT_EQ, part->getFirstNotes()[measureIndex]);
361     }
362 }
363
364 void CounterpointProblem::setStrataAtPosition(int measureIndex, int position, int nVoices,
    IntVarArray& order) {
365     // The order array tells us: order[voice_index] = position_in_sorted_order
366     // We need to find which voice is at each sorted position (0=lowest, 1=middle, etc.)
367
368     // For the first beat of each measure, we already computed the note that really matter for
    each stratum
369     if (position == 0) {
370         rel(*this, lowest->getNotes()[measureIndex*4+position], IRT_EQ, sorted_voices[
    measureIndex][0]);
371         rel(*this, upper_1->getNotes()[measureIndex*4+position], IRT_EQ, sorted_voices[
    measureIndex][1]);
372         if(nVoices >= 3) rel(*this, upper_2->getNotes()[measureIndex*4+position], IRT_EQ,
    sorted_voices[measureIndex][2]);
373         if(nVoices >= 4) rel(*this, upper_3->getNotes()[measureIndex*4+position], IRT_EQ,
    sorted_voices[measureIndex][3]);
374
375     } else {
376         // Lowest stratum
377         BoolVar cfIsLowest(*this, 0, 1);
378         BoolVar cp1IsLowest(*this, 0, 1);
379         BoolVar cp2IsLowest(*this, 0, 1);
380         BoolVar cp3IsLowest(*this, 0, 1);
381
382         rel(*this, order[0], IRT_EQ, 0, Reify(cfIsLowest));      // cantus firmus is at position
     0 (lowest)
383         rel(*this, order[1], IRT_EQ, 0, Reify(cp1IsLowest));     // counterpoint_1 is at
    position 0 (lowest)
384         if(nVoices >= 3) rel(*this, order[2], IRT_EQ, 0, Reify(cp2IsLowest)); // counterpoint_2
    is at position 0 (lowest)
385         if(nVoices >= 4) rel(*this, order[3], IRT_EQ, 0, Reify(cp3IsLowest)); // counterpoint_3
    is at position 0 (lowest)
386
387         // For cantus firmus
388         rel(*this, cfIsLowest >> (lowest->getNotes()[measureIndex*4+position] == cantusFirmus->
    getNotes()[measureIndex]));
389
390         // For counterpoint_1
391         if (counterpoint_1->getSpecies() == FIRST_SPECIES) {
392             rel(*this, cp1IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getFirstNotes()[measureIndex]));
393         } else if (counterpoint_1->getSpecies() == SECOND_SPECIES) {
394             int noteIndex = (position < 2) ? 0 : 2;  // pos 0,1 -> 0; pos 2,3 -> 2
395             rel(*this, cp1IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4+noteIndex]));
396         } else if (counterpoint_1->getSpecies() == THIRD_SPECIES) {
397             rel(*this, cp1IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4+position]));
398         } else if ((counterpoint_1->getSpecies() == FOURTH_SPECIES || counterpoint_1->getSpecies
    () == FIFTH_SPECIES) && measureIndex == 0) {
399             // First measure: use arsis note (index 2) for fourth/fifth species
400             rel(*this, cp1IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4+2]));
401         } else if (counterpoint_1->getSpecies() == FOURTH_SPECIES && measureIndex != 0 &&
    measureIndex != nMeasures-1) {
402             if (position == 2) {
403                 rel(*this, cp1IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4+position]));
404             } else {
405                 rel(*this, cp1IsLowest >> (lowest->getNotes()[measureIndex*4+position] == lowest
    ->getNotes()[measureIndex*4+position-1]));
406             }
407         } else if (counterpoint_1->getSpecies() == FIFTH_SPECIES && measureIndex != 0 &&
    measureIndex != nMeasures-1) {
408             if (position == 2) {
409                 rel(*this, cp1IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4+position]));
410             } else {
411                  IntVar currentSpecies(*this, -1, 4);
412                 rel(*this, currentSpecies, IRT_EQ, counterpoint_1->getSpeciesArray()[
    measureIndex*4+position]);
413                 rel(*this, (cp1IsLowest && currentSpecies == THIRD_SPECIES) >> (lowest->getNotes
    ()[measureIndex*4+position] == counterpoint_1->getNotes()[measureIndex*4+position]));
414                 rel(*this, (cp1IsLowest && currentSpecies != THIRD_SPECIES) >> (lowest->getNotes
    ()[measureIndex*4+position] == lowest->getNotes()[measureIndex*4+position-1]));
```

```
415             }
416         } else {
417             // Default case: use first note of the measure
418             rel(*this, cp1IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4]));
419         }
420 
421         if(nVoices >= 3) {
422             // For counterpoint_2
423             if (counterpoint_2->getSpecies() == FIRST_SPECIES) {
424                 rel(*this, cp2IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getFirstNotes()[measureIndex]));
425             } else if (counterpoint_2->getSpecies() == SECOND_SPECIES) {
426                 int noteIndex = (position < 2) ? 0 : 2;
427                 rel(*this, cp2IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4+noteIndex]));
428             } else if (counterpoint_2->getSpecies() == THIRD_SPECIES) {
429                 rel(*this, cp2IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4+position]));
430             } else if ((counterpoint_2->getSpecies() == FOURTH_SPECIES || counterpoint_2->
    getSpecies() == FIFTH_SPECIES) && measureIndex == 0) {
431                 rel(*this, cp2IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4+2]));
432             } else if (counterpoint_2->getSpecies() == FOURTH_SPECIES && measureIndex != 0 &&
    measureIndex != nMeasures-1) {
433                 if (position == 2) {
434                     rel(*this, cp2IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4+position]));
435                 } else {
436                     rel(*this, cp2IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    lowest->getNotes()[measureIndex*4+position-1]));
437                 }
438             } else if (counterpoint_2->getSpecies() == FIFTH_SPECIES && measureIndex != 0 &&
    measureIndex != nMeasures-1) {
439                 if (position == 2) {
440                     rel(*this, cp2IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4+position]));
441                 } else {
442                     IntVar currentSpecies(*this, -1, 4);
443                     rel(*this, currentSpecies, IRT_EQ, counterpoint_2->getSpeciesArray()[
    measureIndex*4+position]);
444                     rel(*this, (cp2IsLowest && currentSpecies == THIRD_SPECIES) >> (lowest->
    getNotes()[measureIndex*4+position] == counterpoint_2->getNotes()[measureIndex*4+position]))
    ;
445                     rel(*this, (cp2IsLowest && currentSpecies != THIRD_SPECIES) >> (lowest->
    getNotes()[measureIndex*4+position] == lowest->getNotes()[measureIndex*4+position-1]));
446                 }
447             } else {
448                 rel(*this, cp2IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4]));
449             }
450         }
451 
452         if(nVoices >= 4) {
453             // For counterpoint_3
454             if (counterpoint_3->getSpecies() == FIRST_SPECIES) {
455                 rel(*this, cp3IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getFirstNotes()[measureIndex]));
456             } else if (counterpoint_3->getSpecies() == SECOND_SPECIES) {
457                 int noteIndex = (position < 2) ? 0 : 2;
458                 rel(*this, cp3IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4+noteIndex]));
459             } else if (counterpoint_3->getSpecies() == THIRD_SPECIES) {
460                 rel(*this, cp3IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4+position]));
461             } else if ((counterpoint_3->getSpecies() == FOURTH_SPECIES || counterpoint_3->
    getSpecies() == FIFTH_SPECIES) && measureIndex == 0) {
462                 rel(*this, cp3IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4+2]));
463             } else if (counterpoint_3->getSpecies() == FOURTH_SPECIES && measureIndex != 0 &&
    measureIndex != nMeasures-1) {
464                 if (position == 2) {
465                     rel(*this, cp3IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4+position]));
466                 } else {
467                     rel(*this, cp3IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    lowest->getNotes()[measureIndex*4+position-1]));
468                 }
469             } else if (counterpoint_3->getSpecies() == FIFTH_SPECIES && measureIndex != 0 &&
    measureIndex != nMeasures-1) {
470                 if (position == 2) {
471                     rel(*this, cp3IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4+position]));
472                 } else {
473                     IntVar currentSpecies(*this, -1, 4);
474                     rel(*this, currentSpecies, IRT_EQ, counterpoint_3->getSpeciesArray()[
    measureIndex*4+position]);
475                     rel(*this, (cp3IsLowest && currentSpecies == THIRD_SPECIES) >> (lowest->
    getNotes()[measureIndex*4+position] == counterpoint_3->getNotes()[measureIndex*4+position]))
    ;
```

```
476                     rel(*this, (cp3IsLowest && currentSpecies != THIRD_SPECIES) >> (lowest->
    getNotes()[measureIndex*4+position] == lowest->getNotes()[measureIndex*4+position-1]));
477                 }
478             } else {
479                 rel(*this, cp3IsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4]));
480             }
481         }
482 
483         // Set upper strata notes based on voice mapping
484         // For upper_1 (sorted position 1)
485         BoolVar cfIsSecond(*this, 0, 1);
486         BoolVar cp1IsSecond(*this, 0, 1);
487         BoolVar cp2IsSecond(*this, 0, 1);
488         BoolVar cp3IsSecond(*this, 0, 1);
489 
490         rel(*this, order[0], IRT_EQ, 1, Reify(cfIsSecond));      // cantus firmus is at position
     1 (second)
491         rel(*this, order[1], IRT_EQ, 1, Reify(cp1IsSecond));     // counterpoint_1 is at
    position 1 (second)
492         if(nVoices >= 3) rel(*this, order[2], IRT_EQ, 1, Reify(cp2IsSecond)); // counterpoint_2
    is at position 1 (second)
493         if(nVoices >= 4) rel(*this, order[3], IRT_EQ, 1, Reify(cp3IsSecond)); // counterpoint_3
    is at position 1 (second)
494 
495         // CF (whole note)
496         rel(*this, cfIsSecond >> (upper_1->getNotes()[measureIndex*4+position] == cantusFirmus->
    getNotes()[measureIndex]));
497 
498         // cp1: mirror lowest logic
499         if (counterpoint_1->getSpecies() == FIRST_SPECIES) {
500             rel(*this, cp1IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getFirstNotes()[measureIndex]));
501         } else if (counterpoint_1->getSpecies() == SECOND_SPECIES) {
502             int noteIndex = (position < 2) ? 0 : 2;
503             rel(*this, cp1IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4+noteIndex]));
504         } else if (counterpoint_1->getSpecies() == THIRD_SPECIES) {
505             rel(*this, cp1IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4+position]));
506         } else if ((counterpoint_1->getSpecies() == FOURTH_SPECIES || counterpoint_1->getSpecies
    () == FIFTH_SPECIES) && measureIndex == 0) {
507             rel(*this, cp1IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4+2]));
508         } else if (counterpoint_1->getSpecies() == FOURTH_SPECIES && measureIndex != 0 &&
    measureIndex != nMeasures-1) {
509             if (position == 2) {
510                 rel(*this, cp1IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4+position]));
511             } else {
512                 rel(*this, cp1IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    upper_1->getNotes()[measureIndex*4+position-1]));
513             }
514         } else if (counterpoint_1->getSpecies() == FIFTH_SPECIES && measureIndex != 0 &&
    measureIndex != nMeasures-1) {
515             if (position == 2) {
516                 rel(*this, cp1IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4+position]));
517             } else {
518                 IntVar currentSpecies(*this, -1, 4);
519                 rel(*this, currentSpecies, IRT_EQ, counterpoint_1->getSpeciesArray()[
    measureIndex*4+position]);
520                 rel(*this, (cp1IsSecond && currentSpecies == THIRD_SPECIES) >> (upper_1->
    getNotes()[measureIndex*4+position] == counterpoint_1->getNotes()[measureIndex*4+position]))
    ;
521                 rel(*this, (cp1IsSecond && currentSpecies != THIRD_SPECIES) >> (upper_1->
    getNotes()[measureIndex*4+position] == upper_1->getNotes()[measureIndex*4+position-1]));
522             }
523         } else {
524             rel(*this, cp1IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4]));
525         }
526 
527         if(nVoices >= 3) {
528             // cp2: mirror lowest logic
529             if (counterpoint_2->getSpecies() == FIRST_SPECIES) {
530                 rel(*this, cp2IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getFirstNotes()[measureIndex]));
531             } else if (counterpoint_2->getSpecies() == SECOND_SPECIES) {
532                 int noteIndex = (position < 2) ? 0 : 2;
533                 rel(*this, cp2IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4+noteIndex]));
534             } else if (counterpoint_2->getSpecies() == THIRD_SPECIES) {
535                 rel(*this, cp2IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4+position]));
536             } else if ((counterpoint_2->getSpecies() == FOURTH_SPECIES || counterpoint_2->
    getSpecies() == FIFTH_SPECIES) && measureIndex == 0) {
537                 rel(*this, cp2IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4+2]));
538             } else if (counterpoint_2->getSpecies() == FOURTH_SPECIES && measureIndex != 0 &&
```

```
    measureIndex != nMeasures-1) {
539             if (position == 2) {
540                 rel(*this, cp2IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4+position]));
541             } else {
542                 rel(*this, cp2IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    upper_1->getNotes()[measureIndex*4+position-1]));
543             }
544         } else if (counterpoint_2->getSpecies() == FIFTH_SPECIES && measureIndex != 0 &&
    measureIndex != nMeasures-1) {
545             if (position == 2) {
546                 rel(*this, cp2IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4+position]));
547             } else {
548                 IntVar currentSpecies(*this, -1, 4);
549                 rel(*this, currentSpecies, IRT_EQ, counterpoint_2->getSpeciesArray()[
    measureIndex*4+position]);
550                 rel(*this, (cp2IsSecond && currentSpecies == THIRD_SPECIES) >> (upper_1->
    getNotes()[measureIndex*4+position] == counterpoint_2->getNotes()[measureIndex*4+position]))
    ;
551                 rel(*this, (cp2IsSecond && currentSpecies != THIRD_SPECIES) >> (upper_1->
    getNotes()[measureIndex*4+position] == upper_1->getNotes()[measureIndex*4+position-1]));
552             }
553         } else {
554             rel(*this, cp2IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4]));
555         }
556     }
557
558     if(nVoices >= 4) {
559         // cp3: mirror lowest logic
560         if (counterpoint_3->getSpecies() == FIRST_SPECIES) {
561             rel(*this, cp3IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getFirstNotes()[measureIndex]));
562         } else if (counterpoint_3->getSpecies() == SECOND_SPECIES) {
563             int noteIndex = (position < 2) ? 0 : 2;
564             rel(*this, cp3IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4+noteIndex]));
565         } else if (counterpoint_3->getSpecies() == THIRD_SPECIES) {
566             rel(*this, cp3IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4+position]));
567         } else if ((counterpoint_3->getSpecies() == FOURTH_SPECIES || counterpoint_3->
    getSpecies() == FIFTH_SPECIES) && measureIndex == 0) {
568             rel(*this, cp3IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4+2]));
569         } else if (counterpoint_3->getSpecies() == FOURTH_SPECIES && measureIndex != 0 &&
    measureIndex != nMeasures-1) {
570             if (position == 2) {
571                 rel(*this, cp3IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4+position]));
572             } else {
573                 rel(*this, cp3IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    upper_1->getNotes()[measureIndex*4+position-1]));
574             }
575         } else if (counterpoint_3->getSpecies() == FIFTH_SPECIES && measureIndex != 0 &&
    measureIndex != nMeasures-1) {
576             if (position == 2) {
577                 rel(*this, cp3IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4+position]));
578             } else {
579                 IntVar currentSpecies(*this, -1, 4);
580                 rel(*this, currentSpecies, IRT_EQ, counterpoint_3->getSpeciesArray()[
    measureIndex*4+position]);
581                 rel(*this, (cp3IsSecond && currentSpecies == THIRD_SPECIES) >> (upper_1->
    getNotes()[measureIndex*4+position] == counterpoint_3->getNotes()[measureIndex*4+position]))
    ;
582                 rel(*this, (cp3IsSecond && currentSpecies != THIRD_SPECIES) >> (upper_1->
    getNotes()[measureIndex*4+position] == upper_1->getNotes()[measureIndex*4+position-1]));
583             }
584         } else {
585             rel(*this, cp3IsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4]));
586         }
587     }
588
589     // For upper_2 (sorted position 2)
590     if(nVoices >= 3) {
591         BoolVar cfIsThird(*this, 0, 1);
592         BoolVar cp1IsThird(*this, 0, 1);
593         BoolVar cp2IsThird(*this, 0, 1);
594         BoolVar cp3IsThird(*this, 0, 1);
595         rel(*this, order[0], IRT_EQ, 2, Reify(cfIsThird));
596         rel(*this, order[1], IRT_EQ, 2, Reify(cp1IsThird));
597         rel(*this, order[2], IRT_EQ, 2, Reify(cp2IsThird));
598         if(nVoices >= 4) rel(*this, order[3], IRT_EQ, 2, Reify(cp3IsThird));
599
600         rel(*this, cfIsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    cantusFirmus->getNotes()[measureIndex]));
601
602         // cp1: mirror lowest logic
```

```
603             if (counterpoint_1->getSpecies() == FIRST_SPECIES) {
604                 rel(*this, cp1IsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getFirstNotes()[measureIndex]));
605             } else if (counterpoint_1->getSpecies() == SECOND_SPECIES) {
606                 int noteIndex = (position < 2) ? 0 : 2;
607                 rel(*this, cp1IsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4+noteIndex]));
608             } else if (counterpoint_1->getSpecies() == THIRD_SPECIES) {
609                 rel(*this, cp1IsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4+position]));
610             } else if ((counterpoint_1->getSpecies() == FOURTH_SPECIES || counterpoint_1->
    getSpecies() == FIFTH_SPECIES) && measureIndex == 0) {
611                 rel(*this, cp1IsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4+2]));
612             } else if (counterpoint_1->getSpecies() == FOURTH_SPECIES && measureIndex != 0 &&
    measureIndex != nMeasures-1) {
613                 if (position == 2) {
614                     rel(*this, cp1IsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4+position]));
615                 } else {
616                     rel(*this, cp1IsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    upper_2->getNotes()[measureIndex*4+position-1]));
617                 }
618             } else if (counterpoint_1->getSpecies() == FIFTH_SPECIES && measureIndex != 0 &&
    measureIndex != nMeasures-1) {
619                 if (position == 2) {
620                     rel(*this, cp1IsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4+position]));
621                 } else {
622                     IntVar currentSpecies(*this, -1, 4);
623                     rel(*this, currentSpecies, IRT_EQ, counterpoint_1->getSpeciesArray()[
    measureIndex*4+position]);
624                     rel(*this, (cp1IsThird && currentSpecies == THIRD_SPECIES) >> (upper_2->
    getNotes()[measureIndex*4+position] == counterpoint_1->getNotes()[measureIndex*4+position]))
    ;
625                     rel(*this, (cp1IsThird && currentSpecies != THIRD_SPECIES) >> (upper_2->
    getNotes()[measureIndex*4+position] == upper_2->getNotes()[measureIndex*4+position-1]));
626                 }
627             } else {
628                 rel(*this, cp1IsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4]));
629             }
630 
631             // cp2: mirror lowest logic
632             if (counterpoint_2->getSpecies() == FIRST_SPECIES) {
633                 rel(*this, cp2IsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getFirstNotes()[measureIndex]));
634             } else if (counterpoint_2->getSpecies() == SECOND_SPECIES) {
635                 int noteIndex = (position < 2) ? 0 : 2;
636                 rel(*this, cp2IsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4+noteIndex]));
637             } else if (counterpoint_2->getSpecies() == THIRD_SPECIES) {
638                 rel(*this, cp2IsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4+position]));
639             } else if ((counterpoint_2->getSpecies() == FOURTH_SPECIES || counterpoint_2->
    getSpecies() == FIFTH_SPECIES) && measureIndex == 0) {
640                 rel(*this, cp2IsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4+2]));
641             } else if (counterpoint_2->getSpecies() == FOURTH_SPECIES && measureIndex != 0 &&
    measureIndex != nMeasures-1) {
642                 if (position == 2) {
643                     rel(*this, cp2IsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4+position]));
644                 } else {
645                     rel(*this, cp2IsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    upper_2->getNotes()[measureIndex*4+position-1]));
646                 }
647             } else if (counterpoint_2->getSpecies() == FIFTH_SPECIES && measureIndex != 0 &&
    measureIndex != nMeasures-1) {
648                 if (position == 2) {
649                     rel(*this, cp2IsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4+position]));
650                 } else {
651                     IntVar currentSpecies(*this, -1, 4);
652                     rel(*this, currentSpecies, IRT_EQ, counterpoint_2->getSpeciesArray()[
    measureIndex*4+position]);
653                     rel(*this, (cp2IsThird && currentSpecies == THIRD_SPECIES) >> (upper_2->
    getNotes()[measureIndex*4+position] == counterpoint_2->getNotes()[measureIndex*4+position]))
    ;
654                     rel(*this, (cp2IsThird && currentSpecies != THIRD_SPECIES) >> (upper_2->
    getNotes()[measureIndex*4+position] == upper_2->getNotes()[measureIndex*4+position-1]));
655                 }
656             } else {
657                 rel(*this, cp2IsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4]));
658             }
659 
660             if(nVoices >= 4) {
661                 // cp3 may also be third in 4 voices: mirror lowest logic
662                 if (counterpoint_3->getSpecies() == FIRST_SPECIES) {
```

```
663                     rel(*this, cp3IsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getFirstNotes()[measureIndex]));
664                 } else if (counterpoint_3->getSpecies() == SECOND_SPECIES) {
665                     int noteIndex = (position < 2) ? 0 : 2;
666                     rel(*this, cp3IsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4+noteIndex]));
667                 } else if (counterpoint_3->getSpecies() == THIRD_SPECIES) {
668                     rel(*this, cp3IsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4+position]));
669                 } else if ((counterpoint_3->getSpecies() == FOURTH_SPECIES || counterpoint_3->
    getSpecies() == FIFTH_SPECIES) && measureIndex == 0) {
670                     rel(*this, cp3IsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4+2]));
671                 } else if (counterpoint_3->getSpecies() == FOURTH_SPECIES && measureIndex != 0
    && measureIndex != nMeasures-1) {
672                     if (position == 2) {
673                         rel(*this, cp3IsThird >> (upper_2->getNotes()[measureIndex*4+position]
    == counterpoint_3->getNotes()[measureIndex*4+position]));
674                     } else {
675                         rel(*this, cp3IsThird >> (upper_2->getNotes()[measureIndex*4+position]
    == upper_2->getNotes()[measureIndex*4+position-1]));
676                     }
677                 } else if (counterpoint_3->getSpecies() == FIFTH_SPECIES && measureIndex != 0 &&
     measureIndex != nMeasures-1) {
678                     if (position == 2) {
679                         rel(*this, cp3IsThird >> (upper_2->getNotes()[measureIndex*4+position]
    == counterpoint_3->getNotes()[measureIndex*4+position]));
680                     } else {
681                         IntVar currentSpecies(*this, -1, 4);
682                         rel(*this, currentSpecies, IRT_EQ, counterpoint_3->getSpeciesArray()[
    measureIndex*4+position]);
683                         rel(*this, (cp3IsThird && currentSpecies == THIRD_SPECIES) >> (upper_2->
    getNotes()[measureIndex*4+position] == counterpoint_3->getNotes()[measureIndex*4+position]))
    ;
684                         rel(*this, (cp3IsThird && currentSpecies != THIRD_SPECIES) >> (upper_2->
    getNotes()[measureIndex*4+position] == upper_2->getNotes()[measureIndex*4+position-1]));
685                     }
686                 } else {
687                     rel(*this, cp3IsThird >> (upper_2->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4]));
688                 }
689             }
690         }
691 
692         // For upper_3 (sorted position 3)
693         if(nVoices >= 4) {
694             BoolVar cfIsFourth(*this, 0, 1);
695             BoolVar cp1IsFourth(*this, 0, 1);
696             BoolVar cp2IsFourth(*this, 0, 1);
697             BoolVar cp3IsFourth(*this, 0, 1);
698 
699             rel(*this, order[0], IRT_EQ, 3, Reify(cfIsFourth));
700             rel(*this, order[1], IRT_EQ, 3, Reify(cp1IsFourth));
701             rel(*this, order[2], IRT_EQ, 3, Reify(cp2IsFourth));
702             rel(*this, order[3], IRT_EQ, 3, Reify(cp3IsFourth));
703 
704             rel(*this, cfIsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    cantusFirmus->getNotes()[measureIndex]));
705 
706             // cp1: mirror lowest logic
707             if (counterpoint_1->getSpecies() == FIRST_SPECIES) {
708                 rel(*this, cp1IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getFirstNotes()[measureIndex]));
709             } else if (counterpoint_1->getSpecies() == SECOND_SPECIES) {
710                 int noteIndex = (position < 2) ? 0 : 2;
711                 rel(*this, cp1IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4+noteIndex]));
712             } else if (counterpoint_1->getSpecies() == THIRD_SPECIES) {
713                 rel(*this, cp1IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4+position]));
714             } else if ((counterpoint_1->getSpecies() == FOURTH_SPECIES || counterpoint_1->
    getSpecies() == FIFTH_SPECIES) && measureIndex == 0) {
715                 rel(*this, cp1IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4+2]));
716             } else if (counterpoint_1->getSpecies() == FOURTH_SPECIES && measureIndex != 0 &&
    measureIndex != nMeasures-1) {
717                 if (position == 2) {
718                     rel(*this, cp1IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4+position]));
719                 } else {
720                     rel(*this, cp1IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    upper_3->getNotes()[measureIndex*4+position-1]));
721                 }
722             } else if (counterpoint_1->getSpecies() == FIFTH_SPECIES && measureIndex != 0 &&
    measureIndex != nMeasures-1) {
723                 if (position == 2) {
724                     rel(*this, cp1IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4+position]));
725                 } else {
726                     IntVar currentSpecies(*this, -1, 4);
```

```
727                     rel(*this, currentSpecies, IRT_EQ, counterpoint_1->getSpeciesArray()[
    measureIndex*4+position]);
728                     rel(*this, (cp1IsFourth && currentSpecies == THIRD_SPECIES) >> (upper_3->
    getNotes()[measureIndex*4+position] == counterpoint_1->getNotes()[measureIndex*4+position]))
    ;
729                     rel(*this, (cp1IsFourth && currentSpecies != THIRD_SPECIES) >> (upper_3->
    getNotes()[measureIndex*4+position] == upper_3->getNotes()[measureIndex*4+position-1]));
730                 }
731             } else {
732                 rel(*this, cp1IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    counterpoint_1->getNotes()[measureIndex*4]));
733             }
734 
735             // cp2: mirror lowest logic
736             if (counterpoint_2->getSpecies() == FIRST_SPECIES) {
737                 rel(*this, cp2IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getFirstNotes()[measureIndex]));
738             } else if (counterpoint_2->getSpecies() == SECOND_SPECIES) {
739                 int noteIndex = (position < 2) ? 0 : 2;
740                 rel(*this, cp2IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4+noteIndex]));
741             } else if (counterpoint_2->getSpecies() == THIRD_SPECIES) {
742                 rel(*this, cp2IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4+position]));
743             } else if ((counterpoint_2->getSpecies() == FOURTH_SPECIES || counterpoint_2->
    getSpecies() == FIFTH_SPECIES) && measureIndex == 0) {
744                 rel(*this, cp2IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4+2]));
745             } else if (counterpoint_2->getSpecies() == FOURTH_SPECIES && measureIndex != 0 &&
    measureIndex != nMeasures-1) {
746                 if (position == 2) {
747                     rel(*this, cp2IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4+position]));
748                 } else {
749                     rel(*this, cp2IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    upper_3->getNotes()[measureIndex*4+position-1]));
750                 }
751             } else if (counterpoint_2->getSpecies() == FIFTH_SPECIES && measureIndex != 0 &&
    measureIndex != nMeasures-1) {
752                 if (position == 2) {
753                     rel(*this, cp2IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4+position]));
754                 } else {
755                     IntVar currentSpecies(*this, -1, 4);
756                     rel(*this, currentSpecies, IRT_EQ, counterpoint_2->getSpeciesArray()[
    measureIndex*4+position]);
757                     rel(*this, (cp2IsFourth && currentSpecies == THIRD_SPECIES) >> (upper_3->
    getNotes()[measureIndex*4+position] == counterpoint_2->getNotes()[measureIndex*4+position]))
    ;
758                     rel(*this, (cp2IsFourth && currentSpecies != THIRD_SPECIES) >> (upper_3->
    getNotes()[measureIndex*4+position] == upper_3->getNotes()[measureIndex*4+position-1]));
759                 }
760             } else {
761                 rel(*this, cp2IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    counterpoint_2->getNotes()[measureIndex*4]));
762             }
763 
764             // cp3: mirror lowest logic
765             if (counterpoint_3->getSpecies() == FIRST_SPECIES) {
766                 rel(*this, cp3IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getFirstNotes()[measureIndex]));
767             } else if (counterpoint_3->getSpecies() == SECOND_SPECIES) {
768                 int noteIndex = (position < 2) ? 0 : 2;
769                 rel(*this, cp3IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4+noteIndex]));
770             } else if (counterpoint_3->getSpecies() == THIRD_SPECIES) {
771                 rel(*this, cp3IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4+position]));
772             } else if ((counterpoint_3->getSpecies() == FOURTH_SPECIES || counterpoint_3->
    getSpecies() == FIFTH_SPECIES) && measureIndex == 0) {
773                 rel(*this, cp3IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4+2]));
774             } else if (counterpoint_3->getSpecies() == FOURTH_SPECIES && measureIndex != 0 &&
    measureIndex != nMeasures-1) {
775                 if (position == 2) {
776                     rel(*this, cp3IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4+position]));
777                 } else {
778                     rel(*this, cp3IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    upper_3->getNotes()[measureIndex*4+position-1]));
779                 }
780             } else if (counterpoint_3->getSpecies() == FIFTH_SPECIES && measureIndex != 0 &&
    measureIndex != nMeasures-1) {
781                 if (position == 2) {
782                     rel(*this, cp3IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4+position]));
783                 } else {
784                     IntVar currentSpecies(*this, -1, 4);
785                     rel(*this, currentSpecies, IRT_EQ, counterpoint_3->getSpeciesArray()[
    measureIndex*4+position]);
```

```
786                     rel(*this, (cp3IsFourth && currentSpecies == THIRD_SPECIES) >> (upper_3->
    getNotes()[measureIndex*4+position] == counterpoint_3->getNotes()[measureIndex*4+position]))
    ;
787                     rel(*this, (cp3IsFourth && currentSpecies != THIRD_SPECIES) >> (upper_3->
    getNotes()[measureIndex*4+position] == upper_3->getNotes()[measureIndex*4+position-1]));
788                 }
789             } else {
790                 rel(*this, cp3IsFourth >> (upper_3->getNotes()[measureIndex*4+position] ==
    counterpoint_3->getNotes()[measureIndex*4]));
791             }
792         }
793     }
794 }
795
796 void CounterpointProblem::setVoiceLowestFlags(int measureIndex, int nVoices, int size) {
797     // Set the isNotLowest flags for each voice
798     rel(*this, lowest->getFirstNotes()[measureIndex], IRT_NQ, cantusFirmus->getNotes()[
    measureIndex],
799         Reify(cantusFirmus->getIsNotLowest()[measureIndex]));
800
801     if(nVoices >= 2) {
802         // Set isNotLowest boolean for counterpoint_1
803         if(counterpoint_1->getSpecies() == FOURTH_SPECIES && measureIndex != size-1) {
804             rel(*this, expr(*this, (cantusFirmus->getIsNotLowest()[measureIndex]==1) &&
805                 (lowest->getFirstNotes()[measureIndex]==counterpoint_1->getNotes()[(measureIndex
    *4)+2])),
806                 IRT_NQ, 1, Reify(counterpoint_1->getIsNotLowest()[measureIndex]));
807         } else if(counterpoint_1->getSpecies() == FIFTH_SPECIES && measureIndex == 0) {
808             rel(*this, expr(*this, (cantusFirmus->getIsNotLowest()[measureIndex]==1) &&
809                 (lowest->getFirstNotes()[measureIndex]==counterpoint_1->getNotes()[(measureIndex
    *4)+2])),
810                 IRT_NQ, 1, Reify(counterpoint_1->getIsNotLowest()[measureIndex]));
811         } else {
812             rel(*this, expr(*this, (cantusFirmus->getIsNotLowest()[measureIndex]==1) &&
813                 (lowest->getFirstNotes()[measureIndex]==counterpoint_1->getFirstNotes()[
    measureIndex])),
814                 IRT_NQ, 1, Reify(counterpoint_1->getIsNotLowest()[measureIndex]));
815         }
816     }
817
818     if(nVoices == 3) {
819         rel(*this, expr(*this, counterpoint_1->getIsNotLowest()[measureIndex] != cantusFirmus->
    getIsNotLowest()[measureIndex]),
820             IRT_EQ, counterpoint_2->getIsNotLowest()[measureIndex]);
821     }
822
823     if(nVoices == 4) {
824         // Set counterpoint_2 lowest boolean
825         if(counterpoint_2->getSpecies() == FOURTH_SPECIES && measureIndex != size-1) {
826             rel(*this, expr(*this, (cantusFirmus->getIsNotLowest()[measureIndex]==1) &&
827                 (counterpoint_1->getIsNotLowest()[measureIndex]==1) &&
828                 (lowest->getFirstNotes()[measureIndex]==counterpoint_2->getNotes()[(measureIndex
    *4)+2])),
829                 IRT_NQ, 1, Reify(counterpoint_2->getIsNotLowest()[measureIndex]));
830         } else if(counterpoint_2->getSpecies() == FIFTH_SPECIES && measureIndex == 0) {
831             rel(*this, expr(*this, (cantusFirmus->getIsNotLowest()[measureIndex]==1) &&
832                 (counterpoint_1->getIsNotLowest()[measureIndex]==1) &&
833                 (lowest->getFirstNotes()[measureIndex]==counterpoint_2->getNotes()[(measureIndex
    *4)+2])),
834                 IRT_NQ, 1, Reify(counterpoint_2->getIsNotLowest()[measureIndex]));
835         } else {
836             rel(*this, expr(*this, (cantusFirmus->getIsNotLowest()[measureIndex]==1) &&
837                 (counterpoint_1->getIsNotLowest()[measureIndex]==1) &&
838                 (lowest->getFirstNotes()[measureIndex]==counterpoint_2->getFirstNotes()[
    measureIndex])),
839                 IRT_NQ, 1, Reify(counterpoint_2->getIsNotLowest()[measureIndex]));
840         }
841
842         rel(*this, expr(*this, (cantusFirmus->getIsNotLowest()[measureIndex]==1) &&
843             (counterpoint_1->getIsNotLowest()[measureIndex]==1) &&
844             (counterpoint_2->getIsNotLowest()[measureIndex]==1)),
845             IRT_NQ, counterpoint_3->getIsNotLowest()[measureIndex]);
846
847         // Set isHighest flags for 4-voice specific constraints
848         rel(*this, upper_3->getFirstNotes()[measureIndex], IRT_EQ, cantusFirmus->getNotes()[
    measureIndex],
849             Reify(cantusFirmus->getIsHighest()[measureIndex]));
850
851         // ... (similar logic for isHighest flags)
852     }
853 }
854
855 void CounterpointProblem::setMelodicIntervalConstraints(int measureIndex, int nVoices) {
856     // Set melodic interval constraints for the lowest stratum
857     vector<IntVarArray> corresponding_m_intervals;
858
859     corresponding_m_intervals.push_back(cantusFirmus->getMelodicIntervals());
860
861     for(int j = 0; j < nVoices-1; j++) {
862         Part* curr_cp;
```

```
863         if(j == 0) curr_cp = counterpoint_1;
864         else if(j == 1) curr_cp = counterpoint_2;
865         else curr_cp = counterpoint_3;
866 
867         if(curr_cp->getSpecies() == FIRST_SPECIES) {
868             corresponding_m_intervals.push_back(IntVarArray(*this, curr_cp->getMelodicIntervals
    ().slice(0, 4, curr_cp->getMelodicIntervals().size())));
869         } else if(curr_cp->getSpecies() == SECOND_SPECIES) {
870             corresponding_m_intervals.push_back(IntVarArray(*this, curr_cp->getMelodicIntervals
    ().slice(2, 4, curr_cp->getMelodicIntervals().size())));
871         } else if(curr_cp->getSpecies() == THIRD_SPECIES) {
872             corresponding_m_intervals.push_back(IntVarArray(*this, curr_cp->getMelodicIntervals
    ().slice(3, 4, curr_cp->getMelodicIntervals().size())));
873         } else if(curr_cp->getSpecies() == FOURTH_SPECIES) {
874             std::vector<int> selectedIndices;
875             for (int i = 4; i < curr_cp->getMelodicIntervals().size(); i += 4) {
876                 selectedIndices.push_back(i);
877             }
878             selectedIndices.push_back(curr_cp->getMelodicIntervals().size() - 2);
879 
880             IntVarArray selectedIntervals(*this, selectedIndices.size());
881             for (size_t i = 0; i < selectedIndices.size(); ++i) {
882                 selectedIntervals[i] = curr_cp->getMelodicIntervals()[selectedIndices[i]];
883             }
884             corresponding_m_intervals.push_back(selectedIntervals);
885         } else if(curr_cp->getSpecies() == FIFTH_SPECIES) {
886             corresponding_m_intervals.push_back(IntVarArray(*this, curr_cp->getMelodicIntervals
    ().slice(2, 4, curr_cp->getMelodicIntervals().size())));
887         }
888     }
889 
890     rel(*this, (cantusFirmus->getIsNotLowest()[measureIndex]==0) >>
891         (lowest->getMelodicIntervals()[measureIndex-1]==corresponding_m_intervals[0][
    measureIndex-1]));
892     rel(*this, (counterpoint_1->getIsNotLowest()[measureIndex]==0) >>
893         (lowest->getMelodicIntervals()[measureIndex-1]==corresponding_m_intervals[1][
    measureIndex-1]));
894     if(nVoices >= 3) {
895         rel(*this, (counterpoint_2->getIsNotLowest()[measureIndex]==0) >>
896             (lowest->getMelodicIntervals()[measureIndex-1]==corresponding_m_intervals[2][
    measureIndex-1]));
897     }
898     if(nVoices >= 4) {
899         rel(*this, (counterpoint_3->getIsNotLowest()[measureIndex]==0) >>
900             (lowest->getMelodicIntervals()[measureIndex-1]==corresponding_m_intervals[3][
    measureIndex-1]));
901     }
902 }
903 
904 IntVarArray CounterpointProblem::getSolutionArray(){
905     return solutionArray;
906 }
907 
908 /**
909  * Returns the size of the problem
910  * @return an integer representing the size of the vars array           RETURNS THE SIZE OF THE
    SOLUTION_ARRAY
911  */
912 int CounterpointProblem::getSize(){
913     string message = "getSize function called. size = " + std::to_string(solutionArray.size()) +
     "\n";
914     writeToLogFile(message.c_str());
915     return this->solutionArray.size();  /// have to use this-> ?
916 }
917 
918 /**
919  * Returns the values taken by the variables vars in a solution
920  * @todo Modify this to return the solution for your problem. This function uses @param size to
    generate an array of integers
921  * @return an array of integers representing the values of the variables in a solution
922  */
923 int* CounterpointProblem::return_solution(){
924     string message = "return_solution method. Solution : [";
925     int* solution = new int[solutionArray.size()];
926     for(int i = 0; i < solutionArray.size(); i++){
927         solution[i] = solutionArray[i].val();
928         message += std::to_string(solution[i]) + " ";
929     }
930     message += "]\n";
931     writeToLogFile(message.c_str());
932     return solution;
933 }
934 
935 
936 int* CounterpointProblem::get_species_array_5sp(int ctp_index){
937 
938     FifthSpeciesCounterpoint* fifth_sp_ctp;
939     if (ctp_index == 0) fifth_sp_ctp = dynamic_cast<FifthSpeciesCounterpoint*>(counterpoint_1);
940     else if (ctp_index == 1) fifth_sp_ctp = dynamic_cast<FifthSpeciesCounterpoint*>(
    counterpoint_2);
```

```
941     else if (ctp_index == 2) fifth_sp_ctp = dynamic_cast<FifthSpeciesCounterpoint*>(
        counterpoint_3);
942     else{
943         writeToLogFile("invalid value of ctp_index given as argument to get_species_array_5sp");
944         return nullptr;
945     }
946 
947     if (!fifth_sp_ctp){
948         cout << "type cast of part to fifth species failed " << endl;
949         writeToLogFile("type cast of part to fifth species failed");
950         return nullptr;
951     }
952 
953     IntVarArray speciesArray = fifth_sp_ctp->getSpeciesArray();
954 
955     string message = "return_species_array method. Species array : [";
956     int* solution = new int[speciesArray.size()];
957     for(int i = 0; i < speciesArray.size(); i++){
958 
959         switch (speciesArray[i].val())
960         {
961         case FIRST_SPECIES:
962             solution[i] = 1;
963             break;
964 
965         case SECOND_SPECIES:
966             solution[i] = 2;
967             break;
968 
969         case THIRD_SPECIES:
970             solution[i] = 3;
971             break;
972 
973         case FOURTH_SPECIES:
974             solution[i] = 4;
975             break;
976 
977         case -1:
978             solution[i] = 0;
979             break;
980 
981         default:
982             cout << "invalid value in species array" << endl;
983             writeToLogFile("invalid value in species array");
984             return nullptr;
985         }
986         message += std::to_string(solution[i]) + " ";
987     }
988     message += "]\n";
989     writeToLogFile(message.c_str());
990     return solution;
991 }
992 
993 
994 int* CounterpointProblem::get_extended_cp_domain(int ctp_index){
995     vector<int> ext_cp_dom;
996     if(ctp_index == 0) ext_cp_dom = counterpoint_1->getDomain();
997     else if(ctp_index == 1) ext_cp_dom = counterpoint_2->getDomain();
998     else if(ctp_index == 2) ext_cp_dom = counterpoint_3->getDomain();
999     else{
1000        writeToLogFile("invalid value of ctp_index given as argument to get_extended_cp_domain")
        ;
1001        return nullptr;
1002    }
1003    writeToLogFile(int_vector_to_string(ext_cp_dom).c_str());
1004
1005    int* ext_cp_dom_int_ptr = new int[ext_cp_dom.size()];
1006    for(int i = 0; i < ext_cp_dom.size(); i++){
1007        ext_cp_dom_int_ptr[i] = ext_cp_dom[i];
1008        // message += std::to_string(solution[i]) + " ";
1009    }
1010
1011    return ext_cp_dom_int_ptr;
1012 }
1013
1014 int CounterpointProblem::get_ext_cp_domain_size(int ctp_index){
1015     int cpDomSize;
1016     if(ctp_index == 0) cpDomSize = counterpoint_1->getDomain().size();
1017     else if(ctp_index == 1) cpDomSize = counterpoint_2->getDomain().size();
1018     else if(ctp_index == 2) cpDomSize = counterpoint_3->getDomain().size();
1019     else{
1020         writeToLogFile("invalid value of ctp_index given as argument to get_extended_cp_domain")
        ;
1021         return -1;
1022     }
1023 
1024     string message = "getextcpdomainsize function called. size = " + std::to_string(cpDomSize) +
         "\n";
1025     writeToLogFile(message.c_str());
1026     return cpDomSize;
```

```
}

Stratum* CounterpointProblem::getLowest(){
    return lowest;
}

Part* CounterpointProblem::getCantusFirmus(){
    return cantusFirmus;
}

Part* CounterpointProblem::getCounterpoint_1(){
    return counterpoint_1;
}

Part* CounterpointProblem::getCounterpoint_2(){
    return counterpoint_2;
}

Part* CounterpointProblem::getCounterpoint_3(){
    return counterpoint_3;
}


/************************
 * Search engine methods *
 ************************/

//Gecode::Search::TimeStop global_timeout(50000);

/**
 * Creates a search engine for the given problem
 * Should only be used when using OM, otherwise you can create the solver etc in the main file
 * @todo Modify this function to add search options etc
 * @param pb an instance of the Problem class representing a given problem
 * @param type the type of search engine to create (see enumeration in headers/gecode_problem.hpp)
 * @return a search engine for the given problem
 */
Search::Base<CounterpointProblem>* make_solver(CounterpointProblem* pb, int type){
    string message = "make_solver function called. type of solver :\n" + to_string(type) + "\n";
    writeToLogFile(message.c_str());

    Gecode::Search::Options opts;
    /**@todo add here any options you want*/
    //opts.stop = &global_timeout;
    opts.threads = 1;

    if (type == bab_solver)
        return new BAB<CounterpointProblem>(pb, opts);
    else // default case
        return new DFS<CounterpointProblem>(pb, opts);
}

/**
 * Returns the next solution space for the problem
 * Should only be used when using OM
 * @param solver a solver for the problem
 * @return an instance of the Problem class representing the next solution to the problem
 */
CounterpointProblem* get_next_solution_space(Search::Base<CounterpointProblem>* solver){
    string message = "get_next_solution_space function called.\n";
    // RESET TIMEOUT OBJECT HERE
    //global_timeout.reset();
    CounterpointProblem* sol_space = solver->next();
    if (sol_space == nullptr){
        message += "solution_space was null. \n";
        writeToLogFile(message.c_str());
        return NULL;
    }
    message += sol_space->to_string();
    writeToLogFile(message.c_str());
    return sol_space;
}

void CounterpointProblem::computeCombinedCosts(){
    int sz = 2;
    if(counterpoint_2!=nullptr){
        sz++;
    }
    if(counterpoint_3!=nullptr){
        sz++;
    }
    for (size_t i = 0; i < combinedCostNames.size(); i++) {
        string costName = combinedCostNames[i];
        IntVarArgs to_combined(sz);
        // get the cost of the cantus firmus
        vector<string> cf_vec = cantusFirmus->getToCombineCostNames();
        auto cf_it = std::find(cf_vec.begin(), cf_vec.end(), costName);
        if (cf_it != cf_vec.end()) {
            int index = std::distance(cf_vec.begin(), cf_it);
```

```
            to_combined[0] = cantusFirmus->getToCombineCosts()[index];
        } else {
            to_combined[0] = IntVar(*this, 0, 0);
        }
        // get the cost of the counterpoint 1
        vector<string> cp1_vec = counterpoint_1->getToCombineCostNames();
        auto cp1_it = std::find(cp1_vec.begin(), cp1_vec.end(), costName);
        if (cp1_it != cp1_vec.end()) {
            int index = std::distance(cp1_vec.begin(), cp1_it);
            to_combined[1] = counterpoint_1->getToCombineCosts()[index];
        } else {
            to_combined[1] = IntVar(*this, 0, 0);
        }
        // get the cost of the counterpoint 2
        if(counterpoint_2!=nullptr){
            vector<string> cp2_vec = counterpoint_2->getToCombineCostNames();
            auto cp2_it = std::find(cp2_vec.begin(), cp2_vec.end(), costName);
            if (cp2_it != cp2_vec.end()) {
                int index = std::distance(cp2_vec.begin(), cp2_it);
                to_combined[2] = counterpoint_2->getToCombineCosts()[index];
            } else {
                to_combined[2] = IntVar(*this, 0, 0);
            }
        }
        // get the cost of the counterpoint 3
        if(counterpoint_3!=nullptr){
            vector<string> cp3_vec = counterpoint_3->getToCombineCostNames();
            auto cp3_it = std::find(cp3_vec.begin(), cp3_vec.end(), costName);
            if (cp3_it != cp3_vec.end()) {
                int index = std::distance(cp3_vec.begin(), cp3_it);
                to_combined[3] = counterpoint_3->getToCombineCosts()[index];
            } else {
                to_combined[3] = IntVar(*this, 0, 0);
            }
        }
        // sum the costs
        rel(*this, combinedCosts[i], IRT_EQ, expr(*this, sum(to_combined)));
    }
}

void CounterpointProblem::uniteRelaxationCosts(){
    // Collect relaxation cost arrays from all parts
    int totalSize = 0;

    if(cantusFirmus && cantusFirmus->getRelaxationCostArray().size() > 0)
        totalSize += cantusFirmus->getRelaxationCostArray().size();
    if(counterpoint_1 && counterpoint_1->getRelaxationCostArray().size() > 0)
        totalSize += counterpoint_1->getRelaxationCostArray().size();
    if(counterpoint_2 && counterpoint_2->getRelaxationCostArray().size() > 0)
        totalSize += counterpoint_2->getRelaxationCostArray().size();
    if(counterpoint_3 && counterpoint_3->getRelaxationCostArray().size() > 0)
        totalSize += counterpoint_3->getRelaxationCostArray().size();

    // Add problem-level relaxation costs
    totalSize += problemRelaxationCosts.size();

    if(totalSize == 0){
        // No relaxation costs     totalRelaxationCost stays uninitialized
        return;
    }

    IntVarArgs allRelax(totalSize);
    int idx = 0;

    if(cantusFirmus && cantusFirmus->getRelaxationCostArray().size() > 0){
        for(int i = 0; i < cantusFirmus->getRelaxationCostArray().size(); i++){
            allRelax[idx++] = cantusFirmus->getRelaxationCostArray()[i];
        }
    }
    if(counterpoint_1 && counterpoint_1->getRelaxationCostArray().size() > 0){
        for(int i = 0; i < counterpoint_1->getRelaxationCostArray().size(); i++){
            allRelax[idx++] = counterpoint_1->getRelaxationCostArray()[i];
        }
    }
    if(counterpoint_2 && counterpoint_2->getRelaxationCostArray().size() > 0){
        for(int i = 0; i < counterpoint_2->getRelaxationCostArray().size(); i++){
            allRelax[idx++] = counterpoint_2->getRelaxationCostArray()[i];
        }
    }
    if(counterpoint_3 && counterpoint_3->getRelaxationCostArray().size() > 0){
        for(int i = 0; i < counterpoint_3->getRelaxationCostArray().size(); i++){
            allRelax[idx++] = counterpoint_3->getRelaxationCostArray()[i];
        }
    }
    for(int i = 0; i < problemRelaxationCosts.size(); i++){
        allRelax[idx++] = problemRelaxationCosts[i];
    }

    // Sum all relaxation costs into totalRelaxationCost
    totalRelaxationCost = IntVar(*this, 0, totalSize);
```

```
1206     rel(*this, totalRelaxationCost, IRT_EQ, expr(*this, sum(allRelax)));
1207 }
```

A.6.2 CounterpointUtils.cpp

```
1 //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 // This file contains the implementations of the main functions to create problems and
      counterpoints.
4 //
5
6 #include "../../headers/CounterpointProblems/CounterpointUtils.hpp"
7
8
9 Part* create_counterpoint(Home home, int species, int nMeasures, vector<int> cantusFirmus, int
      lowerBound, int upperBound, Stratum* low,
10     CantusFirmus* c, int v_type, vector<int> m_costs, vector<int> g_costs, vector<int> s_costs,
      int bm, int nV){
11     switch(nV) {
12         case TWO_VOICES :
13             switch (species) { /// call the appropriate constructor for the counterpoint
14             case FIRST_SPECIES:
15                 return new FirstSpeciesCounterpoint(home, nMeasures, cantusFirmus, lowerBound,
      upperBound, low, c, v_type, m_costs, g_costs, s_costs, bm, TWO_VOICES);
16                 break;
17             case SECOND_SPECIES:
18                 return new SecondSpeciesCounterpoint(home, nMeasures, cantusFirmus, lowerBound,
      upperBound, low, c, v_type, m_costs, g_costs, s_costs, bm, TWO_VOICES);
19                 break;
20             case THIRD_SPECIES:
21                 return new ThirdSpeciesCounterpoint(home, nMeasures, cantusFirmus, lowerBound,
      upperBound, low, c, v_type, m_costs, g_costs, s_costs, bm, TWO_VOICES);
22                 break;
23             case FOURTH_SPECIES:
24                 return new FourthSpeciesCounterpoint(home, nMeasures, cantusFirmus, lowerBound,
      upperBound, low, c, v_type, m_costs, g_costs, s_costs, bm, TWO_VOICES);
25                 break;
26             case FIFTH_SPECIES:
27                 return new FifthSpeciesCounterpoint(home, nMeasures, cantusFirmus, lowerBound,
      upperBound, low, c, v_type, m_costs, g_costs, s_costs, bm, TWO_VOICES);
28                 break;
29             default:
30                 throw std::invalid_argument("Species not implemented");
31             }
32             break;
33         case THREE_VOICES :
34             switch (species) { /// call the appropriate constructor for the counterpoint
35             case FIRST_SPECIES:
36                 return new FirstSpeciesCounterpoint(home, nMeasures, cantusFirmus, lowerBound,
      upperBound, low, c, v_type, m_costs, g_costs, s_costs, bm,
37                     TWO_VOICES, THREE_VOICES);
38                 break;
39             case SECOND_SPECIES:
40                 return new SecondSpeciesCounterpoint(home, nMeasures, cantusFirmus, lowerBound,
      upperBound, low, c, v_type, m_costs, g_costs, s_costs, bm,
41                     TWO_VOICES, THREE_VOICES);
42                 break;
43             case THIRD_SPECIES:
44                 return new ThirdSpeciesCounterpoint(home, nMeasures, cantusFirmus, lowerBound,
      upperBound, low, c, v_type, m_costs, g_costs, s_costs, bm,
45                     TWO_VOICES, THREE_VOICES);
46                 break;
47             case FOURTH_SPECIES:
48                 return new FourthSpeciesCounterpoint(home, nMeasures, cantusFirmus, lowerBound,
      upperBound, low, c, v_type, m_costs, g_costs, s_costs, bm,
49                     TWO_VOICES, THREE_VOICES);
50                 break;
51             case FIFTH_SPECIES:
52                 return new FifthSpeciesCounterpoint(home, nMeasures, cantusFirmus, lowerBound,
      upperBound, low, c, v_type, m_costs, g_costs, s_costs, bm,
53                     TWO_VOICES, THREE_VOICES);
54                 break;
55             default:
56                 throw std::invalid_argument("Species not implemented");
57             }
58             break;
59         case FOUR_VOICES :
60             switch (species) { /// call the appropriate constructor for the counterpoint
61             case FIRST_SPECIES:
62                 return new FirstSpeciesCounterpoint(home, nMeasures, cantusFirmus, lowerBound,
      upperBound, low, c, v_type, m_costs, g_costs, s_costs, bm,
63                     TWO_VOICES, THREE_VOICES, FOUR_VOICES);
64                 break;
65             case SECOND_SPECIES:
66                 return new SecondSpeciesCounterpoint(home, nMeasures, cantusFirmus, lowerBound,
      upperBound, low, c, v_type, m_costs, g_costs, s_costs, bm,
67                     TWO_VOICES, THREE_VOICES, FOUR_VOICES);
```

```
                break;
            case THIRD_SPECIES:
                return new ThirdSpeciesCounterpoint(home, nMeasures, cantusFirmus, lowerBound, upperBound, low, c, v_type, m_costs, g_costs, s_costs, bm,
                    TWO_VOICES, THREE_VOICES, FOUR_VOICES);
                break;
            case FOURTH_SPECIES:
                return new FourthSpeciesCounterpoint(home, nMeasures, cantusFirmus, lowerBound, upperBound, low, c, v_type, m_costs, g_costs, s_costs, bm,
                    TWO_VOICES, THREE_VOICES, FOUR_VOICES);
                break;
            case FIFTH_SPECIES:
                return new FifthSpeciesCounterpoint(home, nMeasures, cantusFirmus, lowerBound, upperBound, low, c, v_type, m_costs, g_costs, s_costs, bm,
                    TWO_VOICES, THREE_VOICES, FOUR_VOICES);
                break;
            default:
                throw std::invalid_argument("Species not implemented");
            }
            break;
        default :
            throw std::invalid_argument("Number of voices not implemented");
    }
};


CounterpointProblem* create_problem(vector<int> cf, vector<Species> spList, vector<int> v_type, vector<int> m_costs, vector<int> g_costs,
    vector<int> s_costs, vector<int> imp, int bm, ObjectiveMode objMode){

    switch (spList.size())
    {
    case 1:
        return new TwoVoiceCounterpoint(cf, spList[0], v_type[0], m_costs, g_costs, s_costs, imp, bm, objMode);
        break;
    case 2:
        return new ThreeVoiceCounterpoint(cf, spList, v_type, m_costs, g_costs, s_costs, imp, bm, objMode);
        break;
    case 3:
        return new FourVoiceCounterpoint(cf, spList, v_type, m_costs, g_costs, s_costs, imp, bm, objMode);
        break;
    default:
        throw std::invalid_argument("The number of voices you asked for is not implemented (yet).");
        break;
    }
}

bool notInt(char* argv){
    bool noInt = false;
    for(int i = 0; i < strlen(argv); i++){
        if(!isdigit(argv[i])){
            noInt = true;
            break;
        }
    }
    return noInt;
}

bool has_solution(CounterpointProblem* problem) {
    BAB<CounterpointProblem> e(problem);
    if (e.next()) {
        // cout << problem->getCantusFirmus()->getHIntervals() << endl;
        // cout << problem->getCounterpoint_1()->getFirstSpeciesHIntervals() << endl;
        // cout << problem->getCounterpoint_2()->getFirstSpeciesHIntervals() << endl;
        return true;
    }
    return false;
}
```

A.6.3 TwoVoiceCounterpoint.cpp

```
//
// Created by Damien Sprockeels on 13/06/2024.
// Extended and developed by Luc Cleenewerk and Diego de Patoul up to August 2024.
//

#include "../../headers/CounterpointProblems/TwoVoiceCounterpoint.hpp"
#include "../../headers/CounterpointProblems/CounterpointUtils.hpp"

/**
 * Constructor of the class.
 * @param cf a vector<int> representing the cantus firmus.
```

```
 * @param sp the species of the counterpoint. it takes values from the enum "species" in headers/Utilities.hpp
 * @param k the key of the score. it takes values from the notes in headers/Utilities.hpp
 * @param lb the lowest note possible for the counterpoint in MIDI
 * @param ub the highest note possible for the counterpoint in MIDI
 */
TwoVoiceCounterpoint::TwoVoiceCounterpoint(vector<int> cf, Species sp, int v_type, vector<int> m_costs, vector<int> g_costs,
    vector<int> s_costs, vector<int> imp, int bm, ObjectiveMode objMode) :
    CounterpointProblem(cf, v_type, m_costs, g_costs, s_costs, imp, TWO_VOICES, objMode){
    species = sp;
    upper_1 = new Stratum(*this, nMeasures, 0, 127, lowest->getNotes());
    upper_2 = nullptr;
    upper_3 = nullptr;

    counterpoint_1 = create_counterpoint(*this, species, nMeasures, cf, (6 * v_type - 12) + cf[0], (6 * v_type + 12) + cf[0], lowest, cantusFirmus,
        v_type, m_costs, g_costs, s_costs, bm, TWO_VOICES);
    counterpoint_2 = nullptr;
    counterpoint_3 = nullptr;

    vector<Part*> parts = {cantusFirmus, counterpoint_1};

    // G6 : no chromatic melodies (works for 1st, 2nd and 3rd species)
    if (activeConstraints[V2_G6]) {
        G6_noChromaticMelodies(*this, counterpoint_1, species);
    }

    // M2_1 : variety cost (penalize repeated notes in 2-voice mode)
    if (activeConstraints[V2_1M4]) {
        M2_1_varietyCost(*this, parts);
    }

    // 1.H4 (G9)
    if (activeConstraints[V2_G9]) {
        G9_lastChordSameAsFundamental(*this, lowest, cantusFirmus);
    }

    /// H2 from Thibault: The first harmonic interval must be a perfect consonance
    if (activeConstraints[V2_1H2]) {
        //we check for fifth species since it always starts with a break, so it doesn't make sense to apply the constraint in this case
        if(species!=FIFTH_SPECIES){
            H2_1_startWithPerfectConsonance(*this, counterpoint_1);
        }
    }

    /// H3 from Thibault: The last harmonic interval must be a perfect consonance
    if (activeConstraints[V2_1H3]) {
        H3_1_endWithPerfectConsonance(*this, counterpoint_1);
    }

    // H5 from Thibault : The cp and the cf cannot play the same note
    if (activeConstraints[V2_1H5]) {
        H5_1_cpAndCfDifferentNotes(*this, counterpoint_1, cantusFirmus);
    }

    //3.H4 : in the penultimate measure, if the cantusFirmus is in the upper part, then the h_interval of the first note should be a minor third
    if (activeConstraints[SP3_3H4_2V]) {
        // Create Boolean variables for each condition
        BoolVar isThirdSpecies(*this, 0, 1);
        BoolVar isNotLowestCantus(*this, 0, 1);

        rel(*this, isThirdSpecies == (counterpoint_1->getSpecies() == THIRD_SPECIES));
        rel(*this, isNotLowestCantus == (cantusFirmus->getIsNotLowest()[cantusFirmus->getIsNotLowest().size()-2] == 1));

        // Combine the conditions using Gecode's AND operator
        BoolVar is3H4active(*this, 0, 1);
        rel(*this, is3H4active == (isThirdSpecies && isNotLowestCantus));

        // Implication constraint
        rel(*this, is3H4active >>
            (expr(*this, abs(cantusFirmus->getFirstHInterval()[cantusFirmus->getFirstHInterval().size()-2])) == MINOR_THIRD));
    }

    // P8 from Karim : successive fifths and successive octave / unissons are forbidden
    if (activeConstraints[V2_1P8]) {
        P8_noSuccessiveSamePerfectInterval(*this, parts);
    }

    // P9 from Karim : simultaneous repetitions between two voices are forbidden
    if (activeConstraints[V2_1P9]) {
        P9_noSimultaneousRepetition(*this, parts);
    }

    // [Claude Code]
    // Tonal rule (mandatory): a disjunct weak beat must belong to the measure harmony.
```

```
    // Previously applied only in 3v/4v; extending it to 2 voices also closes the gap
    // where 3rd-species weak beats +1/+3 were left unchecked for consonance.
    if (counterpoint_1->getSpecies() == SECOND_SPECIES || counterpoint_1->getSpecies() ==
    THIRD_SPECIES) {
        H2_4_chordMembershipOnDisjunctWeakBeats(*this, counterpoint_1, parts);
    }

    // [Claude Code]
    // Consonance on the three weak beats of 2-voice 3rd species: reuse the proven multi-voice
    // check. The in-species H2_3 only ties the central weak beat and reads a free
    // getConsonance(), so disjunct dissonances (e.g. a leapt-to fourth fa/do) slipped onto
    // beats +1/+3.
    if (counterpoint_1->getSpecies() == THIRD_SPECIES && activeConstraints[SP3_3H2]) {
        H2_3_dissonanceImpliesDiminution_multiVoice(*this, counterpoint_1, parts);
    }

    setStrata();

    unitedCosts = IntVarArray(*this, counterpoint_1->getCosts().size(), 0, 1000000);

    for(int i = 0; i < unitedCosts.size(); i++){
        rel(*this, unitedCosts[i], IRT_EQ, counterpoint_1->getCosts()[i]);
    }

    unitedCostNames = counterpoint_1->getCostNames();

    // compute combined costs
    computeCombinedCosts();

    orderCosts();

    solutionArray = IntVarArray(*this, counterpoint_1->getBranchingNotes().size(), 0, 127);
    rel(*this, solutionArray, IRT_EQ, counterpoint_1->getBranchingNotes());

    branch(*this, lowest->getNotes().slice(0, 4/notesPerMeasure.at(FIRST_SPECIES), lowest->
    getNotes().size()), INT_VAR_DEGREE_MAX(), INT_VAL_SPLIT_MIN());
    if(species==FIFTH_SPECIES){
        branch(*this, counterpoint_1->getSpeciesArray(), INT_VAR_DEGREE_MAX(), INT_VAL_RND(3U));
    }
    if(species==FIFTH_SPECIES){
        branch(*this, counterpoint_1->getCambiataCostArray(),  INT_VAR_DEGREE_MAX(), INT_VAL_MIN
    ());
    }
    if(species==FOURTH_SPECIES || species==FIFTH_SPECIES){
        branch(*this, counterpoint_1->getNoSyncope(), BOOL_VAR_AFC_MAX(), BOOL_VAL_MIN());
    }
    branch(*this, solutionArray, INT_VAR_AFC_MAX(), INT_VAL_RND(1U));
    branch(*this, cost(), INT_VAR_NONE(), INT_VAL_MAX());
}
// COPY CONSTRUCTOR
TwoVoiceCounterpoint::TwoVoiceCounterpoint(TwoVoiceCounterpoint& s) : CounterpointProblem(s){
    species = s.species;
}

IntLexMinimizeSpace* TwoVoiceCounterpoint::copy(){
    return new TwoVoiceCounterpoint(*this);
}


string TwoVoiceCounterpoint::to_string() const {
    string text = "";
    text += CounterpointProblem::to_string();
    text += "Lowest : \n";
    text += cantusFirmus->to_string();
    text += "\n";
    text += "Counterpoint 1 : \n";
    text += counterpoint_1->to_string();
    text += "\n";
    text += "Solution Array : \n";
    text += intVarArray_to_string(solutionArray);
    text += "\n";
    text += counterpoint_1->to_string();
    return text;
}
```

A.6.4 ThreeVoiceCounterpoint.cpp

```
//
// Created by Luc Cleenewerk and Diego de Patoul.
//

#include "../../headers/CounterpointProblems/ThreeVoiceCounterpoint.hpp"
#include "../../headers/CounterpointProblems/CounterpointUtils.hpp"

/**
 * Constructor of the class.
 * @param cf a vector<int> representing the cantus firmus.
```

```
 * @param sp the species of the counterpoint. it takes values from the enum "species" in headers
    /Utilities.hpp
 * @param k the key of the score. it takes values from the notes in headers/Utilities.hpp
 * @param lb the lowest note possible for the counterpoint in MIDI
 * @param ub the highest note possible for the counterpoint in MIDI
 */
ThreeVoiceCounterpoint::ThreeVoiceCounterpoint(vector<int> cf, vector<Species> sp, vector<int>
    v_type, vector<int> m_costs, vector<int> g_costs,
    vector<int> s_costs, vector<int> imp, int bm, ObjectiveMode objMode) :
    CounterpointProblem(cf, -1, m_costs, g_costs, s_costs, imp, THREE_VOICES, objMode){
    species = sp;

    //initialize upper strata
    upper_1 = new Stratum(*this, nMeasures, 0, 127, lowest->getNotes(), THREE_VOICES);
    upper_2 = new Stratum(*this, nMeasures, 0, 127, lowest->getNotes(), THREE_VOICES);
    upper_3 = nullptr;

    //create counterpoints

    counterpoint_1 = create_counterpoint(*this, species[0], nMeasures, cf, (6 * v_type[0] - 12)
    + cf[0], (6 * v_type[0] + 12) + cf[0], lowest,
        cantusFirmus, v_type[0], m_costs, g_costs, s_costs, bm, THREE_VOICES);
    counterpoint_2 = create_counterpoint(*this, species[1], nMeasures, cf, (6 * v_type[1] - 12)
    + cf[0], (6 * v_type[1] + 12) + cf[0], lowest,
        cantusFirmus, v_type[1], m_costs, g_costs, s_costs, bm, THREE_VOICES);
    counterpoint_3 = nullptr;

    //create strata
    setStrata();

    //creating variables

    vector<Part*> parts = {cantusFirmus, counterpoint_1, counterpoint_2};
    int scc_cz = ((cantusFirmus->getSize()/4)-1);
    bool containsThirdSpecies = 0;

    triadCostArray = IntVarArray(*this, counterpoint_1->getFirstHInterval().size(), IntSet({0,
    counterpoint_1->getTriadCost()}));
    successiveCostArray = IntVarArray(*this, scc_cz, IntSet({0, counterpoint_1->getSuccCost()}))
    ;

    // G6 : no chromatic melodies (works for 1st, 2nd and 3rd species)
    if (activeConstraints[V3_G6]) {
        for(int p = 1; p < parts.size(); p++){
            G6_noChromaticMelodies(*this, parts[p], sp[p-1]);
        }
    }

    // 1.H4 (G9) last chord must have the same fundamental as the cf (used throughout the
    composition)
    if (activeConstraints[V3_1H4]) {
        G9_lastChordSameAsFundamental(*this, lowest, cantusFirmus);
    }

    //H5 for three voices
    if (activeConstraints[V3_1H5]) {
        H5_1_differentNotes(*this, parts); //this function modified by Tom Lai
    }

    //H13
    if (activeConstraints[V4_U2]) {
        noMinorSecondBetweenUpper(*this, vector<Stratum*>{upper_1, upper_2});
    }

    //H8 : the triad should be used as much as possible
    if (activeConstraints[V3_1H8]) {
        H8_3v_preferHarmonicTriad(*this, counterpoint_1, triadCostArray, upper_1, upper_2);
    }

    //M4 variety cost (notes should be as diverse as possible)
    if (activeConstraints[V3_1M4]) {
        M2_1_varietyCost(*this, parts);
    }

    //P4 avoid successive perfect consonances
    if (activeConstraints[V3_1P4]) {
        P4_successiveCost(*this, parts, successiveCostArray);
    }

    //P6 : no move in same direction
    if (activeConstraints[V3_1P6]) {
        P6_3v_noMoveInSameDirection(*this, parts);
        // if(counterpoint_1->getSpecies()!=FOURTH_SPECIES&&counterpoint_2->getSpecies()!=
    FOURTH_SPECIES){ //doesn't apply to 4th species
            // P6_3v_noMoveInSameDirection(*this, parts);
        // }
    }

    //P7 : no suxxessive ascending sixths
    if (activeConstraints[V3_1P7]) {
```

```
93          P7_noSuccessiveAscendingSixths(*this, parts);
94      }
95
96      // P8 : successive fifths and successive octave / unissons are forbidden
97      if (activeConstraints[V3_1P8]) {
98          P8_noSuccessiveSamePerfectInterval(*this, parts);
99      }
100
101     // P9 from Karim : simultaneous repetitions between two voices are forbidden
102     if (activeConstraints[V3_1P9]) {
103         P9_noSimultaneousRepetition(*this, parts);
104     }
105
106     //2.M2, have to write it here since it has a weird interaction with the third species
107     if (activeConstraints[V3_2M2]) {
108         M2_2_3v_twoConsecutiveNotesAreNotTheSame(*this, parts, containsThirdSpecies);
109     }
110
111     // [Claude Code]
112     // H2 extended (multi-voice rule): a disjunct weak beat must be consonant with
113     // every other voice's strong beat, not only the lowest stratum. See constraints.cpp.
114     for (Part* p : parts) {
115         if (p->getSpecies() == SECOND_SPECIES && activeConstraints[SP2_2H2]) {
116             H2_2_arsisHarmoniesCannotBeDissonant_multiVoice(*this, p, parts);
117         } else if (p->getSpecies() == THIRD_SPECIES && activeConstraints[SP3_3H2]) {
118             H2_3_dissonanceImpliesDiminution_multiVoice(*this, p, parts);
119         }
120         // Tonal rule (mandatory): disjunct weak beat must belong to the measure harmony.
121         if (activeConstraints[V3_4H2] && (p->getSpecies() == SECOND_SPECIES || p->getSpecies()
    == THIRD_SPECIES)) {
122             H2_4_chordMembershipOnDisjunctWeakBeats(*this, p, parts);
123         }
124     }
125
126     // 1.H13 no minor second interval between upper
127     if (activeConstraints[V4_U2]) {
128         noMinorSecondBetweenUpper(*this, vector<Stratum*>{upper_1, upper_2});
129     }
130
131     //5.R9 two  fifth species counterpoints should be as different as possible
132     if (activeConstraints[V3_5R9]) {
133         R9_5_twoFifthSpeciesDiversity_3v(*this, counterpoint_1, counterpoint_2);
134     }
135
136     solutionArray = IntVarArray(*this, counterpoint_1->getBranchingNotes().size() +
    counterpoint_2->getBranchingNotes().size(), 0, 127);
137
138     unitedCosts = IntVarArray(*this, 14, 0, 10000000);
139     unitedCostNames = {};
140
141     uniteCounterpoints();
142     uniteCosts();
143
144     // for(int i = 0; i < unitedCostNames.size(); i++){
145     //     cout << unitedCostNames[i] << endl;
146     // }
147
148     // compute combined costs
149     computeCombinedCosts();
150
151
152     orderCosts();
153
154     //Branching strategies
155
156     branch(*this, lowest->getNotes().slice(0, 4/notesPerMeasure.at(FIRST_SPECIES), lowest->
    getNotes().size()), INT_VAR_DEGREE_MAX(), INT_VAL_SPLIT_MIN());
157
158     if(species[0]==FIFTH_SPECIES){
159         branch(*this, counterpoint_1->getSpeciesArray(), INT_VAR_DEGREE_MAX(), INT_VAL_RND(3U));
160     }
161     if(species[1]==FIFTH_SPECIES){
162         branch(*this, counterpoint_2->getSpeciesArray(), INT_VAR_DEGREE_MAX(), INT_VAL_RND(3U));
163     }
164
165     if(species[0]==FIFTH_SPECIES){
166         branch(*this, counterpoint_1->getCambiataCostArray(), INT_VAR_DEGREE_MAX(),
    INT_VAL_SPLIT_MIN());
167     }
168     if(species[1]==FIFTH_SPECIES){
169         branch(*this, counterpoint_2->getCambiataCostArray(), INT_VAR_DEGREE_MAX(),
    INT_VAL_SPLIT_MIN());
170     }
171
172     if(species[0]==FOURTH_SPECIES || species[0]==FIFTH_SPECIES){
173         branch(*this, counterpoint_1->getNoSyncope(), BOOL_VAR_AFC_MAX(), BOOL_VAL_MIN());
174     }
175     if(species[1]==FOURTH_SPECIES || species[1]==FIFTH_SPECIES){
176         branch(*this, counterpoint_2->getNoSyncope(), BOOL_VAR_AFC_MAX(), BOOL_VAL_MIN());
177     }
```

```
178
179     // Dispatched through globals (g_solution_var_sel / g_solution_val_sel) so
180     // callers can swap heuristics without recompiling. Defaults are
181     // AFC_MAX + VAL_RND, matching the 2-voice production branching.
182     branch_solution_array_dynamic(*this, solutionArray);
183     branch(*this, cost(), INT_VAR_NONE(), INT_VAL_MIN());
184
185     writeToLogFile(("solution array size : " + std::to_string(solutionArray.size())).c_str());
186 }
187
188 // COPY CONSTRUCTOR
189 ThreeVoiceCounterpoint::ThreeVoiceCounterpoint(ThreeVoiceCounterpoint& s) : CounterpointProblem(
        s){
190     species = s.species;
191 }
192
193 IntLexMinimizeSpace* ThreeVoiceCounterpoint::copy(){
194     return new ThreeVoiceCounterpoint(*this);
195 }
196
197
198 string ThreeVoiceCounterpoint::to_string() const {
199     string text = CounterpointProblem::to_string();
200     text += "Counterpoint 1 : \n";
201     text += counterpoint_1->to_string();
202     text += "\n";
203     text += "Counterpoint 2 : \n";
204     text += counterpoint_2->to_string();
205     text += "\n";
206     text += "Successive cost array : \n";
207     text += intVarArray_to_string(successiveCostArray);
208     text += "\n";
209     text += " Solution array : \n";
210     text += intVarArray_to_string(solutionArray);
211     text += "\n";
212     return text;
213 }
214
215 void ThreeVoiceCounterpoint::uniteCounterpoints(){
216     //this function takes all the notes that are being branched on and adds them one after the
        other to the solution array
217     int idx = 0;
218     for(int i = 0; i < counterpoint_1->getBranchingNotes().size(); i++){
219         rel(*this, solutionArray[idx], IRT_EQ, counterpoint_1->getBranchingNotes()[i]);
220         idx++;
221     }
222     for(int i = 0; i < counterpoint_2->getBranchingNotes().size(); i++){
223         rel(*this, solutionArray[idx], IRT_EQ, counterpoint_2->getBranchingNotes()[i]);
224         idx++;
225     }
226 }
227
228 void ThreeVoiceCounterpoint::uniteCosts(){
229     //this function takes the costs that are present for each species and some 3v specific costs
         and adds them together
230     int cp1_idx = 0;
231     int cp2_idx = 0;
232     for(int i = 0; i < 14; i++){
233         //goes through every possible cost
234         string name = importanceNames[i];
235         //check seperately for the successive cost
236         if(name=="succ"){
237             unitedCostNames.push_back(name);
238             rel(*this, unitedCosts[i], IRT_EQ, expr(*this, sum(IntVarArgs(successiveCostArray)))
        );
239         } else if(name=="triad"){ //check seperately for the triad cost
240             unitedCostNames.push_back(name);
241             rel(*this, unitedCosts[i], IRT_EQ, expr(*this, sum(IntVarArgs(triadCostArray))));
242         } else {
243
244             //decides if the cost is present for any counterpoint
245
246             bool cp1_contains = 0;
247             bool cp2_contains = 0;
248             int sz = 0;
249             for(int t = 0; t < counterpoint_1->getCostNames().size(); t++){
250                 if(name==counterpoint_1->getCostNames()[t]){
251                     cp1_contains=1;
252                     sz++;
253                 }
254             }
255             for(int t = 0; t < counterpoint_2->getCostNames().size(); t++){
256                 if(name==counterpoint_2->getCostNames()[t]){
257                     cp2_contains=1;
258                     sz++;
259                 }
260             }
261             //if the cost is present in no counterpoint -> DONT leave it empty
262             if(!cp1_contains && !cp2_contains){
263                 //unitedCostNames.push_back("NOT ADDED");
```

```
                rel(*this, unitedCosts[i], IRT_EQ, 0);
            } else {
                //if it is in either counterpoint -> add the cost
                unitedCostNames.push_back(name);
                //adds the cost to the IntVarArgs
                IntVarArgs x(sz);
                int idx=0;
                if(cp1_contains){
                    x[idx] = counterpoint_1->getCosts()[cp1_idx];
                    idx++;
                    cp1_idx++;
                }
                if(cp2_contains){
                    x[idx] = counterpoint_2->getCosts()[cp2_idx];
                    idx++;
                    cp2_idx++;
                }
                //sum the cost together if it are present in both counterpoints
                rel(*this, unitedCosts[i], IRT_EQ, expr(*this, sum(x)));
            }
        }
    }
}
```

A.6.5 FourVoiceCounterpoint.cpp

```
//
// Created by Luc Cleenewerk and Diego de Patoul.
//

#include "../../headers/CounterpointProblems/FourVoiceCounterpoint.hpp"
#include "../../headers/CounterpointProblems/CounterpointUtils.hpp"
#include <algorithm>

FourVoiceCounterpoint::FourVoiceCounterpoint(vector<int> cf, vector<Species> sp, vector<int>
    v_type, vector<int> m_costs, vector<int> g_costs,
        vector<int> s_costs, vector<int> imp, int bm, ObjectiveMode objMode):
    CounterpointProblem(cf, -1, m_costs, g_costs, s_costs, imp, FOUR_VOICES, objMode)
{
    species = sp;

    //initialize upper strata

    upper_1 = new Stratum(*this, nMeasures, 0, 127, lowest->getNotes(), THREE_VOICES,
    FOUR_VOICES);
    upper_2 = new Stratum(*this, nMeasures, 0, 127, lowest->getNotes(), THREE_VOICES,
    FOUR_VOICES);
    upper_3 = new Stratum(*this, nMeasures, 0, 127, lowest->getNotes(), THREE_VOICES,
    FOUR_VOICES);

    //create counterpoints

    counterpoint_1 = create_counterpoint(*this, species[0], nMeasures, cf, (6 * v_type[0] - 12)
    + cf[0], (6 * v_type[0] + 12) + cf[0], lowest,
        cantusFirmus, v_type[0], m_costs, g_costs, s_costs, bm, FOUR_VOICES);
    counterpoint_2 = create_counterpoint(*this, species[1], nMeasures, cf, (6 * v_type[1] - 12)
    + cf[0], (6 * v_type[1] + 12) + cf[0], lowest,
        cantusFirmus, v_type[1], m_costs, g_costs, s_costs, bm, FOUR_VOICES);
    counterpoint_3 = create_counterpoint(*this, species[2], nMeasures, cf, (6 * v_type[2] - 12)
    + cf[0], (6 * v_type[2] + 12) + cf[0], lowest,
        cantusFirmus, v_type[2], m_costs, g_costs, s_costs, bm, FOUR_VOICES);

    //create strata

    setStrata();

    //creating variables

    vector<Part*> parts = {cantusFirmus, counterpoint_1, counterpoint_2, counterpoint_3};
    int scc_cz = 3*((cantusFirmus->getSize()/4)-1);
    bool containsThirdSpecies = 0;

    triadCostArray = IntVarArray(*this, counterpoint_1->getFirstHInterval().size(), IntSet({0,
    not_harmonic_triad_cost, double_fifths_cost, double_thirds_cost,
        triad_with_octave_cost}));
    successiveCostArray = IntVarArray(*this, scc_cz, IntSet({0, counterpoint_1->getSuccCost()}))
    ;

    // G6 : no chromatic melodies (works for 1st, 2nd and 3rd species)
    if (activeConstraints[V4_G6]) {
        for(int p = 1; p < parts.size(); p++){
            G6_noChromaticMelodies(*this, parts[p], sp[p-1]);
        }
    }

    // 1.H4 (G9) last chord must have the same fundamental as the cf (used throughout the
    composition)
```

```
52      if (activeConstraints[V4_1H4]) {
53          if (softConstraints[V4_1H4]) {
54              problemRelaxationCosts = IntVarArray(*this, 2, 0, 1);
55              G9_lastChordSameAsFundamental_soft(*this, lowest, cantusFirmus,
    problemRelaxationCosts, 0);
56          } else {
57              G9_lastChordSameAsFundamental(*this, lowest, cantusFirmus);
58          }
59      }
60
61      //H8 : harmonic triads are preferred, adapted for 4 voices
62      if (activeConstraints[V4_1H8]) {
63          H8_4v_preferHarmonicTriad(*this, triadCostArray, upper_1, upper_2, upper_3);
64      }
65
66      //M4 variety cost (notes should be as diverse as possible)
67      if (activeConstraints[V4_1M4]) {
68          M2_1_varietyCost(*this, parts);
69      }
70
71      //P4 avoid successive perfect consonances
72      if (activeConstraints[V4_1P4]) {
73          P4_successiveCost(*this, parts, successiveCostArray);
74      }
75
76      //P6 : no move in same direction
77      if (activeConstraints[V4_1P6]) {
78          P6_4v_noMoveInSameDirection(*this, parts);
79          // if(counterpoint_1->getSpecies()!=FOURTH_SPECIES&&counterpoint_2->getSpecies()!=
    FOURTH_SPECIES&&counterpoint_3->getSpecies()!=FOURTH_SPECIES){
80          //     P6_4v_noMoveInSameDirection(*this, parts);
81          // }
82      }
83
84      //P7 : no suxxessive ascending sixths
85      if (activeConstraints[V4_1P7]) {
86          P7_noSuccessiveAscendingSixths(*this, parts);
87      }
88
89      // P8 from Karim : successive fifths and successive octave / unissons are forbidden
90      if (activeConstraints[V4_1P8]) {
91          P8_noSuccessiveSamePerfectInterval(*this, parts);
92      }
93
94      // P9 from Karim : simultaneous repetitions between two voices are forbidden
95      if (activeConstraints[V4_1P9]) {
96          P9_noSimultaneousRepetition(*this, parts);
97      }
98
99      //2.M2, have to write it here since it has a weird interaction with the third species
100     if (activeConstraints[V4_2M2]) {
101         if (softConstraints[V4_2M2]) {
102             M2_2_3v_twoConsecutiveNotesAreNotTheSame_soft(*this, parts, containsThirdSpecies);
103         } else {
104             M2_2_3v_twoConsecutiveNotesAreNotTheSame(*this, parts, containsThirdSpecies);
105         }
106     }
107
108     // [Claude Code]
109     // H2 extended (multi-voice rule): a disjunct weak beat must be consonant with
110     // every other voice's strong beat, not only the lowest stratum. See constraints.cpp.
111     for (Part* p : parts) {
112         if (activeConstraints[SP2_2H2] && p->getSpecies() == SECOND_SPECIES) {
113             H2_2_arsisHarmoniesCannotBeDissonant_multiVoice(*this, p, parts);
114         } else if (activeConstraints[SP3_3H2] && p->getSpecies() == THIRD_SPECIES) {
115             H2_3_dissonanceImpliesDiminution_multiVoice(*this, p, parts);
116         }
117         // Tonal rule (mandatory): disjunct weak beat must belong to the measure harmony.
118         if (activeConstraints[V4_4H2] && (p->getSpecies() == SECOND_SPECIES || p->getSpecies()
    == THIRD_SPECIES)) {
119             H2_4_chordMembershipOnDisjunctWeakBeats(*this, p, parts);
120         }
121     }
122
123     //two fifth species counterpoints should be as different as possible
124     if (activeConstraints[V4_5R9]) {
125         R9_5_twoFifthSpeciesDiversity_3v(*this, counterpoint_1, counterpoint_3);
126     }
127
128     //no minor second interval between upper
129     if (activeConstraints[V4_U2]) {
130         noMinorSecondBetweenUpper(*this, vector<Stratum*>{upper_1, upper_2, upper_3});
131     }
132
133     solutionArray = IntVarArray(*this, counterpoint_1->getBranchingNotes().size() +
    counterpoint_2->getBranchingNotes().size() +
134         counterpoint_3->getBranchingNotes().size(), 0, 127);
135
136     unitedCosts = IntVarArray(*this, 14, 0, 10000000);
137     unitedCostNames = {};
```

```
138
139     uniteCounterpoints();
140     uniteCosts();
141
142     // Collect relaxation costs from all parts and sum them
143     uniteRelaxationCosts();
144
145     // compute combined costs
146     computeCombinedCosts();
147
148     orderCosts();
149
150     //Branching strategies
151
152     branch(*this, lowest->getNotes().slice(0, 4/notesPerMeasure.at(FIRST_SPECIES), lowest->
        getNotes().size()), INT_VAR_DEGREE_MAX(), INT_VAL_SPLIT_MIN());
153
154     if(species[0]==FIFTH_SPECIES){
155         branch(*this, counterpoint_1->getSpeciesArray(), INT_VAR_DEGREE_MAX(), INT_VAL_RND(3U));
156         branch(*this, counterpoint_1->getCambiataCostArray(), INT_VAR_DEGREE_MAX(),
        INT_VAL_SPLIT_MIN());
157         branch(*this, counterpoint_1->getNoSyncope(), BOOL_VAR_AFC_MAX(), BOOL_VAL_MIN());
158     }
159     if(species[1]==FIFTH_SPECIES){
160         branch(*this, counterpoint_2->getSpeciesArray(), INT_VAR_DEGREE_MAX(), INT_VAL_RND(3U));
161         branch(*this, counterpoint_2->getCambiataCostArray(), INT_VAR_DEGREE_MAX(),
        INT_VAL_SPLIT_MIN());
162         branch(*this, counterpoint_2->getNoSyncope(), BOOL_VAR_AFC_MAX(), BOOL_VAL_MIN());
163     }
164     if(species[2]==FIFTH_SPECIES){
165         branch(*this, counterpoint_3->getSpeciesArray(), INT_VAR_DEGREE_MAX(), INT_VAL_RND(3U));
166         branch(*this, counterpoint_3->getCambiataCostArray(), INT_VAR_DEGREE_MAX(),
        INT_VAL_SPLIT_MIN());
167         branch(*this, counterpoint_3->getNoSyncope(), BOOL_VAR_AFC_MAX(), BOOL_VAL_MIN());
168     }
169
170
171     if(species[0]==FOURTH_SPECIES){
172         branch(*this, counterpoint_1->getNoSyncope(), BOOL_VAR_AFC_MAX(), BOOL_VAL_MIN());
173     }
174     if(species[1]==FOURTH_SPECIES){
175         branch(*this, counterpoint_2->getNoSyncope(), BOOL_VAR_AFC_MAX(), BOOL_VAL_MIN());
176     }
177     if(species[2]==FOURTH_SPECIES){
178         branch(*this, counterpoint_3->getNoSyncope(), BOOL_VAR_AFC_MAX(), BOOL_VAL_MIN());
179     }
180
181     // Branch on counterpoints in order of increasing complexity (fewer notes first)
182     // This allows simpler voices to constrain the search space before complex ones
183     vector<pair<int, Part*>> voicesBySize = {
184         {counterpoint_1->getBranchingNotes().size(), counterpoint_1},
185         {counterpoint_2->getBranchingNotes().size(), counterpoint_2},
186         {counterpoint_3->getBranchingNotes().size(), counterpoint_3}
187     };
188     // Sort by size (ascending order - simplest first)
189     std::sort(voicesBySize.begin(), voicesBySize.end(),
190         [](const pair<int, Part*>& a, const pair<int, Part*>& b) {
191             return a.first < b.first;
192         });
193
194     // Branch on each voice in order of complexity.
195     // Dispatched through globals (g_solution_var_sel / g_solution_val_sel) so
196     // callers can swap heuristics without recompiling. Defaults are
197     // AFC_MAX + VAL_RND, matching the 2-voice production branching.
198     for(const auto& voice : voicesBySize) {
199         branch_solution_array_dynamic(*this, voice.second->getBranchingNotes());
200     }
201
202     branch(*this, cost(), INT_VAR_NONE(), INT_VAL_MAX()); // Solves all "ValOfUnassignedVar"
    problems + accelerate every test
203 }
204
205 // COPY CONSTRUCTOR
206 FourVoiceCounterpoint::FourVoiceCounterpoint(FourVoiceCounterpoint& s) : CounterpointProblem(s){
207     species = s.species;
208 }
209
210 IntLexMinimizeSpace* FourVoiceCounterpoint::copy(){
211     return new FourVoiceCounterpoint(*this);
212 }
213
214 string FourVoiceCounterpoint::to_string() const {
215     string text = CounterpointProblem::to_string();
216     text += "Counterpoint 1 : \n";
217     text += counterpoint_1->to_string();
218     text += "\n";
219     text += "Counterpoint 2 : \n";
220     text += counterpoint_2->to_string();
221     text += "\n";
222     text += "Counterpoint 3 : \n";
```

```
223     text += counterpoint_3->to_string();
224     text += "\n";
225     text += " Solution array : \n";
226     text += intVarArray_to_string(solutionArray);
227     text += "\n";
228     return text;
229 }
230
231 void FourVoiceCounterpoint::uniteCounterpoints(){
232     int idx = 0;
233     for(int i = 0; i < counterpoint_1->getBranchingNotes().size(); i++){
234         rel(*this, solutionArray[idx], IRT_EQ, counterpoint_1->getBranchingNotes()[i]);
235         idx++;
236     }
237     for(int i = 0; i < counterpoint_2->getBranchingNotes().size(); i++){
238         rel(*this, solutionArray[idx], IRT_EQ, counterpoint_2->getBranchingNotes()[i]);
239         idx++;
240     }
241     for(int i = 0; i < counterpoint_3->getBranchingNotes().size(); i++){
242         rel(*this, solutionArray[idx], IRT_EQ, counterpoint_3->getBranchingNotes()[i]);
243         idx++;
244     }
245 }
246
247 void FourVoiceCounterpoint::uniteCosts(){
248     int cp1_idx = 0;
249     int cp2_idx = 0;
250     int cp3_idx = 0;
251     for(int i = 0; i < 14; i++){
252         string name = importanceNames[i];
253         if(name=="succ"){
254             unitedCostNames.push_back(name);
255             rel(*this, unitedCosts[i], IRT_EQ, expr(*this, sum(IntVarArgs(successiveCostArray)))
    );
256         } else if(name=="triad"){
257             unitedCostNames.push_back(name);
258             rel(*this, unitedCosts[i], IRT_EQ, expr(*this, sum(IntVarArgs(triadCostArray))));
259         } else {
260
261             //decides if the cost is present for any counterpoint
262
263             bool cp1_contains = 0;
264             bool cp2_contains = 0;
265             bool cp3_contains = 0;
266             int sz = 0;
267             //determining in how many counterpoints a cost appears
268             for(int t = 0; t < counterpoint_1->getCostNames().size(); t++){
269                 if(name==counterpoint_1->getCostNames()[t]){
270                     cp1_contains=1;
271                     sz++;
272                 }
273             }
274             for(int t = 0; t < counterpoint_2->getCostNames().size(); t++){
275                 if(name==counterpoint_2->getCostNames()[t]){
276                     cp2_contains=1;
277                     sz++;
278                 }
279             }
280             for(int t = 0; t < counterpoint_3->getCostNames().size(); t++){
281                 if(name==counterpoint_3->getCostNames()[t]){
282                     cp3_contains=1;
283                     sz++;
284                 }
285             }
286             //if it is not present in any counterpoint -> DONT leave it empty
287             if(!cp1_contains && !cp2_contains && !cp3_contains){
288                 rel(*this, unitedCosts[i], IRT_EQ, 0);
289                 //unitedCostNames.push_back("NOT ADDED");
290             } else { //else -> add the costs together for that entry
291                 unitedCostNames.push_back(name);
292                 //adds the cost to the IntVarArgs
293                 IntVarArgs x(sz);
294                 int idx=0;
295                 if(cp1_contains){
296                     x[idx] = counterpoint_1->getCosts()[cp1_idx];
297                     idx++;
298                     cp1_idx++;
299                 }
300                 if(cp2_contains){
301                     x[idx] = counterpoint_2->getCosts()[cp2_idx];
302                     idx++;
303                     cp2_idx++;
304                 }
305                 if(cp3_contains){
306                     x[idx] = counterpoint_3->getCosts()[cp3_idx];
307                     idx++;
308                     cp3_idx++;
309                 }
310                 rel(*this, unitedCosts[i], IRT_EQ, expr(*this, sum(x)));
311             }
```

```
312         }
313     }
314 }
```

A.7 Voice and part sources

A.7.1 Voice.cpp

```
1 //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 //
4
5 #include "../../headers/Parts/Voice.hpp"
6
7 Voice::Voice(Home home, int nMes, int lb, int ub){
8     nMeasures = nMes;
9     size = nMes*4;
10
11     lowerBound = lb;
12     upperBound = ub;
13
14     notes = IntVarArray(home, size-3, lowerBound, upperBound);
15     //this is so that species other than first species are allowed to go beneath the lowest
       h_interval, since it is only calculated using the first note
16     //(so 2nd, 3rd or 4th note couldn't be smaller than 0)
17     h_intervals = IntVarArray(home, size-3, -PERFECT_OCTAVE, PERFECT_OCTAVE);
18     m_intervals_brut = IntVarArray(home, notes.size()-1, -PERFECT_OCTAVE, PERFECT_OCTAVE);
19
20 }
21
22
23 string Voice::to_string() const {
24     string text = "voice : ";
25     text += "VOICE TO STRING NOT IMPLEMENTED YET";
26     text += "\n";
27     return text;
28 }
29
30
31 // clone constructor
32 Voice::Voice(Home home, Voice &s){
33     nMeasures = s.nMeasures;
34     size = s.size;
35     lowerBound = s.lowerBound;
36     upperBound = s.upperBound;
37     notes.update(home, s.notes);
38     h_intervals.update(home, s.h_intervals);
39     m_intervals_brut.update(home, s.m_intervals_brut);
40     motions.update(home, s.motions);
41 }
42
43 Voice* Voice::clone(Home home){
44     return new Voice(home, *this);
45 }
46
47 IntVarArgs Voice::getFirstNotes(){
48     return notes.slice(0, 4/notesPerMeasure.at(FIRST_SPECIES),notes.size());
49 }
50
51 IntVarArray Voice::getMelodicIntervals(){
52     return m_intervals_brut;
53 }
54
55 IntVarArray Voice::getHIntervals(){
56     return h_intervals;
57 }
58
59 IntVarArgs Voice::getSecondHInterval(){
60     return h_intervals.slice(2,4,h_intervals.size());
61 }
62
63 IntVarArray Voice::getMotions(){
64     return motions;
65 }
```

A.7.2 Stratum.cpp

```
1 //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 //
4
```

```cpp
#include "../../headers/Parts/Stratum.hpp"

Stratum::Stratum(Home home, int nMes, int lb, int ub) : Voice(home, nMes, lb, ub){
    //notes = IntVarArray(home, nMes*4-3, 0, 127);
    //h_intervals = IntVarArray(home, notes.size(), -PERFECT_OCTAVE, PERFECT_OCTAVE);
    //m_intervals_brut = IntVarArray(home, notes.size()-1, -PERFECT_OCTAVE, PERFECT_OCTAVE);
}

//if the stratum is in the upper strata, this constructor will create the h_intervals to the lowest stratum
Stratum::Stratum(Home home, int nMes, int lb, int ub, IntVarArray lowestNotes) : Stratum(home, nMes, lb, ub){
    for(int i = 0; i < h_intervals.size(); i++){
        rel(home, (h_intervals[i])==((notes[i]-lowestNotes[i])%12));
    }
}

//this constructor applies 3rd voice specific constraints which apply to the upper strata
Stratum::Stratum(Home home, int nMes, int lb, int ub, IntVarArray lowestNotes, int nV) : Stratum(home, nMes, lb, ub, lowestNotes){

    //1.H3 (formerly G8) Last chord can only consist of notes of the harmonic triad
    if (activeConstraints[STRATUM_UPPER_1H3]) {
        dom(home, expr(home, abs(h_intervals[h_intervals.size()-1])), IntSet(IntArgs(TRIAD)));
    }

    //REMOVED
    // //H10 No tenths in last chord
    // if (activeConstraints[STRATUM_UPPER_1H10]) {
    //     rel(home, ((notes[notes.size()-1]-lowestNotes[lowestNotes.size()-1])>12) >> (expr(home, abs(h_intervals[h_intervals.size()-1]))!=MINOR_THIRD &&
    //         expr(home, abs(h_intervals[h_intervals.size()-1]))!=MAJOR_THIRD));
    // }

    //H12 Last chord cannot include a minor third
    if (activeConstraints[STRATUM_UPPER_1H12]) {
        rel(home, expr(home, abs(h_intervals[h_intervals.size()-1])), IRT_NQ, 3);
    }
}

Stratum::Stratum(Home home, int nMes, int lb, int ub, IntVarArray lowestNotes, int nV1, int nV2) : Stratum(home, nMes, lb, ub, lowestNotes){
    //1.H3 (formerly G8) Last chord can only consist of notes of the harmonic triad
    if (activeConstraints[STRATUM_1H3]) {
        dom(home, expr(home, abs(h_intervals[h_intervals.size()-1])), IntSet(IntArgs(TRIAD)));
    }

    //H12 Last chord cannot include a minor third
    if (activeConstraints[STRATUM_1H12]) {
        rel(home, expr(home, abs(h_intervals[h_intervals.size()-1])), IRT_NQ, 3);
    }
}


string Stratum::to_string() const {
    string text = "Notes : ";
    text += intVarArray_to_string(notes);
    text += "\n";
    text += "H intervals : ";
    text += intVarArray_to_string(h_intervals);
    text += "\n";
    text += "M intervals : ";
    text += intVarArray_to_string(m_intervals_brut);
    text += "\n";
    return text;
}


// clone constructor
Stratum::Stratum(Home home,  Stratum &s) : Voice(home, s){
    // update/clone here all variables that are not in voice
}

Stratum* Stratum::clone(Home home){
    return new Stratum(home, *this);
}

void Stratum::setNote(Home home, int index, IntVar note){
    rel(home, notes[index], IRT_EQ, note);
}
```

A.7.3 Part.cpp

```cpp
//
// Created by Damien Sprockeels on 11/06/2024.
// Extended and developed by Luc Cleenewerk and Diego de Patoul up to August 2024.
//
```

```
5
6 #include "../../headers/Parts/Part.hpp"
7
8 /// This class represents a part, so it creates all the variables associated to that part and
      posts the constraints that are species independent
9 Part::Part(Home home, int nMes, Species sp, vector<int> cf, int lb, int ub, int v_type, vector<
      int> m_costs, vector<int> g_costs, vector<int> s_costs,
10     int nV, int bm) :
11     Voice(home, nMes, lb, ub){
12     nVoices          = nV;
13     species = sp;
14     borrowMode       = bm;
15     voice_type       = v_type;
16     //lowest           = low;
17     isNotLowest       = BoolVarArray(home, nMeasures, 0, 1);
18     isHighest         = BoolVarArray(home, nMeasures, 0, 1);
19
20     borrowed_scale = get_all_notes_from_scale(cf[0]%12, BORROWED_SCALE);
21     // D tection automatique de la gamme    partir du cantus firmus complet, plut t
22     // que de toujours imposer MAJOR_SCALE. La premi re note donne la tonique et
23     // l'ensemble des notes du CF permet d'identifier le mode (Ionien, Dorien,
24     // Phrygien, Lydien, Mixolydien, Aeolien, Locrien, mineur harmonique, mineur
25     // m lodique). En cas d'ambigu t  on retient la gamme la plus g n rale.
26     vector<int> detected_scale = detect_scale_for_cf(cf);
27     scale = get_all_notes_from_scale(cf[0]%12, detected_scale);
28     chromatic_scale = get_all_notes_from_scale(cf[0]%12, CHROMATIC_SCALE);
29     cp_range = {};
30
31     secondCost = m_costs[0];
32     thirdCost = m_costs[1];
33     fourthCost = m_costs[2];
34     tritoneCost = m_costs[3];
35     fifthCost = m_costs[4];
36     sixthCost = m_costs[5];
37     seventhCost = m_costs[6];
38     octaveCost = m_costs[7];
39
40     borrowCost = g_costs[0];
41     h_fifthCost = g_costs[1];
42     h_octaveCost = g_costs[2];
43     succCost = g_costs[3];
44     varietyCost = g_costs[4];
45     triadCost = g_costs[5];
46     directMoveCost = g_costs[6];
47     penultCost = g_costs[7];
48
49     penultSixthCost = s_costs[0];
50     cambiataCost = s_costs[1];
51     mSkipCost = s_costs[2];
52     triad3rdCost = s_costs[3];
53     m2ZeroCost = s_costs[4];
54     syncopationCost = s_costs[5];
55     prefSlider = s_costs[6];
56
57     directCost = 0;
58     obliqueCost = 1;
59     contraryCost = 2;
60 }
61
62 string Part::to_string() const{
63     string part = "Part characteristics :\n";
64     part += "Part costs : \n";
65     part += intVarArray_to_string(costs);
66     part += "\n";
67     return part;
68 }
69
70
71
72
73 Part::Part(Home home, Part& s) : Voice(home, s) {
74     nVoices = s.nVoices;
75     borrowMode = s.borrowMode;
76
77     borrowed_scale = s.borrowed_scale;
78     scale = s.scale;
79     chromatic_scale = s.chromatic_scale;
80     species = s.species;
81     voice_type = s.voice_type;
82
83     cp_range = s.cp_range;
84
85     domain = s.domain;
86     off_domain = s.off_domain;
87
88     secondCost = s.secondCost;
89     thirdCost = s.thirdCost;
90     fourthCost = s.fourthCost;
91     tritoneCost = s.tritoneCost;
92     fifthCost = s.fifthCost;
```

```
93      sixthCost = s.sixthCost;
94      seventhCost = s.seventhCost;
95      octaveCost = s.octaveCost;
96
97      borrowCost = s.borrowCost;
98      h_fifthCost = s.h_fifthCost;
99      h_octaveCost = s.h_octaveCost;
100     succCost = s.succCost;
101     varietyCost = s.varietyCost;
102     triadCost = s.triadCost;
103     directMoveCost = s.directMoveCost;
104     penultCost = s.penultCost;
105
106     penultSixthCost = s.penultSixthCost;
107     cambiataCost = s.cambiataCost;
108     mSkipCost = s.mSkipCost;
109     triad3rdCost = s.triad3rdCost;
110     m2ZeroCost = s.m2ZeroCost;
111     syncopationCost = s.syncopationCost;
112     prefSlider = s.prefSlider;
113
114     directCost = s.directCost;
115     obliqueCost = s.obliqueCost;
116     contraryCost = s.contraryCost;
117
118     cost_names = s.cost_names;
119
120     toCombineCosts = s.toCombineCosts;
121     toCombineCostNames = s.toCombineCostNames;
122
123     melodicDegreeCost.update(home, s.melodicDegreeCost);
124     fifthCostArray.update(home, s.fifthCostArray);
125     octaveCostArray.update(home, s.octaveCostArray);
126     is_off.update(home, s.is_off);
127     offCostArray.update(home, s.offCostArray);
128     costs.update(home, s.costs);
129     varietyCostArray.update(home, s.varietyCostArray);
130     directCostArray.update(home, s.directCostArray);
131     isConsonance.update(home, s.isConsonance);
132     isNotLowest.update(home, s.isNotLowest);
133     isHighest.update(home, s.isHighest);
134
135     firstSpeciesHarmonicIntervals.update(home, s.firstSpeciesHarmonicIntervals); // Interval for
        the first note of each measure
136     firstSpeciesNotesCp.update(home, s.firstSpeciesNotesCp);
137     firstSpeciesMelodicIntervals.update(home, s.firstSpeciesMelodicIntervals);
138     firstSpeciesMotions.update(home, s.firstSpeciesMotions);
139     firstSpeciesMotionCosts.update(home, s.firstSpeciesMotionCosts);
140
141     isDiminution.update(home, s.isDiminution);
142     penultCostArray.update(home, s.penultCostArray);
143     secondSpeciesMotions.update(home, s.secondSpeciesMotions);
144     secondSpeciesMelodicIntervals.update(home, s.secondSpeciesMelodicIntervals);
145     secondSpeciesRealMotions.update(home, s.secondSpeciesRealMotions);
146
147     is5QNArray.update(home, s.is5QNArray);
148     thirdSpeciesHarmonicIntervals.update(home, s.thirdSpeciesHarmonicIntervals); // Interval for
        the each measure
149     thirdSpeciesMelodicIntervals.update(home, s.thirdSpeciesMelodicIntervals);
150     cambiataCostArray.update(home, s.cambiataCostArray);
151
152     isNoSyncopeArray.update(home, s.isNoSyncopeArray);
153     snycopeCostArray.update(home, s.snycopeCostArray);
154
155     speciesArray.update(home, s.speciesArray);
156
157     toCombineCosts.update(home, s.toCombineCosts);
158
159     relaxationCostArray.update(home, s.relaxationCostArray);
160 }
161
162 // Virtual clone function
163 Part* Part::clone(Home home) {
164     return new Part(home, *this);
165 }
166
167 IntVarArray Part::getBranchingNotes(){
168     return notes;
169 }
170
171 IntVarArray Part::getPartNotes(){
172     return notes;
173 }
174
175 int Part::getSuccCost(){
176     return succCost;
177 }
178
179 IntVarArray Part::getFirstHInterval(){
180     return h_intervals;
```

```
}

IntVarArray Part::getMotions(){
    return motions;
}

IntVarArray Part::getFirstMInterval(){
    return m_intervals_brut;
}

IntVarArray Part::getCosts(){
    return costs;
}

int Part::getVarietyCost(){
    return varietyCost;
}

IntVar Part::getVarietyArray(int idx){
    return varietyCostArray[idx];
}

int Part::getHIntervalSize(){
    return h_intervals.size();
}

int Part::getTriadCost(){
    return triadCost;
}

int Part::getSecondCost(){
    return secondCost;
}
int Part::getThirdCost(){
    return thirdCost;
}
int Part::getFourthCost(){
    return fourthCost;
}
int Part::getTritoneCost(){
    return tritoneCost;
}
int Part::getFifthCost(){
    return fifthCost;
}
int Part::getSixthCost(){
    return sixthCost;
}
int Part::getSeventhCost(){
    return seventhCost;
}
int Part::getOctaveCost(){
    return octaveCost;
}

IntVarArray Part::getMelodicDegreeCost(){
    return melodicDegreeCost;
}

vector<string> Part::getCostNames(){
    return cost_names;
}

BoolVarArray Part::getConsonance(){
    return isConsonance;
}

void Part::add_cost(Home home, int idx, IntVarArray to_be_added, IntVarArray costs){
    int sz = to_be_added.size();
    IntVarArgs args(sz);
    for(int i = 0; i < sz; i++){
        args[i] = to_be_added[i];
    }
    rel(home, costs[idx], IRT_EQ, expr(home, sum(args)));
}

void Part::add_toCombineCost(Home home, int idx, IntVarArray to_be_added, IntVarArray costs) {
    int sz = to_be_added.size();
    IntVarArgs args(sz);
    for(int i = 0; i < sz; i++){
        args[i] = to_be_added[i];
    }
    rel(home, costs[idx], IRT_EQ, expr(home, sum(args)));
}

IntVarArray Part::getRelaxationCostArray(){
    return relaxationCostArray;
}

void Part::initRelaxationCostArray(Home home, int size){
```

```
271     relaxationCostArray = IntVarArray(home, size, 0, 1);
272 }
273
274 BoolVarArray Part::getIsNotLowest(){
275     return isNotLowest;
276 }
277
278 IntVarArray Part::getFirstSpeciesHIntervals(){
279     return firstSpeciesHarmonicIntervals;
280 }
281
282 IntVarArray Part::getFirstSpeciesNotes(){
283     return firstSpeciesNotesCp;
284 }
285
286 IntVarArray Part::getFirstSpeciesMIntervals(){
287     return firstSpeciesMelodicIntervals;
288 }
289
290 IntVarArray Part::getFifthCostArray(){
291     return fifthCostArray;
292 }
293
294 IntVarArray Part::getOctaveCostArray(){
295     return octaveCostArray;
296 }
297
298 int Part::getHFifthCost(){
299     return h_fifthCost;
300 }
301
302 int Part::getHOctaveCost(){
303     return h_octaveCost;
304 }
305
306 IntVarArray Part::getFirstSpeciesMotions(){
307     return firstSpeciesMotions;
308 }
309
310 IntVarArray Part::getDirectCostArray(){
311     return directCostArray;
312 }
313
314 int Part::getDirectCost(){
315     return directCost;
316 }
317
318 BoolVarArray Part::getIsOffArray(){
319     return is_off;
320 }
321
322 vector<int> Part::getOffDomain(){
323     return off_domain;
324 }
325
326 vector<int> Part::getDomain(){
327     return domain;
328 }
329
330 IntVarArray Part::getOffCostArray(){
331     return offCostArray;
332 }
333
334 int Part::getBorrowCost(){
335     return borrowCost;
336 }
337
338 BoolVarArray Part::getIsDiminution(){
339     return isDiminution;
340 }
341
342 int Part::getPenultCost(){
343     return penultCost;
344 }
345
346 IntVarArray Part::getPenultCostArray(){
347     return penultCostArray;
348 }
349
350 IntVarArray Part::getSecondSpeciesMotions(){
351     return secondSpeciesMotions;
352 }
353
354 IntVarArray Part::getSecondSpeciesMIntervals(){
355     return secondSpeciesMelodicIntervals;
356 }
357
358 IntVarArray Part::getSecondSpeciesRealMotions(){
359     return secondSpeciesRealMotions;
360 }
```

```
361
362 int Part::getDirectMoveCost(){
363     return directMoveCost;
364 }
365
366 BoolVarArray Part::getIs5QNArray(){
367     return is5QNArray;
368 }
369
370 IntVarArray Part::getThirdSpeciesHIntervals(){
371     return thirdSpeciesHarmonicIntervals;
372 }
373
374 IntVarArray Part::getThirdSpeciesMIntervals(){
375     return thirdSpeciesMelodicIntervals;
376 }
377
378 int Part::getCambiataCost(){
379     return cambiataCost;
380 }
381
382 IntVarArray Part::getCambiataCostArray(){
383     return cambiataCostArray;
384 }
385
386 BoolVarArray Part::getNoSyncope(){
387     return isNoSyncopeArray;
388 }
389
390 IntVarArray Part::getSyncopeCostArray(){
391     return snycopeCostArray;
392 }
393
394 IntVarArray Part::getFourthSpeciesMIntervals(){
395     return fourthSpeciesMelodicIntervals;
396 }
397
398 IntVarArray Part::getSpeciesArray(){
399     return speciesArray;
400 }
401
402 BoolVarArray Part::getIsHighest(){
403     return isHighest;
404 }
405
406 IntVarArray Part::getToCombineCosts(){
407     return toCombineCosts;
408 }
409
410 vector<string> Part::getToCombineCostNames(){
411     return toCombineCostNames;
412 }
```

A.7.4 CantusFirmus.cpp

```
1 //
2 // Created by Damien Sprockeels on 12/06/2024.
3 // Extended and developed by Luc Cleenewerk and Diego de Patoul up to August 2024.
4 //
5
6 #include "../../headers/Parts/CantusFirmus.hpp"
7
8 CantusFirmus::CantusFirmus(Home home, int size, vector<int> cf, Stratum* low, int v_type, vector
      <int> m_costs, vector<int> g_costs, vector<int> s_costs, int nV) :
9     Part(home, size, CANTUS_FIRMUS, cf, 0, 127, v_type, m_costs, g_costs, s_costs, nV, -1){
10     cf_vector = cf;
11     notes = IntVarArray(home, size, lowerBound, upperBound);
12     /// Melodic intervals for the first species notes
13     m_intervals_brut = IntVarArray(home, notes.size()-1, -PERFECT_OCTAVE, PERFECT_OCTAVE);
14     h_intervals = IntVarArray(home, size, -PERFECT_OCTAVE, PERFECT_OCTAVE);
15     ///link melodic intervals
16     for(int i = 0; i < m_intervals_brut.size(); i++)
17         rel(home, m_intervals_brut[i], IRT_EQ, expr(home, notes[i+1] - notes[i]));
18     for(int i = 0; i < size; i++){
19         rel(home, h_intervals[i], IRT_EQ, expr(home, (notes[i]-low->getFirstNotes()[i])%12));
20     }
21
22     for(int i = 0; i < size; i++)
23         rel(home, notes[i], IRT_EQ, cf_vector[i]);
24
25     motions = IntVarArray(home, notes.size()-1, IntSet{-1, CONTRARY_MOTION, OBLIQUE_MOTION,
       PARALLEL_MOTION});
26     //create motions
27
28     for(int i = 0; i < motions.size(); i++){
29         //direct motions help creation
30
```

```
        BoolVar both_up = expr(home, (m_intervals_brut[i]>0)&&(low->getMelodicIntervals()[i]>0))
    ; //if both parts are going in the same direction
        BoolVar both_stay = expr(home, (m_intervals_brut[i]==0)&&(low->getMelodicIntervals()[i
    ]==0)); //if both parts are staying
        BoolVar both_down = expr(home, (m_intervals_brut[i]<0)&&(low->getMelodicIntervals()[i
    ]<0)); //if both parts are going down
        //oblique motions help creation
        BoolVar cf_stays_1 = expr(home, (m_intervals_brut[i]>0)&&(low->getMelodicIntervals()[i
    ]==0)); //if the lowest part stays and one goes up
        BoolVar cf_stays_2 = expr(home, (m_intervals_brut[i]<0)&&(low->getMelodicIntervals()[i
    ]==0)); //if the lowest part stays and one goes down
        BoolVar cp_stays_1 = expr(home, (m_intervals_brut[i]==0)&&(low->getMelodicIntervals()[i
    ]>0)); //if the lowest part goes up and one stays
        BoolVar cp_stays_2 = expr(home, (m_intervals_brut[i]==0)&&(low->getMelodicIntervals()[i
    ]<0)); //if the lowest part goes down and one stays
        //contrary motions help creation
        BoolVar cpd_cfu = expr(home, (m_intervals_brut[i]<0)&&(low->getMelodicIntervals()[i]>0))
    ; //if the cf goes up and the cp down
        BoolVar cpu_cfd = expr(home, (m_intervals_brut[i]>0)&&(low->getMelodicIntervals()[i]<0))
    ; //if the cf goes down and the cp up

        //direct constraints
        rel(home, ((both_up || both_stay || both_down) && (isNotLowest[i]==1)) >> (motions[i]==
    PARALLEL_MOTION));
        //oblique constraints
        rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (isNotLowest[i]==1)
    ) >> (motions[i]==OBLIQUE_MOTION));
        //contrary constraints
        rel(home, ((cpd_cfu || cpu_cfd) && (isNotLowest[i]==1)) >> (motions[i]==CONTRARY_MOTION)
    );
        //bass constraints
        rel(home, (isNotLowest[i]==0) >> (motions[i]==-1));
    }



    // 1.H1 cf version (commented by Tom Lai)
    if (activeConstraints[CF_1H1]) {
        if (softConstraints[CF_1H1]) {
            // Soft version: count violations for each non-consonant interval
            IntSet consonantSet({UNISSON, MINOR_THIRD, MAJOR_THIRD, PERFECT_FIFTH, MINOR_SIXTH,
    MAJOR_SIXTH, PERFECT_OCTAVE,
                -MINOR_THIRD, -MAJOR_THIRD, -PERFECT_FIFTH, -MINOR_SIXTH, -MAJOR_SIXTH, -
    PERFECT_OCTAVE});
            initRelaxationCostArray(home, h_intervals.size());
            for(int i = 0; i < h_intervals.size(); i++){
                BoolVar inSet(home, 0, 1);
                dom(home, h_intervals[i], consonantSet, inSet);
                rel(home, (inSet == 1) >> (relaxationCostArray[i] == 0));
                rel(home, (inSet == 0) >> (relaxationCostArray[i] == 1));
            }
        } else {
            dom(home, h_intervals, IntSet({UNISSON, MINOR_THIRD, MAJOR_THIRD, PERFECT_FIFTH,
    MINOR_SIXTH, MAJOR_SIXTH, PERFECT_OCTAVE,
                -MINOR_THIRD, -MAJOR_THIRD, -PERFECT_FIFTH, -MINOR_SIXTH, -MAJOR_SIXTH, -
    PERFECT_OCTAVE}));
        }

        // disArray = IntVarArray(home, notes.size(), IntSet{0, 1});
        // // Define the set of consonant intervals
        // IntSet consonantIntervals({UNISSON, MINOR_THIRD, MAJOR_THIRD, PERFECT_FIFTH,
    MINOR_SIXTH, MAJOR_SIXTH, PERFECT_OCTAVE,
        //     -MINOR_THIRD, -MAJOR_THIRD, -PERFECT_FIFTH, -MINOR_SIXTH, -MAJOR_SIXTH, -
    PERFECT_OCTAVE});

        // // Loop through each harmonic interval
        // for (size_t i = 0; i < h_intervals.size(); i++) {
        //     // Create a Boolean variable to check if h_intervals[i] is in consonantIntervals
        //     BoolVar isConsonant(home, 0, 1);
        //     dom(home, h_intervals[i], consonantIntervals, isConsonant);
        // // Loop through each harmonic interval
        // for (size_t i = 0; i < h_intervals.size(); i++) {
        //     // Create a Boolean variable to check if h_intervals[i] is in consonantIntervals
        //     BoolVar isConsonant(home, 0, 1);
        //     dom(home, h_intervals[i], consonantIntervals, isConsonant);

        //     // If the interval is consonant, set disArray[i] to 0
        //     rel(home, isConsonant >> (disArray[i] == 0));

        //     // Otherwise, set disArray[i] to H1_1_cost
        //     rel(home, !isConsonant >> (disArray[i] == 1));
        // }
    }

    if(nV==TWO_VOICES){
        /// 1.H2 from Thibault: The first harmonic interval must be a perfect consonance
        if (activeConstraints[CF_1H2_2V]) {
            dom(home, h_intervals[0], IntSet(IntArgs(PERFECT_CONSONANCES)));
        }

```

```
103         /// 1.H3 from Thibault: The last harmonic interval must be a perfect consonance
104         if (activeConstraints[CF_1H3_2V]) {
105             dom(home, h_intervals[h_intervals.size()-1], IntSet(IntArgs(PERFECT_CONSONANCES)));
106         }
107 
108         //H7,H8 cf version
109         if (activeConstraints[CF_1H7_2V]) {
110             // rel(home, expr(home, notes[notes.size()-2] - low->getNotes()[low->getNotes().size
    ()-2]), IRT_EQ, MINOR_THIRD, Reify(isNotLowest[isNotLowest.size()-2], RM_IMP));
111             LinIntExpr penultInterval = expr(home, notes[notes.size()-2] - low->getNotes()[low->
    getNotes().size()-2]);
112             rel(home, (isNotLowest[isNotLowest.size()-2]==1) >> (penultInterval==MINOR_THIRD ||
    penultInterval==MAJOR_THIRD));
113         }
114 
115 
116         //P1 from Thibault : Perfect consonances cannot be reached by direct motion
117         if (activeConstraints[CF_1P1]) {
118             for(int j = 0; j < motions.size(); j++){
119                 rel(home, ((h_intervals[j+1]==UNISSON || h_intervals[j+1]==PERFECT_FIFTH)&&
    isNotLowest[j+1]==1) >>
120                     (motions[j]!=PARALLEL_MOTION));
121             }
122         }
123 
124     } else {
125         //H7,H8 cf version, 3v adapted
126         if (activeConstraints[CF_1H7_3V]) {
127             LinIntExpr penultInterval = (notes[notes.size()-2] - low->getNotes()[low->getNotes()
    .size()-2])%12;
128             rel(home, penultInterval==UNISSON || penultInterval==MINOR_THIRD || penultInterval==
    PERFECT_FIFTH || penultInterval==MAJOR_SIXTH);
129         }
130     }
131 
132     //No battuta
133     if (activeConstraints[CF_1P3]) {
134         P3_0_noBattuta(home, this);
135     }
136 
137     // combined costs
138     toCombineCosts = IntVarArray(home, 1, 0, 10000);
139     // toCombineCostNames = {"1H1"};
140     //need to set constraintCosts[0]
141     // add_toCombineCost(home, 0, disArray, toCombineCosts);
142 }
143 
144 string CantusFirmus::to_string() const {
145     string text = "Cantus Firmus notes : ";
146     text += intVarArray_to_string(notes);
147     text += "\n";
148     text += "H intervals brut : ";
149     text += intVarArray_to_string(h_intervals);
150     text += "\n";
151     text += "Motions : ";
152     text += intVarArray_to_string(motions);
153     text += "\n";
154     text += "is Lowest : ";
155     text += boolVarArray_to_string(isNotLowest);
156     text += "\n";
157     text += "is Highest : ";
158     text += boolVarArray_to_string(isHighest);
159     text += "\n";
160     return text;
161 }
162 
163 
164 // clone constructor
165 CantusFirmus::CantusFirmus(Home home, CantusFirmus &s) : Part(home, s){
166 
167 }
168 
169 CantusFirmus* CantusFirmus::clone(Home home){
170     return new CantusFirmus(home, *this);
171 }
172 
173 IntVarArray CantusFirmus::getFirstHInterval(){
174     return h_intervals;
175 }
176 
177 IntVarArray CantusFirmus::getMotions(){
178     return motions;
179 }
180 
181 IntVarArray CantusFirmus::getFirstMInterval(){
182     return m_intervals_brut;
183 }
184 
185 IntVarArgs CantusFirmus::getFirstNotes(){
186     return notes.slice(0, 4/4, notes.size());
```

```
}

int CantusFirmus::getHIntervalSize(){
    return h_intervals.size();
}
```

A.8 Species sources

A.8.1 FirstSpeciesCounterpoint.cpp

```
//
// Created by Damien Sprockeels on 11/06/2024.
// Extended and developed by Luc Cleenewerk and Diego de Patoul up to August 2024.
//

#include "../../headers/Parts/FirstSpeciesCounterpoint.hpp"

/**
 * GENERAL CONSTRUCTOR
 */

FirstSpeciesCounterpoint::FirstSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Species mSpecies, Stratum* low, CantusFirmus* c,
     int v_type, vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV):
        Part(home, nMes, mSpecies, cf, lb, ub, v_type, m_costs, g_costs, s_costs, nV, bm) { /// super constructor

    motherSpecies = mSpecies;
    for(int i = lowerBound; i <= upperBound; i++){
        cp_range.push_back(i);
    }
    /*
    if borrowMode is enabled, the extended domain is extended to make the inclusion of borrowed notes possible. We can see from Fux's examples
    that he does like to borrow notes, so the borrow cost should just do the job and we should still allow borrowed notes, not outright forbid them
    */
    if(borrowMode==1){
        domain = cp_range;
        cout << "[warning] borrow mode 1 enabled     domain unfiltered" << endl;
    } else {
        domain = vector_intersection(cp_range, vector_union(scale, borrowed_scale));
    }
    off_domain = vector_difference(vector_intersection(cp_range, scale), lowerBound, upperBound);  // By default, always allow some specific notes of the major mode

    /// First species notes in the counterpoint
    firstSpeciesNotesCp = IntVarArray(home, nMeasures * notesPerMeasure.at(FIRST_SPECIES), IntSet(IntArgs(domain)));

    if(borrowMode==1 && motherSpecies==FIRST_SPECIES){
        firstSpeciesNotesCp[firstSpeciesNotesCp.size()-2] = IntVar(home, IntSet(IntArgs(vector_intersection(cp_range, chromatic_scale))));
    } else {
        firstSpeciesNotesCp[firstSpeciesNotesCp.size()-2] = IntVar(home, IntSet(IntArgs(domain)));
    }

    rel(home, firstSpeciesNotesCp, IRT_EQ, notes.slice(0,4/notesPerMeasure.at(FIRST_SPECIES), notes.size()));

    //Harmonic intervals
    for (int i = 0; i < h_intervals.size(); i++) {
        rel(home, (h_intervals[i])==(notes[i]-low->getNotes()[i])%12);
    }

    /// Harmonic intervals for the first species notes
    firstSpeciesHarmonicIntervals = IntVarArray(home, nMeasures* notesPerMeasure.at(FIRST_SPECIES), -PERFECT_OCTAVE, PERFECT_OCTAVE);

    for(int i = 0; i < firstSpeciesHarmonicIntervals.size(); i++){
        rel(home, (firstSpeciesHarmonicIntervals[i])==((firstSpeciesNotesCp[i]-low->getNotes()[i*4])%12));
    }

    rel(home, firstSpeciesHarmonicIntervals, IRT_EQ, h_intervals.slice(0,4/notesPerMeasure.at(FIRST_SPECIES),h_intervals.size()));

    /// Melodic intervals for the first species notes
    firstSpeciesMelodicIntervals = IntVarArray(home, nMeasures* notesPerMeasure.at(FIRST_SPECIES) -1, -PERFECT_OCTAVE, PERFECT_OCTAVE);
    for(int i = 0; i < firstSpeciesMelodicIntervals.size(); i++)
      rel(home, firstSpeciesMelodicIntervals[i], IRT_EQ, expr(home, firstSpeciesNotesCp[i+1] - firstSpeciesNotesCp[i]));
```

```
61
62      //create the is off array which determines if a notes is borrowed or not
63      is_off = BoolVarArray(home, notes.size(), 0, 1);
64      initializeIsOffArray(home, this);
65
66      //create off_cost array
67      offCostArray = IntVarArray(home, is_off.size(), IntSet({0, borrowCost}));
68
69      //create the melodic degree cost array
70      melodicDegreeCost = IntVarArray(home, m_intervals_brut.size(), IntSet({secondCost, thirdCost
        , fourthCost, tritoneCost, fifthCost,
71          sixthCost, seventhCost, octaveCost}));
72
73      //create motions arrays
74      firstSpeciesMotions = IntVarArray(home, nMeasures* notesPerMeasure.at(FIRST_SPECIES) -1,
        IntSet{-1, CONTRARY_MOTION, OBLIQUE_MOTION, PARALLEL_MOTION});
75      firstSpeciesMotionCosts = IntVarArray(home, firstSpeciesMotions.size(), IntSet{0, directCost
        , obliqueCost, contraryCost});
76
77      //create motions
78
79      for(int i = 0; i < firstSpeciesMotions.size(); i++){
80
81          //direct motions help creation
82          BoolVar both_up = expr(home, (firstSpeciesMelodicIntervals[i]>0)&&(low->
        getMelodicIntervals()[i]>0)); //if both parts are going in the same direction
83          BoolVar both_stay = expr(home, (firstSpeciesMelodicIntervals[i]==0)&&(low->
        getMelodicIntervals()[i]==0)); //if both parts are staying
84          BoolVar both_down = expr(home, (firstSpeciesMelodicIntervals[i]<0)&&(low->
        getMelodicIntervals()[i]<0)); //if both parts are going down
85          //oblique motions help creation
86          BoolVar cf_stays_1 = expr(home, (firstSpeciesMelodicIntervals[i]>0)&&(low->
        getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes up
87          BoolVar cf_stays_2 = expr(home, (firstSpeciesMelodicIntervals[i]<0)&&(low->
        getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes down
88          BoolVar cp_stays_1 = expr(home, (firstSpeciesMelodicIntervals[i]==0)&&(low->
        getMelodicIntervals()[i]>0)); //if the lowest part goes up and one stays
89          BoolVar cp_stays_2 = expr(home, (firstSpeciesMelodicIntervals[i]==0)&&(low->
        getMelodicIntervals()[i]<0)); //if the lowest part goes down and one stays
90          //contrary motions help creation
91          BoolVar cpd_cfu = expr(home, (firstSpeciesMelodicIntervals[i]<0)&&(low->
        getMelodicIntervals()[i]>0)); //if the cf goes up and the cp down
92          BoolVar cpu_cfd = expr(home, (firstSpeciesMelodicIntervals[i]>0)&&(low->
        getMelodicIntervals()[i]<0)); //if the cf goes down and the cp up
93
94          //direct constraints
95          rel(home, ((both_up || both_stay || both_down) && (this->isNotLowest[i]==1)) >> (
        firstSpeciesMotions[i]==PARALLEL_MOTION));
96          rel(home, ((both_up || both_stay || both_down) && (this->isNotLowest[i]==1)) >> (
        firstSpeciesMotionCosts[i]==directCost));
97          //oblique constraints
98          rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (this->isNotLowest[
        i]==1)) >> (firstSpeciesMotions[i]==OBLIQUE_MOTION));
99          rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (this->isNotLowest[
        i]==1)) >> (firstSpeciesMotionCosts[i]==obliqueCost));
100         //contrary constraints
101         rel(home, ((cpd_cfu || cpu_cfd) && (this->isNotLowest[i]==1)) >> (firstSpeciesMotions[i
        ]==CONTRARY_MOTION));
102         rel(home, ((cpd_cfu || cpu_cfd) && (this->isNotLowest[i]==1)) >> (
        firstSpeciesMotionCosts[i]==contraryCost));
103         //bass constraints
104         rel(home, (this->isNotLowest[i]==0) >> (firstSpeciesMotions[i]==-1));
105         rel(home, (this->isNotLowest[i]==0) >> (firstSpeciesMotionCosts[i]==0));
106
107     }
108
109     // create pefectConsArray
110     fifthCostArray = IntVarArray(home, h_intervals.size(), IntSet({0, h_fifthCost}));
111     octaveCostArray = IntVarArray(home, h_intervals.size(), IntSet({0, h_octaveCost}));
112
113     isConsonance = BoolVarArray(home, h_intervals.size(), 0, 1);
114     //this loop checks that every harmonic interval is a consonance or not
115     for(int i = 0; i < isConsonance.size(); i++){
116         rel(home, expr(home, (h_intervals[i]==UNISSON)||(h_intervals[i]==MINOR_THIRD)||(
        h_intervals[i]==MAJOR_THIRD)||(h_intervals[i]==PERFECT_FIFTH)||
117                 (h_intervals[i]==MINOR_SIXTH)||(h_intervals[i]==MAJOR_SIXTH)||(h_intervals[i]==
        PERFECT_OCTAVE)), IRT_EQ, isConsonance[i]);
118     }
119     /// General rules
120
121     //G4 :
122     if (activeConstraints[SP1_G4]) {
123         G4_counterpointMustBeInTheSameKey(home, this);
124     }
125
126     // G7 : melodic intervals should be small (works for 1st, 2nd and 3rd species)
127     if (activeConstraints[SP1_G7]) {
128         G7_melodicIntervalsShouldBeSmall(home, this, motherSpecies);
129     }
130
```

```
131     /// Harmonic rules
132     /// H1 from Thibault: All harmonic intervals must be consonances
133     if (activeConstraints[SP1_1H1]) {
134         H1_1_harmonicIntervalsAreConsonances(home, this);
135         // disArray = IntVarArray(home, firstSpeciesHarmonicIntervals.size(), IntSet{0, 1});
136         // // Define the set of consonant intervals
137         // IntSet consonantIntervals({UNISSON, MINOR_THIRD, MAJOR_THIRD, PERFECT_FIFTH,
    MINOR_SIXTH, MAJOR_SIXTH, PERFECT_OCTAVE,
138         //     -MINOR_THIRD, -MAJOR_THIRD, -PERFECT_FIFTH, -MINOR_SIXTH, -MAJOR_SIXTH, -
    PERFECT_OCTAVE});
139 
140         // // Loop through each harmonic interval
141         // for (size_t i = 0; i < firstSpeciesHarmonicIntervals.size(); i++) {
142         //     // Create a Boolean variable to check if h_intervals[i] is in consonantIntervals
143         //     BoolVar isConsonant(home, 0, 1);
144         //     dom(home, firstSpeciesHarmonicIntervals[i], consonantIntervals, isConsonant);
145 
146         //     // If the interval is consonant, set disArray[i] to 0
147         //     rel(home, isConsonant >> (disArray[i] == 0));
148 
149         //     // Otherwise, set disArray[i] to H1_1_cost
150         //     rel(home, !isConsonant >> (disArray[i] == 1));
151         // }
152 
153     }
154 
155     //H2 and H3 are found in the TwoVoiceCounterpoint class, since these rules are 2 voice
    specific
156 
157     // H4 : applied as rule G9 in the Two, Three and FourVoice Counterpointproblem
158 
159     // H6 from Thibault : Imperfect consonances are preferred
160     if (activeConstraints[SP1_1H6]) {
161         H6_1_preferImperfectConsonances(home, this);
162     }
163 
164     // combined costs
165     toCombineCosts = IntVarArray(home, 1, 0, 10000);
166     // toCombineCostNames = {"1H1"};
167     // //need to set constraintCosts[0]
168     // add_toCombineCost(home, 0, disArray, toCombineCosts);
169 }
170 
171 /**
172  * 2 VOICES CONSTRUCTOR
173  */
174 
175 FirstSpeciesCounterpoint::FirstSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb,
    int ub, Stratum* low, CantusFirmus* c, int v_type
176     , vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV) :
177         FirstSpeciesCounterpoint(home, nMes, cf, lb, ub, FIRST_SPECIES, low, c, v_type, m_costs,
     g_costs, s_costs, bm, nV) ///call the general constructor
178 {
179     rel(home, firstSpeciesMelodicIntervals, IRT_EQ, m_intervals_brut.slice(0,4/notesPerMeasure.
    at(FIRST_SPECIES),m_intervals_brut.size()));
180 
181     //H7,H8 from Thibault : penultimate note major sixth or minor third
182     // cout << "H7_1_2v_penultimateSixthOrThird" << endl;
183     if (activeConstraints[SP1_1H7_2V]) {
184         H7_1_2v_penultimateSixthOrThird(home, this);
185     }
186 
187     //1.M2 from Thibault: Melodic intervals cannot exceed a minor sixth
188     if (activeConstraints[SP1_1M2_2V]) {
189         M1_1_2v_melodicIntervalsNotExceedMinorSixth(home, this);
190     }
191 
192     // Motion rules
193     //1.P1 from Thibault : Perfect consonances cannot be reached by direct motion
194     if (activeConstraints[SP1_1P1_2V]) {
195         P1_1_2v_noDirectMotionFromPerfectConsonance(home, this);
196     }
197 
198     //P2 from Thibault : already done when creating motions array
199 
200     //P3 from Thibault : no battuta
201     if (activeConstraints[SP1_1P3_2V]) {
202         P3_1_noBattuta(home, this);
203     }
204 
205     varietyCostArray = IntVarArray(home, 3*(firstSpeciesHarmonicIntervals.size()-2), IntSet({0,
    varietyCost}));
206 
207     costs = IntVarArray(home, 6, 0, 1000000);
208     cost_names = {"fifth", "octave", "motion", "melodic", "borrow", "variety"};
209 
210     //set cost[0] to be fifth cost
211     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure.at(
    FIRST_SPECIES), fifthCostArray.size())), costs);
212     //set cost[1] to be octave cost
```

```
213     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(0, 4/notesPerMeasure.at(
        FIRST_SPECIES), octaveCostArray.size())), costs);
214     //set cost[2] to be motion cost
215     add_cost(home, 2, firstSpeciesMotionCosts, costs);
216     //set cost[3] to be melodic cost
217     add_cost(home, 3, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
        FIRST_SPECIES), melodicDegreeCost.size())), costs);
218     //set cost[4] to be off cost
219     add_cost(home, 4, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(FIRST_SPECIES
        ), offCostArray.size())), costs);
220     //set cost[5] to be variety cost
221     add_cost(home, 5, varietyCostArray, costs);
222 }
223
224 /**
225  * 3 VOICES CONSTRUCTOR
226  */
227
228 FirstSpeciesCounterpoint::FirstSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb,
        int ub,  Stratum* low, CantusFirmus* c,  int v_type,
229     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2) :
230     FirstSpeciesCounterpoint(home, nMes, cf, lb, ub, FIRST_SPECIES, low, c, v_type, m_costs,
        g_costs, s_costs, bm, nV2) ///call the general constructor
231 {
232     rel(home, firstSpeciesMelodicIntervals, IRT_EQ, m_intervals_brut.slice(0,4/notesPerMeasure.
        at(FIRST_SPECIES),m_intervals_brut.size()));
233
234     varietyCostArray = IntVarArray(home, 3*(firstSpeciesHarmonicIntervals.size()-2), IntSet({0,
        varietyCost}));
235     directCostArray = IntVarArray(home, firstSpeciesMotions.size()-1,IntSet({0, directMoveCost})
        );
236
237     //1.H7 -- after careful testing, this constraint does not work Fux's examples
238     if (activeConstraints[SP1_1H7_3V]) {
239         H7_1_3v_penultimateSixthOrThird(home, this);
240     }
241
242     //1.M2 from Thibault: Melodic intervals cannot exceed a minor sixth (also include octave?)
243     if (activeConstraints[SP1_1M2_3V]) {
244         M1_1_3v_melodicIntervalsNotExceedMinorSixth(home, this);
245     }
246
247     //1.P1
248     if (activeConstraints[SP1_1P1_3V]) {
249         P1_1_3v_noDirectMotionFromPerfectConsonance(home, this);
250     }
251
252     //1.P3 from Thibault : no battuta
253     if (activeConstraints[SP1_1P3_3V]) {
254         P3_1_noBattuta(home, this);
255     }
256
257     costs = IntVarArray(home, 7, 0, 1000000);
258     cost_names = {"borrow", "fifth", "octave", "variety", "motion", "melodic", "direct"};
259     //need to set cost[0] to be off cost
260     add_cost(home, 0, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(FIRST_SPECIES
        ), offCostArray.size())), costs);
261     //set cost[1] to be fifth cost
262     add_cost(home, 1, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure.at(
        FIRST_SPECIES), fifthCostArray.size())), costs);
263     //set cost[2] to be octave cost
264     add_cost(home, 2, IntVarArray(home, octaveCostArray.slice(0, 4/notesPerMeasure.at(
        FIRST_SPECIES), octaveCostArray.size())), costs);
265     //need to set cost[3] to be variety cost
266     add_cost(home, 3, varietyCostArray, costs);
267     //set cost[4] to be motion cost
268     add_cost(home, 4, firstSpeciesMotionCosts, costs);
269     //set cost[5] to be melodic cost
270     add_cost(home, 5, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
        FIRST_SPECIES), melodicDegreeCost.size())), costs);
271     //need to set cost[6] to be direct cost
272     add_cost(home, 6, directCostArray, costs);
273 }
274
275 /**
276  * 4 VOICES CONSTRUCTOR
277  */
278
279 FirstSpeciesCounterpoint::FirstSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb,
        int ub, Stratum* low, CantusFirmus* c,  int v_type,
280     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2, int
         nV3):
281     FirstSpeciesCounterpoint(home, nMes, cf, lb, ub, FIRST_SPECIES, low, c, v_type, m_costs,
        g_costs, s_costs, bm, nV3)
282 {
283     rel(home, firstSpeciesMelodicIntervals, IRT_EQ, m_intervals_brut.slice(0,4/notesPerMeasure.
        at(FIRST_SPECIES),m_intervals_brut.size()));
284
285     varietyCostArray = IntVarArray(home, 3*(firstSpeciesHarmonicIntervals.size()-2), IntSet({0,
        varietyCost}));
```

```
286     directCostArray = IntVarArray(home, firstSpeciesMotions.size()-1,IntSet({0, 2,
        directMoveCost}));
287
288     /// M2 from Thibault: Melodic intervals cannot exceed a minor sixth (also include octave?)
289     if (activeConstraints[SP1_1M2_4V]) {
290         M1_1_3v_melodicIntervalsNotExceedMinorSixth(home, this);
291     }
292
293     //P1 4 voices version
294     if (activeConstraints[SP1_1P1_4V]) {
295         P1_1_4v_noDirectMotionFromPerfectConsonance(home, this);
296     }
297
298     //P3 from Thibault : no battuta
299     if (activeConstraints[SP1_1P3_4V]) {
300         P3_1_noBattuta(home, this);
301     }
302
303     costs = IntVarArray(home, 7, 0, 1000000);
304     cost_names = {"borrow", "fifth", "octave", "variety", "motion", "melodic", "direct"};
305     //need to set cost[0] to be off cost
306     add_cost(home, 0, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(FIRST_SPECIES
        ), offCostArray.size())), costs);
307     //set cost[1] to be fifth cost
308     add_cost(home, 1, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure.at(
        FIRST_SPECIES), fifthCostArray.size())), costs);
309     //set cost[2] to be octave cost
310     add_cost(home, 2, IntVarArray(home, octaveCostArray.slice(0, 4/notesPerMeasure.at(
        FIRST_SPECIES), octaveCostArray.size())), costs);
311     //need to set cost[3] to be variety cost
312     add_cost(home, 3, varietyCostArray, costs);
313     //set cost[4] to be motion cost
314     add_cost(home, 4, firstSpeciesMotionCosts, costs);
315     //set cost[5] to be melodic cost
316     add_cost(home, 5, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
        FIRST_SPECIES), melodicDegreeCost.size())), costs);
317     //need to set cost[6] to be direct cost
318     add_cost(home, 6, directCostArray, costs);
319 }
320
321 /**
322  * This function returns a string with the characteristics of the counterpoint. It calls the
        to_string() method from
323  * the Part class and adds 1st species specific characteristics.
324  * @return a string representation of the current instance of the FirstSpeciesCounterpoint class
        .
325  */
326 string FirstSpeciesCounterpoint::to_string() const {
327     string text = Part::to_string() + "\nFirst species :\n";
328     text += "First species first notes: " + intVarArray_to_string(firstSpeciesNotesCp) + "\n";
329     text += "First species isLowest array : " + boolVarArray_to_string(isNotLowest) + "\n";
330     text += "First species motions : " + intVarArray_to_string(firstSpeciesMotions) + "\n";
331     text += "First species h intervals : " + intVarArray_to_string(firstSpeciesHarmonicIntervals
        ) + "\n";
332     return text;
333 }
334
335 // clone constructor
336 FirstSpeciesCounterpoint::FirstSpeciesCounterpoint(Home home, FirstSpeciesCounterpoint &s) :
        Part(home, s){
337     motherSpecies = s.motherSpecies;
338     //cantus = s.cantus;
339 }
340
341 FirstSpeciesCounterpoint* FirstSpeciesCounterpoint::clone(Home home){
342     return new FirstSpeciesCounterpoint(home, *this);
343 }
344
345 IntVarArray FirstSpeciesCounterpoint::getBranchingNotes(){
346     return firstSpeciesNotesCp;
347 }
348
349 IntVarArray FirstSpeciesCounterpoint::getFirstHInterval(){
350     return firstSpeciesHarmonicIntervals;
351 }
352
353 IntVarArray FirstSpeciesCounterpoint::getMotions(){
354     return firstSpeciesMotions;
355 }
356
357 IntVarArray FirstSpeciesCounterpoint::getFirstMInterval(){
358     return firstSpeciesMelodicIntervals;
359 }
360
361 int FirstSpeciesCounterpoint::getHIntervalSize(){
362     return firstSpeciesHarmonicIntervals.size();
363 }
```

A.8.2 SecondSpeciesCounterpoint.cpp

```
//
// Created by Damien Sprockeels on 12/06/2024.
// Extended and developed by Luc Cleenewerk and Diego de Patoul up to August 2024.
//

#include "../../headers/Parts/SecondSpeciesCounterpoint.hpp"

/**
 * Second species constructor. Does general constructing. Then calls the general first species constructor
 */
SecondSpeciesCounterpoint::SecondSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, int mSpec, Stratum* low, CantusFirmus* c, int v_type
    , vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV):
    FirstSpeciesCounterpoint(home, size, cf, lb, ub, SECOND_SPECIES, low, c, v_type, m_costs, g_costs, s_costs, bm, nV) /// super constructor. Applies all rules for the first species to the 1st note of each measure
{
    /// Second species notes in the counterpoint
    secondSpeciesNotesCp = IntVarArray(home, (nMeasures*notesPerMeasure.at(SECOND_SPECIES))-1, IntSet(IntArgs(domain)));
    if(borrowMode==1){
        secondSpeciesNotesCp[secondSpeciesNotesCp.size()-2] = IntVar(home, IntSet(IntArgs(vector_intersection(cp_range, chromatic_scale))));
    }
    rel(home, secondSpeciesNotesCp, IRT_EQ, notes.slice(0,4/notesPerMeasure.at(SECOND_SPECIES),(notes.size())));
    /// Harmonic intervals for the second species notes
    secondSpeciesHarmonicIntervals = IntVarArray(home, (nMeasures*notesPerMeasure.at(SECOND_SPECIES))-1, -PERFECT_OCTAVE, PERFECT_OCTAVE);
    // rel(home, secondSpeciesHarmonicIntervals, IRT_EQ, h_intervals.slice(0,4/notesPerMeasure.at(SECOND_SPECIES),(h_intervals.size())));
    for(int i = 0; i < secondSpeciesHarmonicIntervals.size(); i++){
        rel(home, (secondSpeciesHarmonicIntervals[i])==((secondSpeciesNotesCp[i]-low->getNotes()[floor(i/2)*4])%12));
    }

    /// Melodic intervals for the second species notes
    secondSpeciesMelodicIntervals = IntVarArray(home, secondSpeciesNotesCp.size()-1, -MAX_STEP, MAX_STEP);

    /// link melodic intervals
    for(int i = 0; i < secondSpeciesMelodicIntervals.size(); i++)
        rel(home, secondSpeciesMelodicIntervals[i], IRT_EQ, expr(home, secondSpeciesNotesCp[i+1] - secondSpeciesNotesCp[i]));
    rel(home, secondSpeciesMelodicIntervals, IRT_EQ, m_intervals_brut.slice(0,4/notesPerMeasure.at(SECOND_SPECIES),m_intervals_brut.size()));

    secondSpeciesArsisArray = IntVarArray(home, firstSpeciesMelodicIntervals.size(), -MAX_STEP, MAX_STEP);
    for(int i = 0; i < secondSpeciesArsisArray.size()-1; i++){
        rel(home, secondSpeciesArsisArray[i], IRT_EQ, expr(home, (secondSpeciesNotesCp[(i*2)+2]-secondSpeciesNotesCp[(i*2)+1]))); // modified by Tom Lai
    }
    rel(home, secondSpeciesArsisArray[secondSpeciesArsisArray.size()-1], IRT_EQ, expr(home, secondSpeciesNotesCp[secondSpeciesNotesCp.size()-2]-
        secondSpeciesNotesCp[secondSpeciesNotesCp.size()-1]));

    secondSpeciesMotions = IntVarArray(home, (nMeasures* notesPerMeasure.at(FIRST_SPECIES) -1), IntSet{-1, CONTRARY_MOTION, OBLIQUE_MOTION, PARALLEL_MOTION});
    secondSpeciesMotionCosts = IntVarArray(home, (nMeasures* notesPerMeasure.at(FIRST_SPECIES) -1), IntSet{0, directCost, obliqueCost, contraryCost});

    //the second motions are between the two arsis notes as are the first motions between thesis notes. this way of doing it takes the correct
    //m intervals between two arsis notes and then calculates the motion with respect to the cantusFirmus
    for(int i = 0; i < secondSpeciesArsisArray.size(); i++){

        //direct motions help creation
        BoolVar both_up = expr(home, (secondSpeciesArsisArray[i]>0)&&(low->getMelodicIntervals()[i]>0)); //if both parts are going in the same direction
        BoolVar both_stay = expr(home, (secondSpeciesArsisArray[i]==0)&&(low->getMelodicIntervals()[i]==0)); //if both parts are staying
        BoolVar both_down = expr(home, (secondSpeciesArsisArray[i]<0)&&(low->getMelodicIntervals()[i]<0)); //if both parts are going down
        //oblique motions help creation
        BoolVar cf_stays_1 = expr(home, (secondSpeciesArsisArray[i]>0)&&(low->getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes up
        BoolVar cf_stays_2 = expr(home, (secondSpeciesArsisArray[i]<0)&&(low->getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes down
        BoolVar cp_stays_1 = expr(home, (secondSpeciesArsisArray[i]==0)&&(low->getMelodicIntervals()[i]>0)); //if the lowest part goes up and one stays
        BoolVar cp_stays_2 = expr(home, (secondSpeciesArsisArray[i]==0)&&(low->getMelodicIntervals()[i]<0)); //if the lowest part goes down and one stays
        //contrary motions help creation
        BoolVar cpd_cfu = expr(home, (secondSpeciesArsisArray[i]<0)&&(low->getMelodicIntervals()
```

```
    [i]>0)); //if the cf goes up and the cp down
61      BoolVar cpu_cfd = expr(home, (secondSpeciesArsisArray[i]>0)&&(low->getMelodicIntervals()
    [i]<0)); //if the cf goes down and the cp up
62
63          //direct constraints
64          rel(home, ((both_up || both_stay || both_down) && (isNotLowest[i]==1)) >> (
    secondSpeciesMotions[i]==PARALLEL_MOTION));
65          rel(home, ((both_up || both_stay || both_down) && (isNotLowest[i]==1)) >> (
    secondSpeciesMotionCosts[i]==directCost));
66          //oblique constraints
67          rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (isNotLowest[i]==1)
    ) >> (secondSpeciesMotions[i]==OBLIQUE_MOTION));
68          rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (isNotLowest[i]==1)
    ) >> (secondSpeciesMotionCosts[i]==obliqueCost));
69          //contrary constraints
70          rel(home, ((cpd_cfu || cpu_cfd) && (isNotLowest[i]==1)) >> (secondSpeciesMotions[i]==
    CONTRARY_MOTION));
71          rel(home, ((cpd_cfu || cpu_cfd) && (isNotLowest[i]==1)) >> (secondSpeciesMotionCosts[i
    ]==contraryCost));
72          //bass constraints
73          rel(home, (isNotLowest[i]==0) >> (secondSpeciesMotions[i]==-1));
74          rel(home, (isNotLowest[i]==0) >> (secondSpeciesMotionCosts[i]==0));
75      }
76
77      //create real motions. we have the thesis-thesis and arsis-arsis motions. now we need the
        successive m intervals in meas, meaning thesis-arsis inside
78      //a measure. to get this, we iterate in increments of 2 starting from i=0
79      secondSpeciesRealMotions = IntVarArray(home, secondSpeciesMotions.size(), IntSet{-1,
        CONTRARY_MOTION, OBLIQUE_MOTION, PARALLEL_MOTION});
80      secondSpeciesRealMotionCosts = IntVarArray(home, secondSpeciesRealMotions.size(), IntSet{0,
        directCost, obliqueCost, contraryCost});
81      for(int i = 0; i < secondSpeciesRealMotions.size(); i++){
82              rel(home, (expr(home, abs(secondSpeciesMelodicIntervals[i*2])>4)==1) >> (
    secondSpeciesRealMotions[i]==secondSpeciesMotions[i]));
83              rel(home, (expr(home, abs(secondSpeciesMelodicIntervals[i*2])>4)==0) >> (
    secondSpeciesRealMotions[i]==firstSpeciesMotions[i]));
84
85              rel(home, (expr(home, abs(secondSpeciesMelodicIntervals[i*2])>4)==1) >> (
    secondSpeciesRealMotionCosts[i]==secondSpeciesMotionCosts[i]));
86              rel(home, (expr(home, abs(secondSpeciesMelodicIntervals[i*2])>4)==0) >> (
    secondSpeciesRealMotionCosts[i]==firstSpeciesMotionCosts[i]));
87      }
88
89      //create isDiminution Array
90      isDiminution = BoolVarArray(home, (nMeasures* notesPerMeasure.at(FIRST_SPECIES) -1), 0, 1);
91
92      for(int i = 0; i < isDiminution.size(); i++){
93
94          BoolVar btt3 = BoolVar(home, 0, 1);
95          BoolVar btt4 = BoolVar(home, 0, 1);
96          BoolVar bta2nd = BoolVar(home, 0, 1);
97          BoolVar btt3rd = BoolVar(home, 0, 1);
98          BoolVar bat2nd = BoolVar(home, 0, 1);
99          BoolVar band = BoolVar(home, 0, 1);
100
101         rel(home, expr(home, abs(firstSpeciesMelodicIntervals[i])), IRT_EQ, 3, Reify(btt3));
102         rel(home, expr(home, abs(firstSpeciesMelodicIntervals[i])), IRT_EQ, 4, Reify(btt4));
103         rel(home, expr(home, abs(secondSpeciesMelodicIntervals[i*2])), IRT_LQ, 2, Reify(bta2nd))
    ;
104         rel(home, expr(home, abs(secondSpeciesMelodicIntervals[i*2+1])), IRT_LQ, 2, Reify(bat2nd
    ));
105         rel(home, btt3, BOT_OR, btt4, btt3rd);
106         rel(home, bta2nd, BOT_AND, btt3rd, band);
107         rel(home, band, BOT_AND, bat2nd, isDiminution[i]);
108
109     }
110
111     penultCostArray = IntVarArray(home, 1, IntSet({0, penultCost}));
112     /// Constraints
113
114     // 2.H2 : Arsis harmonies cannot be dissonant except if there is a diminution.
115     if (activeConstraints[SP2_2H2]) {
116         if (softConstraints[SP2_2H2]) {
117             H2_2_arsisHarmoniesCannotBeDissonant_soft(home, this);
118         } else {
119             H2_2_arsisHarmoniesCannotBeDissonant(home, this);
120         }
121     }
122
123     //2.M1
124     if (activeConstraints[SP2_2M1]) {
125         M1_2_octaveLeap(home, this, low);
126     }
127
128     //2.P2 : battuta adapted (2.P3 ? Tom Lai)
129     if (activeConstraints[SP2_2P3]) {
130         P3_2_noBattuta(home, this);
131     }
132
133     //1.M2
```

```
134     if (activeConstraints[SP1_1M2_2V]) {
135         for (size_t i = 0; i < secondSpeciesMelodicIntervals.size(); i++)
136         {
137             rel(home, (abs(secondSpeciesMelodicIntervals[i]) <= 8) || (abs(
    secondSpeciesMelodicIntervals[i]) == 12)); // can be negative !!
138         }
139 
140     }
141 
142     // combined costs
143     toCombineCosts = IntVarArray(home, 1, 0, 10000);
144     toCombineCostNames = {"1H1"};
145     //need to set constraintCosts[0]
146     add_toCombineCost(home, 0, disArray, toCombineCosts);
147 }
148 
149 /**
150  * 2 VOICES CONSTRUCTOR
151  */
152 SecondSpeciesCounterpoint::SecondSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb,
     int ub, Stratum* low, CantusFirmus* c, int v_type
153     , vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV) :
154     SecondSpeciesCounterpoint(home, size, cf, lb, ub, SECOND_SPECIES, low, c, v_type, m_costs,
    g_costs, s_costs, bm, nV)
155 {
156     varietyCostArray = IntVarArray(home, 3*(secondSpeciesHarmonicIntervals.size()-2), IntSet({0,
     varietyCost}));
157 
158     costs = IntVarArray(home, 7, 0, 1000000);
159     cost_names = {"fifth", "octave", "motion", "melodic", "borrow", "penult", "variety"};
160 
161     // 2.H3 : penult cost
162     if (activeConstraints[SP2_2H3_2V]) {
163         H3_2_penultimateNoteDomain(home, this);
164     }
165 
166     //can do better than this?
167     //2.M2
168     if (activeConstraints[SP2_2M2_2V]) {
169         M2_2_2v_twoConsecutiveNotesAreNotTheSame(home, this);
170     }
171 
172     //2.P1 : adapted no direct motion rule from first species to real motions
173     if (activeConstraints[SP2_2P1_2V]) {
174         P1_2_2v_noDirectMotionFromPerfectConsonance(home, this);
175     }
176 
177     //set cost[0] to be fifth cost
178     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure.at(
    SECOND_SPECIES), fifthCostArray.size())), costs);
179     //set cost[1] to be octave cost
180     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(0, 4/notesPerMeasure.at(
    SECOND_SPECIES), octaveCostArray.size())), costs);
181     //set cost[2] to be motion cost
182     add_cost(home, 2, secondSpeciesRealMotionCosts, costs);
183     //set cost[3] to be melodic cost
184     add_cost(home, 3, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
    SECOND_SPECIES), melodicDegreeCost.size())), costs);
185     //need to set cost[4] to be off cost
186     add_cost(home, 4, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(
    SECOND_SPECIES), offCostArray.size())), costs);
187     //set cost[5] to be penult sixth cost
188     add_cost(home, 5, penultCostArray, costs);
189     //set cost[6] to be variety cost
190     add_cost(home, 6, varietyCostArray, costs);
191 
192 }
193 
194 /**
195  * 3 VOICES CONSTRUCTOR
196  */
197 SecondSpeciesCounterpoint::SecondSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb,
     int ub, Stratum* low, CantusFirmus* c, int v_type,
198     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2) :
199     SecondSpeciesCounterpoint(home, size, cf, lb, ub, SECOND_SPECIES, low, c, v_type, m_costs,
    g_costs, s_costs, bm, nV2)
200 {
201     costs = IntVarArray(home, 8, 0, 1000000);
202 
203     varietyCostArray = IntVarArray(home, 3*(secondSpeciesHarmonicIntervals.size()-2), IntSet({0,
     varietyCost}));
204     directCostArray = IntVarArray(home, secondSpeciesRealMotions.size()-1,IntSet({0,
    directMoveCost}));
205 
206     // DISABLED
207     // // 2.H3 : penult cost
208     // if (activeConstraints[SP2_2H3_3V]) {
209     //     H3_2_penultimateNoteDomain(home, this);
210     // }
211 
```

```
212     //P1 3 voices version
213     if (activeConstraints[SP2_1P1_3V]) {
214         P1_2_3v_noDirectMotionFromPerfectConsonance(home, this);
215     }
216 
217     cost_names = {"borrow", "fifth", "octave", "variety", "motion", "melodic", "direct", "penult
        "};
218     //need to set cost[0] to be off cost
219     add_cost(home, 0, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(
        SECOND_SPECIES), offCostArray.size())), costs);
220     //set cost[1] to be fifth cost
221     add_cost(home, 1, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure.at(
        SECOND_SPECIES), fifthCostArray.size())), costs);
222     //set cost[2] to be octave cost
223     add_cost(home, 2, IntVarArray(home, octaveCostArray.slice(0, 4/notesPerMeasure.at(
        SECOND_SPECIES), octaveCostArray.size())), costs);
224     //need to set cost[3] to be variety cost
225     add_cost(home, 3, varietyCostArray, costs);
226     //set cost[4] to be motion cost
227     add_cost(home, 4, secondSpeciesRealMotionCosts, costs);
228     //set cost[5] to be melodic cost
229     add_cost(home, 5, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
        SECOND_SPECIES), melodicDegreeCost.size())), costs);
230     //need to set cost[6] to be direct cost
231     add_cost(home, 6, directCostArray, costs);
232     //set cost[7] to be penult sixth cost
233     add_cost(home, 7, penultCostArray, costs);
234 
235 }
236 
237 /**
238  * 4 VOICES CONSTRUCTOR
239  */
240 SecondSpeciesCounterpoint::SecondSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb,
         int ub, Stratum* low, CantusFirmus* c, int v_type,
241     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2, int
         nV3) :
242     SecondSpeciesCounterpoint(home, size, cf, lb, ub, SECOND_SPECIES, low, c, v_type, m_costs,
        g_costs, s_costs, bm, nV3)
243 {
244 
245     costs = IntVarArray(home, 8, 0, 1000000);
246     cost_names = {"fifth", "octave", "motion", "melodic", "borrow", "variety", "direct", "penult
        "};
247 
248     varietyCostArray = IntVarArray(home, 3*(secondSpeciesHarmonicIntervals.size()-2), IntSet({0,
         varietyCost}));
249     directCostArray = IntVarArray(home, secondSpeciesRealMotions.size()-1,IntSet({0, 2,
        directMoveCost}));
250 
251     // DISABLED
252     // 2.H3 : penult cost
253     // if (activeConstraints[SP2_2H3_4V]) {
254     //     H3_2_penultimateNoteDomain(home, this);
255     // }
256 
257     //P1 4 voices version
258     if (activeConstraints[SP2_1P1_4V]) {
259         P1_2_4v_noDirectMotionFromPerfectConsonance(home, this);
260     }
261 
262     //set cost[0] to be fifth cost
263     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure.at(
        SECOND_SPECIES), fifthCostArray.size())), costs);
264     //set cost[1] to be octave cost
265     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(0, 4/notesPerMeasure.at(
        SECOND_SPECIES), octaveCostArray.size())), costs);
266     //set cost[2] to be motion cost
267     add_cost(home, 2, secondSpeciesRealMotionCosts, costs);
268     //set cost[3] to be melodic cost
269     add_cost(home, 3, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
        SECOND_SPECIES), melodicDegreeCost.size())), costs);
270     //need to set cost[4] to be off cost
271     add_cost(home, 4, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(
        SECOND_SPECIES), offCostArray.size())), costs);
272     //need to set cost[5] to be variety cost
273     add_cost(home, 5, varietyCostArray, costs);
274     //need to set cost[6] to be direct cost
275     add_cost(home, 6, directCostArray, costs);
276     //set cost[7] to be penult sixth cost
277     add_cost(home, 7, penultCostArray, costs);
278 }
279 
280 string SecondSpeciesCounterpoint::to_string() const {
281     string text = "\nSecond species diminution array : " + boolVarArray_to_string(isDiminution)
        + "\n";
282     text += "Second species isLowest array : " + boolVarArray_to_string(isNotLowest) + "\n";
283     text += "First species m intervals : " + intVarArray_to_string(firstSpeciesMelodicIntervals)
         + "\n";
284     text += "Second species m intervals : " + intVarArray_to_string(
```

```
    secondSpeciesMelodicIntervals) + "\n";
285     text += "Second species h intervals : " + intVarArray_to_string(
    secondSpeciesHarmonicIntervals) + "\n";
286     text += "Second species is Consonance : " + boolVarArray_to_string(isConsonance) + "\n";
287     text += "Second species notes : " + intVarArray_to_string(notes) + "\n";
288     text += "First species : " + FirstSpeciesCounterpoint::to_string() + "\n";
289     return text;
290 }
291
292
293 SecondSpeciesCounterpoint::SecondSpeciesCounterpoint(Home home, SecondSpeciesCounterpoint &s):
    FirstSpeciesCounterpoint(home, s){
294     secondSpeciesNotesCp.update(home, s.secondSpeciesNotesCp);
295     secondSpeciesHarmonicIntervals.update(home, s.secondSpeciesHarmonicIntervals);
296     secondSpeciesMelodicIntervals.update(home, s.secondSpeciesMelodicIntervals);
297     secondSpeciesMotions.update(home, s.secondSpeciesMotions);
298     secondSpeciesMotionCosts.update(home, s.secondSpeciesMotionCosts);
299     secondSpeciesRealMotionCosts.update(home, s.secondSpeciesRealMotionCosts);
300
301     secondSpeciesArsisArray.update(home, s.secondSpeciesArsisArray);
302 }
303
304 SecondSpeciesCounterpoint* SecondSpeciesCounterpoint::clone(Home home){
305     return new SecondSpeciesCounterpoint(home, *this);
306 }
307
308 IntVarArray SecondSpeciesCounterpoint::getFirstHInterval(){
309     return firstSpeciesHarmonicIntervals;
310 }
311
312 IntVarArray SecondSpeciesCounterpoint::getMotions(){
313     return secondSpeciesMotions;
314 }
315
316 IntVarArray SecondSpeciesCounterpoint::getFirstMInterval(){
317     return firstSpeciesMelodicIntervals;
318 }
319
320 IntVarArray SecondSpeciesCounterpoint::getBranchingNotes(){
321     return secondSpeciesNotesCp;
322 }
323
324 int SecondSpeciesCounterpoint::getHIntervalSize(){
325     return secondSpeciesHarmonicIntervals.size();
326 }
```

A.8.3 ThirdSpeciesCounterpoint.cpp

```
1 //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 //
4
5 #include "../../headers/Parts/ThirdSpeciesCounterpoint.hpp"
6
7 /**
8  * Third species general constructor
9  */
10 ThirdSpeciesCounterpoint::ThirdSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb,
    int ub, int mSpec, Stratum* low, CantusFirmus* c,
11     int v_type, vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV):
12     FirstSpeciesCounterpoint(home, size, cf, lb, ub, THIRD_SPECIES, low, c, v_type, m_costs,
    g_costs, s_costs, bm, nV)
13 {
14     thirdSpeciesNotesCp = IntVarArray(home, notes.size(), IntSet(IntArgs(domain)));
15     if(borrowMode==1){
16         thirdSpeciesNotesCp[thirdSpeciesNotesCp.size()-2] = IntVar(home, IntSet(IntArgs(
    vector_intersection(cp_range, chromatic_scale))));
17     }
18     rel(home, thirdSpeciesNotesCp, IRT_EQ, notes.slice(0,4/notesPerMeasure.at(THIRD_SPECIES),(
    notes.size())));
19
20     thirdSpeciesHarmonicIntervals = IntVarArray(home, h_intervals.size(), -PERFECT_OCTAVE,
    PERFECT_OCTAVE);
21     // rel(home, thirdSpeciesHarmonicIntervals, IRT_EQ, h_intervals);
22     // for(int i = 0; i < thirdSpeciesHarmonicIntervals.size(); i++){
23     //     rel(home, (thirdSpeciesHarmonicIntervals[i])==((thirdSpeciesNotesCp[i]-low->getNotes
    ()[floor(i/4)*4])%12));
24     // }
25
26
27     thirdSpeciesMelodicIntervals = IntVarArray(home, m_intervals_brut.size(), -MAX_STEP,
    MAX_STEP);
28
29     for(int i = 0; i < thirdSpeciesMelodicIntervals.size(); i++)
30         rel(home, thirdSpeciesMelodicIntervals[i], IRT_EQ, expr(home, thirdSpeciesNotesCp[i+1] -
     thirdSpeciesNotesCp[i]));
31
```

```
32      rel(home, thirdSpeciesMelodicIntervals, IRT_EQ, m_intervals_brut.slice(0,4/notesPerMeasure.
        at(THIRD_SPECIES),m_intervals_brut.size()));
33
34      for(int i = 0; i < thirdSpeciesMelodicIntervals.size(); i++){
35          rel(home, expr(home, abs(thirdSpeciesMelodicIntervals[i])), IRT_GR, 0);
36      }
37
38      thirdSpeciesMotions = IntVarArray(home, notes.size()/4, IntSet({-1, CONTRARY_MOTION,
        OBLIQUE_MOTION, PARALLEL_MOTION}));
39      thirdSpeciesMotionCosts = IntVarArray(home, notes.size()/4, IntSet{0, directCost,
        obliqueCost, contraryCost});
40
41      for(int i = 0; i < thirdSpeciesMotions.size(); i++){
42
43          //direct motions help creation
44          BoolVar both_up = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]>0)&&(low->
        getMelodicIntervals()[i]>0)); //if both parts are going in the same direction
45          BoolVar both_stay = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]==0)&&(low->
        getMelodicIntervals()[i]==0)); //if both parts are staying
46          BoolVar both_down = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]<0)&&(low->
        getMelodicIntervals()[i]<0)); //if both parts are going down
47          //oblique motions help creation
48          BoolVar cf_stays_1 = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]>0)&&(low->
        getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes up
49          BoolVar cf_stays_2 = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]<0)&&(low->
        getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes down
50          BoolVar cp_stays_1 = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]==0)&&(low->
        getMelodicIntervals()[i]>0)); //if the lowest part goes up and one stays
51          BoolVar cp_stays_2 = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]==0)&&(low->
        getMelodicIntervals()[i]<0)); //if the lowest part goes down and one stays
52          //contrary motions help creation
53          BoolVar cpd_cfu = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]<0)&&(low->
        getMelodicIntervals()[i]>0)); //if the cf goes up and the cp down
54          BoolVar cpu_cfd = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]>0)&&(low->
        getMelodicIntervals()[i]<0)); //if the cf goes down and the cp up
55
56          //direct constraints
57          rel(home, ((both_up || both_stay || both_down) && (isNotLowest[i])) >> (
        thirdSpeciesMotions[i]==PARALLEL_MOTION));
58          rel(home, ((both_up || both_stay || both_down) && (isNotLowest[i])) >> (
        thirdSpeciesMotionCosts[i]==directCost));
59          //oblique constraints
60          rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (isNotLowest[i]))
        >> (thirdSpeciesMotions[i]==OBLIQUE_MOTION));
61          rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (isNotLowest[i]))
        >> (thirdSpeciesMotionCosts[i]==obliqueCost));
62          //contrary constraints
63          rel(home, ((cpd_cfu || cpu_cfd) && (isNotLowest[i])) >> (thirdSpeciesMotions[i]==
        CONTRARY_MOTION));
64          rel(home, ((cpd_cfu || cpu_cfd) && (isNotLowest[i])) >> (thirdSpeciesMotionCosts[i]==
        contraryCost));
65          //bass constraints
66          rel(home, (!isNotLowest[i]) >> (thirdSpeciesMotions[i]==-1));
67          rel(home, (!isNotLowest[i]) >> (thirdSpeciesMotionCosts[i]==0));
68      }
69
70      thirdSpeciesAbsMelodic = IntVarArray(home, thirdSpeciesMelodicIntervals.size(), UNISSON,
        PERFECT_OCTAVE);
71      for(int i = 0; i < thirdSpeciesAbsMelodic.size(); i++){
72          rel(home, thirdSpeciesAbsMelodic[i] == abs(thirdSpeciesMelodicIntervals[i]));
73      }
74
75      m2IntervalsArray = IntVarArray(home, thirdSpeciesNotesCp.size()-2, -MAX_STEP, MAX_STEP);
76      for(int i = 0; i < thirdSpeciesNotesCp.size()-2; i++){
77          //size-2 of notes (butlast butlast) ; from 2 onwards in notes (rest rest)
78          rel(home, m2IntervalsArray[i], IRT_EQ, expr(home, thirdSpeciesNotesCp[i+2]-
        thirdSpeciesNotesCp[i]));
79      }
80
81      is5QNArray = BoolVarArray(home, nMeasures-1, 0, 1);
82      for(int i = 0; i < is5QNArray.size()*4; i+=4){
83
84          BoolVar b1 = BoolVar(home, 0, 1);
85          BoolVar b2 = BoolVar(home, 0, 1);
86          BoolVar b3 = BoolVar(home, 0, 1);
87          BoolVar b4 = BoolVar(home, 0, 1);
88          BoolVar bb1 = BoolVar(home, 0, 1);
89          BoolVar bb2 = BoolVar(home, 0, 1);
90          BoolVar bb3 = BoolVar(home, 0, 1);
91          BoolVar bb4 = BoolVar(home, 0, 1);
92          BoolVar band1 = BoolVar(home, 0, 1);
93          BoolVar band2 = BoolVar(home, 0, 1);
94          BoolVar band3 = BoolVar(home, 0, 1);
95          BoolVar beq1 = BoolVar(home, 0, 1);
96          BoolVar beq2 = BoolVar(home, 0, 1);
97          BoolVar beq3 = BoolVar(home, 0, 1);
98
99          rel(home, thirdSpeciesAbsMelodic[i], IRT_LQ, 2, Reify(b1));
100         rel(home, thirdSpeciesAbsMelodic[i+1], IRT_LQ, 2, Reify(b2));
101         rel(home, thirdSpeciesAbsMelodic[i+2], IRT_LQ, 2, Reify(b3));
```

```
102         rel(home, thirdSpeciesAbsMelodic[i+3], IRT_LQ, 2, Reify(b4));
103
104         rel(home, thirdSpeciesMelodicIntervals[i], IRT_GQ, 0, Reify(bb1));
105         rel(home, thirdSpeciesMelodicIntervals[i+1], IRT_GQ, 0, Reify(bb2));
106         rel(home, thirdSpeciesMelodicIntervals[i+2], IRT_GQ, 0, Reify(bb3));
107         rel(home, thirdSpeciesMelodicIntervals[i+3], IRT_GQ, 0, Reify(bb4));
108
109         rel(home, b1, BOT_AND, b2, band1);
110         rel(home, b3, BOT_AND, b4, band2);
111         rel(home, band1, BOT_AND, band2, band3);
112
113         rel(home, bb1, BOT_EQV, bb2, beq1);
114         rel(home, bb3, BOT_EQV, bb4, beq2);
115         rel(home, beq1, BOT_EQV, beq2, beq3);
116         rel(home, band3, BOT_AND, beq3, is5QNArray[i/4]);
117     }
118
119     isDiminution = BoolVarArray(home, (nMeasures* notesPerMeasure.at(FIRST_SPECIES) -1), 0, 1);
120     for(int i = 0; i < isDiminution.size()*4; i+=4){
121
122         BoolVar btt3 = BoolVar(home, 0, 1);
123         BoolVar btt4 = BoolVar(home, 0, 1);
124         BoolVar bta2nd = BoolVar(home, 0, 1);
125         BoolVar btt3rd = BoolVar(home, 0, 1);
126         BoolVar bat2nd = BoolVar(home, 0, 1);
127         BoolVar band = BoolVar(home, 0, 1);
128         IntVar addition = expr(home, abs(m2IntervalsArray[i+1]));
129         rel(home, addition, IRT_EQ, 3, Reify(btt3));
130         rel(home, addition, IRT_EQ, 4, Reify(btt4));
131         rel(home, expr(home, abs(thirdSpeciesMelodicIntervals[i+1])), IRT_LQ, 2, Reify(bta2nd));
132         rel(home, expr(home, abs(thirdSpeciesMelodicIntervals[i+2])), IRT_LQ, 2, Reify(bat2nd));
133         rel(home, btt3, BOT_OR, btt4, btt3rd);
134         rel(home, bta2nd, BOT_AND, btt3rd, band);
135         rel(home, band, BOT_AND, bat2nd, isDiminution[i/4]);
136
137     }
138
139     cambiataCostArray = IntVarArray(home, nMeasures-1, IntSet({0, cambiataCost}));
140
141     m2ZeroArray = IntVarArray(home, thirdSpeciesMelodicIntervals.size()-2, IntSet({0, m2ZeroCost
    }));
142
143     //DISABELED
144     // //3.H1 : five consecutive notes by joint degree implies that the first and the third note
     are consonants
145     // if (activeConstraints[SP3_3H1]) {
146     //     H1_3_fiveConsecutiveNotesByJointDegree(home, this);
147     // }
148
149     //3.H2 : any dissonant note implies that it is surrounded by consonant notes
150     if (activeConstraints[SP3_3H2]) {
151         if (softConstraints[SP3_3H2]) {
152             H2_3_dissonanceImpliesDiminution_soft(home, this);
153         } else {
154             H2_3_dissonanceImpliesDiminution(home, this);
155         }
156     }
157     //3.H3 : cambiata cost
158     if (activeConstraints[SP3_3H3]) {
159         H3_3_cambiataCost(home, this);
160     }
161     //3.M1 : each note and its two beats further peer are preferred to be different
162     //i + i+1 + i+2
163     if (activeConstraints[SP3_3M1]) {
164         for(int i = 0; i < m2ZeroArray.size(); i++){
165             rel(home, ((thirdSpeciesMelodicIntervals[i]+thirdSpeciesMelodicIntervals[i+1]+
    thirdSpeciesMelodicIntervals[i+2])==0) >> (m2ZeroArray[i]==m2ZeroCost));
166             rel(home, ((thirdSpeciesMelodicIntervals[i]+thirdSpeciesMelodicIntervals[i+1]+
    thirdSpeciesMelodicIntervals[i+2])!=0) >> (m2ZeroArray[i]==0));
167         }
168     }
169
170     //marcel's rule
171     /*for(int i = 0; i < thirdSpeciesMelodicIntervals.size()-1; i++){
172         BoolVar bSkip = BoolVar(home, 0, 1);
173         BoolVar bMbUp = BoolVar(home, 0, 1);
174         BoolVar bMbDown = BoolVar(home, 0, 1);
175         BoolVar bContrary = BoolVar(home, 0, 1);
176
177         rel(home, expr(home, abs(thirdSpeciesMelodicIntervals[i])), IRT_GR, 2, Reify(bSkip));
178         rel(home, thirdSpeciesMelodicIntervals[i], IRT_GR, 0, Reify(bMbUp));
179         rel(home, thirdSpeciesMelodicIntervals[i+1], IRT_LE, 1, Reify(bMbDown));
180         rel(home, bMbUp, BOT_EQV, bMbDown, bContrary);
181         rel(home, thirdSpeciesMelodicIntervals[i+1], IRT_LQ, 2, Reify(bSkip, RM_IMP));
182         rel(home, bSkip, BOT_IMP, bContrary, 1);
183     }*/
184
185     //no battuta adapted for third species (1.P3 ? Tom Lai)
186     if (activeConstraints[SP3_1P3]) {
187         for(int j = 0; j < thirdSpeciesMotions.size(); j++){
```

```
188             rel(home, expr(home, thirdSpeciesMotions[j]==CONTRARY_MOTION &&
    firstSpeciesHarmonicIntervals[j+1]==0 && thirdSpeciesMelodicIntervals[(j*4)+3]<-4),
189                 BOT_AND, isNotLowest[j], 0);
190         }
191     }
192
193     // 3.M2
194     //no melodic interval between 9 and 11
195     if (activeConstraints[SP3_U1]) {
196         for(int i = 0; i < nMeasures-1; i++){
197             for (size_t j = 0; j < 4; j++)
198             {
199                 rel(home, abs(thirdSpeciesMelodicIntervals[(i*4)+j])!=MAJOR_SIXTH && abs(
    thirdSpeciesMelodicIntervals[(i*4)+j])!=MINOR_SEVENTH && abs(thirdSpeciesMelodicIntervals[(i
    *4)+j])!=MAJOR_SEVENTH);
200             }
201
202
203         }
204     }
205
206     // REMOVED UNDOCUMENTED RULE
207     // //third note of the penultimate measure must be below the fourth one
208     // if (activeConstraints[SP3_U2]) {
209     //     BoolVar isLowest(home, 0, 1);
210     //     rel(home, isNotLowest[isNotLowest.size()-2], IRT_EQ, 0, Reify(isLowest));
211     //     //third note of the penultimate measure must be more distant than a semi tone from
    the last note of the penultimate measure
212     //     rel(home, (!isLowest) >> (thirdSpeciesMelodicIntervals[thirdSpeciesMelodicIntervals.
    size()-2] > 0));
213     // }
214
215     // 3.M4
216     //second one must also be more distant than a semi tone from the last note of the
    penultimate measure
217     if (activeConstraints[SP3_U3]) {
218         rel(home, expr(home, thirdSpeciesMelodicIntervals[thirdSpeciesMelodicIntervals.size()
    -3]+thirdSpeciesMelodicIntervals[thirdSpeciesMelodicIntervals.size()-2])
219         , IRT_NQ, 1);
220     }
221
222     // combined costs
223     toCombineCosts = IntVarArray(home, 1, 0, 10000);
224     toCombineCostNames = {"1H1"};
225     //need to set constraintCosts[0]
226     add_toCombineCost(home, 0, disArray, toCombineCosts);
227 }
228
229 /**
230  * 2 VOICES CONSTRUCTOR
231  */
232 ThirdSpeciesCounterpoint::ThirdSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb,
    int ub, Stratum* low, CantusFirmus* c, int v_type,
233     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV):
234     ThirdSpeciesCounterpoint(home, size, cf, lb, ub, THIRD_SPECIES, low, c, v_type, m_costs,
    g_costs, s_costs, bm, nV)
235 {
236     //3.H4 : in the penultimate measure, if the cantusFirmus is in the upper part, then the
    h_interval of the first note should be a minor third
237     // REPLACED in twoVoiceCounterpoint.cpp
238     // if (activeConstraints[SP3_3H4_2V]) {
239     //     rel(home, (getIsNotLowest()[getIsNotLowest().size()-2]==0) >>
240     //     (expr(home, abs(h_intervals[h_intervals.size()-5]))==MINOR_THIRD));
241     // }
242     varietyCostArray = IntVarArray(home, 3*(thirdSpeciesHarmonicIntervals.size()-2), IntSet({0,
    varietyCost}));
243
244     costs = IntVarArray(home, 8, 0, 10000);
245     cost_names = {"fifth", "octave", "motion", "melodic", "borrow", "cambiata", "m2", "variety"
    };
246
247     //set cost[0] to be fifth cost
248     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure.at(
    THIRD_SPECIES), fifthCostArray.size())), costs);
249     //set cost[1] to be octave cost
250     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(0, 4/notesPerMeasure.at(
    THIRD_SPECIES), octaveCostArray.size())), costs);
251     //set cost[2] to be motion cost
252     add_cost(home, 2, thirdSpeciesMotionCosts, costs);
253     //set cost[3] to be melodic cost
254     add_cost(home, 3, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
    THIRD_SPECIES), melodicDegreeCost.size())), costs);
255     //need to set cost[4] to be off cost
256     add_cost(home, 4, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(THIRD_SPECIES
    ), offCostArray.size())), costs);
257     //need to set cost[5] to be cambiata cost
258     add_cost(home, 5, cambiataCostArray, costs);
259     //need to set cost[6] to be m2Zero cost
260     add_cost(home, 6, m2ZeroArray, costs);
261     //set cost[7] to be variety cost
```

```
262     add_cost(home, 7, varietyCostArray, costs);
263 }
264
265 /**
266  * 3 VOICES CONSTRUCTOR
267  */
268 ThirdSpeciesCounterpoint::ThirdSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb,
    int ub, Stratum* low, CantusFirmus* c, int v_type,
269     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2):
270     ThirdSpeciesCounterpoint(home, size, cf, lb, ub, THIRD_SPECIES, low, c, v_type, m_costs,
    g_costs, s_costs, bm, nV2)
271 {
272     varietyCostArray = IntVarArray(home, 3*(thirdSpeciesHarmonicIntervals.size()-2), IntSet({0,
    varietyCost}));
273     directCostArray = IntVarArray(home, thirdSpeciesMotions.size()-1,IntSet({0, directMoveCost})
    );
274
275     //1.H7,H8 adapted
276     if (activeConstraints[SP3_1H7_3V]) {
277         rel(home, thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.size()-1]==UNISSON
    ||thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.size()-1]==MINOR_THIRD||
    thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.size()-1]==PERFECT_FIFTH||
278         thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.size()-1]==MAJOR_SIXTH);
279
280     }
281
282     //3.H6 : harmonic triad should be used on the second or third beat
283     if (activeConstraints[SP3_3H6_3V]) {
284         thirdHTriadArray = IntVarArray(home, nMeasures-1, IntSet({0, triad3rdCost}));
285         for(int i = 0; i < thirdHTriadArray.size(); i++){
286             rel(home, ((thirdSpeciesHarmonicIntervals[(i*4)+1]!=UNISSON&&
    thirdSpeciesHarmonicIntervals[(i*4)+1]!=MINOR_THIRD&&thirdSpeciesHarmonicIntervals[(i*4)
    +1]!=MAJOR_THIRD&&thirdSpeciesHarmonicIntervals[(i*4)+1]!=PERFECT_FIFTH)&&
287                 (thirdSpeciesHarmonicIntervals[(i*4)+2]!=UNISSON&&thirdSpeciesHarmonicIntervals
    [(i*4)+2]!=MINOR_THIRD&&thirdSpeciesHarmonicIntervals[(i*4)+2]!=MAJOR_THIRD&&
    thirdSpeciesHarmonicIntervals[(i*4)+2]!=PERFECT_FIFTH)) >>
288                 (thirdHTriadArray[i]==triad3rdCost));
289             rel(home, ((thirdSpeciesHarmonicIntervals[(i*4)+1]==UNISSON||
    thirdSpeciesHarmonicIntervals[(i*4)+1]==MINOR_THIRD||thirdSpeciesHarmonicIntervals[(i*4)
    +1]==MAJOR_THIRD||thirdSpeciesHarmonicIntervals[(i*4)+1]==PERFECT_FIFTH)||
290                 (thirdSpeciesHarmonicIntervals[(i*4)+2]==UNISSON||thirdSpeciesHarmonicIntervals
    [(i*4)+2]==MINOR_THIRD||thirdSpeciesHarmonicIntervals[(i*4)+2]==MAJOR_THIRD||
    thirdSpeciesHarmonicIntervals[(i*4)+2]==PERFECT_FIFTH)) >>
291                 (thirdHTriadArray[i]==0));
292         }
293     }
294
295     //1.P1 3 voices version
296     if (activeConstraints[SP3_1P1_3V]) {
297         for(int j = 0; j < firstSpeciesMotions.size()-1; j++){
298             //set a cost when it is reached through direct motion, it is 0 when not
299             rel(home, (thirdSpeciesMotions[j]==2&&(firstSpeciesHarmonicIntervals[j+1]==0||
    firstSpeciesHarmonicIntervals[j+1]==7))>>
300                 (directCostArray[j]==directMoveCost));
301             rel(home, (thirdSpeciesMotions[j]!=2||(firstSpeciesHarmonicIntervals[j+1]!=0&&
    firstSpeciesHarmonicIntervals[j+1]!=7))>>
302                 (directCostArray[j]==0));
303         }
304     }
305
306
307     costs = IntVarArray(home, 10, 0, 10000);
308     cost_names = {"borrow", "fifth", "octave", "variety", "motion", "melodic", "direct", "
    cambiata", "m2", "triad3"};
309     //need to set cost[0] to be off cost
310     add_cost(home, 0, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(THIRD_SPECIES
    ), offCostArray.size())), costs);
311     //set cost[1] to be fifth cost
312     add_cost(home, 1, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure.at(
    THIRD_SPECIES), fifthCostArray.size())), costs);
313     //set cost[2] to be octave cost
314     add_cost(home, 2, IntVarArray(home, octaveCostArray.slice(0, 4/notesPerMeasure.at(
    THIRD_SPECIES), octaveCostArray.size())), costs);
315     //need to set cost[3] to be variety cost
316     add_cost(home, 3, varietyCostArray, costs);
317     //set cost[4] to be motion cost
318     add_cost(home, 4, thirdSpeciesMotionCosts, costs);
319     //set cost[5] to be melodic cost
320     add_cost(home, 5, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
    THIRD_SPECIES), melodicDegreeCost.size())), costs);
321     //need to set cost[6] to be direct cost
322     add_cost(home, 6, directCostArray, costs);
323     //set cost[7] to be cambiata sixth cost
324     add_cost(home, 7, cambiataCostArray, costs);
325     //set cost[8] to be m2 zero cost
326     add_cost(home, 8, m2ZeroArray, costs);
327     //set cost[9] to be triad h third cost
328     add_cost(home, 9, thirdHTriadArray, costs);
329 }
330
```

```
331 /**
332  * 4 VOICES CONSTRUCTOR
333  */
334 ThirdSpeciesCounterpoint::ThirdSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb,
    int ub, Stratum* low, CantusFirmus* c, int v_type,
335     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2, int
     nV3):
336     ThirdSpeciesCounterpoint(home, size, cf, lb, ub, THIRD_SPECIES, low, c, v_type, m_costs,
    g_costs, s_costs, bm, nV3)
337 {
338     varietyCostArray = IntVarArray(home, 3*(thirdSpeciesHarmonicIntervals.size()-2), IntSet({0,
    varietyCost}));
339     directCostArray = IntVarArray(home, thirdSpeciesMotions.size()-1,IntSet({0, 2,
    directMoveCost}));
340
341     //1.H7,H8 adapted
342     if (activeConstraints[SP3_1H7_4V]) {
343         rel(home, thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.size()-1]==UNISSON
    ||thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.size()-1]==MINOR_THIRD||
    thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.size()-1]==PERFECT_FIFTH||
344         thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.size()-1]==MAJOR_SIXTH);
345     }
346
347     //3.H6 : harmonic triad should be used on the second or third beat
348     if (activeConstraints[SP3_3H6_4V]) {
349         thirdHTriadArray = IntVarArray(home, nMeasures-1, IntSet({0, triad3rdCost}));
350         for(int i = 0; i < thirdHTriadArray.size(); i++){
351             rel(home, ((thirdSpeciesHarmonicIntervals[(i*4)+1]!=UNISSON&&
    thirdSpeciesHarmonicIntervals[(i*4)+1]!=MINOR_THIRD&&thirdSpeciesHarmonicIntervals[(i*4)
    +1]!=MAJOR_THIRD&&thirdSpeciesHarmonicIntervals[(i*4)+1]!=PERFECT_FIFTH)&&
352                 (thirdSpeciesHarmonicIntervals[(i*4)+2]!=UNISSON&&thirdSpeciesHarmonicIntervals
    [(i*4)+2]!=MINOR_THIRD&&thirdSpeciesHarmonicIntervals[(i*4)+2]!=MAJOR_THIRD&&
    thirdSpeciesHarmonicIntervals[(i*4)+2]!=PERFECT_FIFTH)) >>
353                 (thirdHTriadArray[i]==triad3rdCost));
354             rel(home, ((thirdSpeciesHarmonicIntervals[(i*4)+1]==UNISSON||
    thirdSpeciesHarmonicIntervals[(i*4)+1]==MINOR_THIRD||thirdSpeciesHarmonicIntervals[(i*4)
    +1]==MAJOR_THIRD||thirdSpeciesHarmonicIntervals[(i*4)+1]==PERFECT_FIFTH)||
355                 (thirdSpeciesHarmonicIntervals[(i*4)+2]==UNISSON||thirdSpeciesHarmonicIntervals
    [(i*4)+2]==MINOR_THIRD||thirdSpeciesHarmonicIntervals[(i*4)+2]==MAJOR_THIRD||
    thirdSpeciesHarmonicIntervals[(i*4)+2]==PERFECT_FIFTH)) >>
356                 (thirdHTriadArray[i]==0));
357         }
358     }
359
360     //1.P1 4 voices version
361     if (activeConstraints[SP3_1P1_4V]) {
362         for(int j = 0; j < firstSpeciesMotions.size()-1; j++){
363             //set a cost when it is reached through direct motion, it is 0 when not
364             rel(home, (thirdSpeciesMotions[j]==2&&(firstSpeciesHarmonicIntervals[j+1]==0||
    firstSpeciesHarmonicIntervals[j+1]==7))>>
365                 (directCostArray[j]==directMoveCost));
366             rel(home, (thirdSpeciesMotions[j]!=2||(firstSpeciesHarmonicIntervals[j+1]!=0&&
    firstSpeciesHarmonicIntervals[j+1]!=7))>>
367                 (directCostArray[j]==0));
368         }
369     }
370
371
372     costs = IntVarArray(home, 10, 0, 10000);
373     cost_names = {"borrow", "fifth", "octave", "variety", "motion", "melodic", "direct", "
    cambiata", "m2", "triad3"};
374     //need to set cost[0] to be off cost
375     add_cost(home, 0, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(THIRD_SPECIES
    ), offCostArray.size())), costs);
376     //set cost[1] to be fifth cost
377     add_cost(home, 1, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure.at(
    THIRD_SPECIES), fifthCostArray.size())), costs);
378     //set cost[2] to be octave cost
379     add_cost(home, 2, IntVarArray(home, octaveCostArray.slice(0, 4/notesPerMeasure.at(
    THIRD_SPECIES), octaveCostArray.size())), costs);
380     //need to set cost[3] to be variety cost
381     add_cost(home, 3, varietyCostArray, costs);
382     //set cost[4] to be motion cost
383     add_cost(home, 4, thirdSpeciesMotionCosts, costs);
384     //set cost[5] to be melodic cost
385     add_cost(home, 5, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
    THIRD_SPECIES), melodicDegreeCost.size())), costs);
386     //need to set cost[6] to be direct cost
387     add_cost(home, 6, directCostArray, costs);
388     //set cost[7] to be cambiata sixth cost
389     add_cost(home, 7, cambiataCostArray, costs);
390     //set cost[8] to be m2 zero cost
391     add_cost(home, 8, m2ZeroArray, costs);
392     //set cost[9] to be triad h third cost
393     add_cost(home, 9, thirdHTriadArray, costs);
394 }
395
396 string ThirdSpeciesCounterpoint::to_string() const {
397     string text = Part::to_string() + "\nThird species isConsonance intervals : \n";
398     text += boolVarArray_to_string(isConsonance) += "\n";
```

```
399     text += "Third species is Lowest : \n";
400     text += boolVarArray_to_string(isNotLowest) += "\n";
401     text += "Third species motions : \n";
402     text += intVarArray_to_string(thirdSpeciesMotions) += "\n";
403     text += "Third species m intervals : \n";
404     text += intVarArray_to_string(thirdSpeciesMelodicIntervals) += "\n";
405     text += "m intervals : \n";
406     text += intVarArray_to_string(m_intervals_brut) += "\n";
407     text += "Third species is5QN array : \n";
408     text += boolVarArray_to_string(is5QNArray) += "\n";
409     return text;
410 }
411
412
413 ThirdSpeciesCounterpoint::ThirdSpeciesCounterpoint(Home home, ThirdSpeciesCounterpoint &s):
        FirstSpeciesCounterpoint(home, s){
414     thirdSpeciesNotesCp.update(home, s.thirdSpeciesNotesCp);
415     thirdSpeciesMotions.update(home, s.thirdSpeciesMotions);
416     thirdSpeciesMotionCosts.update(home, s.thirdSpeciesMotionCosts);
417     m2IntervalsArray.update(home, s.m2IntervalsArray);
418     m2ZeroArray.update(home, s.m2ZeroArray);
419     thirdHTriadArray.update(home, s.thirdHTriadArray);
420     thirdSpeciesAbsMelodic.update(home, s.thirdSpeciesAbsMelodic);
421 }
422
423 ThirdSpeciesCounterpoint* ThirdSpeciesCounterpoint::clone(Home home){
424     return new ThirdSpeciesCounterpoint(home, *this);
425 }
426
427 IntVarArray ThirdSpeciesCounterpoint::getFirstHInterval(){
428     return firstSpeciesHarmonicIntervals;
429 }
430
431 IntVarArray ThirdSpeciesCounterpoint::getMotions(){
432     return thirdSpeciesMotions;
433 }
434
435 IntVarArray ThirdSpeciesCounterpoint::getFirstMInterval(){
436     return firstSpeciesMelodicIntervals;
437 }
438
439 IntVarArray ThirdSpeciesCounterpoint::getBranchingNotes(){
440     return thirdSpeciesNotesCp;
441 }
442
443 int ThirdSpeciesCounterpoint::getHIntervalSize(){
444     return thirdSpeciesHarmonicIntervals.size();
445 }
```

A.8.4 FourthSpeciesCounterpoint.cpp

```
1  //
2  // Created by Luc Cleenewerk and Diego de Patoul.
3  //
4
5  #include "../../headers/Parts/FourthSpeciesCounterpoint.hpp"
6
7  /**
8   * GENERAL CONSTRUCTOR
9   */
10 FourthSpeciesCounterpoint::FourthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb
       , int ub, Species mSpecies, Stratum* low, CantusFirmus* c,
11      int v_type, vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV):
12     Part(home, nMes, mSpecies, cf, lb, ub, v_type, m_costs, g_costs, s_costs, nV, bm)
13 {
14
15     for(int i = lowerBound; i <= upperBound; i++){
16         cp_range.push_back(i);
17     }
18
19     /*
20     if borrowMode is enabled, the domain is extended to make the inclusion of borrowed notes
       possible. We can see from Fux's examples
21     that he does like to borrow notes, so the borrow cost should just do the job and still allow
        borrowed notes, not outright forbid them
22     */
23     if(borrowMode==1){
24         domain = cp_range;
25     } else {
26         domain = vector_intersection(cp_range, vector_union(scale, borrowed_scale)); // By
       default, always allow some specific notes (seconds, major 3th and major 4th)
27     }
28
29     off_domain = vector_difference(vector_intersection(cp_range, scale), lowerBound, upperBound)
       ;
30
```

```
31      fourthSpeciesNotesCp = IntVarArray(home, ((nMeasures*notesPerMeasure.at(FOURTH_SPECIES))-1)
        -1, IntSet(IntArgs(domain)));
32      if(borrowMode==1){
33          fourthSpeciesNotesCp[fourthSpeciesNotesCp.size()-2] = IntVar(home, IntSet(IntArgs(
        vector_intersection(cp_range, chromatic_scale))));
34      }
35      int i;
36      for (i=0; i<fourthSpeciesNotesCp.size()-3; i+=2){// 4th species basics lol
37          rel(home, fourthSpeciesNotesCp[i], IRT_EQ, fourthSpeciesNotesCp[i+1]);
38      }
39
40      sol = IntVarArray(home, fourthSpeciesNotesCp.slice(0,1,fourthSpeciesNotesCp.size()));
41
42      //Harmonic intervals
43      for (int i = 0; i < h_intervals.size(); i++) {
44          rel(home, (h_intervals[i])==(notes[i]-low->getNotes()[i])%12);
45      }
46
47      fourthSpeciesHIntervals = IntVarArray(home, ((nMeasures*notesPerMeasure.at(FOURTH_SPECIES))
        -1)-1, -PERFECT_OCTAVE, PERFECT_OCTAVE);
48
49      for(int i = 0; i < fourthSpeciesHIntervals.size(); i++){
50          rel(home, (fourthSpeciesHIntervals[i])==((fourthSpeciesNotesCp[i]-low->getNotes()[floor
        ((i+1)/2)*4])%12));
51      }
52
53
54      firstHInterval = IntVarArray(home, fourthSpeciesHIntervals.slice(1, 2,
        fourthSpeciesHIntervals.size()));
55
56      //m-intervals : arsis and next thesis
57      //m-succ-intervals : thesis and arsis in same measure
58      //m2-intervals : between n and n+2 for whole counterpoint
59      //m-all-intervals : between all notes
60
61      fourthSpeciesMelodicIntervals = IntVarArray(home, fourthSpeciesNotesCp.size()-1, -
        PERFECT_OCTAVE, PERFECT_OCTAVE);
62      for(int i = 0; i < fourthSpeciesMelodicIntervals.size(); i++){
63          rel(home, fourthSpeciesMelodicIntervals[i] == expr(home, fourthSpeciesNotesCp[i+1] -
        fourthSpeciesNotesCp[i]));
64      }
65
66      m2IntervalsArray = IntVarArray(home, fourthSpeciesNotesCp.size()-2, -PERFECT_OCTAVE,
        PERFECT_OCTAVE);
67      for(int i = 0; i < fourthSpeciesNotesCp.size()-2; i++){
68          rel(home, m2IntervalsArray[i], IRT_EQ, expr(home, fourthSpeciesNotesCp[i+2]-
        fourthSpeciesNotesCp[i]));
69      }
70
71      //create motions array
72      // Arsis motions for the fourth species correspond to the first species motions
73      firstSpeciesMotions = IntVarArray(home, nMeasures-1, IntSet{-1, CONTRARY_MOTION,
        OBLIQUE_MOTION, PARALLEL_MOTION});
74      firstSpeciesMotionCosts = IntVarArray(home, firstSpeciesMotions.size(), IntSet{0, directCost
        , obliqueCost, contraryCost});
75
76      //create motions
77      for(int i = 0; i < firstSpeciesMotions.size(); i++){
78          // cout << "i: " << i << endl;
79          int j = 1;
80          if (i == firstSpeciesMotions.size()-1) {
81              j = 0; // last motion is between the arsis of penultimate measure and the thesis of
        the last measure
82          }
83
84          //direct motions help creation
85          BoolVar both_up = expr(home, (fourthSpeciesMelodicIntervals[i*2+j]>0)&&(low->
        getMelodicIntervals()[i]>0)); //if both parts are going in the same direction
86          BoolVar both_stay = expr(home, (fourthSpeciesMelodicIntervals[i*2+j]==0)&&(low->
        getMelodicIntervals()[i]==0)); //if both parts are staying
87          BoolVar both_down = expr(home, (fourthSpeciesMelodicIntervals[i*2+j]<0)&&(low->
        getMelodicIntervals()[i]<0)); //if both parts are going
88          //oblique motions help creation
89          BoolVar cf_stays_1 = expr(home, (fourthSpeciesMelodicIntervals[i*2+j]>0)&&(low->
        getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes up
90          BoolVar cf_stays_2 = expr(home, (fourthSpeciesMelodicIntervals[i*2+j]<0)&&(low->
        getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes down
91          BoolVar cp_stays_1 = expr(home, (fourthSpeciesMelodicIntervals[i*2+j]==0)&&(low->
        getMelodicIntervals()[i]>0)); //if the lowest part goes up and one stays
92          BoolVar cp_stays_2 = expr(home, (fourthSpeciesMelodicIntervals[i*2+j]==0)&&(low->
        getMelodicIntervals()[i]<0)); //if the lowest part goes down and one stays
93          //contrary motions help creation
94          BoolVar cpd_cfu = expr(home, (fourthSpeciesMelodicIntervals[i*2+j]<0)&&(low->
        getMelodicIntervals()[i]>0)); //if the cf goes up and the cp down
95          BoolVar cpu_cfd = expr(home, (fourthSpeciesMelodicIntervals[i*2+j]>0)&&(low->
        getMelodicIntervals()[i]<0)); //if the cf goes down and the cp up
96
97          //direct constraints
98          rel(home, ((both_up || both_stay || both_down) && (this->isNotLowest[i]==1)) >> (
        firstSpeciesMotions[i]==PARALLEL_MOTION));
```

```
99          rel(home, ((both_up || both_stay || both_down) && (this->isNotLowest[i]==1)) >> (
    firstSpeciesMotionCosts[i]==directCost));
100         //oblique constraints
101         rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (this->isNotLowest[
    i]==1)) >> (firstSpeciesMotions[i]==OBLIQUE_MOTION));
102         rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (this->isNotLowest[
    i]==1)) >> (firstSpeciesMotionCosts[i]==obliqueCost));
103         //contrary constraints
104         rel(home, ((cpd_cfu || cpu_cfd) && (this->isNotLowest[i]==1)) >> (firstSpeciesMotions[i
    ]==CONTRARY_MOTION));
105         rel(home, ((cpd_cfu || cpu_cfd) && (this->isNotLowest[i]==1)) >> (
    firstSpeciesMotionCosts[i]==contraryCost));
106         //bass constraints
107         rel(home, (this->isNotLowest[i]==0) >> (firstSpeciesMotions[i]==-1));
108         rel(home, (this->isNotLowest[i]==0) >> (firstSpeciesMotionCosts[i]==0));
109     }
110
111     is_off = BoolVarArray(home, notes.size(), 0, 1);
112     for(int i = 0; i < is_off.size(); i++){ // Using Reify domain membership
113         BoolVar inOff(home, 0, 1);
114         dom(home, notes[i], IntSet(IntArgs(off_domain)), Reify(inOff, RM_EQV));
115         rel(home, is_off[i] == inOff);
116     }
117
118     //create off_cost array
119     offCostArray = IntVarArray(home, is_off.size(), IntSet({0, borrowCost}));
120     //set the cost for borrowing this note
121     for(int i = 0; i < is_off.size(); i++){
122         rel(home, (is_off[i]==0) >> (offCostArray[i]==0));
123         rel(home, (is_off[i]==1) >> (offCostArray[i]==borrowCost));
124     }
125
126     melodicDegreeCost = IntVarArray(home, m_intervals_brut.size(), IntSet({secondCost, thirdCost
    , fourthCost, tritoneCost, fifthCost,
127         sixthCost, seventhCost, octaveCost}));
128
129     // create pefectConsArray
130     fifthCostArray = IntVarArray(home, h_intervals.size(), IntSet({0, h_fifthCost}));
131     octaveCostArray = IntVarArray(home, h_intervals.size(), IntSet({0, h_octaveCost}));
132
133     isConsonance = BoolVarArray(home, h_intervals.size(), 0, 1);
134     for(int i = 0; i < isConsonance.size(); i++){
135         rel(home, expr(home, (abs(h_intervals[i])==UNISSON)||(abs(h_intervals[i])==MINOR_THIRD)
    ||(abs(h_intervals[i])==MAJOR_THIRD)||(abs(h_intervals[i])==PERFECT_FIFTH)||
136             (abs(h_intervals[i])==MINOR_SIXTH)||(abs(h_intervals[i])==MAJOR_SIXTH)||(abs(
    h_intervals[i])==PERFECT_OCTAVE)), IRT_EQ, isConsonance[i]);
137     }
138
139     isNoSyncopeArray = BoolVarArray(home, nMeasures-1, 0, 1);
140     for(int i = 0; i < isNoSyncopeArray.size(); i++){
141         rel(home, fourthSpeciesMelodicIntervals[(i*2)], IRT_NQ, 0, Reify(isNoSyncopeArray[i]));
142     }
143
144     // G6 : no chromatic melodies
145     if (activeConstraints[SP4_G6]) {
146         for(int i = 0; i < m_intervals_brut.size()-1; i+=4/notesPerMeasure.at(FOURTH_SPECIES)){
147             rel(home, expr(home, m_intervals_brut[i]==1), BOT_AND, expr(home, m_intervals_brut[i
    +1]==1), 0);
148             rel(home, expr(home, m_intervals_brut[i]==-1), BOT_AND, expr(home, m_intervals_brut[
    i+1]==-1), 0);
149         }
150     }
151
152     // G7 : melodic intervals should be small (works for 1st, 2nd and 3rd species)
153     if (activeConstraints[SP4_G7]) {
154         int idx = 0;
155         for(int i = 0; i < m_intervals_brut.size(); i+=4/notesPerMeasure.at(FOURTH_SPECIES)){
156             rel(home, (abs(m_intervals_brut[i])<MINOR_THIRD) >> (melodicDegreeCost[i]==
    secondCost));
157             rel(home, (abs(m_intervals_brut[i])==MINOR_THIRD || abs(m_intervals_brut[i])==
    MAJOR_THIRD) >> (melodicDegreeCost[i]==thirdCost));
158             rel(home, (abs(m_intervals_brut[i])==PERFECT_FOURTH) >> (melodicDegreeCost[i]==
    fourthCost));
159             rel(home, (abs(m_intervals_brut[i])==TRITONE) >> (melodicDegreeCost[i]==tritoneCost)
    );
160             rel(home, (abs(m_intervals_brut[i])==PERFECT_FIFTH) >> (melodicDegreeCost[i]==
    fifthCost));
161             rel(home, (abs(m_intervals_brut[i])==MINOR_SIXTH || abs(m_intervals_brut[i])==
    MAJOR_SIXTH) >> (melodicDegreeCost[i]==sixthCost));
162             rel(home, (abs(m_intervals_brut[i])==MINOR_SEVENTH || abs(m_intervals_brut[i])==
    MAJOR_SEVENTH) >> (melodicDegreeCost[i]==seventhCost));
163             rel(home, (abs(m_intervals_brut[i])==PERFECT_OCTAVE) >> (melodicDegreeCost[i]==
    octaveCost));
164             idx++;
165         }
166     }
167
168     m2ZeroArray = IntVarArray(home, ((fourthSpeciesNotesCp.size()-1)/2)-2, IntSet({0, m2ZeroCost
    }));
169     snycopeCostArray = IntVarArray(home, (fourthSpeciesMelodicIntervals.size())/2, IntSet({0,
```

```
    syncopationCost}));
170
171     //link them
172     rel(home, fourthSpeciesNotesCp, IRT_EQ, notes.slice(2, 4/notesPerMeasure.at(FOURTH_SPECIES),
         notes.size()));
173     rel(home, fourthSpeciesMelodicIntervals, IRT_EQ, m_intervals_brut.slice(2,4/notesPerMeasure.
        at(FOURTH_SPECIES),m_intervals_brut.size()));
174     // rel(home, fourthSpeciesHIntervals, IRT_EQ, h_intervals.slice(2, 4/notesPerMeasure.at(
        FOURTH_SPECIES), h_intervals.size()));
175
176     // H6 from Thibault : Imperfect consonances are preferred
177     if (activeConstraints[SP4_1H6]) {
178         for(int i = 0; i < h_intervals.size(); i++){
179
180             rel(home, octaveCostArray[i], IRT_EQ, h_octaveCost, Reify(expr(home, h_intervals[i
        ]==UNISSON), RM_PMI));
181             rel(home, octaveCostArray[i], IRT_EQ, 0, Reify(expr(home, h_intervals[i]!=UNISSON),
        RM_PMI));
182
183             rel(home, fifthCostArray[i], IRT_EQ, h_fifthCost, Reify(expr(home, h_intervals[i]==
        PERFECT_FIFTH), RM_PMI));
184             rel(home, fifthCostArray[i], IRT_EQ, 0, Reify(expr(home, h_intervals[i]!=
        PERFECT_FIFTH), RM_PMI));
185
186         }
187     }
188
189     //4.H1 : arsis harmonies must be consonant (could be the reason for 3 / 4 voice bugs)
190     if (activeConstraints[SP4_4H1]) {
191         for(int i = 2; i < h_intervals.size(); i+=4){
192             dom(home, expr(home, abs(h_intervals[i])), IntSet(CONSONANCES));
193         }
194     }
195
196     //4.M1 Arsis half notes should be the same as their next halves in thesis
197     if (activeConstraints[SP4_4M1]) {
198         for(int i = 0; i < snycopeCostArray.size(); i++){
199             rel(home, (isNoSyncopeArray[i]==1) >> (snycopeCostArray[i]==syncopationCost));
200             rel(home, (isNoSyncopeArray[i]==0) >> (snycopeCostArray[i]==0));
201         }
202     }
203
204     //4.M2 notes and two beats further are preferred to be different
205     if (activeConstraints[SP4_4M2]) {
206         for(int i = 0; i < m2ZeroArray.size(); i++){
207             rel(home, (fourthSpeciesNotesCp[(i*2)]==fourthSpeciesNotesCp[(i*2)+4]) >> (
        m2ZeroArray[i]==m2ZeroCost));
208             rel(home, (fourthSpeciesNotesCp[(i*2)]!=fourthSpeciesNotesCp[(i*2)+4]) >> (
        m2ZeroArray[i]==0));
209         }
210     }
211
212     //4.P1
213     if (activeConstraints[SP4_4P1]) {
214         for(int i = 1; i < nMeasures-2; i++){
215             rel(home, (isConsonance[(i*4)]==0) >> ((fourthSpeciesMelodicIntervals[(i*2)-1]==-
        MINOR_SECOND) ||
216                 (fourthSpeciesMelodicIntervals[(i*2)-1]==-MAJOR_SECOND)));
217         }
218     }
219
220     //4.P2
221     if (activeConstraints[SP4_4P2]) {
222         for(int i = 0; i < nMeasures-1; i++){
223             BoolVar buni = BoolVar(home, 0, 1);
224             BoolVar band = BoolVar(home, 0, 1);
225             rel(home, fourthSpeciesHIntervals[(i*2)], IRT_EQ, UNISSON, buni);
226             rel(home, expr(home, !c->getIsNotLowest()[i]), BOT_AND, buni, band);
227             rel(home, fourthSpeciesHIntervals[i*2], IRT_NQ, 1, Reify(band, RM_IMP));
228             rel(home, fourthSpeciesHIntervals[i*2], IRT_NQ, 2, Reify(band, RM_IMP));
229         }
230     }
231
232     //1.M2 Melodic intervals cannot be greater than Minor Sixth (except octave)
233     if (activeConstraints[SP4_1M2]) {
234         dom(home, fourthSpeciesMelodicIntervals, IntSet({UNISSON, -PERFECT_OCTAVE, -MINOR_SIXTH,
         -PERFECT_FIFTH, -AUGMENTED_FOURTH, -PERFECT_FOURTH,
235             -MAJOR_THIRD, -MINOR_THIRD, -MAJOR_SECOND, -MINOR_SECOND, PERFECT_OCTAVE,
        MINOR_SIXTH, PERFECT_FIFTH, AUGMENTED_FOURTH, PERFECT_FOURTH,
236             MAJOR_THIRD, MINOR_THIRD, MAJOR_SECOND, MINOR_SECOND}));
237     }
238
239     //DISABELED
240     // //no chromatic motion between 3 consecutive notes
241     // if (activeConstraints[SP4_U2]) {
242     //     for(int i = 1; i < fourthSpeciesMelodicIntervals.size()-1; i++){
243     //         BoolVar b1 = BoolVar(home, 0, 1);
244     //         BoolVar b2 = BoolVar(home, 0, 1);
245     //         BoolVar b3 = BoolVar(home, 0, 1);
246     //         BoolVar b4 = BoolVar(home, 0, 1);
```

```
247
248         //          rel(home, fourthSpeciesMelodicIntervals[i+1], IRT_EQ, 1, Reify(b1));
249         //          rel(home, m2IntervalsArray[i], IRT_EQ, 2, Reify(b2));
250         //          rel(home, b1, BOT_AND, b2, 0);
251
252         //          rel(home, fourthSpeciesMelodicIntervals[i+1], IRT_EQ, -1, Reify(b3));
253         //          rel(home, m2IntervalsArray[i], IRT_EQ, -2, Reify(b4));
254         //          rel(home, b3, BOT_AND, b4, 0);
255         //      }
256         // }
257 }
258
259 /**
260  * 2 VOICES CONSTRUCTOR
261  */
262 FourthSpeciesCounterpoint::FourthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb
        , int ub, Stratum* low, CantusFirmus* c,  int v_type,
263     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV):
264     FourthSpeciesCounterpoint(home, nMes, cf, lb, ub, FOURTH_SPECIES, low, c, v_type, m_costs,
        g_costs, s_costs, bm, nV)
265 {
266
267     //4.H2 : If the 4th species is the lowest stratum, then no hamonic seventh
268     if (activeConstraints[SP4_4H2_2V]) {
269         for(int j = 1; j < getIsNotLowest().size(); j++){
270             rel(home, (getIsNotLowest()[j]==0) >> (fourthSpeciesHIntervals[(j*2)-1]!=
        MINOR_SEVENTH && fourthSpeciesHIntervals[(j*2)-1]!=-MINOR_SEVENTH));
271             rel(home, (getIsNotLowest()[j]==0) >> (fourthSpeciesHIntervals[(j*2)-1]!=
        MAJOR_SEVENTH && fourthSpeciesHIntervals[(j*2)-1]!=-MAJOR_SEVENTH));
272         }
273     }
274
275     //4.H3 Penult note condition
276     if (activeConstraints[SP4_4H3_2V]) {
277         rel(home,(isNotLowest[isNotLowest.size()-2]==1) >> (expr(home, abs(h_intervals[
        h_intervals.size()-3]))==MAJOR_SIXTH));
278     }
279
280     //Must start with a perfect consonance (since 1.H2 applies to the first note, which is most
        likely a rest in 4th species)
281     if (activeConstraints[SP4_1H2_2V]) {
282         dom(home, fourthSpeciesHIntervals[0], IntSet({UNISSON, PERFECT_FIFTH, -PERFECT_FIFTH}));
283     }
284
285     varietyCostArray = IntVarArray(home, 3*(getHIntervalSize()-2), IntSet({0, varietyCost}));
286
287     costs = IntVarArray(home, 7, 0, 1000000);
288     cost_names = {"fifth", "octave", "melodic", "borrow", "m2", "syncopation", "variety"};
289
290     //set cost[0] to be fifth cost
291     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(2, 4/notesPerMeasure.at(
        FOURTH_SPECIES), fifthCostArray.size())), costs);
292     //set cost[1] to be octave cost
293     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(2, 4/notesPerMeasure.at(
        FOURTH_SPECIES), octaveCostArray.size())), costs);
294     //set cost[2] to be melodic cost
295     add_cost(home, 2, IntVarArray(home, melodicDegreeCost.slice(2, 4/notesPerMeasure.at(
        FOURTH_SPECIES), melodicDegreeCost.size())), costs);
296     //set cost[3] to be off cost
297     add_cost(home, 3, IntVarArray(home, offCostArray.slice(2, 4/notesPerMeasure.at(
        FOURTH_SPECIES), offCostArray.size())), costs);
298     //need to set cost[4] to be m2Zero cost
299     add_cost(home, 4, m2ZeroArray, costs);
300     //need to set cost[5] to be syncopation cost
301     add_cost(home, 5, snycopeCostArray, costs);
302     //set cost[6] to be variety cost
303     add_cost(home, 6, varietyCostArray, costs);
304 }
305
306 /**
307  * 3 VOICES CONSTRUCTOR
308  */
309 FourthSpeciesCounterpoint::FourthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb
        , int ub, Stratum* low, CantusFirmus* c,  int v_type,
310     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2):
311     FourthSpeciesCounterpoint(home, nMes, cf, lb, ub, FOURTH_SPECIES, low, c, v_type, m_costs,
        g_costs, s_costs, bm, nV2)
312 {
313     varietyCostArray = IntVarArray(home, 3*(getHIntervalSize()-2), IntSet({0, varietyCost}));
314
315     //4.P5 -- after careful testing, Fux does not seem to follow this rule in many of his
        examples. Suspended for now, but implementation left in case the decision is taken to
        reactivate it.
316     // -- It had very bad impact on generation duration, so it is recommended to test it before
        enabling it again
317     //if (activeConstraints[SP4_4P5_3V] && false) {
318     //    for(int j = 1; j < nMeasures-1; j++){
319     //        rel(home, (low->getMelodicIntervals()[j]==0)>>(expr(home, abs(
        fourthSpeciesHIntervals[(j*2)+1]))==MINOR_SECOND || expr(home, abs(fourthSpeciesHIntervals[(
        j*2)+1]))==MAJOR_SECOND
```

```
320     //            || expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==PERFECT_FOURTH || expr(
        home, abs(fourthSpeciesHIntervals[(j*2)+1]))==AUGMENTED_FOURTH ||
321     //            expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==MINOR_SEVENTH || expr(home,
         abs(fourthSpeciesHIntervals[(j*2)+1]))==MAJOR_SEVENTH));
322     //    }
323     //}
324
325
326     costs = IntVarArray(home, 7, 0, 1000000);
327     cost_names = {"fifth", "octave", "melodic", "borrow", "m2", "syncopation", "variety"};
328
329     //set cost[0] to be fifth cost
330     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(2, 4/notesPerMeasure.at(
        FOURTH_SPECIES), fifthCostArray.size())), costs);
331     //set cost[1] to be octave cost
332     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(2, 4/notesPerMeasure.at(
        FOURTH_SPECIES), octaveCostArray.size())), costs);
333     //set cost[2] to be melodic cost
334     add_cost(home, 2, IntVarArray(home, melodicDegreeCost.slice(2, 4/notesPerMeasure.at(
        FOURTH_SPECIES), melodicDegreeCost.size())), costs);
335     //set cost[3] to be off cost
336     add_cost(home, 3, IntVarArray(home, offCostArray.slice(2, 4/notesPerMeasure.at(
        FOURTH_SPECIES), offCostArray.size())), costs);
337     //need to set cost[4] to be m2Zero cost
338     add_cost(home, 4, m2ZeroArray, costs);
339     //need to set cost[5] to be syncopation cost
340     add_cost(home, 5, snycopeCostArray, costs);
341     //need to set cost[6] to be variety cost
342     add_cost(home, 6, varietyCostArray, costs);
343 }
344
345 /**
346  * 4 VOICES CONSTRUCTOR
347  */
348 FourthSpeciesCounterpoint::FourthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb
        , int ub, Stratum* low, CantusFirmus* c,  int v_type,
349     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2, int
         nV3):
350     FourthSpeciesCounterpoint(home, nMes, cf, lb, ub, FOURTH_SPECIES, low, c, v_type, m_costs,
        g_costs, s_costs, bm, nV2)
351 {
352     varietyCostArray = IntVarArray(home, 3*(getHIntervalSize()-2), IntSet({0, varietyCost}));
353
354     //4.P5 -- after careful testing, Fux does not seem to follow this rule in many of his
        examples. Suspended for now, but implementation left in case the decision is taken to
        reactivate it.
355     // -- It had very bad impact on generation duration, so it is recommended to test it before
        enabling it again
356     //if (activeConstraints[SP4_4P5_4V]) {
357     //    for(int j = 1; j < nMeasures-1; j++){
358     //        rel(home, (low->getMelodicIntervals()[j]==0)>>(expr(home, abs(
        fourthSpeciesHIntervals[(j*2)+1]))==MINOR_SECOND || expr(home, abs(fourthSpeciesHIntervals[(
        j*2)+1]))==MAJOR_SECOND
359     //            || expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==PERFECT_FOURTH || expr(
        home, abs(fourthSpeciesHIntervals[(j*2)+1]))==AUGMENTED_FOURTH ||
360     //            expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==MINOR_SEVENTH || expr(home,
         abs(fourthSpeciesHIntervals[(j*2)+1]))==MAJOR_SEVENTH));
361     //    }
362     //}
363
364
365     costs = IntVarArray(home, 7, 0, 1000000);
366     cost_names = {"fifth", "octave", "melodic", "borrow", "m2", "syncopation", "variety"};
367
368     //set cost[0] to be fifth cost
369     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(2, 4/notesPerMeasure.at(
        FOURTH_SPECIES), fifthCostArray.size())), costs);
370     //set cost[1] to be octave cost
371     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(2, 4/notesPerMeasure.at(
        FOURTH_SPECIES), octaveCostArray.size())), costs);
372     //set cost[2] to be melodic cost
373     add_cost(home, 2, IntVarArray(home, melodicDegreeCost.slice(2, 4/notesPerMeasure.at(
        FOURTH_SPECIES), melodicDegreeCost.size())), costs);
374     //set cost[3] to be off cost
375     add_cost(home, 3, IntVarArray(home, offCostArray.slice(2, 4/notesPerMeasure.at(
        FOURTH_SPECIES), offCostArray.size())), costs);
376     //need to set cost[4] to be m2Zero cost
377     add_cost(home, 4, m2ZeroArray, costs);
378     //need to set cost[5] to be syncopation cost
379     add_cost(home, 5, snycopeCostArray, costs);
380     //need to set cost[6] to be variety cost
381     add_cost(home, 6, varietyCostArray, costs);
382 }
383
384 /** j < nMeasures-1; j++){
385     //     rel(home, (low->getMelodicIntervals()[j]==0)>>(expr(home, abs(fourthSpeciesHIntervals
        [(j*2)+1]))==MINOR_SECOND || expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==MAJOR_SECOND
386     //         || expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==PERFECT_FOURTH || expr(home,
         abs(fourthSpeciesHIntervals[(j*2)+1]))==AUGMENTED_FOURTH ||
387     //         expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==MINOR_SEVENTH || expr(home, abs
```

```
    (fourthSpeciesHIntervals[(j*2)+1]))==MAJOR_SEVENTH));
    // }
 * This function returns a string with the characteristics of the counterpoint. It calls the to_string() method from
 * the Part class and adds 1st species specific characteristics.
 * @return a string representation of the current instance of the FirstSpeciesCounterpoint class.
 */
string FourthSpeciesCounterpoint::to_string() const {
    string text = Part::to_string() + "\nFourth species :\n";
    text += "No Syncope array : " + boolVarArray_to_string(isNoSyncopeArray) + "\n";
    text += "Fourth species notes : " + intVarArray_to_string(fourthSpeciesNotesCp) + "\n";
    text += "Fourth species h intervals : " + intVarArray_to_string(fourthSpeciesHIntervals) + "\n";
    text += "Fourth species m intervals : " + intVarArray_to_string(fourthSpeciesMelodicIntervals) + "\n";
    text += "is Consonance array : " + boolVarArray_to_string(isConsonance) + "\n";
    return text;
}

// clone constructor
FourthSpeciesCounterpoint::FourthSpeciesCounterpoint(Home home, FourthSpeciesCounterpoint &s) : Part(home, s){
    fourthSpeciesNotesCp.update(home, s.fourthSpeciesNotesCp);
    fourthSpeciesHIntervals.update(home, s.fourthSpeciesHIntervals);
    fourthSpeciesMelodicIntervals.update(home, s.fourthSpeciesMelodicIntervals);
    m2IntervalsArray.update(home, s.m2IntervalsArray);
    firstHInterval.update(home, s.firstHInterval);
    m2ZeroArray.update(home, s.m2ZeroArray);
    sol.update(home, s.sol);
}

FourthSpeciesCounterpoint* FourthSpeciesCounterpoint::clone(Home home){
    return new FourthSpeciesCounterpoint(home, *this);
}

IntVarArray FourthSpeciesCounterpoint::getBranchingNotes(){
    return sol;
}

IntVarArray FourthSpeciesCounterpoint::getFirstHInterval(){
    return firstHInterval;
}

IntVarArray FourthSpeciesCounterpoint::getMotions(){
    return motions;
}

IntVarArray FourthSpeciesCounterpoint::getFirstMInterval(){
    return m2IntervalsArray;
}

int FourthSpeciesCounterpoint::getHIntervalSize(){
    return fourthSpeciesHIntervals.size();
}
```

A.8.5 FifthSpeciesCounterpoint.cpp

```
//
// Created by Luc Cleenewerk and Diego de Patoul.
//

#include "../../headers/Parts/FifthSpeciesCounterpoint.hpp"

/**
 * GENERAL CONSTRUCTOR
 */
FifthSpeciesCounterpoint::FifthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Species mSpecies, Stratum* low, CantusFirmus* c,
    int v_type, vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV):
    Part(home, nMes, mSpecies, cf, lb, ub, v_type, m_costs, g_costs, s_costs, nV, bm)
{

    for(int i = lowerBound; i <= upperBound; i++){
        cp_range.push_back(i);
    }
    /*
    if borrowMode is enabled, the domain is extended to make the inclusion of borrowed notes possible. We can see from Fux's examples
    that he does like to borrow notes, so the borrow cost should just do the job and still allow borrowed notes, not outright forbid them
    */
    if(borrowMode==1){
        domain = cp_range;
    } else {
        domain = vector_intersection(cp_range, vector_union(scale, borrowed_scale)); // By default, always allow some specific notes of the major mode
```

```
26      }
27
28      off_domain = vector_difference(vector_intersection(cp_range, scale), lowerBound, upperBound)
        ;
29
30      solutionLength = notes.size();
31      speciesArray = IntVarArray(home, solutionLength, IntSet({-1, THIRD_SPECIES, FOURTH_SPECIES})
        );
32      isNthSpeciesArray = BoolVarArray(home, notes.size()*5, 0, 1);
33      isConstrainedArray = BoolVarArray(home, solutionLength, 0, 1);
34      isThirdSpeciesArray = BoolVarArray(home, notes.size(), 0, 1);
35      isFourthSpeciesArray = BoolVarArray(home, notes.size(), 0, 1);
36
37      /**
38       * CREATE THE SPECIES ARRAY
39       */
40      createSpeciesArrays(home);
41
42      fifthSpeciesNotesCp = IntVarArray(home, notes.size(), IntSet(IntArgs(domain)));
43      if(borrowMode==1){
44          fifthSpeciesNotesCp[fifthSpeciesNotesCp.size()-2] = IntVar(home, IntSet(IntArgs(
        vector_intersection(cp_range, chromatic_scale))));
45      }
46
47
48      fifthSpeciesHIntervals = IntVarArray(home, h_intervals.size(), -MAX_STEP, MAX_STEP);
49
50      firstSpeciesHarmonicIntervals = IntVarArray(home, fifthSpeciesHIntervals.slice(0, 4,
        fifthSpeciesHIntervals.size()));
51
52          //Harmonic intervals
53      for (int i = 0; i < h_intervals.size(); i++) {
54          BoolVar isFourthSpeciesNote(home, 0, 1);
55          rel(home, isFourthSpeciesArray[i], IRT_EQ, 1, Reify(isFourthSpeciesNote));
56          BoolVar isJoinedDegree(home, 0, 1);
57          if (i < h_intervals.size() - 2) {
58              rel(home, expr(home, abs(fifthSpeciesNotesCp[i+2] - fifthSpeciesNotesCp[i])), IRT_LQ
        , 2, Reify(isJoinedDegree));
59          } else {
60              rel(home, isJoinedDegree == 0); // last two notes cannot be joined
61          }
62          rel(home, (isFourthSpeciesNote == 0) >> (h_intervals[i] == (fifthSpeciesNotesCp[i] - low
        ->getNotes()[i]) % 12)); // thesis
63          rel(home, (isFourthSpeciesNote==1 && isJoinedDegree == 0) >> (h_intervals[i]==(notes[i]-
        low->getNotes()[i])%12));       // thesis
64          if (i < h_intervals.size() - 2) {
65              rel(home, (isFourthSpeciesNote==1 && isJoinedDegree == 1) >> (h_intervals[i]==(notes
        [i+2]-low->getNotes()[i])%12));       // arsis
66          }
67      }
68
69      // for(int i = 0; i < fifthSpeciesHIntervals.size(); i++){
70      //     rel(home, (fifthSpeciesHIntervals[i])==((fifthSpeciesNotesCp[i]-low->getNotes()[floor
        (i/4)*4])%12));
71      // }
72      // //rel(home, (fifthSpeciesHIntervals[0])==((fifthSpeciesNotesCp[2]-low->getNotes()[0])%12)
        );
73
74      fifthSpeciesSuccMIntervals = IntVarArray(home, m_intervals_brut.size(), -PERFECT_OCTAVE,
        PERFECT_OCTAVE);
75
76      firstSpeciesMelodicIntervals = IntVarArray(home, nMeasures-1, -PERFECT_OCTAVE,
        PERFECT_OCTAVE);
77
78      for(int i = 0; i < fifthSpeciesSuccMIntervals.size(); i++){
79          rel(home, fifthSpeciesSuccMIntervals[i] == (fifthSpeciesNotesCp[i+1]-fifthSpeciesNotesCp
        [i]));
80      }
81
82      for(int i = 0; i < firstSpeciesMelodicIntervals.size(); i++){
83          rel(home, firstSpeciesMelodicIntervals[i] == (fifthSpeciesNotesCp[(i+1)*4]-
        fifthSpeciesNotesCp[i*4]));
84      }
85
86      fifthSpeciesMIntervals = IntVarArray(home, m_intervals_brut.size(), -16, 16);
87      for(int i = 0; i < fifthSpeciesMIntervals.size(); i+=4){
88          rel(home, fifthSpeciesMIntervals[i+2] == (fifthSpeciesNotesCp[i+4]-fifthSpeciesNotesCp[i
        +2]));
89          rel(home, fifthSpeciesMIntervals[i+3] == (fifthSpeciesNotesCp[i+4]-fifthSpeciesNotesCp[i
        +3]));
90      }
91
92      fifthSpeciesMTAIntervals = IntVarArray(home, nMeasures-1, -16, 16);
93      for(int i = 0; i < fifthSpeciesMTAIntervals.size(); i++){
94          rel(home, fifthSpeciesMTAIntervals[i] == (fifthSpeciesNotesCp[(i*4)+2]-
        fifthSpeciesNotesCp[(i*4)]));
95      }
96
97      m2Len = ((nMeasures-1)*4)-1;
98      fifthSpeciesM2Intervals = IntVarArray(home, m2Len, -16, 16);
```

```
99      for(int i = 0; i < m2Len; i++){
100         rel(home, fifthSpeciesM2Intervals[i] == (fifthSpeciesNotesCp[i+2]-fifthSpeciesNotesCp[i
        ]));
101     }
102
103     fifthSpeciesMAllIntervals = IntVarArray(home, m_intervals_brut.size(), -PERFECT_OCTAVE,
        PERFECT_OCTAVE);
104     for(int i = 0; i < fifthSpeciesMAllIntervals.size(); i++){
105         rel(home, fifthSpeciesMAllIntervals[i] == (fifthSpeciesNotesCp[i+1]-fifthSpeciesNotesCp[
        i]));
106     }
107
108     fifthSpeciesMotions = IntVarArray(home, nMeasures-1, IntSet({-1, CONTRARY_MOTION,
        OBLIQUE_MOTION, PARALLEL_MOTION}));
109     fifthSpeciesMotionsCosts = IntVarArray(home, nMeasures-1, IntSet({0, contraryCost,
        obliqueCost, directCost}));
110
111     //create motions
112     for(int i = 0; i < fifthSpeciesMotions.size(); i++){
113         //direct motions help creation
114
115         BoolVar both_up = expr(home, (fifthSpeciesMIntervals[(i*4)+3]>0)&&(low->
        getMelodicIntervals()[i]>0)); //if both parts are going in the same direction
116         BoolVar both_stay = expr(home, (fifthSpeciesMIntervals[(i*4)+3]==0)&&(low->
        getMelodicIntervals()[i]==0)); //if both parts are staying
117         BoolVar both_down = expr(home, (fifthSpeciesMIntervals[(i*4)+3]<0)&&(low->
        getMelodicIntervals()[i]<0)); //if both parts are going down
118         //oblique motions help creation
119         BoolVar cf_stays_1 = expr(home, (fifthSpeciesMIntervals[(i*4)+3]>0)&&(low->
        getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes up
120         BoolVar cf_stays_2 = expr(home, (fifthSpeciesMIntervals[(i*4)+3]<0)&&(low->
        getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes down
121         BoolVar cp_stays_1 = expr(home, (fifthSpeciesMIntervals[(i*4)+3]==0)&&(low->
        getMelodicIntervals()[i]>0)); //if the lowest part goes up and one stays
122         BoolVar cp_stays_2 = expr(home, (fifthSpeciesMIntervals[(i*4)+3]==0)&&(low->
        getMelodicIntervals()[i]<0)); //if the lowest part goes down and one stays
123         //contrary motions help creation
124         BoolVar cpd_cfu = expr(home, (fifthSpeciesMIntervals[(i*4)+3]<0)&&(low->
        getMelodicIntervals()[i]>0)); //if the cf goes up and the cp down
125         BoolVar cpu_cfd = expr(home, (fifthSpeciesMIntervals[(i*4)+3]>0)&&(low->
        getMelodicIntervals()[i]<0)); //if the cf goes down and the cp up
126
127         //direct constraints
128         rel(home, ((both_up || both_stay || both_down) && (this->isNotLowest[i]==1)) >> (
        fifthSpeciesMotions[i]==PARALLEL_MOTION));
129         rel(home, ((both_up || both_stay || both_down) && (this->isNotLowest[i]==1)) >> (
        fifthSpeciesMotionsCosts[i]==directCost));
130         //oblique constraints
131         rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (this->isNotLowest[
        i]==1)) >> (fifthSpeciesMotions[i]==OBLIQUE_MOTION));
132         rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (this->isNotLowest[
        i]==1)) >> (fifthSpeciesMotionsCosts[i]==obliqueCost));
133         //contrary constraints
134         rel(home, ((cpd_cfu || cpu_cfd) && (this->isNotLowest[i]==1)) >> (fifthSpeciesMotions[i
        ]==CONTRARY_MOTION));
135         rel(home, ((cpd_cfu || cpu_cfd) && (this->isNotLowest[i]==1)) >> (
        fifthSpeciesMotionsCosts[i]==contraryCost));
136         //bass constraints
137         rel(home, (this->isNotLowest[i]==0) >> (fifthSpeciesMotions[i]==-1));
138         rel(home, (this->isNotLowest[i]==0) >> (fifthSpeciesMotionsCosts[i]==0));
139
140     }
141
142     is5QNArray = BoolVarArray(home, nMeasures-1, 0, 1);
143
144     for(int i = 0; i < is5QNArray.size()*4; i+=4){
145
146         BoolVar b1 = BoolVar(home, 0, 1);
147         BoolVar b2 = BoolVar(home, 0, 1);
148         BoolVar b3 = BoolVar(home, 0, 1);
149         BoolVar b4 = BoolVar(home, 0, 1);
150         BoolVar bb1 = BoolVar(home, 0, 1);
151         BoolVar bb2 = BoolVar(home, 0, 1);
152         BoolVar bb3 = BoolVar(home, 0, 1);
153         BoolVar bb4 = BoolVar(home, 0, 1);
154         BoolVar band1 = BoolVar(home, 0, 1);
155         BoolVar band2 = BoolVar(home, 0, 1);
156         BoolVar band3 = BoolVar(home, 0, 1);
157         BoolVar beq1 = BoolVar(home, 0, 1);
158         BoolVar beq2 = BoolVar(home, 0, 1);
159         BoolVar beq3 = BoolVar(home, 0, 1);
160
161         rel(home, expr(home, abs(fifthSpeciesSuccMIntervals[i])), IRT_LQ, 2, Reify(b1));
162         rel(home, expr(home, abs(fifthSpeciesSuccMIntervals[i+1])), IRT_LQ, 2, Reify(b2));
163         rel(home, expr(home, abs(fifthSpeciesSuccMIntervals[i+2])), IRT_LQ, 2, Reify(b3));
164         rel(home, expr(home, abs(fifthSpeciesSuccMIntervals[i+3])), IRT_LQ, 2, Reify(b4));
165
166         rel(home, fifthSpeciesSuccMIntervals[i], IRT_GQ, 0, Reify(bb1));
167         rel(home, fifthSpeciesSuccMIntervals[i+1], IRT_GQ, 0, Reify(bb2));
168         rel(home, fifthSpeciesSuccMIntervals[i+2], IRT_GQ, 0, Reify(bb3));
```

```
169         rel(home, fifthSpeciesSuccMIntervals[i+3], IRT_GQ, 0, Reify(bb4));
170
171         rel(home, b1, BOT_AND, b2, band1);
172         rel(home, b3, BOT_AND, b4, band2);
173         rel(home, band1, BOT_AND, band2, band3);
174
175         rel(home, bb1, BOT_EQV, bb2, beq1);
176         rel(home, bb3, BOT_EQV, bb4, beq2);
177         rel(home, beq1, BOT_EQV, beq2, beq3);
178         rel(home, band3, BOT_AND, beq3, is5QNArray[i/4]);
179     }
180
181     isMostlyThirdArray = BoolVarArray(home, nMeasures-1, 0, 1);
182     for(int i = 0; i < isMostlyThirdArray.size(); i++){
183         BoolVar b23 = BoolVar(home, 0, 1);
184         /**
185          * EXPLANATION :
186          * -for each measure there are 20 entries in the isNthSpeciesArray (5 possibilities for
    each note, and a measure has 4 notes)
187          * -therefor, we multiply i by 20 to always land at the start of a measure (0 is the
    first note of the first measure, 20 the first of the second
188          *  measure etc etc)
189          * -since each note has 5 entries to check for which type of species it is, to land on
    the first entry of the second note, we do 20+5
190          * -now to check if the second note is of the third species, we do 20+5+3 (second
    measure, second note, check for third species)
191          */
192         rel(home, isNthSpeciesArray[(i*20)+8], BOT_AND, isNthSpeciesArray[(i*20)+13], b23);
193         rel(home, b23, BOT_AND, isNthSpeciesArray[(i*20)+18], isMostlyThirdArray[i]);
194
195     }
196
197     isConsonance = BoolVarArray(home, notes.size(), 0, 1);
198
199     //create isConsonance array
200     for(int i = 0; i < isConsonance.size(); i++){
201         rel(home, expr(home, (h_intervals[i]==UNISSON)||(h_intervals[i]==MINOR_THIRD)||(
    h_intervals[i]==MAJOR_THIRD)||(h_intervals[i]==PERFECT_FIFTH)||
202             (h_intervals[i]==MINOR_SIXTH)||(h_intervals[i]==MAJOR_SIXTH)||(h_intervals[i]==
    PERFECT_OCTAVE)||(h_intervals[i]==-MINOR_THIRD)||
203             (h_intervals[i]==-MAJOR_THIRD)||(h_intervals[i]==-PERFECT_FIFTH)||(h_intervals[i]==-
    MINOR_SIXTH)||(h_intervals[i]==-MAJOR_SIXTH)||
204             (h_intervals[i]==-PERFECT_OCTAVE)), IRT_EQ, isConsonance[i]);
205     }
206
207     //create isThird and isFourth arrays
208     for(int i = 0; i < notes.size(); i++){
209         rel(home, isThirdSpeciesArray[i], IRT_EQ, isNthSpeciesArray[(i*5)+3]);
210         rel(home, isFourthSpeciesArray[i], IRT_EQ, isNthSpeciesArray[(i*5)+4]);
211     }
212
213     isDiminution = BoolVarArray(home, nMeasures-1, 0, 1);
214     for(int i = 0; i < isDiminution.size()*4; i+=4){
215
216         BoolVar btt3 = BoolVar(home, 0, 1);
217         BoolVar btt4 = BoolVar(home, 0, 1);
218         BoolVar bta2nd = BoolVar(home, 0, 1);
219         BoolVar btt3rd = BoolVar(home, 0, 1);
220         BoolVar bat2nd = BoolVar(home, 0, 1);
221         BoolVar band = BoolVar(home, 0, 1);
222         IntVar addition = expr(home, abs(fifthSpeciesM2Intervals[i+1]));
223         rel(home, addition, IRT_EQ, 3, Reify(btt3));
224         rel(home, addition, IRT_EQ, 4, Reify(btt4));
225         rel(home, fifthSpeciesSuccMIntervals[i+1], IRT_LQ, 2, Reify(bta2nd));
226         rel(home, fifthSpeciesSuccMIntervals[i+2], IRT_LQ, 2, Reify(bat2nd));
227         rel(home, btt3, BOT_OR, btt4, btt3rd);
228         rel(home, bta2nd, BOT_AND, btt3rd, band);
229         rel(home, band, BOT_AND, bat2nd, isDiminution[i/4]);
230
231     }
232
233     //set cambiata array
234     isNotCambiata = BoolVarArray(home, nMeasures-1, 0, 1);
235     for(int i = 0; i < isNotCambiata.size(); i++){
236         rel(home, ((fifthSpeciesHIntervals[(i*4)+1]==UNISSON || fifthSpeciesHIntervals[(i*4)
    +1]==PERFECT_FIFTH)&&(fifthSpeciesHIntervals[(i*4)+2]==UNISSON || fifthSpeciesHIntervals[(i
    *4)+2]==PERFECT_FIFTH)
237             &&(abs(fifthSpeciesSuccMIntervals[(i*4)+1])<=2)) >> (isNotCambiata[i]==1));
238         rel(home, ((fifthSpeciesHIntervals[(i*4)+1]!=UNISSON && fifthSpeciesHIntervals[(i*4)
    +1]!=PERFECT_FIFTH)||(fifthSpeciesHIntervals[(i*4)+2]!=UNISSON && fifthSpeciesHIntervals[(i
    *4)+2]!=PERFECT_FIFTH)
239             ||(abs(fifthSpeciesSuccMIntervals[(i*4)+1])>2)) >> (isNotCambiata[i]==0));
240     }
241
242     is_off = BoolVarArray(home, notes.size(), 0, 1);
243     initializeIsOffArray(home, this);
244
245     isNoSyncopeArray = BoolVarArray(home, nMeasures-1, 0, 1);
246     for(int i = 0; i < isNoSyncopeArray.size(); i++){
247         rel(home, fifthSpeciesMIntervals[(i*2)+2], IRT_NQ, 0, Reify(isNoSyncopeArray[i]));
```

```
248     }
249     rel(home, fifthSpeciesNotesCp, IRT_EQ, notes.slice(0,4/notesPerMeasure.at(FIFTH_SPECIES),(
        notes.size())));
250     rel(home, fifthSpeciesHIntervals, IRT_EQ, h_intervals.slice(0,4/notesPerMeasure.at(
        FIFTH_SPECIES),h_intervals.size()));
251     rel(home, fifthSpeciesSuccMIntervals, IRT_EQ, m_intervals_brut.slice(0,4/notesPerMeasure.at(
        FIFTH_SPECIES),m_intervals_brut.size()));
252
253     /** ===========================================================================
254      *                              HARMONIC CONSTRAINS
255      *  ===========================================================================
256      */
257
258     // 1.H1 every thesis note should be consonant
259     if (activeConstraints[SP5_H4]) {
260         for(int i = 0; i < isConsonance.size(); i+=4){ //checks every thesis note
261             rel(home, isNthSpeciesArray[(i*5)+1], BOT_IMP, isConsonance[i], 1); //first species
        check
262             rel(home, isNthSpeciesArray[(i*5)+2], BOT_IMP, isConsonance[i], 1); //second species
         check
263             rel(home, isThirdSpeciesArray[i], BOT_IMP, isConsonance[i], 1);     //third species
        check
264             if(i!=isConsonance.size()-1){
265                 //rel(home, isFourthSpeciesArray[i+2], BOT_IMP, isConsonance[i+2], 1);  //fourth
         species check
266             }
267             if(i!=0 && i!=isConsonance.size()-1){
268                 //rel(home, expr(home, isFourthSpeciesArray[i]==1 && isNoSyncopeArray[i]==0),
        BOT_IMP, isConsonance[i],1);
269             }
270         }
271
272         for(int i = 4; i < isConsonance.size()-1; i+=4){ //start at the first note of the second
         measure
273             rel(home, expr(home, isFourthSpeciesArray[i]==1 && isNoSyncopeArray[i/4]==1),
        BOT_IMP, isConsonance[i], 1);
274         }
275     }
276
277     // 3.H1 five consecutive notes by joint degree (3rd species)
278     if (activeConstraints[SP5_H5]) {
279         for(int i = 0; i < is5QNArray.size(); i++){
280             BoolVar b = BoolVar(home, 0, 1);
281             rel(home, is5QNArray[i], BOT_AND, isMostlyThirdArray[i], b);
282             rel(home, b, BOT_IMP, isConsonance[i*4+2], 1);
283         }
284     }
285
286     // [Claude Code]
287     // 3.H2 a dissonant 3rd-species weak beat (florid in quarters) must be a genuine passing
288     // tone      approached AND left by step. The original rule had two gaps: it only tied the
289     // CENTRAL weak beat (+2), leaving +1 and +3 free to carry a leapt-to dissonance; and the
290     // `|| isThirdSpeciesArray[(i*4)+2]` term made every third-species central beat trivially
291     // exempt. We now check the THREE weak beats (+1,+2,+3) of every third-species note,
292     // mirroring H2_3_..._multiVoice used by the pure 3rd species. The check is gated on
293     // isThirdSpeciesArray[j] so 1st/2nd/4th-species cells (whole/half/tied notes) are untouched
        .
294     if (activeConstraints[SP5_H6]) {
295         for(int i = 0; i < isDiminution.size(); i++){
296             for(int k = 1; k <= 3; k++){
297                 int j = i*4 + k; // weak-beat instant
298                 IntVar inMove  = expr(home, abs(notes[j]   - notes[j-1]));
299                 IntVar outMove = expr(home, abs(notes[j+1] - notes[j]));
300                 BoolVar isPassing = expr(home, inMove >= 1 && inMove <= 2 && outMove >= 1 &&
        outMove <= 2);
301                 // isThirdSpeciesArray[j] => (isConsonance[j] || isPassing)
302                 rel(home, isThirdSpeciesArray[j], BOT_IMP, expr(home, isConsonance[j] ||
        isPassing), 1);
303             }
304         }
305     }
306
307     /** ===========================================================================
308      *                              MELODIC CONSTRAINS
309      *  ===========================================================================
310      */
311
312     //no melodic interval between MAJOR_SIXTH and MAJOR_SEVENTH
313     if (activeConstraints[SP5_M1]) {
314         for(int i = 0; i < fifthSpeciesMAllIntervals.size(); i++){
315             rel(home, expr(home, abs(fifthSpeciesMAllIntervals[i])), IRT_NQ, MAJOR_SIXTH, Reify(
        expr(home, isConstrainedArray[i]==1 && isConstrainedArray[i+1]==1), RM_IMP));
316             rel(home, expr(home, abs(fifthSpeciesMAllIntervals[i])), IRT_NQ, MINOR_SEVENTH,
        Reify(expr(home, isConstrainedArray[i]==1 && isConstrainedArray[i+1]==1), RM_IMP));
317             rel(home, expr(home, abs(fifthSpeciesMAllIntervals[i])), IRT_NQ, MAJOR_SEVENTH,
        Reify(expr(home, isConstrainedArray[i]==1 && isConstrainedArray[i+1]==1), RM_IMP));
318         }
319     }
320
321     //no unisson between two consecutive notes
```

```
322     if (activeConstraints[SP5_M2]) {
323         for(int i = 0; i < nMeasures-1; i++){
324             rel(home, fifthSpeciesNotesCp[(i*4)], IRT_NQ, fifthSpeciesNotesCp[(i*4)+1], Reify(
    expr(home, (isConstrainedArray[(i*4)]==1) && (isConstrainedArray[(i*4)+1]==1)), RM_IMP));
325             rel(home, fifthSpeciesNotesCp[(i*4+1)], IRT_NQ, fifthSpeciesNotesCp[(i*4)+2], Reify(
    expr(home, (isConstrainedArray[(i*4+1)]==1) && (isConstrainedArray[(i*4)+2]==1)), RM_IMP));
326             rel(home, fifthSpeciesNotesCp[(i*4)+2], IRT_NQ, fifthSpeciesNotesCp[(i*4)+3], Reify(
    expr(home, (isConstrainedArray[(i*4)+2]==1) && (isConstrainedArray[(i*4)+3]==1)), RM_IMP));
327             rel(home, fifthSpeciesNotesCp[(i*4)+3], IRT_NQ, fifthSpeciesNotesCp[(i*4)+4], Reify(
    expr(home, (isConstrainedArray[(i*4)+3]==1) && (isConstrainedArray[(i*4)+4]==1)), RM_IMP));
328         }
329     }
330
331     //REMOVED
332     // //no more than minor sixth interval between arsis and thesis notes
333     // if (activeConstraints[SP5_M3]) {
334     //     for(int i = 0; i < nMeasures-1; i++){
335     //         rel(home, expr(home, abs(fifthSpeciesMTAIntervals[i])), IRT_NQ, MAJOR_SIXTH,
    Reify(expr(home, isConstrainedArray[i+1]==1), RM_IMP));
336     //         rel(home, expr(home, abs(fifthSpeciesMTAIntervals[i])), IRT_NQ, MINOR_SEVENTH,
    Reify(expr(home, isConstrainedArray[i+1]==1), RM_IMP));
337     //         rel(home, expr(home, abs(fifthSpeciesMTAIntervals[i])), IRT_NQ, MAJOR_SEVENTH,
    Reify(expr(home, isConstrainedArray[i+1]==1), RM_IMP));
338     //     }
339     // }
340
341
342     // 4.M2 ? no same syncopation
343     if (activeConstraints[SP5_M4]) {
344         for(int i = 1; i < nMeasures-1; i++){
345             rel(home, fifthSpeciesNotesCp[(i*4)], IRT_NQ, fifthSpeciesNotesCp[(i*4)+2],
346                 Reify(expr(home, isFourthSpeciesArray[(i*4)]==1 && isFourthSpeciesArray[(i*4)
    +2]==1 && isConstrainedArray[(i*4)+2]), RM_IMP));
347
348             rel(home, (isFourthSpeciesArray[(i*4)]==1 && isFourthSpeciesArray[(i*4)+2]==1 &&
    isConstrainedArray[(i*4)+2]==1) >> (fifthSpeciesNotesCp[(i*4)]!=fifthSpeciesNotesCp[(i*4)
    +2]));
349         }
350     }
351
352     //Third note of penult measure must be below the fourth one
353     if (activeConstraints[SP5_3M3]) {
354         rel(home, fifthSpeciesSuccMIntervals[fifthSpeciesSuccMIntervals.size()-3], IRT_GR,
    MINOR_SECOND,
355         Reify(isThirdSpeciesArray[isThirdSpeciesArray.size()-2], RM_IMP));
356     }
357     //Second and third note distant by more than one semi tone from fourth note
358     if (activeConstraints[SP5_3M4]) {
359         rel(home, expr(home, abs(fifthSpeciesM2Intervals[fifthSpeciesM2Intervals.size()-2])),
    IRT_NQ, 1,
360             Reify(isThirdSpeciesArray[isThirdSpeciesArray.size()-2], RM_IMP));
361     }
362
363
364     /** ============================================================================
365      *                              MOTION CONSTRAINS
366      *  ============================================================================
367      */
368
369     //no battuta kind of motion
370     if (activeConstraints[SP5_P1]) {
371         for(int i = 0; i < fifthSpeciesMotions.size(); i++){
372             BoolVar isCM = BoolVar(home, 0, 1);
373             BoolVar isOct = BoolVar(home, 0, 1);
374             BoolVar isCPDown = BoolVar(home, 0, 1);
375             BoolVar band1 = BoolVar(home, 0, 1);
376             BoolVar band2 = BoolVar(home, 0, 1);
377             BoolVar band3 = BoolVar(home, 0, 1);
378
379             rel(home, fifthSpeciesMotions[i], IRT_EQ, CONTRARY_MOTION, Reify(isCM));
380             rel(home, expr(home, (fifthSpeciesHIntervals[((i+1)*4)]==UNISSON ||
    fifthSpeciesHIntervals[((i+1)*4)]==PERFECT_OCTAVE ||
381                 fifthSpeciesHIntervals[((i+1)*4)]==-PERFECT_OCTAVE)) >> isOct);
382             rel(home, fifthSpeciesMIntervals[(i*4)+3], IRT_LE, -4, Reify(isCPDown));
383
384             rel(home, isCM, BOT_AND, isOct, band1);
385             rel(home, band1, BOT_AND, isCPDown, band2);
386             rel(home, band2, BOT_AND, expr(home, c->getNotes()[i] <= fifthSpeciesNotesCp[(i*4)
    +3]), band3);
387             rel(home, isThirdSpeciesArray[(i*4)+3], BOT_AND, band3, 0);
388         }
389     }
390
391     //dissonant notes must be followed by the consonant note below
392     if (activeConstraints[SP5_4P1]) {
393         for(int i = 0; i < fifthSpeciesMTAIntervals.size(); i++){
394             //isFourthArray first note
395             //is Consonance array first note
396             //mTa intervals, general
397             BoolVar bNot = BoolVar(home, 0, 1);
```

```
398             BoolVar isCst = BoolVar(home, 0, 1);
399             BoolVar bAnd = BoolVar(home, 0, 1);
400 
401             rel(home, isConsonance[i*4], BOT_EQV, BoolVar(home, 0, 0), bNot);
402             rel(home, bNot, BOT_AND, isFourthSpeciesArray[i*4], bAnd);
403             rel(home, fifthSpeciesMTAIntervals[i], IRT_LE, 0, Reify(bAnd, RM_IMP));
404             rel(home, fifthSpeciesMTAIntervals[i], IRT_GQ, -2, Reify(bAnd, RM_IMP));
405         }
406     }
407 
408     //DISABELED
409     // //no second dissonant note if the cantusFirmus is at the bass
410     // if (activeConstraints[SP5_P3]) {
411     //     for(int i = 0; i < nMeasures-1; i++){
412     //         BoolVar bUni = BoolVar(home, 0, 1);
413     //         BoolVar bAnd = BoolVar(home, 0, 1);
414     //         BoolVar isCst = BoolVar(home, 0, 1);
415     //         BoolVar bAndCst = BoolVar(home, 0, 1);
416 
417     //         rel(home, isFourthSpeciesArray[((i+1)*4)], BOT_EQV, isCst, 1);
418     //         rel(home, fifthSpeciesHIntervals[(i*4)+2], IRT_EQ, 0, Reify(bUni));
419     //         rel(home, isNotLowest[i], BOT_AND, bUni, bAnd);
420     //         rel(home, bAnd, BOT_AND, isCst, bAndCst);
421     //         rel(home, fifthSpeciesHIntervals[((i+1)*4)], IRT_NQ, 1, Reify(bAndCst, RM_IMP));
422     //         rel(home, fifthSpeciesHIntervals[((i+1)*4)], IRT_NQ, 2, Reify(bAndCst, RM_IMP));
423     //     }
424     // }
425 
426     //DISABLED RULE
427     // //marcel's rule
428     // if (activeConstraints[SP5_P4]) {
429     //     for(int i = 0; i < fifthSpeciesSuccMIntervals.size()-1; i++){
430     //         BoolVar bSkip = BoolVar(home, 0, 1);
431     //         BoolVar bMbUp = BoolVar(home, 0, 1);
432     //         BoolVar bMbDown = BoolVar(home, 0, 1);
433     //         BoolVar bContrary = BoolVar(home, 0, 1);
434 
435     //         rel(home, expr(home, abs(fifthSpeciesSuccMIntervals[i])), IRT_GR, 2, Reify(bSkip)
    );
436     //         rel(home, fifthSpeciesSuccMIntervals[i], IRT_GR, 0, Reify(bMbUp));
437     //         rel(home, fifthSpeciesSuccMIntervals[i+1], IRT_LE, 1, Reify(bMbDown));
438     //         rel(home, bMbUp, BOT_EQV, bMbDown, bContrary);
439     //         rel(home, fifthSpeciesSuccMIntervals[i+1], IRT_LQ, 2, Reify(bSkip, RM_IMP));
440     //         rel(home, bSkip, BOT_IMP, bContrary, 1);
441     //     }
442     // }
443 
444     /** ============================================================================
445      *                               COST CONSTRAINS
446      *  ============================================================================
447      */
448 
449     //Imperfect consonances are preferred
450     fifthCostArray = IntVarArray(home, notes.size(), IntSet({0, fifthCost}));
451     octaveCostArray = IntVarArray(home, notes.size(), IntSet({0, octaveCost}));
452 
453     //set fifth cost
454     for(int i = 0; i < notes.size(); i++){
455         BoolVar b = BoolVar(home, 0, 1);
456         BoolVar band = BoolVar(home, 0, 1);
457 
458         rel(home, fifthSpeciesHIntervals[i], IRT_EQ, 7, Reify(b));
459         rel(home, b, BOT_AND, isConstrainedArray[i], band);
460         ite(home, band, IntVar(home, fifthCost, fifthCost), IntVar(home, 0, 0), fifthCostArray[i
    ]);
461     }
462 
463     //set octave cost
464     for(int i = 0; i < notes.size(); i++){
465         BoolVar b = BoolVar(home, 0, 1);
466         BoolVar band = BoolVar(home, 0, 1);
467         BoolVar band2 = BoolVar(home, 0, 1);
468 
469         rel(home, fifthSpeciesHIntervals[i], IRT_EQ, 7, Reify(b));
470         rel(home, isNotLowest[floor(i/4)], BOT_AND, isConstrainedArray[i], band2);
471         rel(home, b, BOT_AND, band2, band);
472         ite(home, band, IntVar(home, octaveCost, octaveCost), IntVar(home, 0, 0),
    octaveCostArray[i]);
473     }
474 
475     //create off_cost array
476     offCostArray = IntVarArray(home, is_off.size(), IntSet({0, borrowCost}));
477     //set the cost for borrowing this note (G4 constraint, modified)
478     for(int i = 0; i < is_off.size(); i++){
479         rel(home, (is_off[i]==0) >> (offCostArray[i]==0));
480         rel(home, (is_off[i]==1 && isConstrainedArray[i]==1) >> (offCostArray[i]==borrowCost));
481         rel(home, (is_off[i]==1 && isConstrainedArray[i]==0) >> (offCostArray[i]==0));
482     }
483 
484     melodicDegreeCost = IntVarArray(home, m_intervals_brut.size(), IntSet({0, secondCost,
```

```
    thirdCost, fourthCost, tritoneCost, fifthCost,
485         sixthCost, seventhCost, octaveCost}));
486
487     //G7 (modified)
488     for(int i = 0; i < m_intervals_brut.size(); i+=4/notesPerMeasure.at(FIFTH_SPECIES)){
489         rel(home, (isConstrainedArray[i]==0 || isConstrainedArray[i+1]==0) >> (melodicDegreeCost
    [i]==0));
490         rel(home, (abs(m_intervals_brut[i])<MINOR_THIRD && isConstrainedArray[i]==1 &&
    isConstrainedArray[i+1]==1) >> (melodicDegreeCost[i]==secondCost));
491         rel(home, ((abs(m_intervals_brut[i])==MINOR_THIRD || abs(m_intervals_brut[i])==
    MAJOR_THIRD)&& isConstrainedArray[i]==1&& isConstrainedArray[i+1]==1) >> (melodicDegreeCost[
    i]==thirdCost));
492         rel(home, (abs(m_intervals_brut[i])==PERFECT_FOURTH && isConstrainedArray[i]==1&&
    isConstrainedArray[i+1]==1) >> (melodicDegreeCost[i]==fourthCost));
493         rel(home, (abs(m_intervals_brut[i])==TRITONE && isConstrainedArray[i]==1&&
    isConstrainedArray[i+1]==1) >> (melodicDegreeCost[i]==tritoneCost));
494         rel(home, (abs(m_intervals_brut[i])==PERFECT_FIFTH && isConstrainedArray[i]==1&&
    isConstrainedArray[i+1]==1) >> (melodicDegreeCost[i]==fifthCost));
495         rel(home, ((abs(m_intervals_brut[i])==MINOR_SIXTH || abs(m_intervals_brut[i])==
    MAJOR_SIXTH)&& isConstrainedArray[i]==1&& isConstrainedArray[i+1]==1) >> (melodicDegreeCost[
    i]==sixthCost));
496         rel(home, ((abs(m_intervals_brut[i])==MINOR_SEVENTH || abs(m_intervals_brut[i])==
    MAJOR_SEVENTH)&& isConstrainedArray[i]==1&& isConstrainedArray[i+1]==1) >> (
    melodicDegreeCost[i]==seventhCost));
497         rel(home, (abs(m_intervals_brut[i])==PERFECT_OCTAVE && isConstrainedArray[i]==1&&
    isConstrainedArray[i+1]==1) >> (melodicDegreeCost[i]==octaveCost));
498     }
499
500     cambiataCostArray = IntVarArray(home, nMeasures-1, IntSet({0, cambiataCost}));
501
502     for(int i = 0; i < cambiataCostArray.size(); i++){
503         BoolVar band = BoolVar(home, 0, 1);
504         rel(home, isNotCambiata[i], BOT_AND, isMostlyThirdArray[i], band);
505         ite(home, band, IntVar(home, cambiataCost, cambiataCost), IntVar(home, 0, 0),
    cambiataCostArray[i]);
506     }
507
508     m2ZeroCostArray = IntVarArray(home, m2Len, IntSet({0, m2ZeroCost}));
509
510     for(int i = 0; i < fifthSpeciesM2Intervals.size(); i++){
511         BoolVar b = BoolVar(home, 0, 1);
512         BoolVar band = BoolVar(home, 0, 1);
513         BoolVar band2 = BoolVar(home, 0, 1);
514
515         rel(home, fifthSpeciesM2Intervals[i], IRT_EQ, 0, Reify(b));
516         rel(home, isConstrainedArray[i], BOT_AND, isConstrainedArray[i+2], band2);
517         rel(home, b, BOT_AND, band2, band);
518         ite(home, band, IntVar(home, m2ZeroCost, m2ZeroCost), IntVar(home, 0, 0),
    m2ZeroCostArray[i]);
519     }
520
521     snycopeCostArray = IntVarArray(home, (fifthSpeciesMIntervals.size())/4, IntSet({0,
    syncopationCost}));
522     for(int i = 0; i < snycopeCostArray.size(); i++){
523         BoolVar b = BoolVar(home, 0, 1);
524         BoolVar band = BoolVar(home, 0, 1);
525
526         rel(home, fifthSpeciesMIntervals[(i*4)+2], IRT_NQ, 0, Reify(b));
527         rel(home, b, BOT_AND, isFourthSpeciesArray[(i*4)+2], band);
528         ite(home, band, IntVar(home, syncopationCost, syncopationCost), IntVar(home, 0, 0),
    snycopeCostArray[i]);
529     }
530 }
531
532 /**
533  * 2 VOICES CONSTRUCTOR
534  */
535 FifthSpeciesCounterpoint::FifthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb,
    int ub, Stratum* low, CantusFirmus* c,  int v_type,
536     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV):
537 FifthSpeciesCounterpoint(home, nMes, cf, lb, ub, FIFTH_SPECIES, low, c, v_type, m_costs, g_costs
    , s_costs, bm, nV)
538 {
539     //following two rel are H2_1 but modified, so we check in TwoVoiceProblem that that
    constrained isn't activated if fifth species
540     if (activeConstraints[SP5_2V_1]) {
541         //if the first note is constrained, then it must be a perfect consonance (works)
542         rel(home, (isConstrainedArray[0]==1) >> (fifthSpeciesHIntervals[0]==UNISSON ||
    fifthSpeciesHIntervals[0]==PERFECT_FIFTH));
543         //else if the first note is not constrained, then the third note must be a perfect
    consonance (works)
544         rel(home, (isConstrainedArray[0]==0) >> (fifthSpeciesHIntervals[2]==UNISSON ||
    fifthSpeciesHIntervals[2]==PERFECT_FIFTH));
545     }
546
547
548     //If the cantusFirmus is in the upper part, then no hamonic seventh
549     if (activeConstraints[SP5_2V_2]) {
550         for(int j = 1; j < c->getIsNotLowest().size(); j++){
551             rel(home, (getIsNotLowest()[j]==0 && isFourthSpeciesArray[j*4]) >> (
```

```
    firstSpeciesHarmonicIntervals[j]!=MINOR_SEVENTH && firstSpeciesHarmonicIntervals[j]!=-
    MINOR_SEVENTH));
552         rel(home, (getIsNotLowest()[j]==0 && isFourthSpeciesArray[j*4]) >> (
    firstSpeciesHarmonicIntervals[j]!=MAJOR_SEVENTH && firstSpeciesHarmonicIntervals[j]!=-
    MAJOR_SEVENTH));
553     }
554 }
555
556
557 //H3_1 can be activated as-is
558 if (activeConstraints[SP5_2V_3]) {
559     //penultimate note (only applies if it is 3rd species)
560     BoolVar bandThen_1 = BoolVar(home, 0, 1);
561     BoolVar testNot_1 = BoolVar(home, 0, 1);
562     BoolVar bandElse_1 = BoolVar(home, 0, 1);
563     BoolVar test_1 = expr(home, c->getNotes()[c->getNotes().size()-2] <= fifthSpeciesNotesCp
    [fifthSpeciesNotesCp.size()-2]);
564
565     rel(home, test_1, BOT_AND, isThirdSpeciesArray[isThirdSpeciesArray.size()-2], bandThen_1
    );
566     rel(home, test_1, BOT_EQV, BoolVar(home, 0, 0), testNot_1);
567     rel(home, testNot_1, BOT_AND, isThirdSpeciesArray[isThirdSpeciesArray.size()-2],
    bandElse_1);
568
569     rel(home, expr(home, fifthSpeciesNotesCp[fifthSpeciesNotesCp.size()-2]-c->getNotes()[c->
    getNotes().size()-2]), IRT_EQ, MAJOR_SIXTH, Reify(bandThen_1, RM_IMP));
570     rel(home, expr(home, fifthSpeciesNotesCp[fifthSpeciesNotesCp.size()-2]-c->getNotes()[c->
    getNotes().size()-2]), IRT_EQ, MINOR_THIRD, Reify(bandElse_1, RM_IMP));
571
572     //penultimate note (only applies if it is 4th species)
573     BoolVar bandThen = BoolVar(home, 0, 1);
574     BoolVar testNot = BoolVar(home, 0, 1);
575     BoolVar bandElse = BoolVar(home, 0, 1);
576     BoolVar test = expr(home, c->getNotes()[c->getNotes().size()-2] <= fifthSpeciesNotesCp[
    fifthSpeciesNotesCp.size()-3]);
577
578     rel(home, test, BOT_AND, isThirdSpeciesArray[isThirdSpeciesArray.size()-2], bandThen);
579     rel(home, test, BOT_EQV, BoolVar(home, 0, 0), testNot);
580     rel(home, testNot, BOT_AND, isThirdSpeciesArray[isThirdSpeciesArray.size()-2], bandElse)
    ;
581     rel(home, expr(home, fifthSpeciesNotesCp[fifthSpeciesNotesCp.size()-3]-c->getNotes()[c->
    getNotes().size()-2]), IRT_EQ, MAJOR_SIXTH, Reify(bandThen, RM_IMP));
582     rel(home, expr(home, fifthSpeciesNotesCp[fifthSpeciesNotesCp.size()-3]-c->getNotes()[c->
    getNotes().size()-2]), IRT_EQ, MINOR_THIRD, Reify(bandElse, RM_IMP));
583 }
584
585
586 //no direct motion to reach a perfect consonance
587 if (activeConstraints[SP5_2V_4]) {
588     for(int i = 0; i < fifthSpeciesMotions.size(); i++){
589         BoolVar isThirdSpeciesAndPCons = BoolVar(home, 0, 1);
590         BoolVar isPConsAndNotLowest = BoolVar(home, 0, 1);
591
592         rel(home, isThirdSpeciesArray[i*4], BOT_AND, expr(home, fifthSpeciesHIntervals[(i*4)
    +3]==UNISSON || fifthSpeciesHIntervals[(i*4)+3]==PERFECT_FIFTH ||
593             fifthSpeciesHIntervals[(i*4)+3]==-PERFECT_FIFTH), isThirdSpeciesAndPCons);
594         rel(home, isNotLowest[i], BOT_AND, isThirdSpeciesAndPCons, isPConsAndNotLowest);
595
596         rel(home, fifthSpeciesMotions[i], IRT_NQ, PARALLEL_MOTION, Reify(isPConsAndNotLowest
    , RM_IMP));
597
598     }
599 }
600
601
602 varietyCostArray = IntVarArray(home, 3*(fifthSpeciesHIntervals.size()-2), IntSet({0,
    varietyCost}));
603
604 costs = IntVarArray(home, 9, 0, 1000000);
605 cost_names = {"fifth", "octave", "borrow", "melodic", "motion", "cambiata", "m2", "
    syncopation", "variety"};
606
607 //set cost[0] to be fifth cost
608 add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure.at(
    FIFTH_SPECIES), fifthCostArray.size())), costs);
609 //set cost[1] to be octave cost
610 add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(0, 4/notesPerMeasure.at(
    FIFTH_SPECIES), octaveCostArray.size())), costs);
611 //set cost[2] to be off cost
612 add_cost(home, 2, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(FIFTH_SPECIES
    ), offCostArray.size())), costs);
613 //set cost[3] to be melodic cost
614 add_cost(home, 3, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
    FIFTH_SPECIES), melodicDegreeCost.size())), costs);
615 //set cost[4] to be motion cost
616 add_cost(home, 4, fifthSpeciesMotionsCosts, costs);
617 //need to set cost[5] to be cambiata cost
618 add_cost(home, 5, cambiataCostArray, costs);
619 //need to set cost[6] to be m2Zero cost
620 add_cost(home, 6, m2ZeroCostArray, costs);
```

```
621     //need to set cost[7] to be syncopation cost
622     add_cost(home, 7, snycopeCostArray, costs);
623     //set cost[8] to be variety cost
624     add_cost(home, 8, varietyCostArray, costs);
625 }
626 
627 /**
628  * 3 VOICES CONSTRUCTOR
629  */
630 FifthSpeciesCounterpoint::FifthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb,
    int ub, Stratum* low, CantusFirmus* c,  int v_type,
631     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2):
632 FifthSpeciesCounterpoint(home, nMes, cf, lb, ub, FIFTH_SPECIES, low, c, v_type, m_costs, g_costs
    , s_costs, bm, nV2)
633 {
634     directCostArray = IntVarArray(home, fifthSpeciesMotions.size()-1,IntSet({0, directMoveCost})
    );
635     varietyCostArray = IntVarArray(home, 3*(fifthSpeciesHIntervals.size()-2), IntSet({0,
    varietyCost}));
636 
637     //3.H6 : harmonic triad should be used on the second or third beat
638     if (activeConstraints[SP5_3V_1]) {
639         thirdHTriadArray = IntVarArray(home, nMeasures-1, IntSet({0, triad3rdCost}));
640         for(int i = 0; i < thirdHTriadArray.size(); i++){
641             rel(home, ((fifthSpeciesHIntervals[(i*4)+1]!=UNISSON&&fifthSpeciesHIntervals[(i*4)
    +1]!=MINOR_THIRD&&fifthSpeciesHIntervals[(i*4)+1]!=MAJOR_THIRD&&fifthSpeciesHIntervals[(i*4)
    +1]!=PERFECT_FIFTH)&&
642                 (fifthSpeciesHIntervals[(i*4)+2]!=UNISSON&&fifthSpeciesHIntervals[(i*4)+2]!=
    MINOR_THIRD&&fifthSpeciesHIntervals[(i*4)+2]!=MAJOR_THIRD&&fifthSpeciesHIntervals[(i*4)+2]!=
    PERFECT_FIFTH)) >>
643                 (thirdHTriadArray[i]==triad3rdCost));
644             rel(home, ((fifthSpeciesHIntervals[(i*4)+1]==UNISSON||fifthSpeciesHIntervals[(i*4)
    +1]==MINOR_THIRD||fifthSpeciesHIntervals[(i*4)+1]==MAJOR_THIRD||fifthSpeciesHIntervals[(i*4)
    +1]==PERFECT_FIFTH)||
645                 (fifthSpeciesHIntervals[(i*4)+2]==UNISSON||fifthSpeciesHIntervals[(i*4)+2]==
    MINOR_THIRD||fifthSpeciesHIntervals[(i*4)+2]==MAJOR_THIRD||fifthSpeciesHIntervals[(i*4)+2]==
    PERFECT_FIFTH)) >>
646                 (thirdHTriadArray[i]==0));
647         }
648     }
649 
650     //1.P1 3 voices version
651     if (activeConstraints[SP5_3V_2]) {
652         for(int j = 0; j < fifthSpeciesMotions.size()-1; j++){
653             //set a cost when it is reached through direct motion, it is 0 when not
654             rel(home, (fifthSpeciesMotions[j]==2&&(firstSpeciesHarmonicIntervals[j+1]==0||
    firstSpeciesHarmonicIntervals[j+1]==7))>>
655                 (directCostArray[j]==directMoveCost));
656             rel(home, (fifthSpeciesMotions[j]!=2||(firstSpeciesHarmonicIntervals[j+1]!=0&&
    firstSpeciesHarmonicIntervals[j+1]!=7))>>
657                 (directCostArray[j]==0));
658         }
659     }
660 
661 
662     costs = IntVarArray(home, 11, 0, 1000000);
663     cost_names = {"fifth", "octave", "borrow", "melodic", "motion", "cambiata", "m2", "
    syncopation", "direct", "variety", "triad3"};
664 
665     //set cost[0] to be fifth cost
666     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(1, 4/notesPerMeasure.at(
    FIFTH_SPECIES), fifthCostArray.size())), costs);
667     //set cost[1] to be octave cost
668     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(1, 4/notesPerMeasure.at(
    FIFTH_SPECIES), octaveCostArray.size())), costs);
669     //set cost[2] to be off cost
670     add_cost(home, 2, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(FIFTH_SPECIES
    ), offCostArray.size())), costs);
671     //set cost[3] to be melodic cost
672     add_cost(home, 3, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
    FIFTH_SPECIES), melodicDegreeCost.size())), costs);
673     //set cost[4] to be motion cost
674     add_cost(home, 4, fifthSpeciesMotionsCosts, costs);
675     //need to set cost[5] to be cambiata cost
676     add_cost(home, 5, cambiataCostArray, costs);
677     //need to set cost[6] to be m2Zero cost
678     add_cost(home, 6, m2ZeroCostArray, costs);
679     //need to set cost[7] to be syncopation cost
680     add_cost(home, 7, snycopeCostArray, costs);
681     //need to set cost[8] to be syncopation cost
682     add_cost(home, 8, directCostArray, costs);
683     //need to set cost[8] to be syncopation cost
684     add_cost(home, 9, varietyCostArray, costs);
685     //set cost[10] to be triad h third cost
686     add_cost(home, 10, thirdHTriadArray, costs);
687 }
688 
689 /**
690  * 4 VOICES CONSTRUCTOR
691  */
```

```
692 FifthSpeciesCounterpoint::FifthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb,
    int ub, Stratum* low, CantusFirmus* c,  int v_type,
693     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2, int
     nV3):
694 FifthSpeciesCounterpoint(home, nMes, cf, lb, ub, FIFTH_SPECIES, low, c, v_type, m_costs, g_costs
    , s_costs, bm, nV3)
695 {
696     directCostArray = IntVarArray(home, fifthSpeciesMotions.size()-1,IntSet({0, directMoveCost})
    );
697     varietyCostArray = IntVarArray(home, 3*(fifthSpeciesHIntervals.size()-2), IntSet({0,
    varietyCost}));
698
699     //3.H6 : harmonic triad should be used on the second or third beat
700     if (activeConstraints[SP5_4V_1]) {
701         thirdHTriadArray = IntVarArray(home, nMeasures-1, IntSet({0, triad3rdCost}));
702         for(int i = 0; i < thirdHTriadArray.size(); i++){
703             rel(home, ((fifthSpeciesHIntervals[(i*4)+1]!=UNISSON&&fifthSpeciesHIntervals[(i*4)
    +1]!=MINOR_THIRD&&fifthSpeciesHIntervals[(i*4)+1]!=MAJOR_THIRD&&fifthSpeciesHIntervals[(i*4)
    +1]!=PERFECT_FIFTH)&&
704                 (fifthSpeciesHIntervals[(i*4)+2]!=UNISSON&&fifthSpeciesHIntervals[(i*4)+2]!=
    MINOR_THIRD&&fifthSpeciesHIntervals[(i*4)+2]!=MAJOR_THIRD&&fifthSpeciesHIntervals[(i*4)+2]!=
    PERFECT_FIFTH)) >>
705                 (thirdHTriadArray[i]==triad3rdCost));
706             rel(home, ((fifthSpeciesHIntervals[(i*4)+1]==UNISSON||fifthSpeciesHIntervals[(i*4)
    +1]==MINOR_THIRD||fifthSpeciesHIntervals[(i*4)+1]==MAJOR_THIRD||fifthSpeciesHIntervals[(i*4)
    +1]==PERFECT_FIFTH)||
707                 (fifthSpeciesHIntervals[(i*4)+2]==UNISSON||fifthSpeciesHIntervals[(i*4)+2]==
    MINOR_THIRD||fifthSpeciesHIntervals[(i*4)+2]==MAJOR_THIRD||fifthSpeciesHIntervals[(i*4)+2]==
    PERFECT_FIFTH)) >>
708                 (thirdHTriadArray[i]==0));
709         }
710     }
711
712
713     //1.P1 4 voices version
714     if (activeConstraints[SP5_4V_2]) {
715         for(int j = 0; j < fifthSpeciesMotions.size()-1; j++){
716             //set a cost when it is reached through direct motion, it is 0 when not
717             rel(home, (fifthSpeciesMotions[j]==2&&(firstSpeciesHarmonicIntervals[j+1]==0||
    firstSpeciesHarmonicIntervals[j+1]==7))>>
718                 (directCostArray[j]==directMoveCost));
719             rel(home, (fifthSpeciesMotions[j]!=2||(firstSpeciesHarmonicIntervals[j+1]!=0&&
    firstSpeciesHarmonicIntervals[j+1]!=7))>>
720                 (directCostArray[j]==0));
721         }
722     }
723
724     costs = IntVarArray(home, 11, 0, 1000000);
725     cost_names = {"fifth", "octave", "borrow", "melodic", "motion", "cambiata", "m2", "
    syncopation", "direct", "variety", "triad3"};
726
727     //set cost[0] to be fifth cost
728     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(1, 4/notesPerMeasure.at(
    FIFTH_SPECIES), fifthCostArray.size())), costs);
729     //set cost[1] to be octave cost
730     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(1, 4/notesPerMeasure.at(
    FIFTH_SPECIES), octaveCostArray.size())), costs);
731     //set cost[2] to be off cost
732     add_cost(home, 2, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(FIFTH_SPECIES
    ), offCostArray.size())), costs);
733     //set cost[3] to be melodic cost
734     add_cost(home, 3, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
    FIFTH_SPECIES), melodicDegreeCost.size())), costs);
735     //set cost[4] to be motion cost
736     add_cost(home, 4, fifthSpeciesMotionsCosts, costs);
737     //need to set cost[5] to be cambiata cost
738     add_cost(home, 5, cambiataCostArray, costs);
739     //need to set cost[6] to be m2Zero cost
740     add_cost(home, 6, m2ZeroCostArray, costs);
741     //need to set cost[7] to be syncopation cost
742     add_cost(home, 7, snycopeCostArray, costs);
743     //need to set cost[8] to be syncopation cost
744     add_cost(home, 8, directCostArray, costs);
745     //need to set cost[8] to be syncopation cost
746     add_cost(home, 9, varietyCostArray, costs);
747     //set cost[10] to be triad h third cost
748     add_cost(home, 10, thirdHTriadArray, costs);
749 }
750
751 /**
752  * This function returns a string with the characteristics of the counterpoint. It calls the
    to_string() method from
753  * the Part class and adds 1st species specific characteristics.
754  * @return a string representation of the current instance of the FirstSpeciesCounterpoint class
    .
755  */
756 string FifthSpeciesCounterpoint::to_string() const {
757     string text = Part::to_string() + "\n Fifth species :\n";
758     text += "Fifth species notes : " + intVarArray_to_string(fifthSpeciesNotesCp) + "\n";
759     text += "M Intervals array : " + intVarArray_to_string(fifthSpeciesSuccMIntervals) + "\n";
```

```
760     text += "isMostlyThird : " + boolVarArray_to_string(isMostlyThirdArray) + "\n";
761     text += "not Lowest array : " + boolVarArray_to_string(isNotLowest) + "\n";
762     text += "Species array : " + intVarArray_to_string(speciesArray) + "\n";
763     return text;
764 }
765
766 string FifthSpeciesCounterpoint::getSolutionRhythm() const {
767     // isThirdSpeciesArray is the only rhythm classification that survives cloning into the
768     // solution space (it is update()'d in the copy ctor); speciesArray is not, so we expose
769     // the 3rd-species flags directly. Each entry corresponds 1:1 to a flat-MIDI cell.
770     return boolVarArray_to_string(isThirdSpeciesArray);
771 }
772
773 // clone constructor
774 FifthSpeciesCounterpoint::FifthSpeciesCounterpoint(Home home, FifthSpeciesCounterpoint &s) :
        Part(home, s){
775     solutionLength = s.solutionLength;
776     m2Len = s.m2Len;
777     fifthSpeciesNotesCp.update(home, s.fifthSpeciesNotesCp);
778     fifthSpeciesHIntervals.update(home, s.fifthSpeciesHIntervals);
779     firstHInterval.update(home, s.firstHInterval);
780     isNthSpeciesArray.update(home, s.isNthSpeciesArray);
781     isConstrainedArray.update(home, s.isConstrainedArray);
782     fifthSpeciesSuccMIntervals.update(home, s.fifthSpeciesSuccMIntervals);
783     fifthSpeciesMIntervals.update(home, s.fifthSpeciesMIntervals);
784     fifthSpeciesMTAIntervals.update(home, s.fifthSpeciesMTAIntervals);
785     fifthSpeciesM2Intervals.update(home, s.fifthSpeciesM2Intervals);
786     fifthSpeciesMAllIntervals.update(home, s.fifthSpeciesMAllIntervals);
787     fifthSpeciesMotions.update(home, s.fifthSpeciesMotions);
788     fifthSpeciesMotionsCosts.update(home, s.fifthSpeciesMotionsCosts);
789     isMostlyThirdArray.update(home, s.isMostlyThirdArray);
790     isFourthSpeciesArray.update(home, s.isFourthSpeciesArray);
791     isThirdSpeciesArray.update(home, s.isThirdSpeciesArray);
792     isNotCambiata.update(home, s.isNotCambiata);
793     m2ZeroCostArray.update(home, s.m2ZeroCostArray);
794     thirdHTriadArray.update(home, s.thirdHTriadArray);
795 }
796
797 FifthSpeciesCounterpoint* FifthSpeciesCounterpoint::clone(Home home){
798     return new FifthSpeciesCounterpoint(home, *this);
799 }
800
801 IntVarArray FifthSpeciesCounterpoint::getBranchingNotes(){
802     return fifthSpeciesNotesCp;
803 }
804
805 IntVarArray FifthSpeciesCounterpoint::getFirstHInterval(){
806     return firstSpeciesHarmonicIntervals;
807 }
808
809 IntVarArray FifthSpeciesCounterpoint::getMotions(){
810     return fifthSpeciesMotions;
811 }
812
813 IntVarArray FifthSpeciesCounterpoint::getFirstMInterval(){
814     return firstSpeciesMelodicIntervals;
815 }
816
817 int FifthSpeciesCounterpoint::getHIntervalSize(){
818     return fifthSpeciesHIntervals.size();
819 }
820
821 void FifthSpeciesCounterpoint::createSpeciesArrays(Home home){
822     IntVarArray countThird = IntVarArray(home, solutionLength, 0, 1);
823     IntVarArray countFourth = IntVarArray(home, solutionLength, 0, 1);
824     int nThirdInt = floor((prefSlider-1)*0.66);
825     int nFourthInt = floor(prefSlider*0.5);
826     IntVar sumThird = IntVar(home, nThirdInt, solutionLength);
827     IntVar sumFourth = IntVar(home, nFourthInt, solutionLength);
828
829     //Count 3rd and 4th species
830     for(int i = 0; i < solutionLength; i++){
831         BoolVar b = BoolVar(home, 0, 1);
832         rel(home, speciesArray[i], IRT_EQ, THIRD_SPECIES, Reify(b));
833         ite(home, b, IntVar(home, 1, 1), IntVar(home, 0, 0), countThird[i]);
834     }
835
836     for(int i = 0; i < solutionLength; i++){
837         BoolVar b = BoolVar(home, 0, 1);
838         rel(home, speciesArray[i], IRT_EQ, FOURTH_SPECIES, Reify(b));
839         ite(home, b, IntVar(home, 1, 1), IntVar(home, 0, 0), countFourth[i]);
840     }
841
842     //sum the counts
843     //rel(home, sumThird, IRT_EQ, expr(home, sum(countThird)));
844     //rel(home, sumFourth, IRT_EQ, expr(home, sum(countFourth)));
845
846     //no 4th species in second and fourth positions in the species array
847     for(int i = 1; i < solutionLength; i+=2){
848         rel(home, speciesArray[i], IRT_NQ, FOURTH_SPECIES);
```

```
849     }
850
851     //4th species in the third position is followed by no species (no note / constraint) and
        then a 4th species
852     for(int i = 0; i < solutionLength-1; i+=4){
853         BoolVar b34 = BoolVar(home, 0, 1);
854         BoolVar b14 = BoolVar(home, 0, 1);
855
856         rel(home, speciesArray[i+2], IRT_EQ, FOURTH_SPECIES, Reify(b34));
857         //the following line works to allow quarter notes after fourth species, but behaves
        weird
858         //rel(home, expr(home, speciesArray[i+4]==FOURTH_SPECIES), BOT_OR, expr(home,
        speciesArray[i+4]==THIRD_SPECIES), b14);
859         rel(home, speciesArray[i+4], IRT_EQ, FOURTH_SPECIES, Reify(b14));
860         rel(home, speciesArray[i+3], IRT_EQ, -1, Reify(b34, RM_IMP));
861         rel(home, b34, BOT_EQV, b14, 1);
862     }
863
864     //only 3rd and 4th species
865     for(int i = 0; i < speciesArray.size(); i++){
866         rel(home, speciesArray[i], IRT_NQ, FIRST_SPECIES);
867         rel(home, speciesArray[i], IRT_NQ, SECOND_SPECIES);
868     }
869
870     //first and penultimate measures are 4th species
871     rel(home, speciesArray[0], IRT_EQ, -1);
872     rel(home, speciesArray[1], IRT_EQ, -1);
873     rel(home, speciesArray[2], IRT_EQ, FOURTH_SPECIES);
874
875     rel(home, speciesArray[solutionLength-5], IRT_EQ, FOURTH_SPECIES);
876     rel(home, speciesArray[solutionLength-4], IRT_EQ, -1);
877     rel(home, speciesArray[solutionLength-3], IRT_EQ, FOURTH_SPECIES);
878
879     //no silence after third species notes
880     for(int i = 0; i < solutionLength-1; i++){
881         BoolVar b = BoolVar(home, 0, 1);
882         rel(home, speciesArray[i], IRT_EQ, THIRD_SPECIES, Reify(b));
883         rel(home, speciesArray[i+1], IRT_NQ, -1, Reify(b, RM_IMP));
884     }
885
886     //no silence after fourth species notes
887     for(int i = 0; i < solutionLength-5; i++){
888         BoolVar b = BoolVar(home, 0, 1);
889         rel(home, speciesArray[i], IRT_EQ, FOURTH_SPECIES, Reify(b));
890         rel(home, speciesArray[i+4], IRT_NQ, -1, Reify(b, RM_IMP));
891     }
892
893     //maximum two consecutive measures without a fourth species
894     for(int i = 2; i < solutionLength-13; i+=4){
895         BoolVar b1n4 = BoolVar(home, 0, 1);
896         BoolVar b2n4 = BoolVar(home, 0, 1);
897         BoolVar band = BoolVar(home, 0, 1);
898
899         rel(home, speciesArray[i+4], IRT_NQ, FOURTH_SPECIES, Reify(b1n4));
900         rel(home, speciesArray[i+8], IRT_NQ, FOURTH_SPECIES, Reify(b2n4));
901         rel(home, b1n4, BOT_AND, b2n4, band);
902         rel(home, speciesArray[i+12], IRT_EQ, FOURTH_SPECIES, Reify(band, RM_IMP));
903     }
904
905     /**
906      * CREATE NTH SPECIES ARRAY
907      */
908     // cout << speciesArray.size() << endl;
909     // cout << isNthSpeciesArray.size() << endl;
910     for(int i = 0; i < isNthSpeciesArray.size(); i+=5){
911         rel(home, speciesArray[floor(i/5)], IRT_EQ, -1, Reify(isNthSpeciesArray[i]));
912         rel(home, speciesArray[floor(i/5)], IRT_EQ, FIRST_SPECIES, Reify(isNthSpeciesArray[i+1])
        );
913         rel(home, speciesArray[floor(i/5)], IRT_EQ, SECOND_SPECIES, Reify(isNthSpeciesArray[i
        +2]));
914         rel(home, speciesArray[floor(i/5)], IRT_EQ, THIRD_SPECIES, Reify(isNthSpeciesArray[i+3])
        );
915         rel(home, speciesArray[floor(i/5)], IRT_EQ, FOURTH_SPECIES, Reify(isNthSpeciesArray[i
        +4]));
916     }
917
918     /**
919      * CREATE IS CONSTRAINED ARRAY
920      */
921     for(int i = 0; i < isConstrainedArray.size(); i++){
922         rel(home, isNthSpeciesArray[i*5], IRT_EQ, 0, Reify(isConstrainedArray[i]));
923     }
924 }
```

A.9 Constraints and utility sources

A.9.1 constraints.cpp

```
//
// Created by Luc Cleenewerk and Diego de Patoul.
// This file contains the implementations of the functions that post the constraints.
//

#include "../../headers/Parts/constraints.hpp"

void initializeIsOffArray(Home home, Part* part){
    for(int i = 0; i < part->getIsOffArray().size(); i++){                                  //loop goes through every note of the counterpoint
        IntVarArray res = IntVarArray(home, part->getOffDomain().size(), 0, 1);
        IntVar sm = IntVar(home, 0, part->getOffDomain().size());
        for(int l = 0; l < part->getOffDomain().size(); l++){                               //loop goes through the domain of borrowed notes
            BoolVar b1 = BoolVar(home, 0, 1);
            rel(home, part->getNotes()[i], IRT_EQ, part->getOffDomain()[l], Reify(b1)); //checks if a note is borrowed
            ite(home, b1, IntVar(home, 1, 1), IntVar(home, 0, 0), res[l]);              //if it is borrowed, set res[l] to one
        }
        IntVarArgs x(res.size());
        for(int t = 0; t < part->getOffDomain().size(); t++){
            x[t] = res[t];                                                              //create an IntVarArgs array of the contents of res[l] (for sum)
        }
        rel(home, sm, IRT_EQ, expr(home, sum(x)));                                      //set the sm variable to be the sum of the IntVarArgs
        rel(home, sm, IRT_GR, 0, Reify(part->getIsOffArray()[i]));                      //if the sum is greater than 0, then the note is borrowed (off key)
    }
}

void R9_5_twoFifthSpeciesDiversity_3v(Home home, Part* cp1, Part* cp2){
    if(cp1->getSpecies()==FIFTH_SPECIES && cp2->getSpecies()==FIFTH_SPECIES){
        BoolVarArray isSameSpecies = BoolVarArray(home, cp1->getNotes().size(), 0, 1);
        IntVarArray isSameSpeciesInt = IntVarArray(home, cp1->getNotes().size(), 0, 1);
        IntVar percentageSame = IntVar(home, 0, cp1->getNotes().size());
        for(int i = 0; i < cp1->getNotes().size(); i++){
            rel(home, cp1->getSpeciesArray()[i], IRT_EQ, cp2->getSpeciesArray()[i], Reify(isSameSpecies[i]));
            rel(home, isSameSpeciesInt[i], IRT_EQ, 1, Reify(isSameSpecies[i]));
        }
        rel(home, percentageSame, IRT_EQ, expr(home, sum(isSameSpeciesInt)));
        rel(home, percentageSame, IRT_LE, floor(cp1->getNotes().size()/2));
    }
}

void noMinorSecondBetweenUpper(Home home, vector<Stratum*> strata){
    // For all pairs of voices s1 < s2:
    for(int s1 = 0; s1 < strata.size(); s1++){
        for(int s2 = s1+1; s2 < strata.size(); s2++){
            // For each measure i:
            for(int i = 0; i < strata[1]->getNMeasures(); i++){
                rel(home, expr(home, abs(strata[s2]->getNotes()[i*4] - strata[s1]->getNotes()[i*4]) % 12), IRT_NQ, MINOR_SECOND);
            }
        }
    }
}

void G4_counterpointMustBeInTheSameKey(Home home, Part* part){
    for(int i = 0; i < part->getIsOffArray().size(); i++){
        rel(home, (part->getIsOffArray()[i]==0) >> (part->getOffCostArray()[i]==0));
             //sets no cost if not borrowed
        rel(home, (part->getIsOffArray()[i]==1) >> (part->getOffCostArray()[i]==part->getBorrowCost())); //sets a cost if borrowed
    }
}

void G6_noChromaticMelodies(Home home, Part* part, int mSpec){

    //forbids that three notes one after the other increase by a step, making the melody chromatic

    for(int i = 0; i < part->getMelodicIntervals().size()-1; i+=4/notesPerMeasure.at(mSpec)){
        rel(home, expr(home, part->getMelodicIntervals()[i]==1), BOT_AND, expr(home, part->getMelodicIntervals()[i+1]==1), 0);
        rel(home, expr(home, part->getMelodicIntervals()[i]==-1), BOT_AND, expr(home, part->getMelodicIntervals()[i+1]==-1), 0);
    }
}

void G7_melodicIntervalsShouldBeSmall(Home home, Part* part, int mSpec){
```

```
70
71      //loop goes through every note of the melodic interval and sets the cost
72
73      for(int i = 0; i < part->getMelodicIntervals().size(); i+=4/notesPerMeasure.at(mSpec)){
74          rel(home, (abs(part->getMelodicIntervals()[i])<MINOR_THIRD) >> (part->
        getMelodicDegreeCost()[i]==part->getSecondCost()));
75          rel(home, (abs(part->getMelodicIntervals()[i])==MINOR_THIRD || abs(part->
        getMelodicIntervals()[i])==MAJOR_THIRD) >> (part->getMelodicDegreeCost()[i]==part->
        getThirdCost()));
76          rel(home, (abs(part->getMelodicIntervals()[i])==PERFECT_FOURTH) >> (part->
        getMelodicDegreeCost()[i]==part->getFourthCost()));
77          rel(home, (abs(part->getMelodicIntervals()[i])==TRITONE) >> (part->getMelodicDegreeCost
        ()[i]==part->getTritoneCost()));
78          rel(home, (abs(part->getMelodicIntervals()[i])==PERFECT_FIFTH) >> (part->
        getMelodicDegreeCost()[i]==part->getFifthCost()));
79          rel(home, (abs(part->getMelodicIntervals()[i])==MINOR_SIXTH || abs(part->
        getMelodicIntervals()[i])==MAJOR_SIXTH) >> (part->getMelodicDegreeCost()[i]==part->
        getSixthCost()));
80          rel(home, (abs(part->getMelodicIntervals()[i])==MINOR_SEVENTH || abs(part->
        getMelodicIntervals()[i])==MAJOR_SEVENTH) >> (part->getMelodicDegreeCost()[i]==part->
        getSeventhCost()));
81          rel(home, (abs(part->getMelodicIntervals()[i])==PERFECT_OCTAVE) >> (part->
        getMelodicDegreeCost()[i]==part->getOctaveCost()));
82      }
83  }
84
85  void G9_lastChordSameAsFundamental(Home home, Stratum* lowest, Part* cantusFirmus){
86
87      //we check the last note of the cantusFirmus to set the last note of the counterpoint to be
        in the same key
88
89      rel(home, expr(home, lowest->getNotes()[lowest->getNotes().size()-1]%12), IRT_EQ, expr(home,
         cantusFirmus->getNotes()[cantusFirmus->getNotes().size()-1]%12));
90      rel(home, expr(home, lowest->getNotes()[0]%12), IRT_EQ, expr(home, cantusFirmus->getNotes()
        [0]%12)); // Modified by Tom Lai
91  }
92
93  void H1_1_harmonicIntervalsAreConsonances(Home home, Part* part){
94
95      //we also include negative values, since the fourth species could render the harmonic
        intervals to be negative because of its deplacement
96      //mostly the harmonic intervals are going to be positive
97
98      dom(home, part->getFirstSpeciesHIntervals(), IntSet({UNISSON, MINOR_THIRD, MAJOR_THIRD,
        PERFECT_FIFTH, MINOR_SIXTH, MAJOR_SIXTH, PERFECT_OCTAVE,
99          -MINOR_THIRD, -MAJOR_THIRD, -PERFECT_FIFTH, -MINOR_SIXTH, -MAJOR_SIXTH, -PERFECT_OCTAVE
        }));
100 }
101
102 void H1_3_fiveConsecutiveNotesByJointDegree(Home home, Part* part){
103     for(int i = 0; i < part->getIs5QNArray().size(); i++){
104         rel(home, (part->getIs5QNArray()[i]==1) >> (part->getConsonance()[(i*4)+2]==1));
105     }
106 }
107
108 void H2_1_startWithPerfectConsonance(Home home, Part* part){
109     dom(home, part->getHIntervals()[0], IntSet(IntArgs(PERFECT_CONSONANCES)));
110 }
111
112 void H2_2_arsisHarmoniesCannotBeDissonant(Home home, Part* part){
113
114     for(int i = 0; i < part->getNMeasures()-1; i++){
115         if(i != part->getNMeasures()-2){ //if it is the penultimate measure
116             for(int d : DISONANCE){
117                 rel(home, part->getHIntervals()[(i*4)+2], IRT_EQ, d, Reify(part->getIsDiminution
        ()[i], RM_PMI));
118             }
119         }
120     }
121 }
122
123 void H2_3_dissonanceImpliesDiminution(Home home, Part* part){
124     // NOTE: this in-species version only ties the central weak beat (+2) to the per-measure
125     // isDiminution flag, and reads getConsonance() which is NOT wired for 3rd species (so it
126     // is effectively a no-op on consonance). Full weak-beat consonance for 2-voice 3rd
127     // species is enforced separately by H2_3_..._multiVoice, called from TwoVoiceCounterpoint
128     // (it computes the interval from the notes, not from getConsonance). Kept for the gated
129     // SP3_3H2 path; harmless.
130     for(int i = 0; i < part->getIsDiminution().size(); i++){
131         rel(home, part->getConsonance()[(i*4)+2], BOT_OR, part->getIsDiminution()[i], 1);
132     }
133 }
134
135 // Returns the index, in another voice's note array, of its strong-beat (downbeat) note
136 // for measure i. Only strong beats are guaranteed pinned: a slow voice's intermediate
137 // cells (i*4+1, +2, +3) are free variables (see FirstSpeciesCounterpoint.cpp slice).
138 // The CantusFirmus stores one note per measure (index i); other parts store four.
139 static int strongBeatIndex(Part* voice, int i){
140     return (voice->getSpecies() == CANTUS_FIRMUS) ? i : i*4;
141 }
```

```
142
143 // [Claude Code]
144 void H2_2_arsisHarmoniesCannotBeDissonant_multiVoice(Home home, Part* secondSpPart, vector<Part
    *> otherVoices){
145     // h_intervals stored on each Part are computed only against the lowest stratum.
146     // In 3+ voice settings, the dissonance between the 2nd-species arsis and any
147     // OTHER voice is therefore never checked. Fux rule: a disjunct weak beat must be
148     // consonant with every sounding voice. We compare the arsis (index i*4+2, pinned
149     // for 2nd species) against each other voice's STRONG beat (only pinned reference),
150     // and tie the dissonance escape to isDiminution[i] (the 2nd-species passing flag).
151     for(int i = 0; i < secondSpPart->getNMeasures()-1; i++){
152         if(i == secondSpPart->getNMeasures()-2) continue; // skip penultimate (same as base H2_2
    )
153         for(Part* other : otherVoices){
154             if(other == secondSpPart) continue;
155             IntVar interval(home, 0, 11);
156             rel(home, interval == abs(secondSpPart->getNotes()[i*4+2] - other->getNotes()[
    strongBeatIndex(other, i)]) % 12);
157             for(int d : DISONANCE){
158                 rel(home, interval, IRT_EQ, d, Reify(secondSpPart->getIsDiminution()[i], RM_PMI)
    );
159             }
160         }
161     }
162 }
163
164 // [Claude Code]
165 void H2_3_dissonanceImpliesDiminution_multiVoice(Home home, Part* thirdSpPart, vector<Part*>
    otherVoices){
166     // 3rd species has 4 notes per measure: 1 strong beat (i*4) + 3 weak beats (i*4+1,+2,+3).
167     // The base H2_3 only checks the central weak beat (+2) against the bass. Here we check
168     // ALL three weak beats against EVERY other voice's strong beat. A weak beat may be
169     // dissonant only if it is a genuine passing tone: approached AND left by step (1..2
170     // semitones). All four cells are pinned for 3rd species, so melodic moves use them directly
    .
171     int nM = thirdSpPart->getNMeasures();
172     for(int i = 0; i < nM-1; i++){
173         for(int k = 1; k <= 3; k++){
174             int j = i*4 + k; // weak-beat instant (pinned for 3rd species)
175             IntVar inMove  = expr(home, abs(thirdSpPart->getNotes()[j]   - thirdSpPart->getNotes
    ()[j-1]));
176             IntVar outMove = expr(home, abs(thirdSpPart->getNotes()[j+1] - thirdSpPart->getNotes
    ()[j]));
177             BoolVar inStep  = expr(home, inMove  >= 1 && inMove  <= 2);
178             BoolVar outStep = expr(home, outMove >= 1 && outMove <= 2);
179             BoolVar isPassing = expr(home, inStep && outStep); // approached AND left by step
180             for(Part* other : otherVoices){
181                 if(other == thirdSpPart) continue;
182                 IntVar interval(home, 0, 11);
183                 rel(home, interval == abs(thirdSpPart->getNotes()[j] - other->getNotes()[
    strongBeatIndex(other, i)]) % 12);
184                 for(int d : DISONANCE){
185                     rel(home, interval, IRT_EQ, d, Reify(isPassing, RM_PMI));
186                 }
187             }
188         }
189     }
190 }
191
192 // Index used to read a voice's harmony note for the SECOND half of a measure.
193 // Only a 4th-species voice changes harmony mid-measure: its suspension (downbeat) resolves
194 // on the 3rd beat (Bitsch  59 ), so it contributes its resolution note (i*4+2). Every other
195 // voice holds its downbeat note across the whole measure.
196 static int resolutionBeatIndex(Part* voice, int i){
197     return (voice->getSpecies() == FOURTH_SPECIES) ? i*4+2 : strongBeatIndex(voice, i);
198 }
199
200 // [Claude Code]
201 void H2_4_chordMembershipOnDisjunctWeakBeats(Home home, Part* sp, vector<Part*> allVoices){
202     // MANDATORY tonal rule (Bitsch  59 ). A disjunct weak-beat note must belong to the
203     // harmony of its half-measure. The harmony is the IMPLIED DIATONIC TRIAD: the
204     // structural (downbeat) notes of all voices outline a chord, and a disjunct weak beat
205     // must be one of that chord's tones     including a chord tone that is not literally
206     // sounding on a downbeat (e.g. the third of a {root, fifth} dyad). This generalises
207     // the older "must equal a sounding downbeat pitch class" rule, which wrongly rejected
208     // legitimate chord tones whenever the triad was incompletely voiced on the downbeat.
209     //
210     // Implementation: the key's diatonic triads are built by stacking thirds inside the
211     // scale (one triad per degree, tonic-independent). A weak beat is legal when, for at
212     // least one diatonic triad T, the weak-beat pitch class AND every structural pitch
213     // class lie in T. A fallback keeps the old "matches a sounding downbeat PC" test so
214     // the rule never gets stricter than before on chromatic/borrowed harmonies that fit
215     // no diatonic triad. Genuine passing tones (approached AND left by step) are exempt;
216     // the penultimate measure (cadence) is exempt.
217     //
218     // Harmony of the half-measure: weak beat +1 reads the 1st-half downbeats; weak beats
219     // +2/+3 read the 2nd-half harmony (a 4th-species voice resolves on its 3rd beat).
220     int spc = sp->getSpecies();
221     vector<int> offsets;
222     if(spc == SECOND_SPECIES) offsets = {2};
```

```
223     else if(spc == THIRD_SPECIES) offsets = {1,2,3};
224     else return;
225 
226     // Diatonic pitch classes of the key, then the diatonic triads (stacked thirds).
227     bool present[12] = {false};
228     for(int n : sp->getScale()) present[((n % 12) + 12) % 12] = true;
229     vector<int> scalePCs;
230     for(int p = 0; p < 12; p++) if(present[p]) scalePCs.push_back(p);
231     int N = (int)scalePCs.size();
232     vector<vector<int>> triads;
233     if(N >= 5){ // genuine diatonic scale (7 for major/minor)
234         for(int j = 0; j < N; j++)
235             triads.push_back({scalePCs[j], scalePCs[(j+2)%N], scalePCs[(j+4)%N]});
236     }
237 
238     int nM = sp->getNMeasures();
239     for(int i = 0; i < nM-1; i++){
240         if(i == nM-2) continue; // penultimate measure exception (cadence)
241         for(int k : offsets){
242             int j = i*4 + k;
243             // passing-tone test (approached AND left by step)
244             BoolVar isPassing(home, 0, 1);
245             if(spc == THIRD_SPECIES){
246                 IntVar inMove  = expr(home, abs(sp->getNotes()[j]   - sp->getNotes()[j-1]));
247                 IntVar outMove = expr(home, abs(sp->getNotes()[j+1] - sp->getNotes()[j]));
248                 isPassing = expr(home, (inMove >= 1 && inMove <= 2) && (outMove >= 1 && outMove
    <= 2));
249             } else { // SECOND_SPECIES: neighbours of the arsis are the real (pinned) thesis
    notes
250                 IntVar inMove  = expr(home, abs(sp->getNotes()[i*4+2]   - sp->getNotes()[i*4]));
251                 IntVar outMove = expr(home, abs(sp->getNotes()[(i+1)*4] - sp->getNotes()[i*4+2])
    );
252                 isPassing = expr(home, (inMove >= 1 && inMove <= 2) && (outMove >= 1 && outMove
    <= 2));
253             }
254             // weak beat +1 belongs to the 1st-half harmony, +2/+3 to the 2nd-half harmony
255             bool firstHalf = (k == 1);
256             IntVar pc = expr(home, sp->getNotes()[j] % 12);
257 
258             // Fallback (old rule): weak-beat PC equals a sounding structural PC.
259             BoolVarArgs inChord;
260             for(Part* v : allVoices){
261                 int idx = firstHalf ? strongBeatIndex(v, i) : resolutionBeatIndex(v, i);
262                 inChord << expr(home, pc == (v->getNotes()[idx] % 12));
263             }
264             BoolVar anyInChord = expr(home, sum(inChord) >= 1);
265 
266             // Implied-triad branch: some diatonic triad T contains the weak-beat PC and
267             // every structural PC of the half-measure.
268             BoolVarArgs triadOk;
269             for(vector<int>& T : triads){
270                 IntArgs Targs(T);
271                 IntSet Tset(Targs);
272                 BoolVar weakInT(home, 0, 1);
273                 dom(home, pc, Tset, Reify(weakInT));
274                 BoolVarArgs structInT;
275                 for(Part* v : allVoices){
276                     int idx = firstHalf ? strongBeatIndex(v, i) : resolutionBeatIndex(v, i);
277                     IntVar spc_pc = expr(home, v->getNotes()[idx] % 12);
278                     BoolVar inT(home, 0, 1);
279                     dom(home, spc_pc, Tset, Reify(inT));
280                     structInT << inT;
281                 }
282                 BoolVar allStructInT = expr(home, sum(structInT) == (int)structInT.size());
283                 triadOk << expr(home, weakInT && allStructInT);
284             }
285             BoolVar inTriad = (triadOk.size() > 0)
286                 ? expr(home, sum(triadOk) >= 1)
287                 : BoolVar(home, 0, 0); // no diatonic triads -> rely on fallback only
288 
289             // disjunct => belongs to the (implied) chord of its half-measure
290             rel(home, isPassing, BOT_OR, expr(home, anyInChord || inTriad), 1);
291         }
292     }
293 }
294 
295 void H3_1_endWithPerfectConsonance(Home home, Part* part){
296     dom(home, part->getHIntervals()[part->getHIntervals().size()-1], IntSet(IntArgs(
    PERFECT_CONSONANCES)));
297 }
298 
299 void H3_2_penultimateNoteDomain(Home home, Part* part){
300     dom(home, expr(home, abs(part->getHIntervals()[part->getHIntervals().size()-5])), IntSet({
    UNISSON, PERFECT_FIFTH, MINOR_SIXTH, MAJOR_SIXTH}));
301 
302     rel(home, (part->getHIntervals()[part->getHIntervals().size()-5]!=PERFECT_FIFTH) >> (part->
    getPenultCostArray()[0]==part->getPenultCost()));
303     rel(home, (part->getHIntervals()[part->getHIntervals().size()-5]==PERFECT_FIFTH) >> (part->
    getPenultCostArray()[0]==0));
304 }
```

```
305
306 void H3_3_cambiataCost(Home home, Part* part){
307     for(int i = 0; i < part->getCambiataCostArray().size(); i++){
308         rel(home, ((part->getThirdSpeciesHIntervals()[(i*4)+1]==UNISSON || part->
    getThirdSpeciesHIntervals()[(i*4)+1]==PERFECT_FIFTH)&&(part->getThirdSpeciesHIntervals()[(i
    *4)+2]==UNISSON || part->getThirdSpeciesHIntervals()[(i*4)+2]==PERFECT_FIFTH)
309             &&(abs(part->getThirdSpeciesMIntervals()[(i*4)+1])<=2)) >> (part->
    getCambiataCostArray()[i]==part->getCambiataCost()));
310         rel(home, ((part->getThirdSpeciesHIntervals()[(i*4)+1]!=UNISSON && part->
    getThirdSpeciesHIntervals()[(i*4)+1]!=PERFECT_FIFTH)||(part->getThirdSpeciesHIntervals()[(i
    *4)+2]!=UNISSON && part->getThirdSpeciesHIntervals()[(i*4)+2]!=PERFECT_FIFTH)
311             ||(abs(part->getThirdSpeciesMIntervals()[(i*4)+1])>2)) >> (part->
    getCambiataCostArray()[i]==0));
312     }
313 }
314
315 void H5_1_cpAndCfDifferentNotes(Home home, Part* part, Part* cf){
316     // cout << part->getFirstSpeciesNotes() << endl;
317     for(int i = 1; i < part->getFirstSpeciesNotes().size()-1; i++){
318         rel(home, part->getFirstSpeciesNotes()[i], IRT_NQ, cf->getNotes()[i]);
319     }
320 }
321
322 /**
323  * Modified by Tom Lai
324  * Check that voices do not play the same notes (except for the first and last measure)
325  * @param home
326  * @param parts vector of parts with parts[0] being the cantusFirmus
327  */
328 void H5_1_differentNotes(Home home, vector<Part*> parts){
329     // iterate over all pairs of parts
330     for(int v1 = 0; v1 < parts.size(); v1++){
331         for(int v2 = v1+1; v2 < parts.size(); v2++){
332             if(parts[v1]->getSpecies()==CANTUS_FIRMUS){
333                 // check voice v2 doesn't play same note as cantusFirmus
334                 for(int i = 1; i < parts[v1]->getNotes().size()-1; i++){
335                     rel(home, parts[v1]->getNotes()[i], IRT_NQ, parts[v2]->getNotes()[i*4]);
336                 }
337             } else {
338                 // check voice v1 doesn't play same note as voice v2 with v1 and v2 are
    counterpoints
339                 for(int i = 4; i < parts[v1]->getNotes().size()-1; i+=4){
340                     rel(home, parts[v1]->getNotes()[i], IRT_NQ, parts[v2]->getNotes()[i]);
341                 }
342             }
343         }
344     }
345 }
346
347 void H6_1_preferImperfectConsonances(Home home, Part* part){
348     for(int i = 0; i < part->getHIntervals().size(); i++){
349         //set the octave cost
350         rel(home, part->getOctaveCostArray()[i], IRT_EQ, part->getHOctaveCost(), Reify(expr(home
    , part->getHIntervals()[i]==UNISSON), RM_PMI));
351         rel(home, part->getOctaveCostArray()[i], IRT_EQ, 0, Reify(expr(home, part->getHIntervals
    ()[i]!=UNISSON), RM_PMI));
352
353         //set the fifth cost
354         rel(home, part->getFifthCostArray()[i], IRT_EQ, part->getHFifthCost(), Reify(expr(home,
    part->getHIntervals()[i]==PERFECT_FIFTH), RM_PMI));
355         rel(home, part->getFifthCostArray()[i], IRT_EQ, 0, Reify(expr(home, part->getHIntervals
    ()[i]!=PERFECT_FIFTH), RM_PMI));
356     }
357 }
358
359 void H7_1_2v_penultimateSixthOrThird(Home home, Part* part){
360     // If the part is not the lowest voice, then it must have a major sixth in the penultimate
361     IntVar penultimateInterval(home, -PERFECT_OCTAVE, PERFECT_OCTAVE);
362     if(part->getSpecies()==FIRST_SPECIES){
363         penultimateInterval = part->getHIntervals()[part->getHIntervals().size()-5];
364     } else if(part->getSpecies()==SECOND_SPECIES || part->getSpecies()==FOURTH_SPECIES){
365         penultimateInterval = part->getHIntervals()[part->getHIntervals().size()-3];
366     } else if(part->getSpecies()==THIRD_SPECIES){
367         penultimateInterval = part->getHIntervals()[part->getHIntervals().size()-2];
368     } else if(part->getSpecies()==FIFTH_SPECIES){
369         rel(home, expr(home, part->getSpeciesArray()[part->getSpeciesArray().size()-3] ==
    FOURTH_SPECIES) >>
370                 (penultimateInterval == part->getHIntervals()[part->getHIntervals().size()-3]));
371         rel(home, expr(home, part->getSpeciesArray()[part->getSpeciesArray().size()-2] ==
    THIRD_SPECIES) >>
372                 (penultimateInterval == part->getHIntervals()[part->getHIntervals().size()-2]));
373     }
374     rel(home, (part->getIsNotLowest()[part->getIsNotLowest().size()-2]==1) >> (
    penultimateInterval==MAJOR_SIXTH));
375 }
376
377 void H7_1_3v_penultimateSixthOrThird(Home home, Part* part){
378     // If the part is not the lowest voice, then it must have a major sixth in the penultimate
    measure
379     // The penultimate measure is the second to last measure, so we check the second to last
```

```
    interval
380     // We also check that the penultimate interval is a minor third, perfect fifth, or major
    sixth
381     IntVar penultimateInterval(home, -PERFECT_OCTAVE, PERFECT_OCTAVE);
382     if(part->getSpecies()==FIRST_SPECIES){
383         penultimateInterval = part->getFirstSpeciesHIntervals()[part->getFirstSpeciesHIntervals
    ().size()-2];
384     } else if(part->getSpecies()==SECOND_SPECIES || part->getSpecies()==FOURTH_SPECIES){
385         penultimateInterval = part->getSecondHInterval()[part->getSecondHInterval().size()-2];
386     } else if(part->getSpecies()==THIRD_SPECIES){
387         penultimateInterval = part->getThirdSpeciesHIntervals()[part->getThirdSpeciesHIntervals
    ().size()-2];
388     } else if(part->getSpecies()==FIFTH_SPECIES){
389         rel(home, expr(home, part->getSpeciesArray()[part->getSpeciesArray().size()-3] ==
    FOURTH_SPECIES) >>
390                 (penultimateInterval == part->getHIntervals()[part->getHIntervals().size()-3]));
391         rel(home, expr(home, part->getSpeciesArray()[part->getSpeciesArray().size()-2] ==
    THIRD_SPECIES) >>
392                 (penultimateInterval == part->getHIntervals()[part->getHIntervals().size()-2]));
393     }
394     rel(home, (part->getIsNotLowest()[part->getIsNotLowest().size()-2]==1) >> (
    penultimateInterval==MAJOR_SIXTH || penultimateInterval==MINOR_THIRD || penultimateInterval
    ==MAJOR_THIRD || penultimateInterval==PERFECT_FIFTH));
395 }
396
397 void H8_3v_preferHarmonicTriad(Home home, Part* part, IntVarArray triadCostArray, Stratum*
    upper1, Stratum* upper2){
398     for(int i = 0; i < triadCostArray.size(); i++){
399         BoolVar h1_3 = BoolVar(home, 0 ,1);          //check if the first interval is a minor
    third
400         BoolVar h1_4 = BoolVar(home, 0 ,1);          //check if the first interval is a major
    third
401         BoolVar h1_third = BoolVar(home, 0 ,1);      //check is the first interval is a third
402         BoolVar h1_7 = BoolVar(home, 0 ,1);          //check if the first interval is a perfect
    fifth
403
404         BoolVar h2_3 = BoolVar(home, 0 ,1);          //check if the second interval is a minor
    third
405         BoolVar h2_4 = BoolVar(home, 0 ,1);          //check if the second interval is a major
    third
406         BoolVar h2_third = BoolVar(home, 0 ,1);      //check if the second interval is a third
407         BoolVar h2_7 = BoolVar(home, 0 ,1);          //check if the second interval is a perfect
    fifth
408
409         BoolVar h_firstPoss = BoolVar(home, 0, 1);  //check if the first interval is a third and
     the second a perfect fifth
410         BoolVar h_secondPoss = BoolVar(home, 0, 1); //check if the second interval is a third
    and the first a perfect fifth
411         BoolVar triad = BoolVar(home, 0, 1);         //check if it is a triad
412         BoolVar not_triad = BoolVar(home, 0, 1);     //check if it is not a triad
413
414         //check for the first possibility
415         rel(home, expr(home, abs(upper1->getHIntervals()[i*4])), IRT_EQ, 3, Reify(h1_3));
416         rel(home, expr(home, abs(upper1->getHIntervals()[i*4])), IRT_EQ, 4, Reify(h1_4));
417         rel(home, expr(home, abs(upper2->getHIntervals()[i*4])), IRT_EQ, 7, Reify(h2_7));
418         rel(home, h1_3, BOT_OR, h1_4, h1_third);
419         rel(home, h1_third, BOT_AND, h2_7, h_firstPoss);
420
421         //check for the second possibility
422         rel(home, expr(home, abs(upper2->getHIntervals()[i*4])), IRT_EQ, 3, Reify(h2_3));
423         rel(home, expr(home, abs(upper2->getHIntervals()[i*4])), IRT_EQ, 4, Reify(h2_4));
424         rel(home, expr(home, abs(upper1->getHIntervals()[i*4])), IRT_EQ, 7, Reify(h1_7));
425         rel(home, h2_3, BOT_OR, h2_4, h2_third);
426         rel(home, h2_third, BOT_AND, h1_7, h_secondPoss);
427
428         rel(home, h_firstPoss, BOT_OR, h_secondPoss, triad);   //is a triad if it is either the
    first or the second possibility
429         rel(home, triad, BOT_XOR, not_triad, 1);               //set not triad
430         rel(home, triadCostArray[i], IRT_EQ, 0, Reify(triad, RM_IMP));
431         rel(home, triadCostArray[i], IRT_EQ, part->getTriadCost(), Reify(not_triad, RM_IMP));
432     }
433 }
434
435 void H8_4v_preferHarmonicTriad(Home home, IntVarArray triadCostArray, Stratum* upper1, Stratum*
    upper2, Stratum* upper3){
436     // cout << upper1->getHIntervals().size() << endl;
437     // cout << upper2->getHIntervals().size() << endl;
438     // cout << upper3->getHIntervals().size() << endl;
439     for(int i = 0; i < triadCostArray.size(); i++){
440
441         IntVar H_b = upper1->getHIntervals()[i*4];
442         IntVar H_c = upper2->getHIntervals()[i*4];
443         IntVar H_d = upper3->getHIntervals()[i*4];
444
445
446         BoolVar H_b_is_third    = expr(home, H_b==MINOR_THIRD || H_b==MAJOR_THIRD);
447         BoolVar H_b_is_fifth    = expr(home, H_b==PERFECT_FIFTH);
448         BoolVar H_b_is_octave   = expr(home, H_b==UNISSON);
449
450         BoolVar H_c_is_third    = expr(home, H_c==MINOR_THIRD || H_c==MAJOR_THIRD);
```

```
451         BoolVar H_c_is_fifth    = expr(home, H_c==PERFECT_FIFTH);
452         BoolVar H_c_is_octave   = expr(home, H_c==UNISSON);
453 
454         BoolVar H_d_is_third    = expr(home, H_d==MINOR_THIRD || H_d==MAJOR_THIRD);
455         BoolVar H_d_is_fifth    = expr(home, H_d==PERFECT_FIFTH);
456         BoolVar H_d_is_octave   = expr(home, H_d==UNISSON);
457 
458 
459         // worst case : not a harmonic triad with at least a third and a fifth
460         BoolVar no_fifth_or_no_third = expr(home, H_b_is_third + H_c_is_third + H_d_is_third ==
    0 || H_b_is_fifth + H_c_is_fifth + H_d_is_fifth == 0);
461         BoolVar note_outside_harmonic_triad = expr(home, H_b_is_third + H_b_is_fifth +
    H_b_is_octave == 0 || H_c_is_third + H_c_is_fifth + H_c_is_octave == 0 || H_d_is_third +
    H_d_is_fifth + H_d_is_octave == 0);
462 
463         rel(home, (note_outside_harmonic_triad || no_fifth_or_no_third) >> (triadCostArray[i] ==
     not_harmonic_triad_cost));
464 
465         // now we are left with only combinations with at a third, a fifth, and another note of
    the harmonic triad.
466         // Doubling the fifth
467         rel(home, expr(home, (H_b_is_fifth + H_c_is_fifth + H_d_is_fifth == 2) && !
    note_outside_harmonic_triad && !no_fifth_or_no_third) >> (triadCostArray[i] ==
    double_fifths_cost));
468 
469         // Doubling the third, and ensure it is the same type of third (not a major and a minor)
470         rel(home, expr(home, (H_b_is_third + H_c_is_third + H_d_is_third == 2) && !
    note_outside_harmonic_triad && !no_fifth_or_no_third) >> (triadCostArray[i] ==
    double_thirds_cost));
471         rel(home, (H_b_is_third && H_c_is_third) >> (H_b == H_c));
472         rel(home, (H_b_is_third && H_d_is_third) >> (H_b == H_d));
473         rel(home, (H_c_is_third && H_d_is_third) >> (H_c == H_d));
474 
475         // triad with octave
476         rel(home, (H_b_is_fifth && H_c_is_third && H_d_is_octave) >> (triadCostArray[i] ==
    triad_with_octave_cost)); // 5 3 8
477         rel(home, (H_b_is_third && H_c_is_octave && H_d_is_fifth) >> (triadCostArray[i] ==
    triad_with_octave_cost)); // 3 8 5
478         rel(home, (H_b_is_third && H_c_is_fifth && H_d_is_octave) >> (triadCostArray[i] ==
    triad_with_octave_cost)); // 3 5 8
479         rel(home, (H_b_is_octave && H_c_is_third && H_d_is_fifth) >> (triadCostArray[i] ==
    triad_with_octave_cost)); // 8 3 5
480         rel(home, (H_b_is_octave && H_c_is_fifth && H_d_is_third) >> (triadCostArray[i] ==
    triad_with_octave_cost)); // 8 5 3
481 
482         // Best case : 5 8 3
483         rel(home, (H_b_is_fifth && H_c_is_octave && H_d_is_third) >> (triadCostArray[i] == 0));
484 
485     }
486 }
487 
488 void M1_1_2v_melodicIntervalsNotExceedMinorSixth(Home home, Part* part){
489     rel(home, part->getFirstSpeciesMIntervals(), IRT_LQ, MINOR_SIXTH);
490     rel(home, part->getFirstSpeciesMIntervals(), IRT_GQ, -MINOR_SIXTH);
491 }
492 
493 void M1_1_3v_melodicIntervalsNotExceedMinorSixth(Home home, Part* part){
494     dom(home, part->getMelodicIntervals(), IntSet({-UNISSON, -MINOR_SECOND, -MAJOR_SECOND, -
    MINOR_THIRD, -MAJOR_THIRD, -PERFECT_FOURTH, -TRITONE,
495             -PERFECT_FIFTH, -MINOR_SIXTH, -PERFECT_OCTAVE, UNISSON, MINOR_SECOND, MAJOR_SECOND,
    MINOR_THIRD, MAJOR_THIRD, PERFECT_FOURTH, TRITONE,
496             PERFECT_FIFTH, MINOR_SIXTH, PERFECT_OCTAVE}));
497 }
498 
499 void M1_2_octaveLeap(Home home, Part* part, Stratum* low){
500     for(int j = 0; j < part->getFirstSpeciesMIntervals().size(); j++){
501         rel(home, part->getFirstSpeciesMIntervals()[j], IRT_EQ, 12, Reify(expr(home,(abs(part->
    getFirstSpeciesHIntervals()[j])>=4)&&
502             (expr(home, expr(home, abs(low->getMelodicIntervals()[j])>0) == part->getIsNotLowest
    ()[j]))), RM_PMI));
503     }
504 }
505 
506 void M2_2_2v_twoConsecutiveNotesAreNotTheSame(Home home, Part* part){
507     rel(home, part->getSecondSpeciesMIntervals(), IRT_NQ, 0);
508 }
509 
510 void M2_2_3v_twoConsecutiveNotesAreNotTheSame(Home home, vector<Part*> parts, bool
    containsThirdSpecies){
511     for(int i = 1; i < parts.size(); i++){
512         if(parts[i]->getSpecies()==THIRD_SPECIES){
513             containsThirdSpecies=1;
514             break;
515         }
516     }
517     for(int p1 = 1; p1 < parts.size(); p1++){
518         for(int p2 = 1; p2 < parts.size(); p2++){
519             if(p1!=p2 && parts[p1]->getSpecies()==SECOND_SPECIES){
520                 if(!containsThirdSpecies){
521                     for(int i = 0; i < parts[p1]->getBranchingNotes().size()-4; i++){
```

```
522                         rel(home, parts[p1]->getMelodicIntervals().slice(0, notesPerMeasure.at(
    SECOND_SPECIES), parts[p1]->getMelodicIntervals().size())[i],
523                                 IRT_NQ, 0);
524                     }
525                 } else {
526                     for(int i = 0; i < parts[p1]->getBranchingNotes().size()-4; i++){
527                         rel(home, parts[p1]->getMelodicIntervals().slice(0, notesPerMeasure.at(
    SECOND_SPECIES), parts[p1]->getMelodicIntervals().size())[i],
528                                 IRT_NQ, 0);
529                     }
530                     //no unison in the two last notes
531                     rel(home, parts[p1]->getBranchingNotes()[parts[p1]->getBranchingNotes().size
    ()-2], IRT_NQ, parts[p1]->getBranchingNotes()[parts[p1]->getBranchingNotes().size()-1]);
532                 }
533             }
534         }
535     }
536 }
537
538 void M2_1_varietyCost(Home home, vector<Part*> parts){
539     for(int i = 1; i < parts.size(); i++){
540         Part* p = parts[i];
541         // cout << "Branching notes size : " << endl;
542         // cout << p->getBranchingNotes().size() << endl;
543         int temp = 0;
544         IntVarArray notes = p->getBranchingNotes();
545         for(int j = 0; j < p->getHIntervalSize()-1; j++){
546             // cout << j << endl;
547             int upbnd = 0;
548             if(j+3<p->getHIntervalSize()){
549                 upbnd = j+4;
550             } else {upbnd = p->getHIntervalSize();}
551             for(int k = j+1; k < upbnd;k++){
552                 //setting a cost if notes inside a window are the same in a part
553                 rel(home, (notes[j]==notes[k])>>(p->getVarietyArray(temp)==p->getVarietyCost()))
    ;
554                 rel(home, (notes[j]!=notes[k])>>(p->getVarietyArray(temp)==0));
555                 temp++;
556             }
557
558         }
559     }
560 }
561
562 void P1_1_2v_noDirectMotionFromPerfectConsonance(Home home, Part* part){
563     for(int j = 0; j < part->getFirstSpeciesMotions().size(); j++){
564
565         rel(home, ((part->getFirstSpeciesHIntervals()[j+1]==UNISSON || part->
    getFirstSpeciesHIntervals()[j+1]==PERFECT_FIFTH)&&part->getIsNotLowest()[j+1]==1) >>
566             (part->getFirstSpeciesMotions()[j]!=PARALLEL_MOTION));
567
568     }
569 }
570
571 void P1_1_3v_noDirectMotionFromPerfectConsonance(Home home, Part* part){
572     for(int j = 0; j < part->getFirstSpeciesMotions().size()-1; j++){
573         //set a cost when it is reached through direct motion, it is 0 when not
574         rel(home, (part->getFirstSpeciesMotions()[j]==2&&(part->getFirstSpeciesHIntervals()[j
    +1]==0||part->getFirstSpeciesHIntervals()[j+1]==7))>>
575             (part->getDirectCostArray()[j]==part->getDirectCost()));
576         rel(home, (part->getFirstSpeciesMotions()[j]!=2||(part->getFirstSpeciesHIntervals()[j
    +1]!=0&&part->getFirstSpeciesHIntervals()[j+1]!=7))>>
577             (part->getDirectCostArray()[j]==0));
578     }
579 }
580
581 void P1_1_4v_noDirectMotionFromPerfectConsonance(Home home, Part* part){
582     for(int j = 0; j < part->getFirstSpeciesMotions().size()-1; j++){
583
584         //if the counterpoint is the lowest part -> the cost is 0
585         rel(home, (!part->getIsNotLowest()[j]) >> (part->getDirectCostArray()[j]==0));
586
587         //if the counterpoint is neither the lowest nor the highest part, and the condition is
    true, then add a cost of 2
588         rel(home, (part->getIsNotLowest()[j] && !part->getIsHighest()[j] && (part->
    getFirstSpeciesMotions()[j]==2&&(part->getFirstSpeciesHIntervals()[j+1]==0||
589             expr(home, abs(part->getFirstSpeciesHIntervals()[j+1]))==7))) >> (part->
    getDirectCostArray()[j]==2));
590         //else the cost is 0
591         rel(home, (part->getIsNotLowest()[j] && !part->getIsHighest()[j] && (part->
    getFirstSpeciesMotions()[j]!=2||(part->getFirstSpeciesHIntervals()[j+1]!=0
592             && expr(home, abs(part->getFirstSpeciesHIntervals()[j+1]))!=7))) >> (part->
    getDirectCostArray()[j]==0));
593
594         //if the counterpoint is the highest part and the condition is true, then add a direct
    move cost (8 in this case)
595         rel(home, (part->getIsHighest()[j] && (part->getFirstSpeciesMotions()[j]==2&&(part->
    getFirstSpeciesHIntervals()[j+1]==0||
596             expr(home, abs(part->getFirstSpeciesHIntervals()[j+1]))==7))) >> (part->
    getDirectCostArray()[j]==part->getDirectMoveCost()));
```

```
597         //else the cost is 0
598         rel(home, (part->getIsHighest()[j] && (part->getFirstSpeciesMotions()[j]!=2||(part->
    getFirstSpeciesHIntervals()[j+1]!=0 &&
599             expr(home, abs(part->getFirstSpeciesHIntervals()[j+1]))!=7))) >> (part->
    getDirectCostArray()[j]==0));
600
601     }
602 }
603
604 void P1_2_2v_noDirectMotionFromPerfectConsonance(Home home, Part* part){
605     for(int j = 0; j < part->getSecondSpeciesRealMotions().size(); j++){
606         rel(home, ((part->getFirstSpeciesHIntervals()[j+1]==UNISSON || part->
    getFirstSpeciesHIntervals()[j+1]==PERFECT_FIFTH)&&part->getIsNotLowest()[j+1]==1) >>
607             (part->getSecondSpeciesRealMotions()[j]!=PARALLEL_MOTION));
608     }
609 }
610
611 void P1_2_3v_noDirectMotionFromPerfectConsonance(Home home, Part* part){
612
613     for(int j = 0; j < part->getFirstSpeciesMotions().size()-1; j++){
614         rel(home, (part->getSecondSpeciesRealMotions()[j]==2&&(part->getFirstSpeciesHIntervals()
    [j+1]==0||part->getFirstSpeciesHIntervals()[j+1]==7))>>
615             (part->getDirectCostArray()[j]==part->getDirectMoveCost()));
616         rel(home, (part->getSecondSpeciesRealMotions()[j]!=2||(part->getFirstSpeciesHIntervals()
    [j+1]!=0&&part->getFirstSpeciesHIntervals()[j+1]!=7))>>
617             (part->getDirectCostArray()[j]==0));
618     }
619
620 }
621
622 void P1_2_4v_noDirectMotionFromPerfectConsonance(Home home, Part* part){
623     for(int j = 0; j < part->getFirstSpeciesMotions().size()-1; j++){
624
625         rel(home, (!part->getIsNotLowest()[j]) >> (part->getDirectCostArray()[j]==0));
626
627         rel(home, (part->getIsNotLowest()[j] && !part->getIsHighest()[j] && (part->
    getSecondSpeciesRealMotions()[j]==2&&(part->getFirstSpeciesHIntervals()[j+1]==0||
628             part->getFirstSpeciesHIntervals()[j+1]==7))) >> (part->getDirectCostArray()[j]==2));
629         rel(home, (part->getIsNotLowest()[j] && !part->getIsHighest()[j] && (part->
    getSecondSpeciesRealMotions()[j]!=2||(part->getFirstSpeciesHIntervals()[j+1]!=0 &&
630             part->getFirstSpeciesHIntervals()[j+1]!=7))) >> (part->getDirectCostArray()[j]==0));
631
632         rel(home, (part->getIsHighest()[j] && (part->getSecondSpeciesRealMotions()[j]==2&&(part
    ->getFirstSpeciesHIntervals()[j+1]==0||
633             part->getFirstSpeciesHIntervals()[j+1]==7))) >> (part->getDirectCostArray()[j]==part
    ->getDirectMoveCost()));
634         rel(home, (part->getIsHighest()[j] && (part->getSecondSpeciesRealMotions()[j]!=2||(part
    ->getFirstSpeciesHIntervals()[j+1]!=0 &&
635             part->getFirstSpeciesHIntervals()[j+1]!=7))) >> (part->getDirectCostArray()[j]==0));
636
637     }
638 }
639
640 void P3_0_noBattuta(Home home, Part* part){
641     for(int j = 0; j < part->getMotions().size(); j++){
642         rel(home, expr(home, part->getMotions()[j]==CONTRARY_MOTION && part->getHIntervals()[j
    +1]==0 &&
643             part->getMelodicIntervals()[j]<-4), BOT_AND, part->getIsNotLowest()[j], 0);
644     }
645 }
646
647 void P3_1_noBattuta(Home home, Part* part){
648     for(int j = 0; j < part->getFirstSpeciesMotions().size(); j++){
649         rel(home, expr(home, part->getFirstSpeciesMotions()[j]==CONTRARY_MOTION && part->
    getFirstSpeciesHIntervals()[j+1]==0 &&
650             part->getFirstSpeciesMIntervals()[j]<-4), BOT_AND, part->getIsNotLowest()[j], 0);
651     }
652 }
653
654 void P3_2_noBattuta(Home home, Part* part){
655     for(int j = 0; j < part->getSecondSpeciesMotions().size(); j++){
656         rel(home, expr(home, part->getSecondSpeciesMotions()[j]==CONTRARY_MOTION && part->
    getFirstSpeciesHIntervals()[j+1]==0 &&
657             part->getFirstSpeciesMIntervals()[j]<-4), BOT_AND, part->getIsNotLowest()[j], 0);
658     }
659 }
660
661 void setCostFromBool(Home home, BoolVar trigger, IntVar target, int cost) {
662     rel(home, (trigger == 1) >> (target == cost));
663     rel(home, (trigger == 0) >> (target == 0));
664 }
665
666
667 void applySecondSpeciesSuccessiveCost(Home home, Part* secondSpeciesPart, const IntVarArray&
    hIntervals, const BoolVarArray& isPerfectConsonance, IntVarArray successiveCostArray, int&
    idx, int appliedCost) {
668     for (int i = 0; i < isPerfectConsonance.size()-1; i++) {
669         BoolVar firstNotFifth(home, 0, 1);
670         BoolVar secondNotFifth(home, 0, 1);
671         BoolVar notSuccessiveFifths(home, 0, 1);
```

```
672
673         BoolVar successivePerfect(home, 0, 1);
674         BoolVar successivePerfectNotFifths(home, 0, 1);
675
676         BoolVar melodicNotThree(home, 0, 1);
677         BoolVar melodicNotFour(home, 0, 1);
678         BoolVar melodicNotThird(home, 0, 1);
679
680         BoolVar firstFifth(home, 0, 1);
681         BoolVar secondFifth(home, 0, 1);
682         BoolVar successiveFifths(home, 0, 1);
683         BoolVar successiveFifthsWithoutThirdMotion(home, 0, 1);
684
685         BoolVar applyCost(home, 0, 1);
686
687         // Are the two harmonic intervals successive fifths
688         rel(home, hIntervals[i], IRT_NQ, 7, Reify(firstNotFifth));
689         rel(home, hIntervals[i+1], IRT_NQ, 7, Reify(secondNotFifth));
690         rel(home, firstNotFifth, BOT_OR, secondNotFifth, notSuccessiveFifths);
691
692         // Are they two successive perfect consonances
693         rel(home, isPerfectConsonance[i], BOT_AND, isPerfectConsonance[i+1], successivePerfect);
694         rel(home, successivePerfect, BOT_AND, notSuccessiveFifths, successivePerfectNotFifths);
695
696         // Special exception in second species:
697         // if melodic interval is != 3 and != 4
698         rel(home, secondSpeciesPart->getMelodicIntervals()[i*2], IRT_NQ, 3, melodicNotThree);
699         rel(home, secondSpeciesPart->getMelodicIntervals()[i*2], IRT_NQ, 4, melodicNotFour);
700         rel(home, melodicNotThree, BOT_AND, melodicNotFour, melodicNotThird);
701
702         rel(home, hIntervals[i], IRT_EQ, 7, Reify(firstFifth));
703         rel(home, hIntervals[i+1], IRT_EQ, 7, Reify(secondFifth));
704         rel(home, firstFifth, BOT_AND, secondFifth, successiveFifths);
705         rel(home, melodicNotThird, BOT_AND, successiveFifths, successiveFifthsWithoutThirdMotion
    );
706
707         rel(home, successivePerfectNotFifths, BOT_OR, successiveFifthsWithoutThirdMotion,
    applyCost);
708
709         setCostFromBool(home, applyCost, successiveCostArray[idx], appliedCost);
710         idx++;
711     }
712 }
713
714 void P4_successiveCost(Home home, vector<Part*> parts, IntVarArray successiveCostArray){
715     int idx = 0;
716
717     for (int v1 = 1; v1 < parts.size(); v1++) {
718         for (int v2 = v1+1; v2 < parts.size(); v2++) {
719             Part* p1 = parts[v1];
720             Part* p2 = parts[v2];
721
722             IntVarArray notes1 = p1->getNotes();
723             IntVarArray notes2 = p2->getNotes();
724             if (notes1.size() != notes2.size()){ // p1 is cantus firmus
725                 notes1 = expandCantusNotes(home, notes1);
726             }
727
728             const int nMeasures = p1->getNMeasures();
729             const bool p1Second = p1->getSpecies() == SECOND_SPECIES;
730             const bool p2Second = p2->getSpecies() == SECOND_SPECIES;
731             const bool p1Fourth = p1->getSpecies() == FOURTH_SPECIES;
732             const bool p2Fourth = p2->getSpecies() == FOURTH_SPECIES;
733
734             IntVarArray hIntervals12(home, nMeasures, 0, MAJOR_SEVENTH);
735             if (p1Fourth || p2Fourth) {
736                 // For fourth species, compare the syncopated note (index +2) with the
737                 // structurally relevant note of the other voice, measure by measure.
738                 for (int i = 0; i < nMeasures-1; i++) {
739                     const int idx1 = p1Fourth ? i*4+2 : i*4;
740                     const int idx2 = p2Fourth ? i*4+2 : i*4;
741                     rel(home, hIntervals12[i] == (abs(notes1[idx1] - notes2[idx2]) % 12));
742                 }
743                 rel(home, hIntervals12[nMeasures-1] == (abs(notes1[notes1.size()-1] - notes2[
    notes2.size()-1]) % 12)); // Final note
744             } else {
745                 for (int i = 0; i < nMeasures; i++) {
746                     rel(home, hIntervals12[i] == (abs(notes1[i*4] - notes2[i*4]) % 12));
747                 }
748             }
749
750             BoolVarArray isPCons12(home, hIntervals12.size(), 0, 1);
751             for (int i = 0; i < hIntervals12.size(); i++) {
752                 rel(home, expr(home, hIntervals12[i] == UNISSON), BOT_OR, expr(home,
    hIntervals12[i] == PERFECT_FIFTH), isPCons12[i]);
753             }
754
755             // ----- Applying constrains -----
756             if (p1Fourth || p2Fourth) { // Second species has priorities because it accepts more
757                 for (int i = 0; i < hIntervals12.size()-1; i++) {
```

```
758                     BoolVar firstNotFifth(home, 0, 1);
759                     BoolVar secondNotFifth(home, 0, 1);
760                     BoolVar notSuccessiveFifths(home, 0, 1);
761 
762                     BoolVar successivePerfect(home, 0, 1);
763                     BoolVar successivePerfectNotFifths(home, 0, 1);
764 
765                     rel(home, hIntervals12[i], IRT_NQ, 7, Reify(firstNotFifth));
766                     rel(home, hIntervals12[i+1], IRT_NQ, 7, Reify(secondNotFifth));
767                     rel(home, firstNotFifth, BOT_OR, secondNotFifth, notSuccessiveFifths);
768 
769                     rel(home, isPCons12[i], BOT_AND, isPCons12[i+1], successivePerfect);
770                     rel(home, successivePerfect, BOT_AND, notSuccessiveFifths,
    successivePerfectNotFifths);
771 
772                     setCostFromBool(home, successivePerfectNotFifths, successiveCostArray[idx],
    p1->getSuccCost());
773                     idx++;
774                 }
775             } else if (p1Second || p2Second){
776                 if (p1Second) {
777                     applySecondSpeciesSuccessiveCost(home, p1, hIntervals12, isPCons12,
    successiveCostArray, idx, p1->getSuccCost());
778                 }
779                 if (p2Second) {
780                     applySecondSpeciesSuccessiveCost(home, p2, hIntervals12, isPCons12,
    successiveCostArray, idx, p1->getSuccCost());
781                 }
782             } else {
783                 for (int i = 0; i < isPCons12.size()-1; i++) {
784                     BoolVar successivePerfect(home, 0, 1);
785                     rel(home, isPCons12[i], BOT_AND, isPCons12[i+1], successivePerfect);
786                     setCostFromBool(home, successivePerfect, successiveCostArray[idx], p1->
    getSuccCost());
787                     idx++;
788                 }
789             }
790         }
791     }
792 }
793 
794 void P6_3v_noMoveInSameDirection(Home home, vector<Part*> parts){
795     for(int i = 0; i < parts[0]->getFirstSpeciesMotions().size(); i++){
796         rel(home, expr(home, parts[0]->getFirstSpeciesMotions()[i]==2 && parts[1]->
    getFirstSpeciesMotions()[i]==2), BOT_AND, expr(home, parts[2]->getFirstSpeciesMotions()[i
    ]==2), 0);
797     }
798 }
799 
800 void P6_4v_noMoveInSameDirection(Home home, vector<Part*> parts){
801     for(int i = 0; i < parts[0]->getFirstSpeciesMotions().size(); i++){
802         rel(home, expr(home, parts[0]->getFirstSpeciesMotions()[i]==2 && parts[1]->
    getFirstSpeciesMotions()[i]==2 && parts[2]->getFirstSpeciesMotions()[i]==2), BOT_AND, expr(
    home, parts[3]->getFirstSpeciesMotions()[i]==2), 0);
803     }
804 }
805 
806 void P7_noSuccessiveAscendingSixths(Home home, vector<Part*> parts){
807     for(int p1 = 0; p1 < parts.size(); p1++){
808         for(int p2 = p1+1; p2 < parts.size(); p2++){
809             for(int j = 1; j < parts[p1]->getFirstHInterval().size()-1; j++){
810                 rel(home, ((parts[p1]->getFirstHInterval()[j-1]!=MINOR_SIXTH && parts[p1]->
    getFirstHInterval()[j-1]!=MAJOR_SIXTH) &&
811                     (parts[p2]->getFirstHInterval()[j-1]!=MINOR_SIXTH && parts[p2]->
    getFirstHInterval()[j-1]!=MAJOR_SIXTH)) || (
812                         (parts[p1]->getFirstHInterval()[j]!=-MINOR_SIXTH && parts[p1]->
    getFirstHInterval()[j]!=-MAJOR_SIXTH) &&
813                     (parts[p2]->getFirstHInterval()[j]!=- MINOR_SIXTH && parts[p2]->
    getFirstHInterval()[j]!=-MAJOR_SIXTH)) || (
814                         parts[p1]->getFirstMInterval()[j]>0 || parts[p2]->getFirstMInterval()[j]
     > 0));
815             }
816         }
817     }
818 }
819 
820 void noSuccessiveSamePerfectIntervalOnIndices(Home home, Part* p1, Part* p2, vector<int>&
    indices) {
821     int nIndices = indices.size();
822     bool p1Fourth = p1->getSpecies() == FOURTH_SPECIES;
823     bool p2Fourth = p2->getSpecies() == FOURTH_SPECIES;
824     if (p1Fourth || p2Fourth) { return; } // Authorized with 4th species (Bitsch, rule 43, p.25)
825 
826     IntVarArray notes1 = p1->getNotes();
827     IntVarArray notes2 = p2->getNotes();
828     if (notes1.size() < notes2.size()){ // p1 is cantus firmus
829         notes1 = expandCantusNotes(home, notes1);
830     }
831     if (notes1.size() > notes2.size()){ // p2 is cantus firmus
832         notes2 = expandCantusNotes(home, notes2);
```

```
833     }
834
835     // Building intervals array
836     IntVarArray hIntervals12(home, nIndices, 0, MAJOR_SEVENTH);
837     for (int i = 0; i < nIndices-1; i++) {
838         int idx = indices[i];
839         rel(home, hIntervals12[i] == (abs(notes2[idx] - notes1[idx]) % 12));
840     }
841     rel(home, hIntervals12[nIndices-1] == (abs(notes2[indices.back()] - notes1[indices.back()])
    % 12)); // Final note
842
843     // Constraint (Bitsch, rule 42, p.24)
844     for (int i = 0; i < hIntervals12.size()-1; i++) {
845         rel(home, expr(home, hIntervals12[i] == UNISSON), BOT_AND, expr(home, hIntervals12[i+1]
    == UNISSON), 0); // No successive unisson
846         rel(home, expr(home, hIntervals12[i] == PERFECT_FIFTH), BOT_AND, expr(home, hIntervals12
    [i+1] == PERFECT_FIFTH), 0); // No successive fifth
847         rel(home, expr(home, hIntervals12[i] == TRITONE), BOT_AND, expr(home, hIntervals12[i+1]
    == PERFECT_FIFTH), 0); // No  fifth --> perfect fifth
848     }
849 }
850
851 void P8_noSuccessiveSamePerfectInterval(Home home, vector<Part*> parts) {
852     for (int v1 = 0; v1 < parts.size(); v1++) {
853         for (int v2 = v1 + 1; v2 < parts.size(); v2++) {
854             Part* p1 = parts[v1];
855             Part* p2 = parts[v2];
856
857             // ----- successive beats -----
858             vector<int> beatIndices = createRangeVector(0, 2*(p1->getNMeasures()-1), 2); //
    thesis + arsis
859             beatIndices.push_back(4*(p1->getNMeasures()-1)); // last thesis
860             noSuccessiveSamePerfectIntervalOnIndices(home, p1, p2, beatIndices);
861
862             // ----- successive notes -----
863             bool p1Third = p1->getSpecies() == THIRD_SPECIES;
864             bool p2Third = p2->getSpecies() == THIRD_SPECIES;
865             if (p1Third || p2Third){ // If no 3d species, already taken care of in arsis or
    thesis
866                 const int nMeasures = p1->getNMeasures();
867                 vector<int> noteIndices = createRangeVector(0, nMeasures*4-3, 1);
868                 noSuccessiveSamePerfectIntervalOnIndices(home, p1, p2, noteIndices);
869             }
870         }
871     }
872 }
873
874 void noSimultaneousRepetitionOnIndices(Home home, Part* p1, Part* p2, const vector<int>& indices
    ) {
875     // P9 forbids two voices from BOTH repeating their note across the same step (a fully
876     // static instant). The check compares note VALUES at successive beats, so it cannot tell a
877     // re-articulated note (a real repetition) from a merely SUSTAINED one (a held cantus-firmus
878     // whole note, or a tie across the bar). Species that rely on ties trip this false positive:
879     //    - 4th species: the suspension tie is mandatory (arsis_m == downbeat_{m+1}); against the
880     //      held CF the AND-of-repetitions is true on every measure -> infeasible everywhere.
881     //    - 5th species: ties are optional, so it stays feasible, but P9 effectively bans every
882     //      suspension against the (always held) CF, suppressing legitimate florid writing.
883     // Both must therefore be exempt. (Authors already noted "If 4th species, doesn't really
    have
884     // sense"; the same reasoning extends to the suspensions of 5th species.)
885     Species s1 = (Species)p1->getSpecies(), s2 = (Species)p2->getSpecies();
886     if (s1 == FOURTH_SPECIES || s2 == FOURTH_SPECIES ||
887         s1 == FIFTH_SPECIES  || s2 == FIFTH_SPECIES) return;
888
889     IntVarArray notes1 = p1->getNotes();
890     IntVarArray notes2 = p2->getNotes();
891
892     if (notes1.size() != notes2.size()){ // p1 is cantus firmus
893         notes1 = expandCantusNotes(home, notes1);
894     }
895
896     for (int i = 0; i < (int)indices.size()-1; i++) {
897         const int t1 = indices[i];
898         const int t2 = indices[i+1];
899
900         BoolVar p1Repeats(home, 0, 1);
901         BoolVar p2Repeats(home, 0, 1);
902
903         rel(home, notes1[t1], IRT_EQ, notes1[t2], Reify(p1Repeats));
904         rel(home, notes2[t1], IRT_EQ, notes2[t2], Reify(p2Repeats));
905
906         rel(home, p1Repeats, BOT_AND, p2Repeats, 0); // Hard forbid
907     }
908 }
909
910 void P9_noSimultaneousRepetition(Home home, vector<Part*> parts) {
911     int count = 0;
912     for (int v1 = 0; v1 < parts.size(); v1++) {
913         for (int v2 = v1 + 1; v2 < parts.size(); v2++) {
914             count++;
```

```
915             Part* p1 = parts[v1];
916             Part* p2 = parts[v2];
917 
918             // ----- successive beats -----
919             vector<int> thesisIndices = createRangeVector(0, p1->getNMeasures(), 4);
920             noSimultaneousRepetitionOnIndices(home, p1, p2, thesisIndices);
921 
922             // Checking again with arsis beats
923             vector<int> beatIndices = createRangeVector(0, 2*(p1->getNMeasures()-1), 2); //
    thesis + arsis
924             beatIndices.push_back(4*(p1->getNMeasures()-1)); // last thesis
925             noSimultaneousRepetitionOnIndices(home, p1, p2, beatIndices);
926 
927             // ----- successive notes -----
928             bool p1Third = p1->getSpecies() == THIRD_SPECIES;
929             bool p2Third = p2->getSpecies() == THIRD_SPECIES;
930             bool p1Fourth = p1->getSpecies() == FOURTH_SPECIES;
931             bool p2Fourth = p2->getSpecies() == FOURTH_SPECIES;
932 
933             // If no 3d species, already taken care of in arsis or thesis
934             // If 4th species, doesn't really have sense
935             if ((p1Third || p2Third) && !p1Fourth && !p2Fourth){
936                 const int nMeasures = p1->getNMeasures();
937                 vector<int> noteIndices = createRangeVector(0, nMeasures*4-3, 1);
938                 noSimultaneousRepetitionOnIndices(home, p1, p2, noteIndices);
939             }
940         }
941     }
942 }
943 
944 /* ================================================
945  *          SOFT (RELAXED) CONSTRAINT VERSIONS
946  * ================================================
947  */
948 
949 void G9_lastChordSameAsFundamental_soft(Home home, Stratum* lowest, Part* cantusFirmus,
    IntVarArray relaxCosts, int startIdx){
950     // Soft version: instead of hard equality, create BoolVar for each violation
951     // relaxCosts[startIdx] = 1 if last note pitch class doesn't match
952     // relaxCosts[startIdx+1] = 1 if first note pitch class doesn't match
953 
954     BoolVar lastMatch(home, 0, 1);
955     rel(home, expr(home, lowest->getNotes()[lowest->getNotes().size()-1]%12), IRT_EQ,
956         expr(home, cantusFirmus->getNotes()[cantusFirmus->getNotes().size()-1]%12), Reify(
    lastMatch, RM_EQV));
957     // violation = !match => cost = 1 - match
958     rel(home, (lastMatch == 1) >> (relaxCosts[startIdx] == 0));
959     rel(home, (lastMatch == 0) >> (relaxCosts[startIdx] == 1));
960 
961     BoolVar firstMatch(home, 0, 1);
962     rel(home, expr(home, lowest->getNotes()[0]%12), IRT_EQ,
963         expr(home, cantusFirmus->getNotes()[0]%12), Reify(firstMatch, RM_EQV));
964     rel(home, (firstMatch == 1) >> (relaxCosts[startIdx+1] == 0));
965     rel(home, (firstMatch == 0) >> (relaxCosts[startIdx+1] == 1));
966 }
967 
968 void M2_2_3v_twoConsecutiveNotesAreNotTheSame_soft(Home home, vector<Part*> parts, bool
    containsThirdSpecies){
969     // Soft version: instead of hard IRT_NQ 0, count violations
970     for(int i = 1; i < parts.size(); i++){
971         if(parts[i]->getSpecies()==THIRD_SPECIES){
972             containsThirdSpecies=1;
973             break;
974         }
975     }
976     for(int p1 = 1; p1 < parts.size(); p1++){
977         for(int p2 = 1; p2 < parts.size(); p2++){
978             if(p1!=p2 && parts[p1]->getSpecies()==SECOND_SPECIES){
979                 int nConstraints = parts[p1]->getBranchingNotes().size()-4;
980                 if(containsThirdSpecies) nConstraints += 1; // +1 for the last two notes
981                 parts[p1]->initRelaxationCostArray(home, nConstraints);
982 
983                 int relaxIdx = 0;
984                 for(int i = 0; i < (int)parts[p1]->getBranchingNotes().size()-4; i++){
985                     BoolVar isZero(home, 0, 1);
986                     rel(home, parts[p1]->getMelodicIntervals().slice(0, notesPerMeasure.at(
    SECOND_SPECIES), parts[p1]->getMelodicIntervals().size())[i],
987                         IRT_EQ, 0, Reify(isZero, RM_EQV));
988                     // violation when interval == 0
989                     rel(home, (isZero == 1) >> (parts[p1]->getRelaxationCostArray()[relaxIdx] ==
     1));
990                     rel(home, (isZero == 0) >> (parts[p1]->getRelaxationCostArray()[relaxIdx] ==
     0));
991                     relaxIdx++;
992                 }
993                 if(containsThirdSpecies){
994                     // soft version of: last two notes must be different
995                     BoolVar lastTwoSame(home, 0, 1);
996                     rel(home, parts[p1]->getBranchingNotes()[parts[p1]->getBranchingNotes().size
    ()-2], IRT_EQ,
```

```
                        parts[p1]->getBranchingNotes()[parts[p1]->getBranchingNotes().size()-1],
     Reify(lastTwoSame, RM_EQV));
                    rel(home, (lastTwoSame == 1) >> (parts[p1]->getRelaxationCostArray()[
    relaxIdx] == 1));
                    rel(home, (lastTwoSame == 0) >> (parts[p1]->getRelaxationCostArray()[
    relaxIdx] == 0));
                    relaxIdx++;
                }
            }
        }
    }
}

void H2_3_dissonanceImpliesDiminution_soft(Home home, Part* part){
    // Soft version of the in-species H2_3 (central weak beat only). See the note on the hard
    // version: full weak-beat consonance for 2-voice 3rd species is enforced by the
    // multi-voice variant from TwoVoiceCounterpoint.
    int nConstraints = part->getIsDiminution().size();
    part->initRelaxationCostArray(home, nConstraints);

    for(int i = 0; i < part->getIsDiminution().size(); i++){
        BoolVar ok(home, 0, 1);
        rel(home, part->getConsonance()[(i*4)+2], BOT_OR, part->getIsDiminution()[i], ok);
        rel(home, (ok == 1) >> (part->getRelaxationCostArray()[i] == 0));
        rel(home, (ok == 0) >> (part->getRelaxationCostArray()[i] == 1));
    }
}

void H2_2_arsisHarmoniesCannotBeDissonant_soft(Home home, Part* part){
    // Soft version: instead of hard reify RM_PMI, count violations when arsis is dissonant and
    not diminution
    int nConstraints = 0;
    for(int i = 0; i < part->getNMeasures()-1; i++){
        if(i != part->getNMeasures()-2) nConstraints++;
    }
    part->initRelaxationCostArray(home, nConstraints);

    int relaxIdx = 0;
    for(int i = 0; i < part->getNMeasures()-1; i++){
        if(i != part->getNMeasures()-2){
            // Original uses multiple rel() with RM_PMI per dissonance
            // Soft: check if arsis is dissonant and not matched by diminution => violation
            // We reify the original constraint: consonance at arsis OR isDiminution
            // If not consonant at arsis AND not diminution => violation
            BoolVar arsisConsonant(home, 0, 1);
            dom(home, part->getHIntervals()[(i*4)+2],
                IntSet({UNISSON, MINOR_THIRD, MAJOR_THIRD, PERFECT_FIFTH, MINOR_SIXTH,
    MAJOR_SIXTH, PERFECT_OCTAVE,
                    -MINOR_THIRD, -MAJOR_THIRD, -PERFECT_FIFTH, -MINOR_SIXTH, -MAJOR_SIXTH, -
    PERFECT_OCTAVE}),
                arsisConsonant);
            // ok = consonant OR diminution
            BoolVar ok(home, 0, 1);
            rel(home, arsisConsonant, BOT_OR, part->getIsDiminution()[i], ok);
            rel(home, (ok == 1) >> (part->getRelaxationCostArray()[relaxIdx] == 0));
            rel(home, (ok == 0) >> (part->getRelaxationCostArray()[relaxIdx] == 1));
            relaxIdx++;
        }
    }
}
```

A.9.2 Utilities.cpp

```
//
// Created by Luc Cleenewerk and Diego de Patoul.
// This file contains the implementations of general utility functions.
//

#include "../headers/Utilities.hpp"

vector<bool> activeConstraints = std::vector<bool>(consSize, false);
vector<bool> softConstraints = std::vector<bool>(consSize, false);

string get_constraint_name(int constraint) {
    if (constraint < 0 || constraint >= constraintNames.size()) {
        return "Unknown constraint";
    }
    return constraintNames[constraint];
}

/**
 * For a given set of intervals between notes that loops and a starting note, returns all the
    possible notes
 * @param note the starting note
 * @param intervals the set of intervals between notes. It must make a loop. For example, to get
     all notes from a major
```

```
 * scale from note, use {2, 2, 1, 2, 2, 2, 1}. To get all notes from a minor chord, use {3, 4, 5}.
 * @return vector<int> all the notes
 */
vector<int> get_all_notes_from_interval_loop(int n, vector<int> intervals)
{
    int note = n % PERFECT_OCTAVE; // bring the root back to [12,23] in case the argument is wrong
    vector<int> notes;

    int i = 0;
    while (note <= 127)
    {
        notes.push_back(note);
        note += intervals[i % intervals.size()];
        ++i;
    }
    return notes;
}

/**
 * For a given tonality (root + mode), returns all the possible notes
 * @param root the root of the tonality (in [0,11])
 * @param scale the set of tones and semitones that define the scale
 * @return vector<int> all the possible notes from that tonality
 */
vector<int> get_all_notes_from_scale(int root, vector<int> scale)
{
    return get_all_notes_from_interval_loop(root, scale);
}


// Liste des gammes candidates pour la d tection automatique.
// Les noms affich s utilisent "Majeur" / "Mineur" (au lieu d'Ionien/Aeolien)
// pour la lisibilit  utilisateur. Les pentatoniques et blues sont incluses.
static const vector<pair<string, vector<int>>> SCALE_CANDIDATES = {
    {"Majeur",                MAJOR_SCALE},        // = Ionien
    {"Mineur naturel",        AEOLIAN_SCALE},      // = mineur naturel
    {"Dorien",                DORIAN_SCALE},
    {"Mixolydien",            MIXOLYDIAN_SCALE},
    {"Lydien",                LYDIAN_SCALE},
    {"Phrygien",              PHRYGIAN_SCALE},
    {"Locrien",               LOCRIAN_SCALE},
    {"Mineur harmonique",     HARMONIC_MINOR_SCALE},
    {"Mineur m lodique",      MELODIC_MINOR_SCALE},
    {"Pentatonique majeure",  PENTATONIC_MAJOR_SCALE},
    {"Pentatonique mineure",  PENTATONIC_MINOR_SCALE},
    {"Blues majeure",         BLUES_MAJOR_SCALE},
    {"Blues mineure",         BLUES_MINOR_SCALE}
};

static pair<string, vector<int>> detect_scale_pair(const vector<int>& cf) { // Pair name, scale
    if (cf.empty()) {
        return {"Majeur [fallback CF vide]", MAJOR_SCALE};
    }
    int root = ((cf[0] % 12) + 12) % 12;

    // Classes de hauteurs pr sentes dans le CF
    set<int> cf_pcs;
    for (int n : cf) cf_pcs.insert(((n % 12) + 12) % 12);

    // On cherche la gamme qui couvre le plus grand nombre de notes du CF.
    // En cas d'  galit    on garde la premi re candidate (= la plus  g  n  rale).
    int best_covered = -1;
    int best_idx = -1;
    for (size_t i = 0; i < SCALE_CANDIDATES.size(); ++i) {
        set<int> scale_pcs;
        int cur = root;
        scale_pcs.insert(cur);
        for (int step : SCALE_CANDIDATES[i].second) {
            cur = (cur + step) % 12;
            scale_pcs.insert(cur);
        }
        int covered = 0;
        for (int pc : cf_pcs) {
            if (scale_pcs.find(pc) != scale_pcs.end()) ++covered;
        }
        if (covered > best_covered) {
            best_covered = covered;
            best_idx = static_cast<int>(i);
            if (covered == static_cast<int>(cf_pcs.size())) {
                // couverture parfaite : on s'arr te (gamme la plus  g  n  rale qui couvre tout)
                break;
            }
        }
    }
    if (best_idx < 0) {
        return {"Majeur [fallback]", MAJOR_SCALE};
    }
    auto chosen = SCALE_CANDIDATES[best_idx];
```

```
110     if (best_covered < static_cast<int>(cf_pcs.size())) {
111         chosen.first += " [couverture partielle " + std::to_string(best_covered)
112                         + "/" + std::to_string(cf_pcs.size()) + "]";
113     }
114     return chosen;
115 }
116
117 vector<int> detect_scale_for_cf(const vector<int>& cf) {
118     return detect_scale_pair(cf).second;
119 }
120
121 string detect_scale_name_for_cf(const vector<int>& cf) {
122     return detect_scale_pair(cf).first;
123 }
124
125 /**
126  * For a given chord (root + mode), returns all the possible notes
127  * @param root the root of the chord
128  * @param quality the set of tones and semitones that define the chord
129  * @return vector<int> all the possible notes from that chord
130  */
131 vector<int> get_all_notes_from_chord(int root, vector<int> quality)
132 {
133     return get_all_notes_from_interval_loop(root, quality);
134 }
135
136 /**
137  * Get all values for a given note
138  * @param note a note
139  * @return vector<int> a vector containing all the given notes
140  */
141 vector<int> get_all_given_note(int note)
142 {
143     int current = note % PERFECT_OCTAVE;
144     vector<int> notes;
145     while (current < 127)
146     {
147         notes.push_back(current);
148         current += 12;
149     }
150     return notes;
151 }
152
153 /**
154  * Transforms a vector of integers into a string
155  * @param vector a vector of integers
156  * @return string the string representation of the vector
157  */
158 string int_vector_to_string(vector<int> vector){
159     string s;
160     for (int i = 0; i < vector.size(); ++i) {
161         s += to_string(vector[i]);
162         if(i != vector.size() - 1)
163             s += " , ";
164     }
165     return s;
166 }
167
168 /**
169  * Transforms an IntVarArray into a string, with values separated by a space (for csv
      utilisation)
170  * @param array IntVarArray
171  * @return string the string representation of the vector
172  */
173 string int_var_array_to_string(IntVarArray array){
174     ostringstream oss;
175     oss << array;
176     string my_str = oss.str();
177     return my_str.substr(1,my_str.size()-2);
178 }
179
180 /**
181  * Transforms an int* into a vector<int>
182  * @param ptr an int* pointer
183  * @param size the size of the array
184  * @return a vector<int> containing the same values as the array
185  */
186 vector<int> int_pointer_to_vector(int* ptr, int size){
187     vector<int> v;
188     for(int i = 0; i < size; i++){
189         v.push_back(ptr[i]);
190     }
191     return v;
192 }
193
194 /**
195  * Union algorithm found and adapted from GeeksForGeeks.com
196  */
197 vector<int> vector_union(vector<int> v1, vector<int> v2){
198     vector<int> v(v1.size()
```

```
                    + v2.size());
    vector<int>::iterator it, st;

    it = set_union(v1.begin(), v1.end(),
                   v2.begin(),
                   v2.end(), v.begin());

    vector<int> res = {};
    for(st = v.begin(); st != it; ++st){
        res.push_back(*st);
    }
    return res;
}

/**
 * Intersection algorithm found and adapted from GeeksForGeeks.com
 */
vector<int> vector_intersection(vector<int> v1, vector<int> v2){
    vector<int> v(v1.size()
                  + v2.size());
    vector<int>::iterator it, st;

    it = set_intersection(v1.begin(), v1.end(),
                   v2.begin(),
                   v2.end(), v.begin());

    vector<int> res = {};
    for(st = v.begin(); st != it; ++st){
        res.push_back(*st);
    }
    return res;
}

/**
 * Difference algorithm found and adapted from GeeksForGeeks.com
 */
vector<int> vector_difference(vector<int> v1, int lb, int ub){
    vector<int> v = {};
    for(int i = lb; i <= ub; i++){
        int t = 0;
        for(int j = 0; j < v1.size(); j++){
            if(i==v1[j]){
                t=1;
                break;
            }
        }
        if(t==0){
            v.push_back(i);
        }
    }
    return v;
}

/**
 * Prints the Search::Statistics object into a readable format
 * @param stats a Search::Statistics object representing the statistics of a search
 * @return The string representation of the statistics object
 */
string statistics_to_string(Search::Statistics stats){
    string s = "Nodes traversed: " + to_string(stats.node) + "\n";
    s += "Failed nodes explored: " + to_string(stats.fail) + "\n";
    s += "Restarts performed: " + to_string(stats.restart) + "\n";
    s += "Propagators executed: " + to_string(stats.propagate) + "\n";
    s += "No goods generated: " + to_string(stats.nogood) + "\n";
    s += "Maximal depth of explored tree: " + to_string(stats.depth) + "\n";
    return s;
}

/**
 * Prints the Search::Statistics object into a csv format (coma separated)
 * @param stats a Search::Statistics object representing the statistics of a search
 * @return The string representation of the statistics object
 */
string statistics_to_csv_string(Search::Statistics stats){
    string s = to_string(stats.node) + ",";
    s += to_string(stats.fail) + ",";
    s += to_string(stats.restart) + ",";
    s += to_string(stats.propagate) + ",";
    s += to_string(stats.nogood) + ",";
    s += to_string(stats.depth) + ",";
    return s;
}

/**
 * Returns the value of a variable as a string
 * @param var an integer variable
 * @param absolute a boolean indicating if the value should be returned as an absolute value (
    default is false)
 * @return a string representing the value of the variable
 */
```

```
288 string intVar_to_string(const IntVar &var, bool absolute) {
289     if (var.assigned()){
290         if(absolute)
291             return to_string(abs(var.val()));
292         return to_string(var.val());
293     }
294     return "<not assigned>";
295 }
296
297 /**
298  * Returns the values of an array of variables as a string
299  * @param vars an array of integer variables
300  * @return a string representing the values of the variables
301  */
302 string intVarArray_to_string(IntVarArray vars){
303     int s = vars.size();
304     string res = "{";
305     for(int i = 0; i < s; i++){
306         res += intVar_to_string(vars[i]);
307         if(i != s - 1)
308             res += ", ";
309     }
310     res += "}";
311     return res;
312 }
313
314 /**
315  * Returns the value of a variable as a string
316  * @param var an integer variable
317  * @param absolute a boolean indicating if the value should be returned as an absolute value (
        default is false)
318  * @return a string representing the value of the variable
319  */
320 string boolVar_to_string(const BoolVar &var, bool absolute) {
321     if (var.assigned()){
322         if(absolute)
323             return to_string(abs(var.val()));
324         return to_string(var.val());
325     }
326     return "<not assigned>";
327 }
328
329 /**
330  * Returns the values of an array of variables as a string
331  * @param vars an array of integer variables
332  * @return a string representing the values of the variables
333  */
334 string boolVarArray_to_string(BoolVarArray vars){
335     int s = vars.size();
336     string res = "{";
337     for(int i = 0; i < s; i++){
338         res += boolVar_to_string(vars[i]);
339         if(i != s - 1)
340             res += ", ";
341     }
342     res += "}";
343     return res;
344 }
345
346 /**
347  * Returns the values of an array of variables as a string
348  * @param vars an array of integer variables
349  * @return a string representing the values of the variables, ignoring unassigned variables
350  */
351 string cleanIntVarArray_to_string(IntVarArray vars){
352     int s = vars.size();
353     string res = "{";
354     for(int i = 0; i < s; i++){
355         if (vars[i].assigned()){
356             res += intVar_to_string(vars[i]);
357             if(i != s - 1)
358                 res += ", ";
359         }
360     }
361     res += "}";
362     return res;
363 }
364
365 /**
366  * Returns the values of an IntVarArgs as a string
367  * @param args an IntVarArgs
368  * @return a string representing the values
369  */
370 string intVarArgs_to_string(IntVarArgs args){
371     int s = args.size();
372     string res;
373     for(int i = 0; i < s; i++){
374         res += intVar_to_string(args[i], 0);
375         if(i != s - 1)
376             res += ", ";
```

```
377     }
378     return res;
379 }
380 
381 Species int_to_species(int sp) {
382     switch (sp) {
383         case 1: return FIRST_SPECIES;
384         case 2: return SECOND_SPECIES;
385         case 3: return THIRD_SPECIES;
386         case 4: return FOURTH_SPECIES;
387         case 5: return FIFTH_SPECIES;
388         default: return THIRD_SPECIES;
389     }
390 }
391 
392 string midi_to_french(int note) {
393     static const string names[] = {"Do","Do#","R  ","Mib","Mi","Fa","Fa#","Sol","Lab","La","Sib"
    ,"Si"};
394     int oct = note / 12 - 1;
395     return names[note % 12] + to_string(oct);
396 }
397 
398 
399 /**
400  * Returns the name of a note based on its MIDI value
401  * @param note an integer
402  */
403 string midi_to_letter(int note){
404     return noteNames[note % PERFECT_OCTAVE];
405 }
406 
407 /**
408  * Returns the name of a mode based on its integer value
409  * @param mode an integer
410  * @return a string representing the name of the mode
411  */
412 string mode_int_to_name(int mode){
413     return modeNames[mode];
414 }
415 
416 /**
417  * returns a string with the time
418  * @return a string with the time
419  */
420 string time(){
421     /// date and time for logs
422     std::time_t currentTime = std::time(nullptr); // Get the current time
423     std::string timeString = std::asctime(std::localtime(&currentTime)); // Convert to string
424     return timeString;
425 }
426 
427 /**
428  * Write a text into a log file
429  * Useful for debugging in the OM environment
430  * @param message the text to write
431  */
432 void write_to_log_file(const char *message, const string& filename) { // TODO : TO ACTIVATE
    LOGGING, UNCOOMMENT THE CONTENTS OF THIS FUNCTION AND THE NEXT FUNCTION.
433     // const char* homeDir = std::getenv("HOME"); // Get the user's home directory
434     // if (homeDir) {
435     //     std::string filePath(homeDir);
436     //     filePath += "/Documents/Libraries/MusicConstraints/out/" + filename; // Specify the
    desired file path, such as $HOME/log.txt
437 
438     //     std::ofstream myfile(filePath, std::ios::app); // append mode
439     //     if (myfile.is_open()) {
440     //         myfile << message << endl;
441     //         myfile.close();
442     //     }
443     // }
444 }
445 
446 /**
447  * Write a text into a log file
448  * @param message the text to write
449  */
450 void writeToLogFile(const char* message){ // TODO : TO ACTIVATE LOGGING, UNCOOMMENT THE CONTENTS
     OF THIS FUNCTION AND THE PREVIOUS FUNCTION.
451     // std::time_t currentTime = std::time(nullptr); // Get the current time
452     // std::string timeString = std::asctime(std::localtime(&currentTime)); // Convert to string
453 
454     // const char* homeDir = std::getenv("HOME"); // Get the user's home directory
455     // if (homeDir) {
456     //     std::string filePath(homeDir);
457     //     filePath += "/log.txt"; // Specify the desired file path, such as $HOME/log.txt
458 
459     //     std::ofstream myfile(filePath, std::ios::app); // append mode
460     //     if (myfile.is_open()) {
461     //         myfile <<timeString<< endl << message << endl;
462     //         myfile.close();
```

```
463     //     }
464     // }
465 }
466 
467 
468 /* ================================================
469  *          CONSTRAINT HELPERS
470  * ================================================
471  */
472 
473 IntVarArray expandCantusNotes(Home home, IntVarArray cantus){
474     IntVarArray expandedCantus = IntVarArray(home, 4*cantus.size()-3, 0, 127);
475     for (int i = 0; i < expandedCantus.size(); i++){
476         expandedCantus[i] = cantus[i/4];
477     }
478     return expandedCantus;
479 }
480 
481 vector<int> createRangeVector(int from, int to, int multiplier) {
482     vector<int> indices;
483     for (int i = from; i < to; i++) {
484         indices.push_back(i*multiplier);
485     }
486     return indices;
487 }
488 
489 BranchVarSel g_solution_var_sel = BR_VAR_AFC_MAX;
490 BranchValSel g_solution_val_sel = BR_VAL_RND;
491 unsigned int g_solution_val_rnd_seed = 1U;
492 
493 const char* branch_var_sel_name(BranchVarSel s) {
494     switch (s) {
495         case BR_VAR_SIZE_MIN:   return "size-min";
496         case BR_VAR_SIZE_MAX:   return "size-max";
497         case BR_VAR_DEGREE_MAX: return "degree-max";
498         case BR_VAR_AFC_MAX:    return "afc-max";
499         case BR_VAR_ACTION_MAX: return "action-max";
500         case BR_VAR_NONE:       return "none";
501     }
502     return "unknown";
503 }
504 
505 const char* branch_val_sel_name(BranchValSel s) {
506     switch (s) {
507         case BR_VAL_MIN:       return "min";
508         case BR_VAL_MAX:       return "max";
509         case BR_VAL_MED:       return "med";
510         case BR_VAL_RND:       return "rnd";
511         case BR_VAL_SPLIT_MIN: return "split-min";
512         case BR_VAL_SPLIT_MAX: return "split-max";
513     }
514     return "unknown";
515 }
516 
517 BranchVarSel parse_branch_var_sel(const string& s) {
518     if (s == "size-min")   return BR_VAR_SIZE_MIN;
519     if (s == "size-max")   return BR_VAR_SIZE_MAX;
520     if (s == "degree-max") return BR_VAR_DEGREE_MAX;
521     if (s == "afc-max")    return BR_VAR_AFC_MAX;
522     if (s == "action-max") return BR_VAR_ACTION_MAX;
523     if (s == "none")       return BR_VAR_NONE;
524     return BR_VAR_AFC_MAX;
525 }
526 
527 BranchValSel parse_branch_val_sel(const string& s) {
528     if (s == "min")       return BR_VAL_MIN;
529     if (s == "max")       return BR_VAL_MAX;
530     if (s == "med")       return BR_VAL_MED;
531     if (s == "rnd")       return BR_VAL_RND;
532     if (s == "split-min") return BR_VAL_SPLIT_MIN;
533     if (s == "split-max") return BR_VAL_SPLIT_MAX;
534     return BR_VAL_RND;
535 }
536 
537 static IntVarBranch resolve_var_branch(BranchVarSel s) {
538     switch (s) {
539         case BR_VAR_SIZE_MIN:   return INT_VAR_SIZE_MIN();
540         case BR_VAR_SIZE_MAX:   return INT_VAR_SIZE_MAX();
541         case BR_VAR_DEGREE_MAX: return INT_VAR_DEGREE_MAX();
542         case BR_VAR_AFC_MAX:    return INT_VAR_AFC_MAX();
543         case BR_VAR_ACTION_MAX: return INT_VAR_ACTION_MAX();
544         case BR_VAR_NONE:       return INT_VAR_NONE();
545     }
546     return INT_VAR_AFC_MAX();
547 }
548 
549 static IntValBranch resolve_val_branch(BranchValSel s, unsigned int seed) {
550     switch (s) {
551         case BR_VAL_MIN:       return INT_VAL_MIN();
552         case BR_VAL_MAX:       return INT_VAL_MAX();
```

```
        case BR_VAL_MED:       return INT_VAL_MED();
        case BR_VAL_RND:       return INT_VAL_RND(seed);
        case BR_VAL_SPLIT_MIN: return INT_VAL_SPLIT_MIN();
        case BR_VAL_SPLIT_MAX: return INT_VAL_SPLIT_MAX();
    }
    return INT_VAL_RND(seed);
}

void branch_solution_array_dynamic(Home home, const IntVarArgs& vars) {
    branch(home, vars,
           resolve_var_branch(g_solution_var_sel),
           resolve_val_branch(g_solution_val_sel, g_solution_val_rnd_seed));
}
```

A.9.3 Generations.cpp

```
// [Claude Code]
// ========================================================================
// Generations.cpp     moteur de g n ration de contrepoints.
// Utilise GenerationCase + bench BAB avec StagnationStop. Toutes les
// donn es autrefois hardcod es (registre CF, presets de param tres) sont
// charg es depuis FuxCP/config/*.csv.
// ========================================================================

#include "../headers/Generations.hpp"

using namespace std;
using namespace std::chrono;
using namespace Gecode;

class StagnationStop : public Search::Stop {
    Search::TimeStop maxStop;
    steady_clock::time_point lastImprovement;
    int stagnation_ms;
    bool stagnation_triggered = false;
    bool max_triggered = false;

public:
    StagnationStop(int max_ms, int stag_ms)
        : maxStop(max_ms), stagnation_ms(stag_ms),
          lastImprovement(steady_clock::now()) {}

    void notifyImprovement() { lastImprovement = steady_clock::now(); }

    bool stop(const Search::Statistics& s, const Search::Options& o) {
        if (maxStop.stop(s, o)) { max_triggered = true; return true; }
        if (stagnation_ms > 0) {
            auto el = duration_cast<milliseconds>(steady_clock::now() - lastImprovement).count()
    ;
            if (el > stagnation_ms) { stagnation_triggered = true; return true; }
        }
        return false;
    }
    string stopReason() const {
        if (stagnation_triggered) return "STAGNATION";
        if (max_triggered)        return "TIMEOUT_MAX";
        return "EXHAUSTIVE";
    }
};

// ============================================================
// Helpers
// ============================================================

static const int CHECKPOINT_INTERVAL = 1000;

static string species_tag_for(const GenerationCase& gc) {
    string tag = to_string(gc.n_voices) + "voices_1";  // CF en premi re position
    for (int s : gc.spList) tag += "_" + to_string(s+1);
    tag += "_vtype";
    for (int s : gc.v_type) tag += "_" + to_string(s);
    return tag;
}

static void mkdir_p(const string& path) {
    string cmd = "mkdir -p '" + path + "'";
    int r = system(cmd.c_str());
    (void)r;
}


// ============================================================
// Construction d'un GenerationCase    partir des CSV + arguments
// ============================================================

bool resolve_cf(GenerationCase& gc, bool verbose) {
    string detected_scale_name;
```

```
71      if (gc.use_preset_cf){
72          string config_dir = ConfigLoader::find_config_dir();
73          auto cfs = ConfigLoader::load_cantus_firmus(config_dir);
74
75          auto it = cfs.find(gc.cf_id);
76          if (it == cfs.end()) {
77              cerr << "CF inconnu : " << gc.cf_id << endl;
78              return false;
79          }
80          gc.cf = it->second.notes;
81          if (gc.cf.empty()) {
82              cerr << "CF id=" << gc.cf_id << " : aucune note dans cantus_firmus.csv" << endl;
83              return false;
84          }
85          gc.cf_name  = it->second.name;
86          gc.cf_scale = it->second.scale;
87          detected_scale_name = detect_scale_name_for_cf(gc.cf);
88      }
89      else { // Direct cf
90          if (gc.cf.empty()) {
91              cerr << "Direct CF provided but empty." << endl;
92              return false;
93          }
94
95          detected_scale_name = detect_scale_name_for_cf(gc.cf);
96          if (gc.cf_scale.empty()) { gc.cf_scale = detected_scale_name; }
97          if (gc.cf_name.empty()){ gc.cf_name  = "direct_cf_" + gc.cf_scale; }
98      }
99
100     if (verbose){
101         cout << "Cantus firmus -> gamme d tect e : "
102         << detected_scale_name << " (tonique = "
103         << noteNames[((gc.cf[0] % 12) + 12) % 12] << ")"
104         << " -> gamme sp cifi e : " << gc.cf_scale << endl;
105     }
106
107     return true;
108 }
109
110 bool apply_preset(GenerationCase& gc) {
111     string config_dir = ConfigLoader::find_config_dir();
112     auto presets = ConfigLoader::load_presets(config_dir);
113
114     auto it = presets.find(gc.preset_name);
115     if (it == presets.end()) {
116         cerr << "Preset inconnu : " << gc.preset_name << endl;
117         return false;
118     }
119     const auto& p = it->second;
120     gc.melodic_params  = p.melodic_params;
121     gc.general_params  = p.general_params;
122     gc.specific_params = p.specific_params;
123     gc.importance      = p.importance;
124     gc.borrow_mode     = p.borrow_mode;
125     return true;
126 }
127
128 // ============================================================
129 // Logging (txt + csv)     une seule g n ration
130 // ============================================================
131
132 static void write_txt_report(const string& path, const GenerationCase& gc,
133                              const GenerationResult& gr) {
134     ofstream r(path);
135     r << "========================================\n"
136       << "  R SULTATS : " << species_tag_for(gc) << "\n"
137       << "========================================\n\n";
138     r << "--- Configuration ---\n";
139     if (gc.use_preset_cf) r << "CF                : " << gc.cf_id << " (" << gc.cf_name << ")\n"
    ;
140     else                  r << "CF                : " << gc.cf_name << "\n";
141     r  << "Gamme indicative  : " << gc.cf_scale << "\n"
142       << "Preset            : " << gc.preset_name << "\n"
143       << "Borrow mode       : " << gc.borrow_mode << "\n"
144       << "Mode d'objectif   : " << obj_mode_label(gc.obj_mode) << "\n"
145       << "Nombre de voix    : " << gc.n_voices << "\n"
146       << "Esp ces (incl CF) : 1";
147     for (int s : gc.spList) r << " " << s;
148     r << "\nv_type            :";
149     for (int v : gc.v_type) r << " " << v;
150     r << "\nTimeout max (ms)  : " << gc.timeout_ms << "\n"
151       << "Stagnation (ms)   : " << gc.stagnation_ms << "\n"
152       << "Cantus firmus     :";
153     for (int n : gc.cf) r << " " << n;
154     r << "\n                  ";
155     for (int n : gc.cf) r << " " << midi_to_french(n);
156     r << "\n\n";
157
158     r << "--- M triques de recherche ---\n"
159       << "Temps total (ms)           : " << fixed << setprecision(1) << gr.ms_total << "\n"
```

```
      << "Temps 1 re  solution (ms)    : " << fixed << setprecision(1) << gr.ms_first_solution << "\n"
      << "Temps derni re am lio. (ms) : " << fixed << setprecision(1) << gr.ms_last_improve << "\n"
      << "Solutions trouv es          : " << gr.nb_solutions << "\n"
      << "Am liorations de co t       : " << gr.nb_improvements << "\n"
      << cost_label(gc.obj_mode) << "           : " << gr.best_cost << "\n"
      << "Terminaison                 : " << gr.termination;
    if (gr.termination == "EXHAUSTIVE") r << " (optimal)";
    r << "\n\n";

    r << "--- Statistiques Gecode ---\n"
      << "N uds      : " << gr.stats.node      << "\n"
      << " checs     : " << gr.stats.fail      << "\n"
      << "Restarts   : " << gr.stats.restart   << "\n"
      << "Propagat.  : " << gr.stats.propagate << "\n"
      << "Profondeur : " << gr.stats.depth     << "\n\n";

    if (!gr.improvements.empty()) {
        r << "--- Am liorations ---\n"
          << setw(8) << "Sol#" << setw(14) << "Temps (ms)" << setw(12) << "Co t" << "\n";
        for (auto& t : gr.improvements) {
            r << setw(8) << get<0>(t)
              << setw(14) << fixed << setprecision(1) << get<1>(t)
              << setw(12) << fixed << setprecision(1) << get<2>(t) << "\n";
        }
        r << "\n";
    }
    if (!gr.checkpoints.empty()) {
        r << "--- Checkpoints (" << CHECKPOINT_INTERVAL << " sol) ---\n"
          << setw(8) << "Sol#" << setw(14) << "Temps (ms)" << setw(14) << "Best cost" << "\n";
        for (auto& t : gr.checkpoints) {
            r << setw(8) << get<0>(t)
              << setw(14) << fixed << setprecision(1) << get<1>(t)
              << setw(14) << fixed << setprecision(1) << get<2>(t) << "\n";
        }
        r << "\n";
    }
    if (!gr.solutions_log.empty()) {
        r << "--- \u00c9volution du vecteur de co\u00fbts (toutes les solutions BAB) ---\n"
          << setw(8) << "Sol#" << setw(14) << "Temps (ms)"
          << setw(12) << "Cost" << "  Vecteur lex\n";
        for (auto& t : gr.solutions_log) {
            r << setw(8) << get<0>(t)
              << setw(14) << fixed << setprecision(1) << get<1>(t)
              << setw(12) << fixed << setprecision(1) << get<2>(t)
              << "  " << get<3>(t) << "\n";
        }
        r << "\n";
    }
    if (!gr.full_solution.empty()) {
        r << "--- Solution (notes MIDI) ---\n";
        for (int n : gr.full_solution) r << n << " ";
        r << "\n";
    }
    // 5th-species rhythm: per cell, 1 = genuine 3rd-species attack (consonance governed by
    // SP5_H6), 0 = phantom continuation/syncope cell (rendered as a quarter but not checked).
    for (size_t i = 0; i < gr.rhythm_dump.size(); ++i) {
        if (!gr.rhythm_dump[i].empty())
            r << "--- isThirdSpeciesArray (CP" << (i + 1) << ") ---\n" << gr.rhythm_dump[i] << "\n";
    }
    r.close();
}

static void write_csv_report(const string& path, const GenerationCase& gc,
                             const GenerationResult& gr) {
    ofstream c(path);

    c << "cf_id,cf_name,preset,borrow_mode,nb_voix,especes,v_type,"
         "timeout_ms,stagnation_ms,temps_total_ms,temps_premiere_solution_ms,"
         "temps_derniere_amelioration_ms,nb_solutions,nb_ameliorations,cout_final,"
         "terminaison,noeuds,echecs,redemarrages,propagations,profondeur_max\n";
    if (gc.use_preset_cf) c << gc.cf_id ;
    else                  c << "/" << "\n";
    c << "," << gc.cf_name << "," << gc.preset_name << ","
      << gc.borrow_mode << "," << gc.n_voices << ",";
    c << "1";
    for (int s : gc.spList) c << "-" << s;
    c << ",";
    for (size_t i = 0; i < gc.v_type.size(); ++i) {
        if (i > 0) c << "-";
        c << gc.v_type[i];
    }
    c << "," << gc.timeout_ms << "," << gc.stagnation_ms
      << "," << fixed << setprecision(1) << gr.ms_total
      << "," << fixed << setprecision(1) << gr.ms_first_solution
      << "," << fixed << setprecision(1) << gr.ms_last_improve
      << "," << gr.nb_solutions
      << "," << gr.nb_improvements
```

```
      << "," << fixed << setprecision(1) << gr.best_cost
      << "," << gr.termination
      << "," << gr.stats.node
      << "," << gr.stats.fail
      << "," << gr.stats.restart
      << "," << gr.stats.propagate
      << "," << gr.stats.depth << "\n";

    if (!gr.improvements.empty()) {
        c << "\n# Am liorations\nsolution_num,temps_ms,cout\n";
        for (auto& t : gr.improvements)
            c << get<0>(t) << "," << fixed << setprecision(1) << get<1>(t)
              << "," << fixed << setprecision(1) << get<2>(t) << "\n";
    }
    if (!gr.checkpoints.empty()) {
        c << "\n# Checkpoints\nsolution_num,temps_ms,meilleur_cout\n";
        for (auto& t : gr.checkpoints)
            c << get<0>(t) << "," << fixed << setprecision(1) << get<1>(t)
              << "," << fixed << setprecision(1) << get<2>(t) << "\n";
    }
    if (!gr.solutions_log.empty()) {
        c << "\n# Evolution couts\nsolution_num,temps_ms,cout_scalaire,vecteur_lex\n";
        for (auto& t : gr.solutions_log) {
            string lex = get<3>(t);
            // CSV-safe : remplacer virgules par points-virgules dans le vecteur lex
            for (char& ch : lex) if (ch == ',') ch = ';';
            c << get<0>(t) << "," << fixed << setprecision(1) << get<1>(t)
              << "," << fixed << setprecision(1) << get<2>(t)
              << ",\"" << lex << "\"\n";
        }
    }
    c.close();
}

static void write_error_txt(const string& path, const GenerationCase& gc,
                            const GenerationResult& gr) {
    ofstream e(path);
    e << "========================================\n"
      << "  AUCUNE SOLUTION : " << species_tag_for(gc) << "\n"
      << "========================================\n\n";
    if (gc.use_preset_cf) e << "CF                : " << gc.cf_id << " (" << gc.cf_name << ")\n"
    ;
    else                  e << "CF                : " << gc.cf_name << "\n";
    e  << "Preset            : " << gc.preset_name << "\n"
      << "Nb voix         : " << gc.n_voices << "\n"
      << "Esp ces           : 1";
    for (int s : gc.spList) e << " " << s;
    e << "\nTimeout (ms)    : " << gc.timeout_ms << "\n"
      << "Stagnation (ms): " << gc.stagnation_ms << "\n"
      << "Terminaison     : " << gr.termination << "\n"
      << "Temps    coul      : " << fixed << setprecision(1) << gr.ms_total << " ms\n"
      << "N uds             : " << gr.stats.node << "\n"
      << " checs            : " << gr.stats.fail << "\n";
    e.close();
}

// ============================================================
// Helper : retourne true si l'it ration n doit afficher du logging progressif
// Affiche    : 1, 5, 10, 20, 30, 40, 50, 100, 200, 500, 1000, 1500, ...
// ============================================================
static bool should_log_iteration(int n) {
    if (n <= 50) {
        return n == 1 || n == 5 || n == 10 || n == 20 || n == 30 || n == 40 || n == 50;
    }
    if (n <= 500) return n % 100 == 0;
    if (n <= 2000) return n % 500 == 0;
    return n % 1000 == 0;
}

// ============================================================
// Bench BAB ( quivalent  generation_BAB_bench de Generations.cpp)
// ============================================================

static GenerationResult run_bench(CounterpointProblem* problem, GenerationCase& gc, bool verbose
    =true) {
    GenerationResult result;
    result.resolved_cf = gc.cf;
    result.resolved_cf_name = gc.cf_name;
    result.resolved_cf_scale = gc.cf_scale;

    StagnationStop stopObj(gc.timeout_ms, gc.stagnation_ms);
    Search::Options opt;
    opt.stop = &stopObj;

    auto t0 = steady_clock::now();
    auto ms_since = [&]() {
        return duration<double, milli>(steady_clock::now() - t0).count();
    };

    BAB<CounterpointProblem> e(problem, opt);
```

```
335     CounterpointProblem* best = nullptr;
336     int iteration = 0;
337 
338     while (CounterpointProblem* s = e.next()) {
339         stopObj.notifyImprovement();
340         iteration++;
341         result.nb_solutions++;
342         double now = ms_since();
343         double cost = s->getCost();
344         string lex = intVarArgs_to_string(s->cost());
345 
346         // Trace live : on voit l'  volution  des co ts en suivant le log.
347         // En mode lex on affiche le vecteur complet (le plus parlant) ;
348         // en mode total/mixed/pond on affiche le scalaire optimis  directement.
349         if (verbose) {
350             if (gc.obj_mode == OBJECTIVE_LEX) {
351                 cout << "  [Sol " << result.nb_solutions
352                     << " | t=" << fixed << setprecision(0) << now << "ms"
353                     << " | somme=" << fixed << setprecision(1) << cost
354                     << " | lex=" << lex << "]" << endl;
355             } else {
356                 cout << "  [Sol " << result.nb_solutions
357                     << " | t=" << fixed << setprecision(0) << now << "ms"
358                     << " | " << cost_label(gc.obj_mode) << "="
359                     << fixed << setprecision(1) << cost << "]" << endl;
360             }
361         }
362 
363         // Logging progressif des it rations (affiche tous les 1, 5, 10, 20, 50, 100, 500,
    1000...)
364         if (should_log_iteration(iteration)) {
365             // Log even without verbose to check the algorithm is progressing
366             cout << "    -> It ration " << iteration << " | t="
367             << fixed << setprecision(0) << now << "ms | meilleur="
368             << fixed << setprecision(1) << result.best_cost << endl;
369         }
370 
371 
372         result.solutions_log.emplace_back(result.nb_solutions, now, cost, lex);
373 
374         if (result.nb_solutions == 1) result.ms_first_solution = now;
375 
376         if (cost < result.best_cost) {
377             result.nb_improvements++;
378             delete best;
379             best = s;
380             result.best_cost = cost;
381             result.ms_last_improve = now;
382             result.improvements.emplace_back(result.nb_solutions, now, cost);
383             stopObj.notifyImprovement();
384         } else {
385             delete s;
386         }
387         if (result.nb_solutions % CHECKPOINT_INTERVAL == 0) {
388             result.checkpoints.emplace_back(result.nb_solutions, now, result.best_cost);
389         }
390     }
391 
392     bool stopped = e.stopped();
393     result.termination = stopped ? stopObj.stopReason() : "EXHAUSTIVE";
394     result.stats = e.statistics();
395     result.ms_total = ms_since();
396 
397     if (best) {
398         IntVarArray sa = best->getSolutionArray();
399         result.full_solution.reserve(sa.size());
400         result.success = true;
401         for (int i = 0; i < sa.size(); ++i) result.full_solution.push_back(sa[i].val());
402         // Capture each counterpoint's 3rd-species classification before freeing the space
403         // (empty for non-5th species). Order matches the voice order in the solution.
404         for (Part* cp : {best->getCounterpoint_1(), best->getCounterpoint_2(), best->
    getCounterpoint_3()})
405             if (cp) result.rhythm_dump.push_back(cp->getSolutionRhythm());
406         delete best;
407     } else {
408         result.success = false;
409     }
410     return result;
411 }
412 
413 // ============================================================
414 // Ex cution d'un GenerationCase complet (probl me + bench + sauvegarde)
415 // ============================================================
416 
417 GenerationResult Generations::run_generation_case(GenerationCase& gc, bool save_outputs, bool
    verbose) {
418     string species_tag = species_tag_for(gc);
419     string cf_prefix = gc.use_preset_cf ? ("cf" + to_string(gc.cf_id)) : gc.cf_name;
420     string default_filenames = cf_prefix + "_" + species_tag;
421 
```

```
422     string base_dir = gc.output_root;
423     if (!gc.output_subdir.empty()) base_dir += "/" + gc.output_subdir;
424     base_dir += "/" + to_string(gc.n_voices) + "voices";
425     string txt_dir  = base_dir + "/txt";
426     string csv_dir  = base_dir + "/csv";
427     string midi_dir = base_dir + "/midi";
428     mkdir_p(txt_dir); mkdir_p(csv_dir); mkdir_p(midi_dir);
429
430     // Description du CF ( c r   e  1 seule fois)
431     if (save_outputs){
432         string cf_desc = base_dir + "/cantus_firmus_" + cf_prefix + ".txt";
433         if (!ifstream(cf_desc).good()) {
434             ofstream f(cf_desc);
435             for (size_t i = 0; i < gc.cf.size(); ++i) { if (i) f << " "; f << gc.cf[i]; }
436             f << "\n";
437             for (size_t i = 0; i < gc.cf.size(); ++i) { if (i) f << " "; f << midi_to_french(gc.
    cf[i]); }
438             f << "\n" << gc.cf_name << " (gamme indicative : " << gc.cf_scale << ")\n";
439         }
440     }
441
442     if (gc.use_preset_cf) cout << "\n=== G n ration : cf" << gc.cf_id << " (" << gc.cf_name <<
     ") | ";
443     else                  cout << "\n=== G n ration : cf" << gc.cf_name << " | ";
444     cout << species_tag << " | preset=" << gc.preset_name
445         << " | obj=" << obj_mode_short(gc.obj_mode)
446         << " | timeout=" << gc.timeout_ms << "ms | stagn=" << gc.stagnation_ms << "ms ===" <<
    endl;
447
448     // Activation des contraintes musicales (par d faut toutes actives)
449     fill(activeConstraints.begin(), activeConstraints.end(), true);
450
451     // Cr ation du probl me via create_problem (m thode  unifi e , cf Generations.cpp)
452     CounterpointProblem* problem = create_problem(
453         gc.cf, gc.spList, gc.v_type,
454         gc.melodic_params, gc.general_params, gc.specific_params,
455         gc.importance, gc.borrow_mode, gc.obj_mode);
456
457     GenerationResult result = run_bench(problem, gc, verbose);
458     delete problem;
459
460     if (!result.full_solution.empty()) {
461         // D coupage des voix pour MIDI
462         vector<pair<vector<int>, Species>> voices;
463         int offset = 0;
464         for (int v = 0; v < (int)gc.spList.size(); ++v) {
465             int sz = branchingNotesSize(gc.spList[v], gc.cf.size());
466             vector<int> notes(result.full_solution.begin() + offset,
467                               result.full_solution.begin() + offset + sz);
468             voices.emplace_back(notes, gc.spList[v]);
469             offset += sz;
470         }
471         result.generated_voices = voices;
472
473         if (save_outputs){
474             string midi_path = midi_dir + "/" + default_filenames + ".mid";
475             string txt_path = txt_dir + "/" + default_filenames + ".txt";
476             string csv_path = csv_dir + "/" + default_filenames + ".csv";
477
478             saveMidiMultiVoice(midi_path, gc.cf, voices);
479             write_txt_report(txt_dir + "/" + default_filenames + ".txt", gc, result);
480             write_csv_report(csv_dir + "/" + default_filenames + ".csv", gc, result);
481
482             result.midi_path = midi_path;
483             result.txt_path = txt_path;
484             result.csv_path = csv_path;
485         }
486
487         if (verbose){
488             cout << "  -> Solutions=" << result.nb_solutions
489              << " Am liorations=" << result.nb_improvements
490              << " " << cost_label(gc.obj_mode) << "=" << result.best_cost
491              << " Temps=" << fixed << setprecision(0) << result.ms_total << "ms"
492              << " (" << result.termination << ")" << "\n"
493              << "  -> Best = " << int_vector_to_string(result.full_solution) << endl;
494         }
495     } else {
496
497         if (save_outputs){
498             string error_path = txt_dir + "/error_" + default_filenames + ".txt";
499             write_error_txt(error_path, gc, result);
500             result.error_txt_path = error_path;
501         }
502
503         if (verbose){
504             cout << "  -> AUCUNE SOLUTION (" << result.termination << ", "
505              << fixed << setprecision(0) << result.ms_total << " ms)" << endl;
506         }
507     }
508
```

```
    // Fill GenerationResult
    return result;
}
```

A.9.4 GenerationsAPI.cpp

```
// ====================================================================
// GenerationAPI.cpp     public API wrapper around the generation engine.
//
// Role:
//   This file exposes a simple programmatic entrypoint for other parts of
//   the project (or external integrations) that need to generate a
//   counterpoint without knowing the internal generation pipeline.
//
// Design:
//   - The actual generation logic stays in Generations.cpp.
//   - This API only converts a clean external input (GenerationInput)
//     into the internal generation case format (GenCase), then delegates
//     execution to the reusable engine function.
//
// Main entrypoint:
//   GenerationResult generate_counterpoint(const GenerationInput& input)
//
// Expected use:
//   - plugin / UI / other project modules
//   - direct generation from a user-provided cantus firmus
//   - integration without exposing CLI-specific options or internals
//
// Not handled here:
//   - command-line parsing
//   - listing presets / cantus firmi
//   - advanced testing workflows
//
// Notes:
//   - The cantus firmus can be provided only as MIDI notes.
//   - Presets and configuration data are still resolved through the
//     shared generation engine.
// ====================================================================

#include "../headers/GenerationsAPI.hpp"

static GenerationCase build_gc_from_api_input(const GenerationInput& input) {
    GenerationCase gc;
    gc.cf = input.cf_notes;
    gc.use_preset_cf = false;
    gc.n_voices = input.species.size()+1;
    gc.spList = input.species;
    gc.v_type = input.v_type;

    if (input.preset_name != "") gc.preset_name = input.preset_name;
    if (input.timeout_ms > -1) gc.timeout_ms = input.timeout_ms;
    if (input.stagnation_ms > -1) gc.stagnation_ms = input.stagnation_ms;
    gc.obj_mode = input.obj_mode;
    if (input.output_root != "") gc.output_root = input.output_root;
    if (input.output_subdir != "") gc.output_subdir = input.output_subdir;

    // cf resolution & presets
    if (!resolve_cf(gc, input.verbose)) { cerr << "  Error in cf resolution !" << endl; }
    if (!apply_preset(gc)){ cerr << "  Error in presets resolution !" << endl; }

    return gc;
}

GenerationResult generate_counterpoint(const GenerationInput& input) {
    GenerationCase gc = build_gc_from_api_input(input);
    return Generations::run_generation_case(gc, input.save_outputs, input.verbose);
}

GenerationResult generate_counterpoint(vector<int> cf_notes, vector<Species> species, vector<int
    > v_type) {
    GenerationInput input;
    input.cf_notes = cf_notes;
    input.species = species;
    input.v_type = v_type;
    return generate_counterpoint(input);
}
```

A.9.5 ConfigLoader.cpp

```
// [Claude Code]
// ConfigLoader.cpp
// Impl mentation du chargement des CSV de configuration.

#include "../headers/ConfigLoader.hpp"
```

```
6
7  #include <fstream>
8  #include <sstream>
9  #include <iostream>
10 #include <sys/stat.h>
11 #include <cstdlib>
12
13 using namespace std;
14
15 // ---------------- Helpers ----------------
16
17 string ConfigLoader::trim(const string& s) {
18     size_t a = s.find_first_not_of(" \t\r\n");
19     if (a == string::npos) return "";
20     size_t b = s.find_last_not_of(" \t\r\n");
21     return s.substr(a, b - a + 1);
22 }
23
24 vector<string> ConfigLoader::split(const string& s, char sep) {
25     vector<string> out;
26     string cur;
27     for (char c : s) {
28         if (c == sep) { out.push_back(cur); cur.clear(); }
29         else cur.push_back(c);
30     }
31     out.push_back(cur);
32     return out;
33 }
34
35 vector<int> ConfigLoader::parse_int_list(const string& s, char sep) {
36     vector<int> out;
37     string t = trim(s);
38     if (t.empty()) return out;
39     for (auto& part : split(t, sep)) {
40         string p = trim(part);
41         if (!p.empty()) out.push_back(atoi(p.c_str()));
42     }
43     return out;
44 }
45
46 static bool file_exists(const string& p) {
47     struct stat st;
48     return stat(p.c_str(), &st) == 0;
49 }
50
51 string ConfigLoader::find_config_dir() {
52     // Cherche le dossier config/ dans plusieurs emplacements probables
53     // (selon le CWD : FuxCP/c++/, FuxCP/, ou ailleurs).
54     const vector<string> candidates = {
55         "../config", "config", "../../config",
56         "FuxCP/config", "../FuxCP/config"
57     };
58     for (auto& c : candidates) {
59         if (file_exists(c + "/cantus_firmus.csv")) return c;
60     }
61     cerr << "[ConfigLoader] ATTENTION : dossier config/ introuvable, fallback ../config" << endl
        ;
62     return "../config";
63 }
64
65 // Renvoie true si la ligne doit  tre  ignor e (commentaire ou vide)
66 static bool is_skippable(const string& line) {
67     string t = ConfigLoader::trim(line);
68     return t.empty() || t[0] == '#';
69 }
70
71 // Lit la 1re ligne non-skip (= header) puis renvoie chaque ligne CSV split par ','
72 // Note : pas de support des virgules intra-champ (les listes utilisent ' ' ou '+').
73 static vector<vector<string>> read_csv_rows(const string& path) {
74     vector<vector<string>> rows;
75     ifstream f(path);
76     if (!f.is_open()) {
77         cerr << "[ConfigLoader] Impossible d'ouvrir " << path << endl;
78         return rows;
79     }
80     string line;
81     bool header_seen = false;
82     while (getline(f, line)) {
83         if (is_skippable(line)) continue;
84         if (!header_seen) { header_seen = true; continue; } // skip header
85         rows.push_back(ConfigLoader::split(line, ','));
86     }
87     return rows;
88 }
89
90 // ---------------- Cantus firmus ----------------
91
92 map<int, CFEntry> ConfigLoader::load_cantus_firmus(const string& dir) {
93     map<int, CFEntry> out;
94     auto rows = read_csv_rows(dir + "/cantus_firmus.csv");
```

```
    for (auto& r : rows) {
        if (r.size() < 5) continue;
        CFEntry e;
        e.id          = atoi(trim(r[0]).c_str());
        e.notes       = parse_int_list(r[1], ' ');
        e.name        = trim(r[2]);
        e.scale       = trim(r[3]);
        e.description = trim(r[4]);
        out[e.id] = e;
    }
    return out;
}

// ---------------- Presets ----------------

map<string, PresetEntry> ConfigLoader::load_presets(const string& dir) {
    map<string, PresetEntry> out;
    auto rows = read_csv_rows(dir + "/presets.csv");
    for (auto& r : rows) {
        if (r.size() < 7) continue;
        PresetEntry p;
        p.name              = trim(r[0]);
        p.melodic_params    = parse_int_list(r[1], ' ');
        p.general_params    = parse_int_list(r[2], ' ');
        p.specific_params   = parse_int_list(r[3], ' ');
        p.importance        = parse_int_list(r[4], ' ');
        p.borrow_mode       = atoi(trim(r[5]).c_str());
        p.description       = trim(r[6]);
        out[p.name] = p;
    }
    return out;
}
```

A.9.6 main.cpp

```
// [Claude Code]
#include "../headers/Generations.hpp"

// ============================================================
// Listing helpers
// ============================================================

static void cmd_list_cf(const map<int, CFEntry>& cfs) {
    cout << "Cantus firmus disponibles :" << endl;
    cout << "  ID  | Nom                  | Gamme indicative   | Description" << endl;
    cout << "  ----+----------------------+--------------------+----------------------------"
    << endl;
    for (auto& kv : cfs) {
        cout << "  " << setw(3) << kv.first << " | "
             << setw(20) << left << kv.second.name << " | "
             << setw(18) << left << kv.second.scale << " | "
             << kv.second.description << endl;
    }
}

static void cmd_list_presets(const map<string, PresetEntry>& presets) {
    cout << "Presets disponibles :" << endl;
    for (auto& kv : presets) {
        cout << "  " << setw(15) << left << kv.first
             << " borrow=" << kv.second.borrow_mode
             << "  - " << kv.second.description << endl;
    }
}

// ============================================================
// CLI
// ============================================================

struct CliArgs {
    int    nb_voix       = 0;
    vector<int> sp_input;          // esp ces des voix de contrepoint
    vector<int>    cf;
    int    timeout_ms    = 300000; // 5min
    int    stagnation_ms = 120000; // 2min
    string output_root   = "../../results";
    string output_subdir = "";
    ObjectiveMode obj_mode = OBJECTIVE_PONDERED;
    vector<int> v_types_override;

    int cf_id            = 1;
    bool use_preset_cf   = false;
    string preset_name   = "default";

    bool list_cf = false, list_presets = false;
};

static void print_usage() {
```

```
52      cout << "Usage : ./GenerateCounterpoint <nb_voix> <especes...> [options]\n"
53           << "        ./GenerateCounterpoint --list-cf | --list-presets\n\n"
54           << "Options :\n"
55           << "  --cf-notes n1,n2,... raw cantus firmus MIDI notes\n"
56           << "  -c CF_ID             id du cantus firmus (cf cantus_firmus.csv)\n"
57           << "  --preset NAME        preset de param tres (cf presets.csv)\n"
58           << "  -t timeout_ms        timeout total maximum\n"
59           << "  -s stagnation_ms     arr t si pas d'am lioration depuis N ms (0 = off)\n"
60           << "  -o output_root       racine des r sultats (d faut ../results)\n"
61           << "  --subdir NAME        sous-dossier sous output_root\n"
62           << "  -m lex|total|mixed|pond  mode d'objectif\n"
63           << "  -v vt1,vt2,...       v_types des voix de contrepoint\n";
64  }
65
66  static bool parse_cli(int argc, char* argv[], CliArgs& a) {
67      int i = 1;
68      if (i < argc) {
69          string s = argv[i];
70          if (s == "--list-cf")        { a.list_cf        = true; return true; }
71          if (s == "--list-presets")   { a.list_presets   = true; return true; }
72      }
73      if (argc < 3) { print_usage(); return false; }
74      a.nb_voix = atoi(argv[1]);
75      if (a.nb_voix < 2 || a.nb_voix > 4) {
76          cerr << "nb_voix doit  tre  2, 3 ou 4." << endl;
77          return false;
78      }
79      int n_sp = a.nb_voix - 1;
80      if (argc < 2 + n_sp) {
81          cerr << "Il faut " << n_sp << " esp ces (hors CF)." << endl;
82          return false;
83      }
84      for (int k = 0; k < n_sp; ++k) {
85          int sp = atoi(argv[2 + k]);
86          if (sp < 1 || sp > 5) { cerr << "Esp ce invalide : " << sp << endl; return false; }
87          a.sp_input.push_back(sp);
88      }
89      int j = 2 + n_sp;
90      while (j < argc) {
91          string s = argv[j];
92          auto need = [&](int extra) { return j + extra < argc; };
93          if (s == "--cf-notes" && need(1)) {
94              string raw = argv[++j];
95              a.cf.clear();
96              for (auto& part : ConfigLoader::split(raw, ',')) {
97                  string p = ConfigLoader::trim(part);
98                  if (!p.empty()) a.cf.push_back(atoi(p.c_str()));
99              }
100         }
101         else if      (s == "-c" && need(1))        {
102             a.cf_id = atoi(argv[++j]);
103             a.use_preset_cf = true;
104         }
105         else if (s == "--preset" && need(1))  { a.preset_name = argv[++j]; }
106         else if (s == "-t" && need(1))        { a.timeout_ms = atoi(argv[++j]); }
107         else if (s == "-s" && need(1))        { a.stagnation_ms = atoi(argv[++j]); }
108         else if (s == "-o" && need(1))        { a.output_root = argv[++j]; }
109         else if (s == "--subdir" && need(1))  { a.output_subdir = argv[++j]; }
110         else if (s == "-m" && need(1))        { a.obj_mode = parse_obj_mode(argv[++j]); }
111         else if (s == "-v" && need(1)) {
112             string v = argv[++j];
113             for (auto& part : ConfigLoader::split(v, ',')) {
114                 string p = ConfigLoader::trim(part);
115                 if (!p.empty()) a.v_types_override.push_back(atoi(p.c_str()));
116             }
117         }
118         else { cerr << "Argument inconnu : " << s << endl; return false; }
119         ++j;
120     }
121     return true;
122 }
123
124 static GenerationCase build_gc_from_cli(const CliArgs& a) {
125     GenerationCase gc;
126     gc.cf = a.cf;
127     if (a.use_preset_cf){
128         gc.cf_id          = a.cf_id;
129         gc.use_preset_cf = true;
130     }
131     gc.n_voices       = a.nb_voix;
132     gc.preset_name    = a.preset_name;
133     gc.timeout_ms     = a.timeout_ms;
134     gc.stagnation_ms  = a.stagnation_ms;
135     gc.output_root    = a.output_root;
136     gc.output_subdir  = a.output_subdir;
137     gc.obj_mode       = a.obj_mode;
138     for (int s : a.sp_input) gc.spList.push_back(int_to_species(s));
139     if (!a.v_types_override.empty()) gc.v_type = a.v_types_override;
140     else gc.v_type = vector<int>(a.nb_voix - 1, 0);
141     return gc;
```

```
}

// ============================================================
// main
// ============================================================

int main(int argc, char* argv[]) {
    string config_dir = ConfigLoader::find_config_dir();
    auto cfs      = ConfigLoader::load_cantus_firmus(config_dir);
    auto presets  = ConfigLoader::load_presets(config_dir);

    CliArgs a;
    if (!parse_cli(argc, argv, a)) return 1;

    if (a.list_cf)        { cmd_list_cf(cfs); return 0; }
    if (a.list_presets)   { cmd_list_presets(presets); return 0; }

    GenerationCase gc = build_gc_from_cli(a);
    cout << gc.spList.size() << endl;
    if (!resolve_cf(gc))       return 1;
    if (!apply_preset(gc)) return 1;
    GenerationResult res = Generations::run_generation_case(gc, true, true);
    return res.success ? 0 : 1;
}
```

A.9.7 Midi.cpp

```
#include <iostream>
#include <fstream>
#include <vector>
#include <cstdint>
#include <algorithm>
#include <utility>
#include "../headers/Midi.hpp"


struct MidiEvent {
    uint32_t tick;
    uint8_t status;
    uint8_t data1;
    uint8_t data2;

    bool operator<(const MidiEvent& other) const {
        if (tick != other.tick) return tick < other.tick;
        // NoteOff (0x8x) avant NoteOn (0x9x) au m me tick
        if ((status & 0xF0) != (other.status & 0xF0)) return (status & 0xF0) < (other.status & 0
    xF0);
        return status < other.status;
    }
};

void writeVLQ(std::vector<uint8_t>& buffer, uint32_t value) {
    uint32_t buffer_val = value & 0x7F;
    while ((value >>= 7) > 0) {
        buffer_val <<= 8;
        buffer_val |= 0x80;
        buffer_val += (value & 0x7F);
    }
    while (true) {
        buffer.push_back(buffer_val & 0xFF);
        if (buffer_val & 0x80) buffer_val >>= 8;
        else break;
    }
}

static std::vector<int> smoothen4th(const std::vector<int>& sol) {
    // Deduplicate pairs
    std::vector<int> out;
    out.reserve((sol.size() + 1) / 2);

    for (size_t i = 0; i < sol.size(); ) {
        int v = sol[i];
        out.push_back(v);
        i += 1;
        // skip one if duplicated
        if (i < sol.size() && sol[i] == v) i += 1;
    }
    return out;
}

std::vector<int> extract_notes(CounterpointProblem* best,
                                            vector<Species> spList,
                                            size_t cfSize) {
    auto arr = best->getSolutionArray();
    std::vector<int> out;
    out.reserve(arr.size());
    for (int i = 0; i < arr.size(); ++i) out.push_back(arr[i].val());
```

```
60      return out;
61  }
62
63  int branchingNotesSize(Species sp, int cfSize) {
64      switch(sp) {
65          case FIRST_SPECIES:  return cfSize;
66          case SECOND_SPECIES: return 2 * cfSize - 1;
67          case THIRD_SPECIES:  return 4 * cfSize - 3;
68          case FOURTH_SPECIES: return 2 * cfSize - 2;
69          case FIFTH_SPECIES:  return 4 * cfSize - 3;
70          default:             return cfSize;
71      }
72  }
73
74  static void addVoiceEvents(std::vector<MidiEvent>& events,
75                             const std::vector<int>& raw_notes,
76                             Species species,
77                             uint8_t channel,
78                             uint8_t velocity,
79                             uint32_t rondeDur) {
80      std::vector<int> notes = raw_notes;
81      uint32_t cpDur;
82
83      switch(species) {
84          case SECOND_SPECIES: cpDur = rondeDur / 2; break;
85          case THIRD_SPECIES:  cpDur = rondeDur / 4; break;
86          case FOURTH_SPECIES:
87              cpDur = rondeDur;
88              notes = smoothen4th(raw_notes);
89              break;
90          case FIFTH_SPECIES:  cpDur = rondeDur / 4; break;
91          case CANTUS_FIRMUS:  cpDur = rondeDur;     break;
92          default:             cpDur = rondeDur;     break; // FIRST_SPECIES
93      }
94
95      uint8_t noteOn  = 0x90 | (channel & 0x0F);
96      uint8_t noteOff = 0x80 | (channel & 0x0F);
97
98      if (species == FOURTH_SPECIES && notes.size() > 1) {
99          size_t i;
100         uint32_t start;
101         for (i = 0; i < notes.size() - 2; ++i) {
102             start = i * cpDur + cpDur / 2;
103             events.push_back({start, noteOn, (uint8_t)notes[i], velocity});
104             events.push_back({start + cpDur, noteOff, (uint8_t)notes[i], 0});
105         }
106         start = i * cpDur + cpDur / 2;
107         events.push_back({start, noteOn, (uint8_t)notes[i], velocity});
108         events.push_back({start + cpDur/2, noteOff, (uint8_t)notes[i], 0});
109         i++;
110
111         start = i * cpDur;
112         events.push_back({start, noteOn, (uint8_t)notes[i], velocity});
113         events.push_back({start + rondeDur, noteOff, (uint8_t)notes[i], 0});
114     } else {
115         uint32_t start;
116         for (size_t i = 0; i < notes.size()-1; ++i) {
117             start = i * cpDur;
118             events.push_back({start, noteOn, (uint8_t)notes[i], velocity});
119             events.push_back({start + cpDur, noteOff, (uint8_t)notes[i], 0});
120         }
121         size_t i = notes.size()-1;
122         start = i * cpDur;
123         events.push_back({start, noteOn, (uint8_t)notes[i], velocity});
124         events.push_back({start + rondeDur, noteOff, (uint8_t)notes[i], 0});
125     }
126 }
127
128 static std::vector<uint8_t> buildTrackData(std::vector<MidiEvent>& events) {
129     std::sort(events.begin(), events.end());
130
131     std::vector<uint8_t> trackData;
132     uint32_t lastTick = 0;
133
134     for (const auto& e : events) {
135         writeVLQ(trackData, e.tick - lastTick);
136         trackData.push_back(e.status);
137         trackData.push_back(e.data1);
138         trackData.push_back(e.data2);
139         lastTick = e.tick;
140     }
141
142     trackData.push_back(0x00);
143     trackData.push_back(0xFF); trackData.push_back(0x2F); trackData.push_back(0x00);
144     return trackData;
145 }
146
147 static void writeMidiFile(const std::string& filename,
148                           std::vector<std::vector<MidiEvent>>& tracks,
149                           uint16_t PPQ) {
```

```
    std::ofstream file(filename, std::ios::binary);

    file << "MThd";
    uint32_t hLen = __builtin_bswap32(6);
    file.write((char*)&hLen, 4);
    uint16_t format = __builtin_bswap16(1);  // Format 1 : pistes s par es
    file.write((char*)&format, 2);
    uint16_t ntrks = __builtin_bswap16((uint16_t)tracks.size());
    file.write((char*)&ntrks, 2);
    uint16_t division = __builtin_bswap16(PPQ);
    file.write((char*)&division, 2);

    for (auto& track : tracks) {
        auto data = buildTrackData(track);
        file << "MTrk";
        uint32_t tLen = __builtin_bswap32(data.size());
        file.write((char*)&tLen, 4);
        file.write((char*)data.data(), data.size());
    }

    file.close();
}

void saveMidiMultiVoice(const std::string& filename,
                        const std::vector<int>& cantusFirmus,
                        const std::vector<std::pair<std::vector<int>, Species>>& voices) {
    const uint32_t PPQ = 480;
    uint32_t rondeDur = PPQ * 4;

    std::vector<std::vector<MidiEvent>> tracks(1 + std::min(voices.size(), (size_t)3));

    addVoiceEvents(tracks[0], cantusFirmus, CANTUS_FIRMUS, 0, 64, rondeDur);

    for (size_t v = 0; v < voices.size() && v < 3; ++v) {
        addVoiceEvents(tracks[v + 1], voices[v].first, voices[v].second,
                       (uint8_t)(v + 1), 80, rondeDur);
    }

    writeMidiFile(filename, tracks, PPQ);
}

void saveMidi(const std::string& filename,
              const std::vector<int>& cantusFirmus,
              const std::vector<int>& raw_solution,
              Species species) {

    const uint32_t PPQ = 480;
    uint32_t rondeDur = PPQ * 4;

    std::vector<std::vector<MidiEvent>> tracks(2);

    addVoiceEvents(tracks[0], cantusFirmus, CANTUS_FIRMUS, 0, 64, rondeDur);

    addVoiceEvents(tracks[1], raw_solution, species, 1, 80, rondeDur);

    writeMidiFile(filename, tracks, PPQ);
}
```

A.10 Test sources

A.10.1 fuxTest.cpp

```
//
// Created by Luc Cleenewerk and Diego de Patoul.
// Modified by Tom Lai.
// This file contains the testing framework implementation.
//
#include <iostream>
#include <chrono>
#include <fstream>  // For file operations
#include <cmath>
#include <chrono>
#include <signal.h> // For testing blocking constrains in generations
#include "../../headers/Tests/fuxTest.hpp"
#include "../../headers/GenerationsAPI.hpp"
#include <gecode/int.hh> // Ensure you include the necessary Gecode headers


// ========== Modified by Tom ====================

FuxTest::FuxTest(char* test){
    cantusFirmus = {60,   62,   65,   64,   67,   65,   64,   62,   60};
    cfSize = cantusFirmus.size();
    melodic_params = {0, 1, 2, 576, 5, 10, 25, 40}; // Last ones augmented to avoid crazy leaps
```

```
    general_params = {4, 1, 1, 2, 2, 2, 8, 1};
    specific_params = {8 , 4 , 0 , 2 , 1 , 8 , 50};
    importance = {8,7,5,2,9,3,14,12,6,11,4,10,1,13};
    borrowMode = 1;
    if(strcmp(test, "all")==0){
        cout << "Running all tests..." << endl;
        test_1H1();
        test_1H2();
        test_1H3();
        test_1H4();
        test_1H5();
        // test_1H6();
        test_1H7();
        test_1H10();
        test_1M2();

        test_1H2();
        test_2M2();
        cout << "Tests ended." << endl;
    } else if(strcmp(test, "1H1")==0){
        test_1H1();
    } else if(strcmp(test, "1H2")==0){
        test_1H2();
    } else if(strcmp(test, "1H3")==0){
        test_1H3();
    } else if(strcmp(test, "1H4")==0){
        test_1H4();
    } else if(strcmp(test, "1H5")==0){
        test_1H5();
    // } else if(strcmp(test, "1H6")==0){
    //     test_1H6();
    } else if(strcmp(test, "1H7")==0){
        test_1H7();
    } else if(strcmp(test, "1H10")==0){
        test_1H10();
    } else if(strcmp(test, "1M2")==0){
        test_1M2();
    } else if(strcmp(test, "2H2")==0){
        test_2H2();
    } else if(strcmp(test, "2M2")==0){
        test_2M2();
    } else if(strcmp(test, "figs")==0){
        test_2v_1sp_fig22();
        test_2v_1sp_fig23();
        test_2v_2sp_fig38();
        test_2v_2sp_fig39();
        test_2v_2sp_fig40();
        test_2v_2sp_fig41();
        test_2v_2sp_fig42();
        test_2v_2sp_fig43();
        test_2v_2sp_fig44();
        test_2v_2sp_fig45();
        test_2v_3sp_fig55();
        test_2v_3sp_fig56();
        test_2v_3sp_fig57();
        test_2v_3sp_fig58();
        test_2v_3sp_fig59();
        test_2v_3sp_fig60();
        test_2v_4sp_fig74();
        test_2v_4sp_fig75();
        test_2v_4sp_fig76();
        test_2v_4sp_fig77();
        test_2v_4sp_fig78();
        test_3v_1sp_fig108();
        test_3v_1sp_fig109();
        test_3v_1sp_fig110();
        test_3v_1sp_fig111();
        test_3v_1sp_fig112();
        test_3v_1sp_fig113();
        test_3v_1sp_fig114();
        test_3v_1sp_fig115();
        test_3v_1sp_fig116();
        test_3v_1sp_fig117();
        test_3v_1sp_fig118();
        test_3v_1sp_fig119();
        test_3v_2sp_fig125();
        test_3v_2sp_fig128();
        test_3v_2sp_fig129();
        test_3v_3sp_fig132();
        test_3v_3sp_fig133();
        test_4v_2sp_fig176();
    } else if(strcmp(test, "gen")==0){
        test_gen_custom();
    } else if(strcmp(test, "gen_all")==0){
        test_gen_all();
    } else if(strcmp(test, "gen_classic")==0){
        test_gen_classic();
    } else if(strcmp(test, "gen_fullsp")==0){
        test_gen_multi();
    }
```

```
113     else if(strcmp(test, "sacha")==0){
114         cout << "Running sacha's test..." << endl;
115         test_sacha();
116     }
117     else {
118         std::invalid_argument("Test for constraint not found!");
119     }
120     //      CounterpointProblem* problem;
121     //      problem = dispatcher(test);
122     //      BAB<CounterpointProblem> e(problem);
123     //      int nb_sol = 0;
124     //      while(CounterpointProblem* pb = e.next()){
125     //          nb_sol++;
126     //          cout << pb->to_string() << endl;
127     //          delete pb;
128     //          if (nb_sol >= 1)
129     //              break;
130     //      }
131     //      if(nb_sol==0){
132     //          cout << "This constraint works. It prohibits a forbidden configuration." << endl;
133     //      } else {
134     //          cout << "This constraint does NOT work. It allows a forbidden configuration." <<
    endl;
135     //      }
136     // }
137
138 }
139
140 /*
141 Chromatic melodies are forbidden for two and three voice composition, and to be avoided for four
        voice composition.
142 */
143 void FuxTest::test_G6() {
144     cout << "Start tests G6..." << endl;
145     test_G6_2v_1sp();
146     cout << "End tests G6." << endl;
147 }
148
149 void FuxTest::test_G6_2v_1sp() {
150     // TODO
151     // spList = {FIRST_SPECIES};
152     // v_type = {0};
153     // auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
154     // std::vector<CounterpointProblem*> solutions = get_all_solutions(problem);
155     // for (CounterpointProblem* solution: solutions) {
156     //      if (solution->getSolutionArray()[0].val() % 12 != cantusFirmus[0] % 12) {
157     //          std::cerr   << "/!\\ ERROR test 1H4_2v_1sp: It exists solution but it shouldn't
    with the following configuration:\n"
158     //                      << "cantus firmus: ";
159     //          printVector(cantusFirmus);
160     //          std::cerr   << "solution array: ";
161     //          printIntVarArray(problem->getSolutionArray());
162     //          std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
163     //      }
164     //      if (solution->getSolutionArray()[cfSize-1].val() % 12 != cantusFirmus[0] % 12) {
165     //          std::cerr   << "/!\\ ERROR test 1H4_2v_1sp: It exists solution but it shouldn't
    with the following configuration:\n"
166     //                      << "cantus firmus: ";
167     //          printVector(cantusFirmus);
168     //          std::cerr   << "solution array: ";
169     //          printIntVarArray(problem->getSolutionArray());
170     //          std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
171     //      }
172     // }
173     // delete problem;
174 }
175
176 /*
177 All notes on the downbeat are consonant with the notes (on the downbeat) of the lowest stratum.
178 */
179 void FuxTest::test_1H1() {
180     cout << "Start tests 1H1..." << endl;
181     test_1H1_2v_1sp();
182     test_1H1_3v_1sp();
183     test_1H1_4v_1sp();
184     test_1H1_2v_2sp();
185     test_1H1_3v_2sp();
186     test_1H1_4v_2sp();
187     test_1H1_2v_3sp();
188     test_1H1_3v_3sp();
189     test_1H1_4v_3sp();
190     test_1H1_2v_4sp();
191     test_1H1_3v_4sp();
192     test_1H1_4v_4sp();
193     test_1H1_2v_5sp();
194     cout << "End tests 1H1." << endl;
195 }
196
197 void FuxTest::test_1H1_2v_1sp() {
```

```
198     cout << "Start test 1H1_2v_1sp..." << endl;
199     // define dissonant intervals
200     int dis[] = {1, 2, 5, 6, 10, 11};
201     // define the species
202     spList = {FIRST_SPECIES};
203     // define the voice types (=pitch range):
204     // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
205     v_type = {3};
206     // Test that dissonant notes are forbidden
207     for (int interval : dis) {
208         for (size_t i = 0; i < cfSize; i++) {
209             CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
210             int note = cantusFirmus[i] + 12 + interval; // note is dissonant
211             rel(problem->getHome(), problem->getSolutionArray()[i], IRT_EQ, note); // fix the
    note i
212             if (has_solution(problem)) {
213                 std::cerr   << "/!\\ ERROR test 1H1_2v_1sp: It exists a solution but it shouldn'
    t with the following configuration:\n"
214                             << "Cantus firmus: ";
215                 printVector(cantusFirmus);
216                 std::cerr   << "Solution array: ";
217                 printIntVarArray(problem->getSolutionArray());
218                 std::cerr   << "> Dissonant harmonic at the mesure " << i << std::endl;
219             }
220             delete problem;
221         }
222     }
223     cout << "End test 1H1_2v_1sp..." << endl;
224 }
225
226 void FuxTest::test_1H1_3v_1sp() {
227     cout << "Start test 1H1_3v_1sp..." << endl;
228     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
229     int cons[] = {3, 4, 7, 8, 9}; // conssonant intervals without 0 because 1H5
230     spList = {FIRST_SPECIES, FIRST_SPECIES};
231     v_type = {3, 3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
232     // Test that dissonant notes are forbidden
233     for (int interval : dis) {
234         for (size_t j = 0; j < 2; j++) { // iterate over each solution line
235             for (size_t i = 0; i < cfSize; i++) {
236                 CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
237                 int note = cantusFirmus[i] + 12 + interval; // note is dissonant
238                 rel(problem->getHome(), problem->getSolutionArray()[j*cfSize+i], IRT_EQ, note);
    // fix the note i
239                 if (has_solution(problem)) {
240                     std::cerr   << "/!\\ ERROR test 1H1_3v_1sp: It exists a solution but it
    shouldn't with the following configuration:\n"
241                                 << "Cantus firmus: ";
242                     printVector(cantusFirmus);
243                     std::cerr   << "Solution array: ";
244                     printIntVarArray(problem->getSolutionArray());
245                     std::cerr   << "> Dissonant harmonic at the mesure " << i << std::endl;
246                 }
247                 delete problem;
248             }
249         }
250     }
251     cout << "End test 1H1_3v_1sp..." << endl;
252 }
253
254 void FuxTest::test_1H1_4v_1sp() {
255     cout << "Start test 1H1_4v_1sp..." << endl;
256     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
257     int cons[] = {3, 4, 7, 8, 9}; // conssonant intervals without 0 because 1H5
258     spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
259     v_type = {3, 3, 3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
260     // Test that dissonant notes are forbidden
261     for (int interval : dis) {
262         for (size_t j = 0; j < 2; j++) { // iterate over each solution line
263             for (size_t i = 0; i < cfSize; i++) {
264                 CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
265                 int note = cantusFirmus[i] + 12 + interval; // note is dissonant
266                 rel(problem->getHome(), problem->getSolutionArray()[j*cfSize+i], IRT_EQ, note);
    // fix the note i
267                 if (has_solution(problem)) {
268                     std::cerr   << "/!\\ ERROR test 1H1_4v_1sp: It exists a solution but it
    shouldn't with the following configuration:\n"
269                                 << "Cantus firmus: ";
270                     printVector(cantusFirmus);
271                     std::cerr   << "Solution array: ";
272                     printIntVarArray(problem->getSolutionArray());
273                     std::cerr   << "> Dissonant harmonic at the mesure " << i << std::endl;
274                 }
275                 delete problem;
276             }
277         }
278     }
```

```
279     cout << "End test 1H1_4v_1sp..." << endl;
280 }
281
282 void FuxTest::test_1H1_2v_2sp() {
283     cout << "Start test 1H1_2v_2sp..." << endl;
284     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
285     int cons[] = {3, 4, 7, 8, 9}; // conssonant intervals without 0 because 1H5
286     spList = {SECOND_SPECIES};
287     v_type = {3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
288     // Test that dissonant notes are forbidden
289     for (int interval : dis) {
290         for (size_t i = 1; i < cfSize; i++) {
291             CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
292             int note = cantusFirmus[i] + 12 + interval; // note is dissonant
293             rel(problem->getHome(), problem->getSolutionArray()[i*2], IRT_EQ, note); // fix the
    note i (downbeat)
294             if (has_solution(problem)) {
295                 std::cerr   << "/!\\ ERROR test 1H1_2v_2sp: It exists a solution but it shouldn'
    t with the following configuration:\n"
296                             << "Cantus firmus: ";
297                 printVector(cantusFirmus);
298                 std::cerr   << "Solution array: ";
299                 printIntVarArray(problem->getSolutionArray());
300                 std::cerr   << "> Dissonant harmonic at the mesure " << i << std::endl;
301             }
302             delete problem;
303         }
304     }
305     cout << "End test 1H1_2v_2sp..." << endl;
306 }
307
308 void FuxTest::test_1H1_3v_2sp() {
309     cout << "Start test 1H1_3v_2sp..." << endl;
310     int cpSize = cfSize*2-1; // size of a counterpoint
311     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
312     int cons[] = {3, 4, 7, 8, 9}; // conssonant intervals without 0 because 1H5
313     spList = {SECOND_SPECIES, SECOND_SPECIES};
314     v_type = {3, 3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
315     // Test that dissonant notes are forbidden
316     for (int interval : dis) {
317         for (size_t j = 0; j < 2; j++) { // iterate over each solution line
318             for (size_t i = 1; i < cfSize; i++) {
319                 CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
320                 int note = cantusFirmus[i] + 12 + interval; // note is dissonant
321                 rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize)+(i*2)], IRT_EQ,
    note); // fix the note i
322                 if (has_solution(problem)) {
323                     std::cerr   << "/!\\ ERROR test 1H1_3v_2sp: It exists a solution but it
    shouldn't with the following configuration:\n"
324                                 << "Cantus firmus: ";
325                     printVector(cantusFirmus);
326                     std::cerr   << "Solution array: ";
327                     printIntVarArray(problem->getSolutionArray());
328                     std::cerr   << "> Dissonant harmonic at the mesure " << i << std::endl;
329                 }
330                 delete problem;
331             }
332         }
333     }
334     cout << "End test 1H1_3v_2sp..." << endl;
335 }
336
337 void FuxTest::test_1H1_4v_2sp() {
338     cout << "Start test 1H1_4v_2sp..." << endl;
339     int cpSize = cfSize*2-1; // size of a counterpoint
340     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
341     int cons[] = {3, 4, 7, 8, 9}; // conssonant intervals without 0 because 1H5
342     spList = {SECOND_SPECIES, SECOND_SPECIES, SECOND_SPECIES};
343     v_type = {3, 3, 3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
344     // Test that dissonant notes are forbidden
345     for (int interval : dis) {
346         for (size_t j = 0; j < 3; j++) { // iterate over each solution line
347             for (size_t i = 1; i < cfSize; i++) {
348                 CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
349                 int note = cantusFirmus[i] + 12 + interval; // note is dissonant
350                 rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize)+(i*2)], IRT_EQ,
    note); // fix the note i
351                 if (has_solution(problem)) {
352                     std::cerr   << "/!\\ ERROR test 1H1_4v_2sp: It exists a solution but it
    shouldn't with the following configuration:\n"
353                                 << "Cantus firmus: ";
354                     printVector(cantusFirmus);
355                     std::cerr   << "Solution array: ";
356                     printIntVarArray(problem->getSolutionArray());
357                     std::cerr   << "> Dissonant harmonic at the mesure " << i << std::endl;
358                 }
359                 delete problem;
```

```
360             }
361         }
362     }
363     cout << "End test 1H1_4v_2sp..." << endl;
364 }
365
366 void FuxTest::test_1H1_2v_3sp() {
367     cout << "Start test 1H1_2v_3sp..." << endl;
368     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
369     spList = {THIRD_SPECIES};
370     v_type = {3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
371     // Test that dissonant notes are forbidden
372     for (int interval : dis) {
373         for (size_t i = 0; i < cfSize; i++) {
374             CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
375             int note = cantusFirmus[i] + 12 + interval; // note is dissonant
376             rel(problem->getHome(), problem->getSolutionArray()[i*4], IRT_EQ, note); // fix the
    note i (downbeat)
377             if (has_solution(problem)) {
378                 std::cerr   << "/!\\ ERROR test 1H1_2v_2sp: It exists a solution but it shouldn'
    t with the following configuration:\n"
379                             << "Cantus firmus: ";
380                 printVector(cantusFirmus);
381                 std::cerr   << "Solution array: ";
382                 printIntVarArray(problem->getSolutionArray());
383                 std::cerr   << "> Dissonant harmonic at the mesure " << i << std::endl;
384             }
385             delete problem;
386         }
387     }
388     cout << "End test 1H1_2v_3sp..." << endl;
389 }
390
391 void FuxTest::test_1H1_3v_3sp() {
392     cout << "Start test 1H1_3v_3sp..." << endl;
393     int cpSize = cfSize*4-3; // size of a counterpoint
394     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
395     spList = {THIRD_SPECIES, THIRD_SPECIES};
396     v_type = {3, 3};
397     // Test that dissonant notes are forbidden
398     for (int interval : dis) {
399         for (size_t i = 0; i < cfSize; i++) {
400             for (size_t j = 0; j < 2; j++) {
401                 CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
402                 int note = cantusFirmus[i] + 12 + interval; // note is dissonant
403                 rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize) + (i * 4)],
    IRT_EQ, note); // fix the downbeat of each mesure
404                 if (has_solution(problem)) {
405                     std::cerr   << "/!\\ ERROR test 1H1_3v_3sp: It exists a solution but it
    shouldn't with the following configuration:\n"
406                                 << "Cantus firmus: ";
407                     printVector(cantusFirmus);
408                     std::cerr   << "Solution array: ";
409                     printIntVarArray(problem->getSolutionArray());
410                     std::cerr   << "> Dissonant harmonic at the measure " << i << std::endl;
411                 }
412                 delete problem;
413             }
414         }
415     }
416     cout << "End test 1H1_3v_3sp..." << endl;
417 }
418
419 void FuxTest::test_1H1_4v_3sp() {
420     cout << "Start test 1H1_4v_3sp..." << endl;
421     int cpSize = cfSize*4-3; // size of a counterpoint
422     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
423     spList = {THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES};
424     v_type = {3, 3, 3};
425     // Test that dissonant notes are forbidden
426     for (int interval : dis) {
427         for (size_t i = 0; i < cfSize; i++) {
428             for (size_t j = 0; j < 3; j++) {
429                 CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
430                 int note = cantusFirmus[i] + 12 + interval; // note is dissonant
431                 rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize) + (i * 4)],
    IRT_EQ, note); // fix the downbeat of each mesure
432                 if (has_solution(problem)) {
433                     std::cerr   << "/!\\ ERROR test 1H1_4v_3sp: It exists a solution but it
    shouldn't with the following configuration:\n"
434                                 << "Cantus firmus: ";
435                     printVector(cantusFirmus);
436                     std::cerr   << "Solution array: ";
437                     printIntVarArray(problem->getSolutionArray());
438                     std::cerr   << "> Dissonant harmonic at the measure " << i << std::endl;
439                 }
440                 delete problem;
```

```
441             }
442         }
443     }
444     cout << "End test 1H1_4v_3sp..." << endl;
445 }
446 
447 void FuxTest::test_1H1_2v_4sp() {
448     cout << "Start test 1H1_2v_4sp..." << endl;
449     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
450     spList = {FOURTH_SPECIES};
451     v_type = {3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
452     // Test that dissonant notes are forbidden
453     for (int interval : dis) {
454         for (size_t i = 1; i < cfSize; i++) {
455             CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
456             if (i == cfSize-1) { // last note
457                 int note = cantusFirmus[i] + 12 + interval; // note is dissonant
458                 rel(problem->getHome(), problem->getSolutionArray()[(i*2)-1], IRT_EQ, note); //
    fix the note i (downbeat)
459             } else { // other notes
460                 int note = cantusFirmus[i] + 12 + interval; // note is dissonant
461                 rel(problem->getHome(), problem->getSolutionArray()[i*2], IRT_EQ, note); // fix
    the note i (downbeat)
462             }
463             if (has_solution(problem)) {
464                 std::cerr   << "/!\\ ERROR test 1H1_2v_4sp: It exists a solution but it shouldn'
    t with the following configuration:\n"
465                             << "Cantus firmus: ";
466                 printVector(cantusFirmus);
467                 std::cerr   << "Solution array: ";
468                 printIntVarArray(problem->getSolutionArray());
469                 std::cerr   << "> Dissonant harmonic at the mesure " << i << std::endl;
470             }
471             delete problem;
472         }
473     }
474     cout << "End test 1H1_2v_4sp..." << endl;
475 }
476 
477 void FuxTest::test_1H1_3v_4sp() {
478     cout << "Start test 1H1_3v_4sp..." << endl;
479     int cpSize = cfSize*2-2; // size of a counterpoint
480     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
481     spList = {FOURTH_SPECIES, FOURTH_SPECIES};
482     v_type = {3, 3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
483     // Test that dissonant notes are forbidden
484     for (int interval : dis) {
485         for (size_t i = 1; i < cfSize; i++) {
486             for (size_t j = 0; j < 2; j++) {
487                 CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
488                 if (i == cfSize-1) { // last note
489                     int note = cantusFirmus[i] + 12 + interval; // note is dissonant
490                     rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize)+(i*2)-1],
    IRT_EQ, note); // fix the note i (downbeat)
491                 } else { // other notes
492                     int note = cantusFirmus[i] + 12 + interval; // note is dissonant
493                     rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize)+(i*2)],
    IRT_EQ, note); // fix the note i (downbeat)
494                 }
495                 if (has_solution(problem)) {
496                     std::cerr   << "/!\\ ERROR test 1H1_3v_4sp: It exists a solution but it
    shouldn't with the following configuration:\n"
497                                 << "Cantus firmus: ";
498                     printVector(cantusFirmus);
499                     std::cerr   << "Solution array: ";
500                     printIntVarArray(problem->getSolutionArray());
501                     std::cerr   << "> Dissonant harmonic at the mesure " << i << std::endl;
502                 }
503                 delete problem;
504             }
505         }
506     }
507     cout << "End test 1H1_3v_4sp..." << endl;
508 }
509 
510 void FuxTest::test_1H1_4v_4sp() {
511     cout << "Start test 1H1_4v_4sp..." << endl;
512     int cpSize = cfSize*2-2; // size of a counterpoint
513     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
514     spList = {FOURTH_SPECIES, FOURTH_SPECIES, FOURTH_SPECIES};
515     v_type = {3, 3, 3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
516     // Test that dissonant notes are forbidden
517     for (int interval : dis) {
518         for (size_t i = 1; i < cfSize; i++) {
519             for (size_t j = 0; j < 3; j++) {
520                 CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
521                 if (i == cfSize-1) { // last note
```

```
522                     int note = cantusFirmus[i] + 12 + interval; // note is dissonant
523                     rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize)+(i*2)-1],
    IRT_EQ, note); // fix the note i (downbeat)
524                 } else { // other notes
525                     int note = cantusFirmus[i] + 12 + interval; // note is dissonant
526                     rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize)+(i*2)],
    IRT_EQ, note); // fix the note i (downbeat)
527                 }
528                 if (has_solution(problem)) {
529                     std::cerr   << "/!\\ ERROR test 1H1_4v_4sp: It exists a solution but it
    shouldn't with the following configuration:\n"
530                                 << "Cantus firmus: ";
531                     printVector(cantusFirmus);
532                     std::cerr   << "Solution array: ";
533                     printIntVarArray(problem->getSolutionArray());
534                     std::cerr   << "> Dissonant harmonic at the mesure " << i << std::endl;
535                 }
536                 delete problem;
537             }
538         }
539     }
540     cout << "End test 1H1_4v_4sp..." << endl;
541 }
542
543 void FuxTest::test_1H1_2v_5sp() {
544     cout << "Start test 1H1_2v_5sp..." << endl;
545     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
546     spList = {FIFTH_SPECIES};
547     v_type = {3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
548     // Test that dissonant notes are forbidden
549     for (int interval : dis) {
550         for (size_t i = 0; i < cfSize; i++) {
551             CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
552             int note = cantusFirmus[i] + 12 + interval; // note is dissonant
553             rel(problem->getHome(), problem->getSolutionArray()[i*4], IRT_EQ, note); // fix the
    note i (downbeat)
554             if (has_solution(problem)) {
555                 std::cerr   << "/!\\ ERROR test 1H1_2v_5sp: It exists a solution but it shouldn'
    t with the following configuration:\n"
556                             << "Cantus firmus: ";
557                 printVector(cantusFirmus);
558                 std::cerr   << "Solution array: ";
559                 printIntVarArray(problem->getSolutionArray());
560                 std::cerr   << "> Dissonant harmonic at the mesure " << i << std::endl;
561             }
562             delete problem;
563         }
564     }
565     cout << "End test 1H1_2v_5sp..." << endl;
566 }
567
568 // void FuxTest::test_1H1_3v_5sp() {
569 //     cout << "Start test 1H1_3v_5sp..." << endl;
570 //     int cpSize = cfSize*2-2; // size of a counterpoint
571 //     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
572 //     spList = {FOURTH_SPECIES, FOURTH_SPECIES};
573 //     v_type = {3, 3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
574 //     // Test that dissonant notes are forbidden
575 //     for (int interval : dis) {
576 //         for (size_t i = 1; i < cfSize; i++) {
577 //             for (size_t j = 0; j < 2; j++) {
578 //                 CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
579 //                 if (i == cfSize-1) { // last note
580 //                     int note = cantusFirmus[i] + 12 + interval; // note is dissonant
581 //                     rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize)+(i*2)-1],
    IRT_EQ, note); // fix the note i (downbeat)
582 //                 } else { // other notes
583 //                     int note = cantusFirmus[i] + 12 + interval; // note is dissonant
584 //                     rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize)+(i*2)],
    IRT_EQ, note); // fix the note i (downbeat)
585 //                 }
586 //                 if (has_solution(problem)) {
587 //                     std::cerr   << "/!\\ ERROR test 1H1_3v_4sp: It exists a solution but it
    shouldn't with the following configuration:\n"
588 //                                 << "Cantus firmus: ";
589 //                     printVector(cantusFirmus);
590 //                     std::cerr   << "Solution array: ";
591 //                     printIntVarArray(problem->getSolutionArray());
592 //                     std::cerr   << "> Dissonant harmonic at the mesure " << i << std::endl;
593 //                 }
594 //                 delete problem;
595 //             }
596 //         }
597 //     }
598 //     cout << "End test 1H1_3v_5sp..." << endl;
599 // }
600
601
```

```
602 /*
603 In two voice composition, the first harmonic interval must be a perfect consonance.
604 */
605 void FuxTest::test_1H2() {
606     cout << "Start tests 1H2..." << endl;
607     test_1H2_2v_1sp();
608     test_1H2_2v_2sp();
609     test_1H2_2v_3sp();
610     test_1H2_2v_4sp();
611     cout << "End tests 1H2." << endl;
612 }
613
614 void FuxTest::test_1H2_2v_1sp() {
615     cout << "Start test 1H2_2v_1sp..." << endl;
616     int p_cons[] = {0, 7}; // perfect conssonant intervals
617     int non_p_cons[] = {1, 2, 3, 4, 5, 6, 8, 9, 10, 11}; // non-perfect conssonant intervals
618     spList = {FIRST_SPECIES};
619     v_type = {3};
620     for (int interval : p_cons) {
621         CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
622         rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, cantusFirmus[0]+12+
    interval); // fix the first solution note as perfect consonant with cf
623         if (!has_solution(problem)) {
624             std::cerr   << "/!\\ ERROR test 1H2_2v_1sp: It doesn't exist a solution but it
    should with the following configuration:\n"
625                         << "Cantus firmus: ";
626             printVector(cantusFirmus);
627             std::cerr   << "Solution array: ";
628             printIntVarArray(problem->getSolutionArray());
629             std::cerr   << "> First harmonic interval should be correct" << std::endl;
630         }
631         delete problem;
632     }
633     for (int interval : non_p_cons) {
634         CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
635         rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, cantusFirmus[0]+12+
    interval); // fix the first solution note as non-perfect consonant with cf
636         if (has_solution(problem)) {
637             std::cerr   << "/!\\ ERROR test 1H2_2v_1sp: It exists a solution but it shouldn't
    with the following configuration:\n"
638                         << "Cantus firmus: ";
639             printVector(cantusFirmus);
640             std::cerr   << "Solution array: ";
641             printIntVarArray(problem->getSolutionArray());
642             std::cerr   << "> Non perfect consonance at the first mesure" << std::endl;
643         }
644         delete problem;
645     }
646     cout << "End test 1H2_2v_1sp..." << endl;
647 }
648
649 void FuxTest::test_1H2_2v_2sp() {
650     cout << "Start test 1H2_2v_2sp..." << endl;
651     int p_cons[] = {0, 7}; // perfect conssonant intervals
652     int non_p_cons[] = {1, 2, 3, 4, 5, 6, 8, 9, 10, 11}; // non-perfect conssonant intervals
653     spList = {SECOND_SPECIES};
654     v_type = {3};
655     for (int interval : p_cons) {
656         CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
657         rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, cantusFirmus[0]+12+
    interval); // fix the first solution note as perfect consonant with cf
658         if (!has_solution(problem)) {
659             std::cerr   << "/!\\ ERROR test 1H2_2v_2sp: It doesn't exist a solution but it
    should with the following configuration:\n"
660                         << "Cantus firmus: ";
661             printVector(cantusFirmus);
662             std::cerr   << "Solution array: ";
663             printIntVarArray(problem->getSolutionArray());
664             std::cerr   << "> First harmonic interval should be correct" << std::endl;
665         }
666         delete problem;
667     }
668     for (int interval : non_p_cons) {
669         CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
670         rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, cantusFirmus[0]+12+
    interval); // fix the first solution note as non-perfect consonant with cf
671         if (has_solution(problem)) {
672             std::cerr   << "/!\\ ERROR test 1H2_2v_2sp: It exists a solution but it shouldn't
    with the following configuration:\n"
673                         << "Cantus firmus: ";
674             printVector(cantusFirmus);
675             std::cerr   << "Solution array: ";
676             printIntVarArray(problem->getSolutionArray());
677             std::cerr   << "> Non perfect consonance at the first mesure" << std::endl;
678         }
679         delete problem;
```

```
680     }
681     cout << "End test 1H2_2v_2sp..." << endl;
682 }
683
684 void FuxTest::test_1H2_2v_3sp() {
685     cout << "Start test 1H2_2v_3sp..." << endl;
686     int p_cons[] = {0, 7}; // perfect conssonant intervals
687     int non_p_cons[] = {1, 2, 3, 4, 5, 6, 8, 9, 10, 11}; // non-perfect conssonant intervals
688     spList = {THIRD_SPECIES};
689     v_type = {3};
690     // perfect conssonant are allowed
691     for (int interval : p_cons) {
692         CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
693         int note = cantusFirmus[0] + 12 + interval; // note is perfect consonant
694         rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); // fix the first
    solution note as perfect consonant with cf
695         if (!has_solution(problem)) {
696             std::cerr   << "/!\\ ERROR test 1H2_2v_3sp: It doesn't exist a solution but it
    should with the following configuration:\n"
697                         << "Cantus firmus: ";
698             printVector(cantusFirmus);
699             std::cerr   << "Solution array: ";
700             printIntVarArray(problem->getSolutionArray());
701             std::cerr   << "> First harmonic interval should be correct" << std::endl;
702         }
703         delete problem;
704     }
705     // non-perfect consonant are forbidden
706     for (int interval : non_p_cons) {
707         CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
708         int note = cantusFirmus[0] + 12 + interval; // note is non-perfect consonant
709         rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); // fix the first
    solution note as non-perfect consonant with cf
710         if (has_solution(problem)) {
711             std::cerr   << "/!\\ ERROR test 1H2_2v_3sp: It exists a solution but it shouldn't
    with the following configuration:\n"
712                         << "Cantus firmus: ";
713             printVector(cantusFirmus);
714             std::cerr   << "Solution array: ";
715             printIntVarArray(problem->getSolutionArray());
716             std::cerr   << "> Non perfect consonance at the first mesure" << std::endl;
717         }
718         delete problem;
719     }
720     cout << "End test 1H2_2v_3sp..." << endl;
721 }
722
723 void FuxTest::test_1H2_2v_4sp() {
724     cout << "Start test 1H2_2v_4sp..." << endl;
725     int p_cons[] = {0, 7}; // perfect conssonant intervals
726     int non_p_cons[] = {1, 2, 3, 4, 5, 6, 8, 9, 10, 11}; // non-perfect conssonant intervals
727     spList = {FOURTH_SPECIES};
728     v_type = {3};
729     // perfect conssonant are allowed
730     for (int interval : p_cons) {
731         CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
732         int note = cantusFirmus[0] + 12 + interval; // note is perfect consonant
733         rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); // fix the first
    solution note as perfect consonant with cf
734         if (!has_solution(problem)) {
735             std::cerr   << "/!\\ ERROR test 1H2_2v_4sp: It doesn't exist a solution but it
    should with the following configuration:\n"
736                         << "Cantus firmus: ";
737             printVector(cantusFirmus);
738             std::cerr   << "Solution array: ";
739             printIntVarArray(problem->getSolutionArray());
740             std::cerr   << "> First harmonic interval should be correct" << std::endl;
741         }
742         delete problem;
743     }
744     // non-perfect consonant are forbidden
745     for (int interval : non_p_cons) {
746         CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
747         int note = cantusFirmus[0] + 12 + interval; // note is non-perfect consonant
748         rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); // fix the first
    solution note as non-perfect consonant with cf
749         if (has_solution(problem)) {
750             std::cerr   << "/!\\ ERROR test 1H2_2v_4sp: It exists a solution but it shouldn't
    with the following configuration:\n"
751                         << "Cantus firmus: ";
752             printVector(cantusFirmus);
753             std::cerr   << "Solution array: ";
754             printIntVarArray(problem->getSolutionArray());
755             std::cerr   << "> Non perfect consonance at the first mesure" << std::endl;
756         }
757         delete problem;
```

```
758     }
759     cout << "End test 1H2_2v_4sp..." << endl;
760 }
761 
762 
763 /*
764 In two voice composition, the last harmonic in-terval must be a perfect consonance.
765 When composing for three or four voices, the last chord should be composed only of notes of the
        harmonic triad.
766 */
767 void FuxTest::test_1H3() {
768     cout << "Start tests 1H3..." << endl;
769     test_1H3_2v_1sp();
770     test_1H3_3v_1sp();
771     test_1H3_4v_1sp();
772     test_1H3_2v_2sp();
773     test_1H3_3v_2sp();
774     test_1H3_4v_2sp();
775     test_1H3_2v_3sp();
776     test_1H3_3v_3sp();
777     test_1H3_4v_3sp();
778     test_1H3_2v_4sp();
779     test_1H3_3v_4sp();
780     test_1H3_4v_4sp();
781     cout << "End tests 1H3." << endl;
782 }
783 
784 void FuxTest::test_1H3_2v_1sp() {
785     cout << "Start test 1H3_2v_1sp..." << endl;
786     int p_cons[] = {0, 7}; // perfect conssonant intervals
787     int non_p_cons[] = {1, 2, 3, 4, 5, 6, 8, 9, 10, 11}; // non-perfect conssonant intervals
788     spList = {FIRST_SPECIES};
789     v_type = {0};
790     // non-perfect consonant intervals are forbidden
791     for (int interval : non_p_cons) {
792         CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
        melodic_params, general_params, specific_params, importance, borrowMode);
793         rel(problem->getHome(), problem->getSolutionArray()[cfSize-1], IRT_EQ, cantusFirmus[
        cfSize-1]+interval); // fix the last note with a not perfect consonance
794         if (has_solution(problem)) {
795             std::cerr   << "/!\\ ERROR test 1H3_2v_1sp: It exists solution but it shouldn't with
         the following configuration:\n"
796                         << "cantus firmus: ";
797             printVector(cantusFirmus);
798             std::cerr   << "solution array: ";
799             printIntVarArray(problem->getSolutionArray());
800             std::cerr   << "> Last harmonic interval must be a perfect consonance" << std::endl;
801         }
802         delete problem;
803     }
804     cout << "End test 1H3_2v_1sp..." << endl;
805 }
806 
807 void FuxTest::test_1H3_3v_1sp() {
808     cout << "Start test 1H3_3v_1sp..." << endl;
809     int triad_cons[] = {0, 3, 4, 7}; // harmonic triad intervals
810     int non_triad_cons[] = {1, 2, 5, 6, 8, 9, 10, 11}; // non-harmonic triad intervals
811     spList = {FIRST_SPECIES, FIRST_SPECIES};
812     v_type = {3, 3};
813     // non-perfect consonant intervals are forbidden
814     for (size_t i = 0; i < 2; i++) { // iterate over each solution line
815         for (int interval : non_triad_cons) {
816             CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
        melodic_params, general_params, specific_params, importance, borrowMode);
817             rel(problem->getHome(), problem->getSolutionArray()[(cfSize*i)+cfSize-1], IRT_EQ,
        cantusFirmus[cfSize-1] + 12 + interval); // fix the last note as a note not in harmonic
        triad intervals
818             if (has_solution(problem)) {
819                 std::cerr   << "/!\\ ERROR test 1H3_3v_1sp: It exists solution but it shouldn't
        with the following configuration:\n"
820                             << "cantus firmus: ";
821                 printVector(cantusFirmus);
822                 std::cerr   << "solution array: ";
823                 printIntVarArray(problem->getSolutionArray());
824                 std::cerr   << "> Last harmonic intervals must be in harmonic triad" << std::
        endl;
825             }
826             delete problem;
827         }
828     }
829     // perfect consonant intervals are allowed
830     for (int interval : triad_cons) {
831         for (size_t i = 0; i < 2; i++) { // iterate over each solution line
832             CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
        melodic_params, general_params, specific_params, importance, borrowMode);
833             rel(problem->getHome(), problem->getSolutionArray()[(cfSize*i)+cfSize-1], IRT_EQ,
        cantusFirmus[cfSize-1] + 12 + interval); // fix the last note as a note in harmonic triad
        intervals
834             if (!has_solution(problem)) {
835                 std::cerr   << "/!\\ ERROR test 1H3_3v_1sp: It doesn't exist solution but it
```

```
    should with the following configuration:\n"
836                         << "cantus firmus: ";
837                 printVector(cantusFirmus);
838                 std::cerr   << "solution array: ";
839                 printIntVarArray(problem->getSolutionArray());
840                 std::cerr   << "> Last harmonic intervals must be in harmonic triad" << std::
    endl;
841             }
842             delete problem;
843         }
844     }
845     cout << "End test 1H3_3v_1sp..." << endl;
846 }
847
848 void FuxTest::test_1H3_4v_1sp() {
849     cout << "Start test 1H3_4v_1sp..." << endl;
850     int triad_cons[] = {0, 3, 4, 7}; // harmonic triad intervals
851     int non_triad_cons[] = {1, 2, 5, 6, 8, 9, 10, 11}; // non-harmonic triad intervals
852     spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
853     v_type = {3, 3, 3};
854     // non-perfect consonant intervals are forbidden
855     for (size_t i = 0; i < 3; i++) { // iterate over each solution line
856         for (int interval : non_triad_cons) {
857             CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
858             rel(problem->getHome(), problem->getSolutionArray()[(cfSize*i)+cfSize-1], IRT_EQ,
    cantusFirmus[cfSize-1]+12+interval); // fix the last note as a note not in harmonic triad
    intervals
859             if (has_solution(problem)) {
860                 std::cerr   << "/!\\ ERROR test 1H3_4v_1sp: It exists solution but it shouldn't
    with the following configuration:\n"
861                         << "cantus firmus: ";
862                 printVector(cantusFirmus);
863                 std::cerr   << "solution array: ";
864                 printIntVarArray(problem->getSolutionArray());
865                 std::cerr   << "> Last harmonic intervals must be in harmonic triad" << std::
    endl;
866             }
867             delete problem;
868         }
869     }
870     // perfect consonant intervals are allowed
871     for (int interval : triad_cons) {
872         for (size_t i = 0; i < 3; i++) { // iterate over each solution line
873             CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
874             rel(problem->getHome(), problem->getSolutionArray()[(cfSize*i)+cfSize-1], IRT_EQ,
    cantusFirmus[cfSize-1] + +12 + interval); // fix the last note as a note in harmonic triad
    intervals
875             if (!has_solution(problem)) {
876                 std::cerr   << "/!\\ ERROR test 1H3_4v_1sp: It doesn't exist solution but it
    should with the following configuration:\n"
877                         << "cantus firmus: ";
878                 printVector(cantusFirmus);
879                 std::cerr   << "solution array: ";
880                 printIntVarArray(problem->getSolutionArray());
881                 std::cerr   << "> Last harmonic intervals must be in harmonic triad" << std::
    endl;
882             }
883             delete problem;
884         }
885     }
886     cout << "End test 1H3_4v_1sp..." << endl;
887 }
888
889 void FuxTest::test_1H3_2v_2sp() {
890     cout << "Start test 1H3_2v_2sp..." << endl;
891     int cpSize = cfSize*2-1; // size of a counterpoint
892     int p_cons[] = {0, 7}; // perfect conssonant intervals
893     int non_p_cons[] = {1, 2, 3, 4, 5, 6, 8, 9, 10, 11}; // non-perfect conssonant intervals
894     spList = {SECOND_SPECIES};
895     v_type = {0};
896     // non-perfect consonant intervals are forbidden
897     for (int interval : non_p_cons) {
898         CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
899         int note = cantusFirmus[cfSize-1] + interval;
900         rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ, note); // fix the
     last note with a not perfect consonance
901         if (has_solution(problem)) {
902             std::cerr   << "/!\\ ERROR test 1H3_2v_2sp: It exists solution but it shouldn't with
     the following configuration:\n"
903                         << "cantus firmus: ";
904             printVector(cantusFirmus);
905             std::cerr   << "solution array: ";
906             printIntVarArray(problem->getSolutionArray());
907             std::cerr   << "> Last harmonic interval must be a perfect consonance" << std::endl;
908         }
909         delete problem;
910     }
```

```
911     cout << "End test 1H3_2v_1sp..." << endl;
912 }
913
914 void FuxTest::test_1H3_3v_2sp() {
915     cout << "Start test 1H3_3v_2sp..." << endl;
916     int cpSize = cfSize*2-1; // size of a counterpoint
917     int triad_cons[] = {0, 3, 4, 7}; // harmonic triad intervals
918     int non_triad_cons[] = {1, 2, 5, 6, 8, 9, 10, 11}; // non-harmonic triad intervals
919     spList = {SECOND_SPECIES, SECOND_SPECIES};
920     v_type = {0, 0};
921     // non-perfect consonant intervals are forbidden
922     for (size_t i = 0; i < 2; i++) { // iterate over each solution line
923         for (int interval : non_triad_cons) {
924             CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
925             int note = cantusFirmus[cfSize-1] + interval;
926             rel(problem->getHome(), problem->getSolutionArray()[(cpSize*i)+cpSize-1], IRT_EQ,
    note); // fix the last note as a note not in harmonic triad intervals
927             if (has_solution(problem)) {
928                 std::cerr   << "/!\\ ERROR test 1H3_3v_2sp: It exists solution but it shouldn't
    with the following configuration:\n"
929                             << "cantus firmus: ";
930                 printVector(cantusFirmus);
931                 std::cerr   << "solution array: ";
932                 printIntVarArray(problem->getSolutionArray());
933                 std::cerr   << "> Last harmonic intervals must be in harmonic triad" << std::
    endl;
934             }
935             delete problem;
936         }
937     }
938     cout << "End test 1H3_3v_2sp..." << endl;
939 }
940
941 void FuxTest::test_1H3_4v_2sp() {
942     cout << "Start test 1H3_4v_2sp..." << endl;
943     int cpSize = cfSize*2-1; // size of a counterpoint
944     int triad_cons[] = {0, 3, 4, 7}; // harmonic triad intervals
945     int non_triad_cons[] = {1, 2, 5, 6, 8, 9, 10, 11}; // non-harmonic triad intervals
946     spList = {SECOND_SPECIES, SECOND_SPECIES, SECOND_SPECIES};
947     v_type = {0, 0};
948     // non-perfect consonant intervals are forbidden
949     for (size_t i = 0; i < 3; i++) { // iterate over each solution line
950         for (int interval : non_triad_cons) {
951             CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
952             int note = cantusFirmus[cfSize-1] + interval;
953             rel(problem->getHome(), problem->getSolutionArray()[(cpSize*i)+cpSize-1], IRT_EQ,
    note); // fix the last note as a note not in harmonic triad intervals
954             if (has_solution(problem)) {
955                 std::cerr   << "/!\\ ERROR test 1H3_4v_2sp: It exists solution but it shouldn't
    with the following configuration:\n"
956                             << "cantus firmus: ";
957                 printVector(cantusFirmus);
958                 std::cerr   << "solution array: ";
959                 printIntVarArray(problem->getSolutionArray());
960                 std::cerr   << "> Last harmonic intervals must be in harmonic triad" << std::
    endl;
961             }
962             delete problem;
963         }
964     }
965     cout << "End test 1H3_4v_2sp..." << endl;
966 }
967
968 void FuxTest::test_1H3_2v_3sp() {
969     cout << "Start test 1H3_2v_3sp..." << endl;
970     int cpSize = cfSize*4-3; // size of a counterpoint
971     int p_cons[] = {0, 7}; // perfect conssonant intervals
972     int non_p_cons[] = {1, 2, 3, 4, 5, 6, 8, 9, 10, 11}; // non-perfect conssonant intervals
973     spList = {THIRD_SPECIES};
974     v_type = {0};
975     // non-perfect consonant intervals are forbidden
976     for (int interval : non_p_cons) {
977         CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
978         int note = cantusFirmus[cfSize-1] + interval;
979         rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ, note); // fix the
     last note with a not perfect consonance
980         if (has_solution(problem)) {
981             std::cerr   << "/!\\ ERROR test 1H3_2v_3sp: It exists solution but it shouldn't with
     the following configuration:\n"
982                         << "cantus firmus: ";
983             printVector(cantusFirmus);
984             std::cerr   << "solution array: ";
985             printIntVarArray(problem->getSolutionArray());
986             std::cerr   << "> Last harmonic interval must be a perfect consonance" << std::endl;
987         }
988         delete problem;
989     }
```

```
990     cout << "End test 1H3_2v_3sp..." << endl;
991 }
992
993 void FuxTest::test_1H3_3v_3sp() {
994     cout << "Start test 1H3_3v_3sp..." << endl;
995     int cpSize = cfSize*4-3; // size of a counterpoint
996     int triad_cons[] = {0, 3, 4, 7}; // harmonic triad intervals
997     int non_triad_cons[] = {1, 2, 5, 6, 8, 9, 10, 11}; // non-harmonic triad intervals
998     spList = {THIRD_SPECIES, THIRD_SPECIES};
999     v_type = {0, 0};
1000    // non-perfect consonant intervals are forbidden
1001    for (size_t i = 0; i < 2; i++) { // iterate over each solution line
1002        for (int interval : non_triad_cons) {
1003            CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
1004            int note = cantusFirmus[cfSize-1] + interval;
1005            rel(problem->getHome(), problem->getSolutionArray()[(cpSize*i)+cpSize-1], IRT_EQ,
    note); // fix the last note as a note not in harmonic triad intervals
1006            if (has_solution(problem)) {
1007                std::cerr   << "/!\\ ERROR test 1H3_3v_3sp: It exists solution but it shouldn't
    with the following configuration:\n"
1008                            << "cantus firmus: ";
1009                printVector(cantusFirmus);
1010                std::cerr   << "solution array: ";
1011                printIntVarArray(problem->getSolutionArray());
1012                std::cerr   << "> Last harmonic intervals must be in harmonic triad" << std::
    endl;
1013            }
1014            delete problem;
1015        }
1016    }
1017    cout << "End test 1H3_3v_3sp..." << endl;
1018 }
1019
1020 void FuxTest::test_1H3_4v_3sp() {
1021     cout << "Start test 1H3_4v_3sp..." << endl;
1022     int cpSize = cfSize*4-3; // size of a counterpoint
1023     int triad_cons[] = {0, 3, 4, 7}; // harmonic triad intervals
1024     int non_triad_cons[] = {1, 2, 5, 6, 8, 9, 10, 11}; // non-harmonic triad intervals
1025     spList = {THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES};
1026     v_type = {0, 0, 0};
1027     // non-perfect consonant intervals are forbidden
1028     for (size_t i = 0; i < 3; i++) { // iterate over each solution line
1029         for (int interval : non_triad_cons) {
1030             CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
1031             int note = cantusFirmus[cfSize-1] + interval;
1032             rel(problem->getHome(), problem->getSolutionArray()[(cpSize*i)+cpSize-1], IRT_EQ,
    note); // fix the last note as a note not in harmonic triad intervals
1033             if (has_solution(problem)) {
1034                 std::cerr   << "/!\\ ERROR test 1H3_4v_3sp: It exists solution but it shouldn't
    with the following configuration:\n"
1035                             << "cantus firmus: ";
1036                 printVector(cantusFirmus);
1037                 std::cerr   << "solution array: ";
1038                 printIntVarArray(problem->getSolutionArray());
1039                 std::cerr   << "> Last harmonic intervals must be in harmonic triad" << std::
    endl;
1040             }
1041             delete problem;
1042         }
1043     }
1044     cout << "End test 1H3_4v_3sp..." << endl;
1045 }
1046
1047 void FuxTest::test_1H3_2v_4sp() {
1048     cout << "Start test 1H3_2v_4sp..." << endl;
1049     int cpSize = cfSize*2-2; // size of a counterpoint
1050     int p_cons[] = {0, 7}; // perfect conssonant intervals
1051     int non_p_cons[] = {1, 2, 3, 4, 5, 6, 8, 9, 10, 11}; // non-perfect conssonant intervals
1052     spList = {FOURTH_SPECIES};
1053     v_type = {0};
1054     // non-perfect consonant intervals are forbidden
1055     for (int interval : non_p_cons) {
1056         CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
    melodic_params, general_params, specific_params, importance, borrowMode);
1057         int note = cantusFirmus[cfSize-1] + interval;
1058         rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ, note); // fix the
     last note with a not perfect consonance
1059         if (has_solution(problem)) {
1060             std::cerr   << "/!\\ ERROR test 1H3_2v_4sp: It exists solution but it shouldn't with
     the following configuration:\n"
1061                         << "cantus firmus: ";
1062             printVector(cantusFirmus);
1063             std::cerr   << "solution array: ";
1064             printIntVarArray(problem->getSolutionArray());
1065             std::cerr   << "> Last harmonic interval must be a perfect consonance" << std::endl;
1066         }
1067         delete problem;
1068     }
```

```
1069     cout << "End test 1H3_2v_4sp..." << endl;
1070 }
1071
1072 void FuxTest::test_1H3_3v_4sp() {
1073     cout << "Start test 1H3_3v_4sp..." << endl;
1074     int cpSize = cfSize*2-2; // size of a counterpoint
1075     int triad_cons[] = {0, 3, 4, 7}; // harmonic triad intervals
1076     int non_triad_cons[] = {1, 2, 5, 6, 8, 9, 10, 11}; // non-harmonic triad intervals
1077     spList = {FOURTH_SPECIES, FOURTH_SPECIES};
1078     v_type = {0, 0};
1079     // non-perfect consonant intervals are forbidden
1080     for (size_t i = 0; i < 2; i++) { // iterate over each solution line
1081         for (int interval : non_triad_cons) {
1082             CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
     melodic_params, general_params, specific_params, importance, borrowMode);
1083             int note = cantusFirmus[cfSize-1] + interval;
1084             rel(problem->getHome(), problem->getSolutionArray()[(cpSize*i)+cpSize-1], IRT_EQ,
     note); // fix the last note as a note not in harmonic triad intervals
1085             if (has_solution(problem)) {
1086                 std::cerr   << "/!\\ ERROR test 1H3_3v_4sp: It exists solution but it shouldn't
     with the following configuration:\n"
1087                             << "cantus firmus: ";
1088                 printVector(cantusFirmus);
1089                 std::cerr   << "solution array: ";
1090                 printIntVarArray(problem->getSolutionArray());
1091                 std::cerr   << "> Last harmonic intervals must be in harmonic triad" << std::
     endl;
1092             }
1093             delete problem;
1094         }
1095     }
1096     cout << "End test 1H3_3v_4sp..." << endl;
1097 }
1098
1099 void FuxTest::test_1H3_4v_4sp() {
1100     cout << "Start test 1H3_4v_4sp..." << endl;
1101     int cpSize = cfSize*2-2; // size of a counterpoint
1102     int triad_cons[] = {0, 3, 4, 7}; // harmonic triad intervals
1103     int non_triad_cons[] = {1, 2, 5, 6, 8, 9, 10, 11}; // non-harmonic triad intervals
1104     spList = {FOURTH_SPECIES, FOURTH_SPECIES, FOURTH_SPECIES};
1105     v_type = {0, 0, 0};
1106     // non-perfect consonant intervals are forbidden
1107     for (size_t i = 0; i < 3; i++) { // iterate over each solution line
1108         for (int interval : non_triad_cons) {
1109             CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
     melodic_params, general_params, specific_params, importance, borrowMode);
1110             int note = cantusFirmus[cfSize-1] + interval;
1111             rel(problem->getHome(), problem->getSolutionArray()[(cpSize*i)+cpSize-1], IRT_EQ,
     note); // fix the last note as a note not in harmonic triad intervals
1112             if (has_solution(problem)) {
1113                 std::cerr   << "/!\\ ERROR test 1H3_4v_4sp: It exists solution but it shouldn't
     with the following configuration:\n"
1114                             << "cantus firmus: ";
1115                 printVector(cantusFirmus);
1116                 std::cerr   << "solution array: ";
1117                 printIntVarArray(problem->getSolutionArray());
1118                 std::cerr   << "> Last harmonic intervals must be in harmonic triad" << std::
     endl;
1119             }
1120             delete problem;
1121         }
1122     }
1123     cout << "End test 1H3_4v_4sp..." << endl;
1124 }
1125
1126 /*The key tone is tuned according to the first note of the cantus firmus.
1127 In other words, the lowest stratum at the first and last notes must be the tonic.*/
1128 void FuxTest::test_1H4() {
1129     cout << "Start tests 1H4..." << endl;
1130     test_1H4_2v_1sp();
1131     test_1H4_3v_1sp();
1132     test_1H4_4v_1sp();
1133     test_1H4_2v_2sp();
1134     test_1H4_3v_2sp();
1135     test_1H4_4v_2sp();
1136     test_1H4_2v_3sp();
1137     test_1H4_3v_3sp();
1138     test_1H4_4v_3sp();
1139     test_1H4_2v_4sp();
1140     test_1H4_3v_4sp();
1141     test_1H4_4v_4sp();
1142     cout << "End tests 1H4." << endl;
1143 }
1144
1145 void FuxTest::test_1H4_2v_1sp() {
1146     cout << "Start test 1H4_2v_1sp..." << endl;
1147     spList = {FIRST_SPECIES};
1148     v_type = {-2};
1149     for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
1150         // fix the first note of the lowest stratum
```

```
1151        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
1152        int note = cantusFirmus[0] + i;
1153        rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); // fix the first
    note of the lowest stratum
1154        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1155            std::cerr   << "/!\\ ERROR test 1H4_2v_1sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1156                        << "Cantus firmus: ";
1157            printVector(cantusFirmus);
1158            std::cerr   << "Solution array: ";
1159            printIntVarArray(problem->getSolutionArray());
1160            std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
    endl;
1161        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1162                std::cerr   << "/!\\ ERROR test 1H4_2v_1sp: It exists solution but it shouldn't
    with the following configuration:\n"
1163                            << "cantus firmus: ";
1164                printVector(cantusFirmus);
1165                std::cerr   << "solution array: ";
1166                printIntVarArray(problem->getSolutionArray());
1167                std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
1168        }
1169        delete problem;
1170        // fix the last note of the lowest stratum
1171        problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
    specific_params, importance, borrowMode);
1172        note = cantusFirmus[cfSize-1] + i;
1173        rel(problem->getHome(), problem->getSolutionArray()[cfSize-1], IRT_EQ, note); // fix the
     last note of the lowest stratum
1174        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1175            std::cerr   << "/!\\ ERROR test 1H4_2v_1sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1176                        << "Cantus firmus: ";
1177            printVector(cantusFirmus);
1178            std::cerr   << "Solution array: ";
1179            printIntVarArray(problem->getSolutionArray());
1180            std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
    ;
1181        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
1182                std::cerr   << "/!\\ ERROR test 1H4_2v_1sp: It exists solution but it shouldn't
    with the following configuration:\n"
1183                            << "cantus firmus: ";
1184                printVector(cantusFirmus);
1185                std::cerr   << "solution array: ";
1186                printIntVarArray(problem->getSolutionArray());
1187                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
1188        }
1189        delete problem;
1190    }
1191    cout << "End test 1H4_2v_1sp..." << endl;
1192 }
1193
1194 void FuxTest::test_1H4_3v_1sp() {
1195    cout << "Start test 1H4_3v_1sp..." << endl;
1196    // Lowest stratum is the first solution array
1197    spList = {FIRST_SPECIES, FIRST_SPECIES};
1198    v_type = {-2, 3};
1199    for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
1200        // fix the first note of the lowest stratum
1201        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
1202        int note = cantusFirmus[0] + i;
1203        rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); // fix the first
    note of the lowest stratum
1204        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1205            std::cerr   << "/!\\ ERROR test 1H4_3v_1sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1206                        << "Cantus firmus: ";
1207            printVector(cantusFirmus);
1208            std::cerr   << "Solution array: ";
1209            printIntVarArray(problem->getSolutionArray());
1210            std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
    endl;
1211        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1212                std::cerr   << "/!\\ ERROR test 1H4_3v_1sp: It exists solution but it shouldn't
    with the following configuration:\n"
1213                            << "cantus firmus: ";
1214                printVector(cantusFirmus);
1215                std::cerr   << "solution array: ";
1216                printIntVarArray(problem->getSolutionArray());
1217                std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
1218        }
1219        delete problem;
1220        // fix the last note of the lowest stratum
1221        problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
    specific_params, importance, borrowMode);
1222        note = cantusFirmus[cfSize-1] + i;
1223        rel(problem->getHome(), problem->getSolutionArray()[cfSize-1], IRT_EQ, note); // fix the
     last note of the lowest stratum
```

```
1224        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1225            std::cerr   << "/!\\ ERROR test 1H4_3v_1sp: It doesn't exist a solution but it should with the following configuration:\n"
1226                        << "Cantus firmus: ";
1227            printVector(cantusFirmus);
1228            std::cerr   << "Solution array: ";
1229            printIntVarArray(problem->getSolutionArray());
1230            std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl;
1231        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
1232                std::cerr   << "/!\\ ERROR test 1H4_3v_1sp: It exists solution but it shouldn't with the following configuration:\n"
1233                            << "cantus firmus: ";
1234                printVector(cantusFirmus);
1235                std::cerr   << "solution array: ";
1236                printIntVarArray(problem->getSolutionArray());
1237                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
1238        }
1239        delete problem;
1240    }
1241    // Lowest stratum is the second solution array
1242    spList = {FIRST_SPECIES, FIRST_SPECIES};
1243    v_type = {3, -2};
1244    for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
1245        // fix the first note of the lowest stratum
1246        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
1247        int note = cantusFirmus[0] + i;
1248        rel(problem->getHome(), problem->getSolutionArray()[cfSize], IRT_EQ, note); // fix the first note of the lowest stratum
1249        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1250            std::cerr   << "/!\\ ERROR test 1H4_3v_1sp: It doesn't exist a solution but it should with the following configuration:\n"
1251                        << "Cantus firmus: ";
1252            printVector(cantusFirmus);
1253            std::cerr   << "Solution array: ";
1254            printIntVarArray(problem->getSolutionArray());
1255            std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::endl;
1256        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1257                std::cerr   << "/!\\ ERROR test 1H4_3v_1sp: It exists solution but it shouldn't with the following configuration:\n"
1258                            << "cantus firmus: ";
1259                printVector(cantusFirmus);
1260                std::cerr   << "solution array: ";
1261                printIntVarArray(problem->getSolutionArray());
1262                std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
1263        }
1264        delete problem;
1265        // fix the last note of the lowest stratum
1266        problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
1267        note = cantusFirmus[cfSize-1] + i;
1268        rel(problem->getHome(), problem->getSolutionArray()[(cfSize*2) -1], IRT_EQ, note); // fix the last note of the lowest stratum
1269        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1270            std::cerr   << "/!\\ ERROR test 1H4_3v_1sp: It doesn't exist a solution but it should with the following configuration:\n"
1271                        << "Cantus firmus: ";
1272            printVector(cantusFirmus);
1273            std::cerr   << "Solution array: ";
1274            printIntVarArray(problem->getSolutionArray());
1275            std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl;
1276        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
1277                std::cerr   << "/!\\ ERROR test 1H4_3v_1sp: It exists solution but it shouldn't with the following configuration:\n"
1278                            << "cantus firmus: ";
1279                printVector(cantusFirmus);
1280                std::cerr   << "solution array: ";
1281                printIntVarArray(problem->getSolutionArray());
1282                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
1283        }
1284        delete problem;
1285    }
1286    cout << "End test 1H4_3v_1sp." << endl;
1287 }
1288
1289 void FuxTest::test_1H4_4v_1sp() {
1290    cout << "Start test 1H4_4v_1sp..." << endl;
1291    // Lowest stratum is the first solution array
1292    spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
1293    v_type = {-2, 3, 3};
1294    for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
1295        // fix the first note of the lowest stratum
1296        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
1297        int note = cantusFirmus[0] + i;
1298        rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); // fix the first note of the lowest stratum
```

```
1299        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1300            std::cerr   << "/!\\ ERROR test 1H4_4v_1sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1301                        << "Cantus firmus: ";
1302            printVector(cantusFirmus);
1303            std::cerr   << "Solution array: ";
1304            printIntVarArray(problem->getSolutionArray());
1305            std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
    endl;
1306        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1307                std::cerr   << "/!\\ ERROR test 1H4_4v_1sp: It exists solution but it shouldn't
    with the following configuration:\n"
1308                            << "cantus firmus: ";
1309                printVector(cantusFirmus);
1310                std::cerr   << "solution array: ";
1311                printIntVarArray(problem->getSolutionArray());
1312                std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
1313        }
1314        delete problem;
1315        // fix the last note of the lowest stratum
1316        problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
    specific_params, importance, borrowMode);
1317        note = cantusFirmus[cfSize-1] + i;
1318        rel(problem->getHome(), problem->getSolutionArray()[cfSize-1], IRT_EQ, note); // fix the
     last note of the lowest stratum
1319        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1320            std::cerr   << "/!\\ ERROR test 1H4_4v_1sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1321                        << "Cantus firmus: ";
1322            printVector(cantusFirmus);
1323            std::cerr   << "Solution array: ";
1324            printIntVarArray(problem->getSolutionArray());
1325            std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
    ;
1326        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
1327                std::cerr   << "/!\\ ERROR test 1H4_4v_1sp: It exists solution but it shouldn't
    with the following configuration:\n"
1328                            << "cantus firmus: ";
1329                printVector(cantusFirmus);
1330                std::cerr   << "solution array: ";
1331                printIntVarArray(problem->getSolutionArray());
1332                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
1333        }
1334        delete problem;
1335    }
1336    // Lowest stratum is the second solution array
1337    spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
1338    v_type = {3, -2, 3};
1339    for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
1340        // fix the first note of the lowest stratum
1341        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
1342        int note = cantusFirmus[0] + i;
1343        rel(problem->getHome(), problem->getSolutionArray()[cfSize], IRT_EQ, note); // fix the
    first note of the lowest stratum
1344        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1345            std::cerr   << "/!\\ ERROR test 1H4_4v_1sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1346                        << "Cantus firmus: ";
1347            printVector(cantusFirmus);
1348            std::cerr   << "Solution array: ";
1349            printIntVarArray(problem->getSolutionArray());
1350            std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
    endl;
1351        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1352                std::cerr   << "/!\\ ERROR test 1H4_4v_1sp: It exists solution but it shouldn't
    with the following configuration:\n"
1353                            << "cantus firmus: ";
1354                printVector(cantusFirmus);
1355                std::cerr   << "solution array: ";
1356                printIntVarArray(problem->getSolutionArray());
1357                std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
1358        }
1359        delete problem;
1360        // fix the last note of the lowest stratum
1361        problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
    specific_params, importance, borrowMode);
1362        note = cantusFirmus[cfSize-1] + i;
1363        rel(problem->getHome(), problem->getSolutionArray()[(cfSize*2) -1], IRT_EQ, note); //
    fix the last note of the lowest stratum
1364        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1365            std::cerr   << "/!\\ ERROR test 1H4_4v_1sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1366                        << "Cantus firmus: ";
1367            printVector(cantusFirmus);
1368            std::cerr   << "Solution array: ";
1369            printIntVarArray(problem->getSolutionArray());
1370            std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
    ;
1371        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
```

```
1372                 std::cerr   << "/!\\ ERROR test 1H4_4v_1sp: It exists solution but it shouldn't
     with the following configuration:\n"
1373                             << "cantus firmus: ";
1374                 printVector(cantusFirmus);
1375                 std::cerr   << "solution array: ";
1376                 printIntVarArray(problem->getSolutionArray());
1377                 std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
1378             }
1379             delete problem;
1380         }
1381         // Lowest stratum is the third solution array
1382         spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
1383         v_type = {3, 3, -2};
1384         for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
1385             // fix the first note of the lowest stratum
1386             auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
1387             int note = cantusFirmus[0] + i;
1388             rel(problem->getHome(), problem->getSolutionArray()[cfSize*2], IRT_EQ, note); // fix the
      first note of the lowest stratum
1389             if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1390                 std::cerr   << "/!\\ ERROR test 1H4_4v_1sp: It doesn't exist a solution but it
     should with the following configuration:\n"
1391                             << "Cantus firmus: ";
1392                 printVector(cantusFirmus);
1393                 std::cerr   << "Solution array: ";
1394                 printIntVarArray(problem->getSolutionArray());
1395                 std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
     endl;
1396             } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1397                 std::cerr   << "/!\\ ERROR test 1H4_4v_1sp: It exists solution but it shouldn't
     with the following configuration:\n"
1398                             << "cantus firmus: ";
1399                 printVector(cantusFirmus);
1400                 std::cerr   << "solution array: ";
1401                 printIntVarArray(problem->getSolutionArray());
1402                 std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
1403             }
1404             delete problem;
1405             // fix the last note of the lowest stratum
1406             problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
     specific_params, importance, borrowMode);
1407             note = cantusFirmus[cfSize-1] + i;
1408             rel(problem->getHome(), problem->getSolutionArray()[(cfSize*3) -1], IRT_EQ, note); //
     fix the last note of the lowest stratum
1409             if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1410                 std::cerr   << "/!\\ ERROR test 1H4_4v_1sp: It doesn't exist a solution but it
     should with the following configuration:\n"
1411                             << "Cantus firmus: ";
1412                 printVector(cantusFirmus);
1413                 std::cerr   << "Solution array: ";
1414                 printIntVarArray(problem->getSolutionArray());
1415                 std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
     ;
1416             } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
1417                 std::cerr   << "/!\\ ERROR test 1H4_4v_1sp: It exists solution but it shouldn't
     with the following configuration:\n"
1418                             << "cantus firmus: ";
1419                 printVector(cantusFirmus);
1420                 std::cerr   << "solution array: ";
1421                 printIntVarArray(problem->getSolutionArray());
1422                 std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
1423             }
1424             delete problem;
1425         }
1426         cout << "End test 1H4_4v_1sp." << endl;
1427 }
1428
1429 void FuxTest::test_1H4_2v_2sp() {
1430         cout << "Start test 1H4_2v_2sp..." << endl;
1431         int cpSize = cfSize*2-1; // size of a counterpoint
1432         spList = {SECOND_SPECIES};
1433         v_type = {-2};
1434         for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
1435             // fix the first note of the lowest stratum
1436             auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
1437             int note = cantusFirmus[0] + i;
1438             rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); // fix the first
     note of the lowest stratum
1439             if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1440                 std::cerr   << "/!\\ ERROR test 1H4_2v_2sp: It doesn't exist a solution but it
     should with the following configuration:\n"
1441                             << "Cantus firmus: ";
1442                 printVector(cantusFirmus);
1443                 std::cerr   << "Solution array: ";
1444                 printIntVarArray(problem->getSolutionArray());
1445                 std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
     endl;
1446             } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
```

```
1447                 std::cerr   << "/!\\ ERROR test 1H4_2v_2sp: It exists solution but it shouldn't
     with the following configuration:\n"
1448                             << "cantus firmus: ";
1449                 printVector(cantusFirmus);
1450                 std::cerr   << "solution array: ";
1451                 printIntVarArray(problem->getSolutionArray());
1452                 std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
1453             }
1454             delete problem;
1455             // fix the last note of the lowest stratum
1456             problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
     specific_params, importance, borrowMode);
1457             note = cantusFirmus[cfSize-1] + i;
1458             rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ, note); // fix the
      last note of the lowest stratum
1459             if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1460                 std::cerr   << "/!\\ ERROR test 1H4_2v_2sp: It doesn't exist a solution but it
     should with the following configuration:\n"
1461                             << "Cantus firmus: ";
1462                 printVector(cantusFirmus);
1463                 std::cerr   << "Solution array: ";
1464                 printIntVarArray(problem->getSolutionArray());
1465                 std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
     ;
1466             } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
1467                 std::cerr   << "/!\\ ERROR test 1H4_2v_2sp: It exists solution but it shouldn't
     with the following configuration:\n"
1468                             << "cantus firmus: ";
1469                 printVector(cantusFirmus);
1470                 std::cerr   << "solution array: ";
1471                 printIntVarArray(problem->getSolutionArray());
1472                 std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
1473             }
1474             delete problem;
1475         }
1476         cout << "End test 1H4_2v_2sp..." << endl;
1477 }
1478
1479 void FuxTest::test_1H4_3v_2sp() {
1480         cout << "Start test 1H4_3v_2sp..." << endl;
1481         int cpSize = cfSize*2-1; // size of a counterpoint
1482         // Lowest stratum is the first solution array
1483         spList = {SECOND_SPECIES, SECOND_SPECIES};
1484         v_type = {-2, 3};
1485         for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
1486             // fix the first note of the lowest stratum
1487             auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
1488             int note = cantusFirmus[0] + i;
1489             rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); // fix the first
     note of the lowest stratum
1490             if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1491                 std::cerr   << "/!\\ ERROR test 1H4_3v_2sp: It doesn't exist a solution but it
     should with the following configuration:\n"
1492                             << "Cantus firmus: ";
1493                 printVector(cantusFirmus);
1494                 std::cerr   << "Solution array: ";
1495                 printIntVarArray(problem->getSolutionArray());
1496                 std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
     endl;
1497             } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1498                 std::cerr   << "/!\\ ERROR test 1H4_3v_2sp: It exists solution but it shouldn't
     with the following configuration:\n"
1499                             << "cantus firmus: ";
1500                 printVector(cantusFirmus);
1501                 std::cerr   << "solution array: ";
1502                 printIntVarArray(problem->getSolutionArray());
1503                 std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
1504             }
1505             delete problem;
1506             // fix the last note of the lowest stratum
1507             problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
     specific_params, importance, borrowMode);
1508             note = cantusFirmus[cfSize-1] + i;
1509             rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ, note); // fix the
      last note of the lowest stratum
1510             if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1511                 std::cerr   << "/!\\ ERROR test 1H4_3v_2sp: It doesn't exist a solution but it
     should with the following configuration:\n"
1512                             << "Cantus firmus: ";
1513                 printVector(cantusFirmus);
1514                 std::cerr   << "Solution array: ";
1515                 printIntVarArray(problem->getSolutionArray());
1516                 std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
     ;
1517             } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
1518                 std::cerr   << "/!\\ ERROR test 1H4_3v_2sp: It exists solution but it shouldn't
     with the following configuration:\n"
1519                             << "cantus firmus: ";
1520                 printVector(cantusFirmus);
```

```
1521                std::cerr   << "solution array: ";
1522                printIntVarArray(problem->getSolutionArray());
1523                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
1524            }
1525            delete problem;
1526        }
1527        // Lowest stratum is the second solution array
1528        spList = {SECOND_SPECIES, SECOND_SPECIES};
1529        v_type = {3, -2};
1530        for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
1531            // fix the first note of the lowest stratum
1532            auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
1533            int note = cantusFirmus[0] + i;
1534            rel(problem->getHome(), problem->getSolutionArray()[cpSize], IRT_EQ, note); // fix the
    first note of the lowest stratum
1535            if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1536                std::cerr   << "/!\\ ERROR test 1H4_3v_2sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1537                            << "Cantus firmus: ";
1538                printVector(cantusFirmus);
1539                std::cerr   << "Solution array: ";
1540                printIntVarArray(problem->getSolutionArray());
1541                std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
    endl;
1542            } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1543                std::cerr   << "/!\\ ERROR test 1H4_3v_2sp: It exists solution but it shouldn't
    with the following configuration:\n"
1544                            << "cantus firmus: ";
1545                printVector(cantusFirmus);
1546                std::cerr   << "solution array: ";
1547                printIntVarArray(problem->getSolutionArray());
1548                std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
1549            }
1550            delete problem;
1551            // fix the last note of the lowest stratum
1552            problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
    specific_params, importance, borrowMode);
1553            note = cantusFirmus[cfSize-1] + i;
1554            rel(problem->getHome(), problem->getSolutionArray()[cpSize*2-1], IRT_EQ, note); // fix
    the last note of the lowest stratum
1555            if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1556                std::cerr   << "/!\\ ERROR test 1H4_3v_2sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1557                            << "Cantus firmus: ";
1558                printVector(cantusFirmus);
1559                std::cerr   << "Solution array: ";
1560                printIntVarArray(problem->getSolutionArray());
1561                std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
    ;
1562            } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
1563                std::cerr   << "/!\\ ERROR test 1H4_3v_2sp: It exists solution but it shouldn't
    with the following configuration:\n"
1564                            << "cantus firmus: ";
1565                printVector(cantusFirmus);
1566                std::cerr   << "solution array: ";
1567                printIntVarArray(problem->getSolutionArray());
1568                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
1569            }
1570            delete problem;
1571        }
1572        cout << "End test 1H4_3v_2sp." << endl;
1573    }
1574
1575    void FuxTest::test_1H4_4v_2sp() {
1576        cout << "Start test 1H4_4v_2sp..." << endl;
1577        int cpSize = cfSize*2-1; // size of a counterpoint
1578        // Lowest stratum is the first solution array
1579        spList = {SECOND_SPECIES, SECOND_SPECIES, SECOND_SPECIES};
1580        v_type = {-2, 3, 3};
1581        for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
1582            // fix the first note of the lowest stratum
1583            auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
1584            int note = cantusFirmus[0] + i;
1585            rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); // fix the first
    note of the lowest stratum
1586            if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1587                std::cerr   << "/!\\ ERROR test 1H4_4v_2sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1588                            << "Cantus firmus: ";
1589                printVector(cantusFirmus);
1590                std::cerr   << "Solution array: ";
1591                printIntVarArray(problem->getSolutionArray());
1592                std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
    endl;
1593            } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1594                std::cerr   << "/!\\ ERROR test 1H4_4v_2sp: It exists solution but it shouldn't
    with the following configuration:\n"
1595                            << "cantus firmus: ";
```

```
1596                printVector(cantusFirmus);
1597                std::cerr   << "solution array: ";
1598                printIntVarArray(problem->getSolutionArray());
1599                std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
1600            }
1601            delete problem;
1602            // fix the last note of the lowest stratum
1603            problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
    specific_params, importance, borrowMode);
1604            note = cantusFirmus[cfSize-1] + i;
1605            rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ, note); // fix the
     last note of the lowest stratum
1606            if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1607                std::cerr   << "/!\\ ERROR test 1H4_4v_2sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1608                            << "Cantus firmus: ";
1609                printVector(cantusFirmus);
1610                std::cerr   << "Solution array: ";
1611                printIntVarArray(problem->getSolutionArray());
1612                std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
    ;
1613            } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
1614                std::cerr   << "/!\\ ERROR test 1H4_4v_2sp: It exists solution but it shouldn't
    with the following configuration:\n"
1615                            << "cantus firmus: ";
1616                printVector(cantusFirmus);
1617                std::cerr   << "solution array: ";
1618                printIntVarArray(problem->getSolutionArray());
1619                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
1620            }
1621            delete problem;
1622        }
1623        // Lowest stratum is the second solution array
1624        spList = {SECOND_SPECIES, SECOND_SPECIES, SECOND_SPECIES};
1625        v_type = {3, -2, 3};
1626        for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
1627            // fix the first note of the lowest stratum
1628            auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
1629            int note = cantusFirmus[0] + i;
1630            rel(problem->getHome(), problem->getSolutionArray()[cpSize], IRT_EQ, note); // fix the
    first note of the lowest stratum
1631            if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1632                std::cerr   << "/!\\ ERROR test 1H4_4v_2sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1633                            << "Cantus firmus: ";
1634                printVector(cantusFirmus);
1635                std::cerr   << "Solution array: ";
1636                printIntVarArray(problem->getSolutionArray());
1637                std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
    endl;
1638            } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1639                std::cerr   << "/!\\ ERROR test 1H4_4v_2sp: It exists solution but it shouldn't
    with the following configuration:\n"
1640                            << "cantus firmus: ";
1641                printVector(cantusFirmus);
1642                std::cerr   << "solution array: ";
1643                printIntVarArray(problem->getSolutionArray());
1644                std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
1645            }
1646            delete problem;
1647            // fix the last note of the lowest stratum
1648            problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
    specific_params, importance, borrowMode);
1649            note = cantusFirmus[cfSize-1] + i;
1650            rel(problem->getHome(), problem->getSolutionArray()[cpSize*2 -1], IRT_EQ, note); // fix
    the last note of the lowest stratum
1651            if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1652                std::cerr   << "/!\\ ERROR test 1H4_4v_2sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1653                            << "Cantus firmus: ";
1654                printVector(cantusFirmus);
1655                std::cerr   << "Solution array: ";
1656                printIntVarArray(problem->getSolutionArray());
1657                std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
    ;
1658            } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
1659                std::cerr   << "/!\\ ERROR test 1H4_4v_2sp: It exists solution but it shouldn't
    with the following configuration:\n"
1660                            << "cantus firmus: ";
1661                printVector(cantusFirmus);
1662                std::cerr   << "solution array: ";
1663                printIntVarArray(problem->getSolutionArray());
1664                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
1665            }
1666            delete problem;
1667        }
1668        // Lowest stratum is the third solution array
1669        spList = {SECOND_SPECIES, SECOND_SPECIES, SECOND_SPECIES};
1670        v_type = {3, 3, -2};
```

```
1671     for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
1672         // fix the first note of the lowest stratum
1673         auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
1674         int note = cantusFirmus[0] + i;
1675         rel(problem->getHome(), problem->getSolutionArray()[cpSize*2], IRT_EQ, note); // fix the
      first note of the lowest stratum
1676         if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1677             std::cerr   << "/!\\ ERROR test 1H4_4v_2sp: It doesn't exist a solution but it
     should with the following configuration:\n"
1678                         << "Cantus firmus: ";
1679             printVector(cantusFirmus);
1680             std::cerr   << "Solution array: ";
1681             printIntVarArray(problem->getSolutionArray());
1682             std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
     endl;
1683         } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1684             std::cerr   << "/!\\ ERROR test 1H4_4v_2sp: It exists solution but it shouldn't
     with the following configuration:\n"
1685                         << "cantus firmus: ";
1686             printVector(cantusFirmus);
1687             std::cerr   << "solution array: ";
1688             printIntVarArray(problem->getSolutionArray());
1689             std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
1690         }
1691         delete problem;
1692         // fix the last note of the lowest stratum
1693         problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
     specific_params, importance, borrowMode);
1694         note = cantusFirmus[cfSize-1] + i;
1695         rel(problem->getHome(), problem->getSolutionArray()[(cpSize*3) -1], IRT_EQ, note); //
     fix the last note of the lowest stratum
1696         if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1697             std::cerr   << "/!\\ ERROR test 1H4_4v_2sp: It doesn't exist a solution but it
     should with the following configuration:\n"
1698                         << "Cantus firmus: ";
1699             printVector(cantusFirmus);
1700             std::cerr   << "Solution array: ";
1701             printIntVarArray(problem->getSolutionArray());
1702             std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
     ;
1703         } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
1704             std::cerr   << "/!\\ ERROR test 1H4_4v_2sp: It exists solution but it shouldn't
     with the following configuration:\n"
1705                         << "cantus firmus: ";
1706             printVector(cantusFirmus);
1707             std::cerr   << "solution array: ";
1708             printIntVarArray(problem->getSolutionArray());
1709             std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
1710         }
1711         delete problem;
1712     }
1713     cout << "End test 1H4_4v_2sp." << endl;
1714 }
1715
1716 void FuxTest::test_1H4_2v_3sp() {
1717     cout << "Start test 1H4_2v_3sp..." << endl;
1718     int cpSize = cfSize*4-3; // size of a counterpoint
1719     spList = {THIRD_SPECIES};
1720     v_type = {-2};
1721     for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
1722         // fix the first note of the lowest stratum
1723         auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
1724         int note = cantusFirmus[0] + i;
1725         rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); // fix the first
     note of the lowest stratum
1726         if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1727             std::cerr   << "/!\\ ERROR test 1H4_2v_3sp: It doesn't exist a solution but it
     should with the following configuration:\n"
1728                         << "Cantus firmus: ";
1729             printVector(cantusFirmus);
1730             std::cerr   << "Solution array: ";
1731             printIntVarArray(problem->getSolutionArray());
1732             std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
     endl;
1733         } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1734             std::cerr   << "/!\\ ERROR test 1H4_2v_3sp: It exists solution but it shouldn't
     with the following configuration:\n"
1735                         << "cantus firmus: ";
1736             printVector(cantusFirmus);
1737             std::cerr   << "solution array: ";
1738             printIntVarArray(problem->getSolutionArray());
1739             std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
1740         }
1741         delete problem;
1742         // fix the last note of the lowest stratum
1743         problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
     specific_params, importance, borrowMode);
1744         note = cantusFirmus[cfSize-1] + i;
```

```
1745        rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ, note); // fix the
     last note of the lowest stratum
1746        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1747            std::cerr   << "/!\\ ERROR test 1H4_2v_3sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1748                        << "Cantus firmus: ";
1749            printVector(cantusFirmus);
1750            std::cerr   << "Solution array: ";
1751            printIntVarArray(problem->getSolutionArray());
1752            std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
    ;
1753        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
1754                std::cerr   << "/!\\ ERROR test 1H4_2v_3sp: It exists solution but it shouldn't
    with the following configuration:\n"
1755                            << "cantus firmus: ";
1756                printVector(cantusFirmus);
1757                std::cerr   << "solution array: ";
1758                printIntVarArray(problem->getSolutionArray());
1759                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
1760        }
1761        delete problem;
1762    }
1763    cout << "End test 1H4_2v_3sp..." << endl;
1764 }
1765
1766 void FuxTest::test_1H4_3v_3sp() {
1767    cout << "Start test 1H4_3v_3sp..." << endl;
1768    int cpSize = cfSize*4-3; // size of a counterpoint
1769    // Lowest stratum is the first solution array
1770    spList = {THIRD_SPECIES, THIRD_SPECIES};
1771    v_type = {-2, 3};
1772    for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
1773        // fix the first note of the lowest stratum
1774        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
1775        int note = cantusFirmus[0] + i;
1776        rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); // fix the first
    note of the lowest stratum
1777        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1778            std::cerr   << "/!\\ ERROR test 1H4_3v_3sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1779                        << "Cantus firmus: ";
1780            printVector(cantusFirmus);
1781            std::cerr   << "Solution array: ";
1782            printIntVarArray(problem->getSolutionArray());
1783            std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
    endl;
1784        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1785                std::cerr   << "/!\\ ERROR test 1H4_3v_3sp: It exists solution but it shouldn't
    with the following configuration:\n"
1786                            << "cantus firmus: ";
1787                printVector(cantusFirmus);
1788                std::cerr   << "solution array: ";
1789                printIntVarArray(problem->getSolutionArray());
1790                std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
1791        }
1792        delete problem;
1793        // fix the last note of the lowest stratum
1794        problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
    specific_params, importance, borrowMode);
1795        note = cantusFirmus[cfSize-1] + i;
1796        rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ, note); // fix the
     last note of the lowest stratum
1797        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1798            std::cerr   << "/!\\ ERROR test 1H4_3v_3sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1799                        << "Cantus firmus: ";
1800            printVector(cantusFirmus);
1801            std::cerr   << "Solution array: ";
1802            printIntVarArray(problem->getSolutionArray());
1803            std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
    ;
1804        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
1805                std::cerr   << "/!\\ ERROR test 1H4_3v_3sp: It exists solution but it shouldn't
    with the following configuration:\n"
1806                            << "cantus firmus: ";
1807                printVector(cantusFirmus);
1808                std::cerr   << "solution array: ";
1809                printIntVarArray(problem->getSolutionArray());
1810                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
1811        }
1812        delete problem;
1813    }
1814    // Lowest stratum is the second solution array
1815    spList = {THIRD_SPECIES, THIRD_SPECIES};
1816    v_type = {3, -2};
1817    for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
1818        // fix the first note of the lowest stratum
1819        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
```

```
1820        int note = cantusFirmus[0] + i;
1821        rel(problem->getHome(), problem->getSolutionArray()[cpSize], IRT_EQ, note); // fix the
    first note of the lowest stratum
1822        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1823            std::cerr   << "/!\\ ERROR test 1H4_3v_3sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1824                        << "Cantus firmus: ";
1825            printVector(cantusFirmus);
1826            std::cerr   << "Solution array: ";
1827            printIntVarArray(problem->getSolutionArray());
1828            std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
    endl;
1829        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1830                std::cerr   << "/!\\ ERROR test 1H4_3v_3sp: It exists solution but it shouldn't
    with the following configuration:\n"
1831                            << "cantus firmus: ";
1832                printVector(cantusFirmus);
1833                std::cerr   << "solution array: ";
1834                printIntVarArray(problem->getSolutionArray());
1835                std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
1836        }
1837        delete problem;
1838        // fix the last note of the lowest stratum
1839        problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
    specific_params, importance, borrowMode);
1840        note = cantusFirmus[cfSize-1] + i;
1841        rel(problem->getHome(), problem->getSolutionArray()[cpSize*2-1], IRT_EQ, note); // fix
    the last note of the lowest stratum
1842        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1843            std::cerr   << "/!\\ ERROR test 1H4_3v_3sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1844                        << "Cantus firmus: ";
1845            printVector(cantusFirmus);
1846            std::cerr   << "Solution array: ";
1847            printIntVarArray(problem->getSolutionArray());
1848            std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
    ;
1849        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
1850                std::cerr   << "/!\\ ERROR test 1H4_3v_3sp: It exists solution but it shouldn't
    with the following configuration:\n"
1851                            << "cantus firmus: ";
1852                printVector(cantusFirmus);
1853                std::cerr   << "solution array: ";
1854                printIntVarArray(problem->getSolutionArray());
1855                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
1856        }
1857        delete problem;
1858    }
1859    cout << "End test 1H4_3v_3sp." << endl;
1860 }
1861
1862 void FuxTest::test_1H4_4v_3sp() {
1863    cout << "Start test 1H4_4v_3sp..." << endl;
1864    int cpSize = cfSize*4-3; // size of a counterpoint
1865    // Lowest stratum is the first solution array
1866    spList = {THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES};
1867    v_type = {-2, 3, 3};
1868    for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
1869        // fix the first note of the lowest stratum
1870        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
1871        int note = cantusFirmus[0] + i;
1872        rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); // fix the first
    note of the lowest stratum
1873        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1874            std::cerr   << "/!\\ ERROR test 1H4_4v_3sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1875                        << "Cantus firmus: ";
1876            printVector(cantusFirmus);
1877            std::cerr   << "Solution array: ";
1878            printIntVarArray(problem->getSolutionArray());
1879            std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
    endl;
1880        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1881                std::cerr   << "/!\\ ERROR test 1H4_4v_3sp: It exists solution but it shouldn't
    with the following configuration:\n"
1882                            << "cantus firmus: ";
1883                printVector(cantusFirmus);
1884                std::cerr   << "solution array: ";
1885                printIntVarArray(problem->getSolutionArray());
1886                std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
1887        }
1888        delete problem;
1889        // fix the last note of the lowest stratum
1890        problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
    specific_params, importance, borrowMode);
1891        note = cantusFirmus[cfSize-1] + i;
1892        rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ, note); // fix the
     last note of the lowest stratum
1893        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
```

```
1894            std::cerr   << "/!\\ ERROR test 1H4_4v_2sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1895                        << "Cantus firmus: ";
1896            printVector(cantusFirmus);
1897            std::cerr   << "Solution array: ";
1898            printIntVarArray(problem->getSolutionArray());
1899            std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
    ;
1900        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
1901                std::cerr   << "/!\\ ERROR test 1H4_4v_3sp: It exists solution but it shouldn't
    with the following configuration:\n"
1902                            << "cantus firmus: ";
1903                printVector(cantusFirmus);
1904                std::cerr   << "solution array: ";
1905                printIntVarArray(problem->getSolutionArray());
1906                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
1907        }
1908        delete problem;
1909    }
1910    // Lowest stratum is the second solution array
1911    spList = {THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES};
1912    v_type = {3, -2, 3};
1913    for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
1914        // fix the first note of the lowest stratum
1915        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
1916        int note = cantusFirmus[0] + i;
1917        rel(problem->getHome(), problem->getSolutionArray()[cpSize], IRT_EQ, note); // fix the
    first note of the lowest stratum
1918        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1919            std::cerr   << "/!\\ ERROR test 1H4_4v_3sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1920                        << "Cantus firmus: ";
1921            printVector(cantusFirmus);
1922            std::cerr   << "Solution array: ";
1923            printIntVarArray(problem->getSolutionArray());
1924            std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
    endl;
1925        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1926                std::cerr   << "/!\\ ERROR test 1H4_4v_3sp: It exists solution but it shouldn't
    with the following configuration:\n"
1927                            << "cantus firmus: ";
1928                printVector(cantusFirmus);
1929                std::cerr   << "solution array: ";
1930                printIntVarArray(problem->getSolutionArray());
1931                std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
1932        }
1933        delete problem;
1934        // fix the last note of the lowest stratum
1935        problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
    specific_params, importance, borrowMode);
1936        note = cantusFirmus[cfSize-1] + i;
1937        rel(problem->getHome(), problem->getSolutionArray()[cpSize*2 -1], IRT_EQ, note); // fix
    the last note of the lowest stratum
1938        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1939            std::cerr   << "/!\\ ERROR test 1H4_4v_3sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1940                        << "Cantus firmus: ";
1941            printVector(cantusFirmus);
1942            std::cerr   << "Solution array: ";
1943            printIntVarArray(problem->getSolutionArray());
1944            std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
    ;
1945        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
1946                std::cerr   << "/!\\ ERROR test 1H4_4v_3sp: It exists solution but it shouldn't
    with the following configuration:\n"
1947                            << "cantus firmus: ";
1948                printVector(cantusFirmus);
1949                std::cerr   << "solution array: ";
1950                printIntVarArray(problem->getSolutionArray());
1951                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
1952        }
1953        delete problem;
1954    }
1955    // Lowest stratum is the third solution array
1956    spList = {THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES};
1957    v_type = {3, 3, -2};
1958    for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
1959        // fix the first note of the lowest stratum
1960        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
1961        int note = cantusFirmus[0] + i;
1962        rel(problem->getHome(), problem->getSolutionArray()[cpSize*2], IRT_EQ, note); // fix the
     first note of the lowest stratum
1963        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1964            std::cerr   << "/!\\ ERROR test 1H4_4v_3sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1965                        << "Cantus firmus: ";
1966            printVector(cantusFirmus);
1967            std::cerr   << "Solution array: ";
```

```
1968            printIntVarArray(problem->getSolutionArray());
1969            std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
    endl;
1970        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1971                std::cerr   << "/!\\ ERROR test 1H4_4v_3sp: It exists solution but it shouldn't
    with the following configuration:\n"
1972                            << "cantus firmus: ";
1973                printVector(cantusFirmus);
1974                std::cerr   << "solution array: ";
1975                printIntVarArray(problem->getSolutionArray());
1976                std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
1977        }
1978        delete problem;
1979        // fix the last note of the lowest stratum
1980        problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
    specific_params, importance, borrowMode);
1981        note = cantusFirmus[cfSize-1] + i;
1982        rel(problem->getHome(), problem->getSolutionArray()[(cpSize*3) -1], IRT_EQ, note); //
    fix the last note of the lowest stratum
1983        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1984            std::cerr   << "/!\\ ERROR test 1H4_4v_3sp: It doesn't exist a solution but it
    should with the following configuration:\n"
1985                        << "Cantus firmus: ";
1986            printVector(cantusFirmus);
1987            std::cerr   << "Solution array: ";
1988            printIntVarArray(problem->getSolutionArray());
1989            std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
    ;
1990        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
1991                std::cerr   << "/!\\ ERROR test 1H4_4v_3sp: It exists solution but it shouldn't
    with the following configuration:\n"
1992                            << "cantus firmus: ";
1993                printVector(cantusFirmus);
1994                std::cerr   << "solution array: ";
1995                printIntVarArray(problem->getSolutionArray());
1996                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
1997        }
1998        delete problem;
1999    }
2000    cout << "End test 1H4_4v_3sp." << endl;
2001 }
2002
2003 void FuxTest::test_1H4_2v_4sp() {
2004    cout << "Start test 1H4_2v_4sp..." << endl;
2005    int cpSize = cfSize*2-2; // size of a counterpoint
2006    spList = {FOURTH_SPECIES};
2007    v_type = {-2};
2008    for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
2009        // fix the first note of the lowest stratum
2010        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
2011        int note = cantusFirmus[0] + i;
2012        rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); // fix the first
    note of the lowest stratum
2013        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
2014            std::cerr   << "/!\\ ERROR test 1H4_2v_4sp: It doesn't exist a solution but it
    should with the following configuration:\n"
2015                        << "Cantus firmus: ";
2016            printVector(cantusFirmus);
2017            std::cerr   << "Solution array: ";
2018            printIntVarArray(problem->getSolutionArray());
2019            std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
    endl;
2020        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
2021                std::cerr   << "/!\\ ERROR test 1H4_2v_4sp: It exists solution but it shouldn't
    with the following configuration:\n"
2022                            << "cantus firmus: ";
2023                printVector(cantusFirmus);
2024                std::cerr   << "solution array: ";
2025                printIntVarArray(problem->getSolutionArray());
2026                std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
2027        }
2028        delete problem;
2029        // fix the last note of the lowest stratum
2030        problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
    specific_params, importance, borrowMode);
2031        note = cantusFirmus[cfSize-1] + i;
2032        rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ, note); // fix the
     last note of the lowest stratum
2033        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
2034            std::cerr   << "/!\\ ERROR test 1H4_2v_4sp: It doesn't exist a solution but it
    should with the following configuration:\n"
2035                        << "Cantus firmus: ";
2036            printVector(cantusFirmus);
2037            std::cerr   << "Solution array: ";
2038            printIntVarArray(problem->getSolutionArray());
2039            std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
    ;
2040        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
2041                std::cerr   << "/!\\ ERROR test 1H4_2v_4sp: It exists solution but it shouldn't
```

```
    with the following configuration:\n"
                            << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
        }
        delete problem;
    }
    cout << "End test 1H4_2v_4sp..." << endl;
}

void FuxTest::test_1H4_3v_4sp() {
    cout << "Start test 1H4_3v_4sp..." << endl;
    int cpSize = cfSize*2-2; // size of a counterpoint
    // Lowest stratum is the first solution array
    spList = {FOURTH_SPECIES, FOURTH_SPECIES};
    v_type = {-2, 3};
    for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
        // fix the first note of the lowest stratum
        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
        int note = cantusFirmus[0] + i;
        rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); // fix the first
    note of the lowest stratum
        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_3v_4sp: It doesn't exist a solution but it
    should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
    endl;
        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test 1H4_3v_4sp: It exists solution but it shouldn't
    with the following configuration:\n"
                            << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
        }
        delete problem;
        // fix the last note of the lowest stratum
        problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
    specific_params, importance, borrowMode);
        note = cantusFirmus[cfSize-1] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ, note); // fix the
     last note of the lowest stratum
        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_3v_4sp: It doesn't exist a solution but it
    should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
    ;
        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test 1H4_3v_4sp: It exists solution but it shouldn't
    with the following configuration:\n"
                            << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
        }
        delete problem;
    }
    // Lowest stratum is the second solution array
    spList = {FOURTH_SPECIES, FOURTH_SPECIES};
    v_type = {3, -2};
    for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
        // fix the first note of the lowest stratum
        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
        int note = cantusFirmus[0] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize], IRT_EQ, note); // fix the
    first note of the lowest stratum
        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_3v_4sp: It doesn't exist a solution but it
    should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
    endl;
        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
```

```
2117                 std::cerr   << "/!\\ ERROR test 1H4_3v_4sp: It exists solution but it shouldn't
     with the following configuration:\n"
2118                             << "cantus firmus: ";
2119                 printVector(cantusFirmus);
2120                 std::cerr   << "solution array: ";
2121                 printIntVarArray(problem->getSolutionArray());
2122                 std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
2123             }
2124             delete problem;
2125             // fix the last note of the lowest stratum
2126             problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
     specific_params, importance, borrowMode);
2127             note = cantusFirmus[cfSize-1] + i;
2128             rel(problem->getHome(), problem->getSolutionArray()[cpSize*2-1], IRT_EQ, note); // fix
     the last note of the lowest stratum
2129             if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
2130                 std::cerr   << "/!\\ ERROR test 1H4_3v_4sp: It doesn't exist a solution but it
     should with the following configuration:\n"
2131                             << "Cantus firmus: ";
2132                 printVector(cantusFirmus);
2133                 std::cerr   << "Solution array: ";
2134                 printIntVarArray(problem->getSolutionArray());
2135                 std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
     ;
2136             } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
2137                 std::cerr   << "/!\\ ERROR test 1H4_3v_4sp: It exists solution but it shouldn't
     with the following configuration:\n"
2138                             << "cantus firmus: ";
2139                 printVector(cantusFirmus);
2140                 std::cerr   << "solution array: ";
2141                 printIntVarArray(problem->getSolutionArray());
2142                 std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
2143             }
2144             delete problem;
2145         }
2146         cout << "End test 1H4_3v_4sp." << endl;
2147 }
2148
2149 void FuxTest::test_1H4_4v_4sp() {
2150     cout << "Start test 1H4_4v_4sp..." << endl;
2151     int cpSize = cfSize*2-2; // size of a counterpoint
2152     // Lowest stratum is the first solution array
2153     spList = {FOURTH_SPECIES, FOURTH_SPECIES, FOURTH_SPECIES};
2154     v_type = {-2, 3, 3};
2155     for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
2156         // fix the first note of the lowest stratum
2157         auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
2158         int note = cantusFirmus[0] + i;
2159         rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); // fix the first
     note of the lowest stratum
2160         if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
2161             std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It doesn't exist a solution but it
     should with the following configuration:\n"
2162                         << "Cantus firmus: ";
2163             printVector(cantusFirmus);
2164             std::cerr   << "Solution array: ";
2165             printIntVarArray(problem->getSolutionArray());
2166             std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
     endl;
2167         } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
2168             std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It exists solution but it shouldn't
     with the following configuration:\n"
2169                         << "cantus firmus: ";
2170             printVector(cantusFirmus);
2171             std::cerr   << "solution array: ";
2172             printIntVarArray(problem->getSolutionArray());
2173             std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
2174         }
2175         delete problem;
2176         // fix the last note of the lowest stratum
2177         problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
     specific_params, importance, borrowMode);
2178         note = cantusFirmus[cfSize-1] + i;
2179         rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ, note); // fix the
      last note of the lowest stratum
2180         if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
2181             std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It doesn't exist a solution but it
     should with the following configuration:\n"
2182                         << "Cantus firmus: ";
2183             printVector(cantusFirmus);
2184             std::cerr   << "Solution array: ";
2185             printIntVarArray(problem->getSolutionArray());
2186             std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
     ;
2187         } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
2188             std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It exists solution but it shouldn't
     with the following configuration:\n"
2189                         << "cantus firmus: ";
2190             printVector(cantusFirmus);
```

```
2191                std::cerr   << "solution array: ";
2192                printIntVarArray(problem->getSolutionArray());
2193                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
2194            }
2195            delete problem;
2196        }
2197        // Lowest stratum is the second solution array
2198        spList = {FOURTH_SPECIES, FOURTH_SPECIES, FOURTH_SPECIES};
2199        v_type = {3, -2, 3};
2200        for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
2201            // fix the first note of the lowest stratum
2202            auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
2203            int note = cantusFirmus[0] + i;
2204            rel(problem->getHome(), problem->getSolutionArray()[cpSize], IRT_EQ, note); // fix the
    first note of the lowest stratum
2205            if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
2206                std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It doesn't exist a solution but it
    should with the following configuration:\n"
2207                            << "Cantus firmus: ";
2208                printVector(cantusFirmus);
2209                std::cerr   << "Solution array: ";
2210                printIntVarArray(problem->getSolutionArray());
2211                std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
    endl;
2212            } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
2213                std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It exists solution but it shouldn't
    with the following configuration:\n"
2214                            << "cantus firmus: ";
2215                printVector(cantusFirmus);
2216                std::cerr   << "solution array: ";
2217                printIntVarArray(problem->getSolutionArray());
2218                std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
2219            }
2220            delete problem;
2221            // fix the last note of the lowest stratum
2222            problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
    specific_params, importance, borrowMode);
2223            note = cantusFirmus[cfSize-1] + i;
2224            rel(problem->getHome(), problem->getSolutionArray()[cpSize*2 -1], IRT_EQ, note); // fix
    the last note of the lowest stratum
2225            if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
2226                std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It doesn't exist a solution but it
    should with the following configuration:\n"
2227                            << "Cantus firmus: ";
2228                printVector(cantusFirmus);
2229                std::cerr   << "Solution array: ";
2230                printIntVarArray(problem->getSolutionArray());
2231                std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
    ;
2232            } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
2233                std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It exists solution but it shouldn't
    with the following configuration:\n"
2234                            << "cantus firmus: ";
2235                printVector(cantusFirmus);
2236                std::cerr   << "solution array: ";
2237                printIntVarArray(problem->getSolutionArray());
2238                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
2239            }
2240            delete problem;
2241        }
2242        // Lowest stratum is the third solution array
2243        spList = {FOURTH_SPECIES, FOURTH_SPECIES, FOURTH_SPECIES};
2244        v_type = {3, 3, -2};
2245        for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
2246            // fix the first note of the lowest stratum
2247            auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
2248            int note = cantusFirmus[0] + i;
2249            rel(problem->getHome(), problem->getSolutionArray()[cpSize*2], IRT_EQ, note); // fix the
     first note of the lowest stratum
2250            if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
2251                std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It doesn't exist a solution but it
    should with the following configuration:\n"
2252                            << "Cantus firmus: ";
2253                printVector(cantusFirmus);
2254                std::cerr   << "Solution array: ";
2255                printIntVarArray(problem->getSolutionArray());
2256                std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::
    endl;
2257            } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
2258                std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It exists solution but it shouldn't
    with the following configuration:\n"
2259                            << "cantus firmus: ";
2260                printVector(cantusFirmus);
2261                std::cerr   << "solution array: ";
2262                printIntVarArray(problem->getSolutionArray());
2263                std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
2264            }
2265            delete problem;
```

```
2266        // fix the last note of the lowest stratum
2267        problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
    specific_params, importance, borrowMode);
2268        note = cantusFirmus[cfSize-1] + i;
2269        rel(problem->getHome(), problem->getSolutionArray()[(cpSize*3) -1], IRT_EQ, note); //
    fix the last note of the lowest stratum
2270        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
2271            std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It doesn't exist a solution but it
    should with the following configuration:\n"
2272                        << "Cantus firmus: ";
2273            printVector(cantusFirmus);
2274            std::cerr   << "Solution array: ";
2275            printIntVarArray(problem->getSolutionArray());
2276            std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl
    ;
2277        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
2278                std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It exists solution but it shouldn't
    with the following configuration:\n"
2279                            << "cantus firmus: ";
2280                printVector(cantusFirmus);
2281                std::cerr   << "solution array: ";
2282                printIntVarArray(problem->getSolutionArray());
2283                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
2284        }
2285        delete problem;
2286    }
2287    cout << "End test 1H4_4v_4sp." << endl;
2288 }
2289
2290 /*
2291 The voices cannot play the same note at the same time except in the first and last measure.
2292 */
2293 void FuxTest::test_1H5() {
2294    cout << "Start tests 1H5..." << endl;
2295    test_1H5_2v_1sp();
2296    test_1H5_3v_1sp();
2297    test_1H5_2v_2sp();
2298    test_1H5_3v_2sp();
2299    test_1H5_2v_3sp();
2300    test_1H5_3v_3sp();
2301    test_1H5_2v_4sp();
2302    test_1H5_3v_4sp();
2303    cout << "End tests 1H5." << endl;
2304 }
2305
2306 void FuxTest::test_1H5_2v_1sp() {
2307    cout << "Start test_1H5_2v_1sp ..." << endl;
2308    spList = {FIRST_SPECIES};
2309    v_type = {0};
2310    for (size_t i = 1; i < cfSize-1; i++) {
2311        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
2312        rel(problem->getHome(), problem->getSolutionArray()[i], IRT_EQ, cantusFirmus[i]); // fix
     the solution note as the same note as the cf
2313        if (has_solution(problem)) {
2314            std::cerr   << "/!\\ ERROR test 1H5_2v_1sp: It exists solution but it shouldn't with
     the following configuration:\n"
2315                        << "cantus firmus: ";
2316            printVector(cantusFirmus);
2317            std::cerr   << "solution array: ";
2318            printIntVarArray(problem->getSolutionArray());
2319            std::cerr   << "> Same note played at the mesure: " << i << std::endl;
2320        }
2321        delete problem;
2322    }
2323    cout << "End test 1H5_2v_1sp" << endl;
2324 }
2325
2326 void FuxTest::test_1H5_3v_1sp() {
2327    cout << "Start test 1H5_3v_1sp..." << endl;
2328    spList = {FIRST_SPECIES, FIRST_SPECIES};
2329    v_type = {0, 0};
2330    // // test solution lines != cf
2331    for (size_t j = 0; j < 2; j++) { // iterate over each solution line
2332        for (size_t i = 1; i < cfSize-1; i++) {
2333            auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
2334            rel(problem->getHome(), problem->getSolutionArray()[(j*cfSize) + i], IRT_EQ,
    cantusFirmus[i]); // fix the solution note as the same note as the cf
2335            if (has_solution(problem)) {
2336                std::cerr   << "/!\\ ERROR test 1H5_3v_1sp: It exists solution but it shouldn't
    with the following configuration:\n"
2337                            << "cantus firmus: ";
2338                printVector(cantusFirmus);
2339                std::cerr   << "solution array: ";
2340                printIntVarArray(problem->getSolutionArray());
2341                std::cerr   << "> Same note played at the mesure: " << i << std::endl;
2342            }
2343            delete problem;
2344        }
```

```
2345     }
2346     // test notes of solutions lines are different
2347     for (size_t i = 1; i < cfSize-1; i++) {
2348         for (int j = -6; j < 13; j++) {
2349             auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
2350             int note = cantusFirmus[0] + j;
2351             rel(problem->getHome(), problem->getSolutionArray()[i], IRT_EQ, note); // fix the
     note of the first solution line as the same note as the note of the second solution line
2352             rel(problem->getHome(), problem->getSolutionArray()[cfSize+i], IRT_EQ, note); // fix
      the note of the first solution line as the same note as the note of the second solution
     line
2353             if (has_solution(problem)) {
2354                 std::cerr   << "/!\\ ERROR test 1H5_3v_1sp: It exists solution but it shouldn't
     with the following configuration:\n"
2355                             << "cantus firmus: ";
2356                 printVector(cantusFirmus);
2357                 std::cerr   << "solution array: ";
2358                 printIntVarArray(problem->getSolutionArray());
2359                 std::cerr   << "> Same note played at the mesure: " << i << std::endl;
2360             }
2361             delete problem;
2362         }
2363     }
2364     cout << "End test 1H5_3v_1sp." << endl;
2365 }
2366
2367 void FuxTest::test_1H5_2v_2sp() {
2368     cout << "Start test 1H5_2v_2sp ..." << endl;
2369     spList = {SECOND_SPECIES};
2370     v_type = {0};
2371     for (size_t i = 1; i < cfSize-1; i++) {
2372         auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
2373         rel(problem->getHome(), problem->getSolutionArray()[i*2], IRT_EQ, cantusFirmus[i]); //
     fix the solution note as the same note as the cf
2374         if (has_solution(problem)) {
2375             std::cerr   << "/!\\ ERROR test 1H5_2v_2sp: It exists solution but it shouldn't with
      the following configuration:\n"
2376                         << "cantus firmus: ";
2377             printVector(cantusFirmus);
2378             std::cerr   << "solution array: ";
2379             printIntVarArray(problem->getSolutionArray());
2380             std::cerr   << "> Same note played at the mesure: " << i << std::endl;
2381         }
2382         delete problem;
2383     }
2384     cout << "End test 1H5_2v_2sp" << endl;
2385 }
2386
2387 void FuxTest::test_1H5_3v_2sp() {
2388     cout << "Start test 1H5_3v_2sp..." << endl;
2389     int cpSize = cfSize*2-1; // size of a counterpoint
2390     spList = {SECOND_SPECIES, SECOND_SPECIES};
2391     v_type = {0, 0};
2392     // // test solution lines != cf
2393     for (size_t j = 0; j < 2; j++) { // iterate over each solution line
2394         for (size_t i = 1; i < cfSize-1; i++) { // iterate over each note of the cantus firmus
2395             auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
2396             rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize) + i*2], IRT_EQ,
     cantusFirmus[i]); // fix the solution note as the same note as the cf
2397             if (has_solution(problem)) {
2398                 std::cerr   << "/!\\ ERROR test 1H5_3v_2sp: It exists solution but it shouldn't
     with the following configuration:\n"
2399                             << "cantus firmus: ";
2400                 printVector(cantusFirmus);
2401                 std::cerr   << "solution array: ";
2402                 printIntVarArray(problem->getSolutionArray());
2403                 std::cerr   << "> Same note played at the mesure: " << i << std::endl;
2404             }
2405             delete problem;
2406         }
2407     }
2408     // test notes of solutions lines are different
2409     for (size_t i = 1; i < cfSize-1; i++) {
2410         for (int j = -6; j < 13; j++) { // iterate over each possible note of solution lines
2411             auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
2412             int note = cantusFirmus[0] + j;
2413             rel(problem->getHome(), problem->getSolutionArray()[i*2], IRT_EQ, note); // fix the
     note of the first solution line as the same note as the note of the second solution line
2414             rel(problem->getHome(), problem->getSolutionArray()[cpSize+i*2], IRT_EQ, note); //
     fix the note of the first solution line as the same note as the note of the second solution
     line
2415             if (has_solution(problem)) {
2416                 std::cerr   << "/!\\ ERROR test 1H5_3v_2sp: It exists solution but it shouldn't
     with the following configuration:\n"
2417                             << "cantus firmus: ";
2418                 printVector(cantusFirmus);
```

```
2419                 std::cerr   << "solution array: ";
2420                 printIntVarArray(problem->getSolutionArray());
2421                 std::cerr   << "> Same note played at the mesure: " << i << std::endl;
2422             }
2423             delete problem;
2424         }
2425     }
2426     cout << "End test 1H5_3v_2sp." << endl;
2427 }
2428
2429 void FuxTest::test_1H5_2v_3sp() {
2430     cout << "Start test 1H5_2v_3sp ..." << endl;
2431     spList = {THIRD_SPECIES};
2432     v_type = {0};
2433     for (size_t i = 1; i < cfSize-1; i++) {
2434         auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
2435         rel(problem->getHome(), problem->getSolutionArray()[i*4], IRT_EQ, cantusFirmus[i]); //
     fix the solution note as the same note as the cf
2436         if (has_solution(problem)) {
2437             std::cerr   << "/!\\ ERROR test 1H5_2v_3sp: It exists solution but it shouldn't with
      the following configuration:\n"
2438                         << "cantus firmus: ";
2439             printVector(cantusFirmus);
2440             std::cerr   << "solution array: ";
2441             printIntVarArray(problem->getSolutionArray());
2442             std::cerr   << "> Same note played at the mesure: " << i << std::endl;
2443         }
2444         delete problem;
2445     }
2446     cout << "End test 1H5_2v_3sp" << endl;
2447 }
2448
2449 void FuxTest::test_1H5_3v_3sp() {
2450     cout << "Start test 1H5_3v_3sp..." << endl;
2451     int cpSize = cfSize*4-3; // size of a counterpoint
2452     spList = {THIRD_SPECIES, THIRD_SPECIES};
2453     v_type = {0, 0};
2454     // // test solution lines != cf
2455     for (size_t j = 0; j < 2; j++) { // iterate over each solution line
2456         for (size_t i = 1; i < cfSize-1; i++) { // iterate over each note of the cantus firmus
2457             auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
2458             rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize) + i*4], IRT_EQ,
     cantusFirmus[i]); // fix the solution note as the same note as the cf
2459             if (has_solution(problem)) {
2460                 std::cerr   << "/!\\ ERROR test 1H5_3v_3sp: It exists solution but it shouldn't
     with the following configuration:\n"
2461                             << "cantus firmus: ";
2462                 printVector(cantusFirmus);
2463                 std::cerr   << "solution array: ";
2464                 printIntVarArray(problem->getSolutionArray());
2465                 std::cerr   << "> Same note played at the mesure: " << i << std::endl;
2466             }
2467             delete problem;
2468         }
2469     }
2470     // test notes of solutions lines are different
2471     for (size_t i = 1; i < cfSize-1; i++) {
2472         for (int j = -6; j < 13; j++) { // iterate over each possible note of solution lines
2473             auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
2474             int note = cantusFirmus[0] + j;
2475             rel(problem->getHome(), problem->getSolutionArray()[i*4], IRT_EQ, note); // fix the
     note of the first solution line as the same note as the note of the second solution line
2476             rel(problem->getHome(), problem->getSolutionArray()[cpSize+i*4], IRT_EQ, note); //
     fix the note of the first solution line as the same note as the note of the second solution
     line
2477             if (has_solution(problem)) {
2478                 std::cerr   << "/!\\ ERROR test 1H5_3v_3sp: It exists solution but it shouldn't
     with the following configuration:\n"
2479                             << "cantus firmus: ";
2480                 printVector(cantusFirmus);
2481                 std::cerr   << "solution array: ";
2482                 printIntVarArray(problem->getSolutionArray());
2483                 std::cerr   << "> Same note played at the mesure: " << i << std::endl;
2484             }
2485             delete problem;
2486         }
2487     }
2488     cout << "End test 1H5_3v_3sp." << endl;
2489 }
2490
2491 void FuxTest::test_1H5_2v_4sp() {
2492     cout << "Start test 1H5_2v_4sp ..." << endl;
2493     spList = {FOURTH_SPECIES};
2494     v_type = {0};
2495     for (size_t i = 1; i < cfSize-1; i++) {
2496         auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
```

```
2497            rel(problem->getHome(), problem->getSolutionArray()[i*2-1], IRT_EQ, cantusFirmus[i]); //
        fix the solution note as the same note as the cf
2498            if (has_solution(problem)) {
2499                std::cerr   << "/!\\ ERROR test 1H5_2v_4sp: It exists solution but it shouldn't with
        the following configuration:\n"
2500                            << "cantus firmus: ";
2501                printVector(cantusFirmus);
2502                std::cerr   << "solution array: ";
2503                printIntVarArray(problem->getSolutionArray());
2504                std::cerr   << "> Same note played at the mesure: " << i << std::endl;
2505            }
2506            delete problem;
2507        }
2508        cout << "End test 1H5_2v_4sp" << endl;
2509    }
2510
2511    void FuxTest::test_1H5_3v_4sp() {
2512        cout << "Start test 1H5_3v_4sp..." << endl;
2513        int cpSize = cfSize*2-2; // size of a counterpoint
2514        spList = {FOURTH_SPECIES, FOURTH_SPECIES};
2515        v_type = {0, 0};
2516        // // test solution lines != cf
2517        for (size_t j = 0; j < 2; j++) { // iterate over each solution line
2518            for (size_t i = 1; i < cfSize-1; i++) { // iterate over each note of the cantus firmus
2519                auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
        general_params, specific_params, importance, borrowMode);
2520                rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize) + (i*2)-2], IRT_EQ,
        cantusFirmus[i]); // fix the solution note as the same note as the cf
2521                if (has_solution(problem)) {
2522                    std::cerr   << "/!\\ ERROR test 1H5_3v_4sp: It exists solution but it shouldn't
        with the following configuration:\n"
2523                                << "cantus firmus: ";
2524                    printVector(cantusFirmus);
2525                    std::cerr   << "solution array: ";
2526                    printIntVarArray(problem->getSolutionArray());
2527                    std::cerr   << "> Same note played at the mesure: " << i << std::endl;
2528                }
2529                delete problem;
2530            }
2531        }
2532        // test notes of solutions lines are different
2533        for (size_t i = 1; i < cfSize-1; i++) {
2534            for (int j = -6; j < 13; j++) { // iterate over each possible note of solution lines
2535                auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
        general_params, specific_params, importance, borrowMode);
2536                int note = cantusFirmus[0] + j;
2537                rel(problem->getHome(), problem->getSolutionArray()[i*2-2], IRT_EQ, note); // fix
        the note of the first solution line as the same note as the note of the second solution line
2538                rel(problem->getHome(), problem->getSolutionArray()[cpSize+i*2-2], IRT_EQ, note); //
         fix the note of the first solution line as the same note as the note of the second solution
         line
2539                if (has_solution(problem)) {
2540                    std::cerr   << "/!\\ ERROR test 1H5_3v_4sp: It exists solution but it shouldn't
        with the following configuration:\n"
2541                                << "cantus firmus: ";
2542                    printVector(cantusFirmus);
2543                    std::cerr   << "solution array: ";
2544                    printIntVarArray(problem->getSolutionArray());
2545                    std::cerr   << "> Same note played at the mesure: " << i << std::endl;
2546                }
2547                delete problem;
2548            }
2549        }
2550        cout << "End test 1H5_3v_4sp." << endl;
2551    }
2552
2553    // /*
2554    // For thesis notes, imperfect consonances are preferred to perfect consonances, and fifths are
         preferred to octaves.
2555    // */
2556    // void FuxTest::test_1H6() {
2557    //     cout << "Start test 1H6..." << endl;
2558    //     test_1H6_2v_1sp();
2559    //     cout << "End test 1H6." << endl;
2560    // }
2561
2562    // void FuxTest::test_1H6_2v_1sp() {
2563    //     cout << "test_1H6_2v_1sp" << endl;
2564    //     spList = {FIRST_SPECIES};
2565    //     v_type = {0};
2566    //     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
        general_params, specific_params, importance, borrowMode);
2567    //     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 60);
2568    //     rel(problem->getHome(), problem->getSolutionArray()[1], IRT_EQ, 54);
2569    //     std::vector<CounterpointProblem*> solutions = get_all_solutions(problem);
2570    //     for (CounterpointProblem* solution : solutions) {
2571    //         cout << ">> Solution: " << solution->getSolutionArray() << endl;
2572    //         cout << ">> Costs: " << solution->cost() << endl;
2573    //     }
2574    //     delete problem;
```

```
// }


/*
In two voice composition, the harmonic interval of the penultimate note must be
a major sixth or a minor third depending on whether or not the cantus firmus is the lowest stratum.
In three voice composition, that harmonic interval must be either a minor third, a perfect fifth, a major sixth or an octave.
*/
void FuxTest::test_1H7() {
    cout << "Start test 1H7..." << endl;
    test_1H7_2v_1sp();
    test_1H7_3v_1sp();
    test_1H7_2v_2sp();
    test_1H7_3v_2sp();
    test_1H7_2v_3sp();
    test_1H7_3v_3sp();
    cout << "End test 1H7." << endl;
}

void FuxTest::test_1H7_2v_1sp() {
    cout << "Start test 1H7_2v_1sp..." << endl;
    // Test with cf as the lowest stratum
    spList = {FIRST_SPECIES};
    v_type = {3};
    for (size_t interval = 0; interval < 12; interval++) {
        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
        if (interval == 9) { // correct interval
            rel(problem->getHome(), problem->getSolutionArray()[cfSize-2], IRT_EQ, cantusFirmus[cfSize-2]+12+interval); // fix the penultimate solution note as the major sixth with the cf
            if (!has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test 1H7_2v_1sp: It doesn't exist a solution but it should with the following configuration:\n"
                            << "Cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "Solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> The penultimate note should be correct" << std::endl;
            }
        } else {
            rel(problem->getHome(), problem->getSolutionArray()[cfSize-2], IRT_EQ, cantusFirmus[cfSize-2]+12+interval); // fix the penultimate solution note as a wrong interval with the cf
            if (has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test 1H7_2v_1sp: It exists solution but it shouldn't with the following configuration:\n"
                            << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> The harmonic interval of the penultimate note must be a major sixth" << std::endl;
            }
        }
        delete problem;
    }
    // Test with cf as the highest stratum
    v_type = {-2};
    for (size_t interval = 0; interval < 12; interval++) {
        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
        if (interval == 3) { // correct interval
            rel(problem->getHome(), problem->getSolutionArray()[cfSize-2], IRT_EQ, cantusFirmus[cfSize-2]-interval); // fix the penultimate solution note as the major sixth with the cf
            if (!has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test 1H7_2v_1sp: It doesn't exist a solution but it should with the following configuration:\n"
                            << "Cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "Solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> The penultimate note should be correct" << std::endl;
            }
        } else {
            rel(problem->getHome(), problem->getSolutionArray()[cfSize-2], IRT_EQ, cantusFirmus[cfSize-2]-interval); // fix the penultimate solution note as a wrong interval with the cf
            if (has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test 1H7_2v_1sp: It exists solution but it shouldn't with the following configuration:\n"
                            << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> The harmonic interval of the penultimate note must be a minor third" << std::endl;
            }
        }
        delete problem;
    }
```

```
2651     cout << "End test 1H7_2v_1sp." << endl;
2652 }
2653
2654 void FuxTest::test_1H7_3v_1sp() {
2655     cout << "Start test 1H7_3v_1sp..." << endl;
2656     spList = {FIRST_SPECIES, FIRST_SPECIES};
2657     v_type = {3, 3};
2658     vector<int> intervals = {1, 2, 4, 5, 6, 8, 10, 11}; // forbidden intervals
2659     for (size_t i = 1; i < 3; i++) {
2660         for (int interval : intervals) {
2661         auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
2662         rel(problem->getHome(), problem->getSolutionArray()[cfSize*i-2], IRT_EQ, cantusFirmus[
     cfSize-2]+12+interval); // fix the penultimate solution note as a wrong interval with the cf
2663         if (has_solution(problem)) {
2664             std::cerr   << "/!\\ ERROR test 1H7_3v_1sp: It exists solution but it shouldn't with
      the following configuration:\n"
2665                         << "cantus firmus: ";
2666             printVector(cantusFirmus);
2667             std::cerr   << "solution array: ";
2668             printIntVarArray(problem->getSolutionArray());
2669             std::cerr   << "> The harmonic interval of the penultimate note must be either a
     minor third, a perfect fifth, a major sixth or an octave. " << std::endl;
2670         }
2671         delete problem;
2672         }
2673     }
2674     cout << "End test 1H7_3v_1sp." << endl;
2675 }
2676
2677 void FuxTest::test_1H7_2v_2sp() {
2678     cout << "Start test 1H7_2v_2sp..." << endl;
2679     int cpSize = cfSize*2-1; // size of a counterpoint
2680     // Test with cf as the lowest stratum
2681     spList = {SECOND_SPECIES};
2682     v_type = {3};
2683     for (size_t interval = 0; interval < 12; interval++) {
2684         auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
2685         if (interval == 9) { // correct interval
2686             rel(problem->getHome(), problem->getSolutionArray()[cpSize-2], IRT_EQ, cantusFirmus[
     cfSize-2]+12+interval); // fix the penultimate solution note as the major sixth with the cf
2687             if (!has_solution(problem)) {
2688                 std::cerr   << "/!\\ ERROR test 1H7_2v_2sp: It doesn't exist a solution but it
     should with the following configuration:\n"
2689                             << "Cantus firmus: ";
2690                 printVector(cantusFirmus);
2691                 std::cerr   << "Solution array: ";
2692                 printIntVarArray(problem->getSolutionArray());
2693                 std::cerr   << "> The penultimate note should be correct" << std::endl;
2694             }
2695         } else {
2696             rel(problem->getHome(), problem->getSolutionArray()[cpSize-2], IRT_EQ, cantusFirmus[
     cfSize-2]+12+interval); // fix the penultimate solution note as a wrong interval with the cf
2697             if (has_solution(problem)) {
2698                 std::cerr   << "/!\\ ERROR test 1H7_2v_2sp: It exists solution but it shouldn't
     with the following configuration:\n"
2699                             << "cantus firmus: ";
2700                 printVector(cantusFirmus);
2701                 std::cerr   << "solution array: ";
2702                 printIntVarArray(problem->getSolutionArray());
2703                 std::cerr   << "> The harmonic interval of the penultimate note must be a major
     sixth" << std::endl;
2704             }
2705         }
2706         delete problem;
2707     }
2708     // Test with cf as the highest stratum
2709     v_type = {-2};
2710     for (size_t interval = 0; interval < 12; interval++) {
2711         auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
2712         if (interval == 3) { // correct interval
2713             rel(problem->getHome(), problem->getSolutionArray()[cpSize-2], IRT_EQ, cantusFirmus[
     cfSize-2]-interval); // fix the penultimate solution note as the major sixth with the cf
2714             if (!has_solution(problem)) {
2715                 std::cerr   << "/!\\ ERROR test 1H7_2v_2sp: It doesn't exist a solution but it
     should with the following configuration:\n"
2716                             << "Cantus firmus: ";
2717                 printVector(cantusFirmus);
2718                 std::cerr   << "Solution array: ";
2719                 printIntVarArray(problem->getSolutionArray());
2720                 std::cerr   << "> The penultimate note should be correct" << std::endl;
2721             }
2722         } else {
2723             rel(problem->getHome(), problem->getSolutionArray()[cpSize-2], IRT_EQ, cantusFirmus[
     cfSize-2]-interval); // fix the penultimate solution note as a wrong interval with the cf
2724             if (has_solution(problem)) {
2725                 std::cerr   << "/!\\ ERROR test 1H7_2v_2sp: It exists solution but it shouldn't
     with the following configuration:\n"
```

```
2726                        << "cantus firmus: ";
2727                printVector(cantusFirmus);
2728                std::cerr   << "solution array: ";
2729                printIntVarArray(problem->getSolutionArray());
2730                std::cerr   << "> The harmonic interval of the penultimate note must be a minor
    third" << std::endl;
2731            }
2732        }
2733        delete problem;
2734    }
2735    cout << "End test 1H7_2v_2sp." << endl;
2736 }
2737
2738 void FuxTest::test_1H7_3v_2sp() {
2739    cout << "Start test 1H7_3v_2sp..." << endl;
2740    int cpSize = cfSize*2-1; // size of a counterpoint
2741    spList = {SECOND_SPECIES, SECOND_SPECIES};
2742    v_type = {3, 3};
2743    vector<int> intervals = {1, 2, 4, 5, 6, 8, 10, 11}; // forbidden intervals
2744    for (size_t i = 1; i < 3; i++) {
2745        for (int interval : intervals) {
2746            auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
2747            rel(problem->getHome(), problem->getSolutionArray()[cpSize*i-2], IRT_EQ,
    cantusFirmus[cfSize-2]+12+interval); // fix the penultimate solution note as a wrong
    interval with the cf
2748            if (has_solution(problem)) {
2749                std::cerr   << "/!\\ ERROR test 1H7_3v_2sp: It exists solution but it shouldn't
    with the following configuration:\n"
2750                        << "cantus firmus: ";
2751                printVector(cantusFirmus);
2752                std::cerr   << "solution array: ";
2753                printIntVarArray(problem->getSolutionArray());
2754                std::cerr   << "> The harmonic interval of the penultimate note must be either a
     minor third, a perfect fifth, a major sixth or an octave. " << std::endl;
2755            }
2756            delete problem;
2757        }
2758    }
2759    cout << "End test 1H7_3v_2sp." << endl;
2760 }
2761
2762 void FuxTest::test_1H7_2v_3sp() {
2763    cout << "Start test 1H7_2v_3sp..." << endl;
2764    int cpSize = cfSize*4-3; // size of a counterpoint
2765    // Test with cf as the lowest stratum
2766    spList = {THIRD_SPECIES};
2767    v_type = {3};
2768    for (size_t interval = 0; interval < 12; interval++) {
2769        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
2770        if (interval == 9) { // correct interval
2771            rel(problem->getHome(), problem->getSolutionArray()[cpSize-2], IRT_EQ, cantusFirmus[
    cfSize-2]+12+interval); // fix the penultimate solution note as the major sixth with the cf
2772            if (!has_solution(problem)) {
2773                std::cerr   << "/!\\ ERROR test 1H7_2v_3sp: It doesn't exist a solution but it
    should with the following configuration:\n"
2774                        << "Cantus firmus: ";
2775                printVector(cantusFirmus);
2776                std::cerr   << "Solution array: ";
2777                printIntVarArray(problem->getSolutionArray());
2778                std::cerr   << "> The penultimate note should be correct" << std::endl;
2779            }
2780        } else {
2781            rel(problem->getHome(), problem->getSolutionArray()[cpSize-2], IRT_EQ, cantusFirmus[
    cfSize-2]+12+interval); // fix the penultimate solution note as a wrong interval with the cf
2782            if (has_solution(problem)) {
2783                std::cerr   << "/!\\ ERROR test 1H7_2v_3sp: It exists solution but it shouldn't
    with the following configuration:\n"
2784                        << "cantus firmus: ";
2785                printVector(cantusFirmus);
2786                std::cerr   << "solution array: ";
2787                printIntVarArray(problem->getSolutionArray());
2788                std::cerr   << "> The harmonic interval of the penultimate note must be a major
    sixth" << std::endl;
2789            }
2790        }
2791        delete problem;
2792    }
2793    // Test with cf as the highest stratum
2794    v_type = {-2};
2795    for (size_t interval = 0; interval < 12; interval++) {
2796        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
    general_params, specific_params, importance, borrowMode);
2797        if (interval == 3) { // correct interval
2798            rel(problem->getHome(), problem->getSolutionArray()[cpSize-2], IRT_EQ, cantusFirmus[
    cfSize-2]-interval); // fix the penultimate solution note as the major sixth with the cf
2799            if (!has_solution(problem)) {
2800                std::cerr   << "/!\\ ERROR test 1H7_2v_3sp: It doesn't exist a solution but it
    should with the following configuration:\n"
```

```
2801                        << "Cantus firmus: ";
2802                printVector(cantusFirmus);
2803                std::cerr   << "Solution array: ";
2804                printIntVarArray(problem->getSolutionArray());
2805                std::cerr   << "> The penultimate note should be correct" << std::endl;
2806            }
2807        } else {
2808            rel(problem->getHome(), problem->getSolutionArray()[cpSize-2], IRT_EQ, cantusFirmus[cfSize-2]-interval); // fix the penultimate solution note as a wrong interval with the cf
2809            if (has_solution(problem)) {
2810                std::cerr   << "/!\\ ERROR test 1H7_2v_3sp: It exists solution but it shouldn't with the following configuration:\n"
2811                            << "cantus firmus: ";
2812                printVector(cantusFirmus);
2813                std::cerr   << "solution array: ";
2814                printIntVarArray(problem->getSolutionArray());
2815                std::cerr   << "> The harmonic interval of the penultimate note must be a minor third" << std::endl;
2816            }
2817        }
2818        delete problem;
2819    }
2820    cout << "End test 1H7_2v_3sp." << endl;
2821 }
2822
2823 void FuxTest::test_1H7_3v_3sp() {
2824    cout << "Start test 1H7_3v_3sp..." << endl;
2825    int cpSize = cfSize*4-3; // size of a counterpoint
2826    spList = {THIRD_SPECIES, THIRD_SPECIES};
2827    v_type = {3, 3};
2828    vector<int> intervals = {1, 2, 4, 5, 6, 8, 10, 11}; // forbidden intervals
2829    for (size_t i = 1; i < 3; i++) {
2830        for (int interval : intervals) {
2831            auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
2832            rel(problem->getHome(), problem->getSolutionArray()[cpSize*i-2], IRT_EQ, cantusFirmus[cfSize-2]+12+interval); // fix the penultimate solution note as a wrong interval with the cf
2833            if (has_solution(problem)) {
2834                std::cerr   << "/!\\ ERROR test 1H7_3v_3sp: It exists solution but it shouldn't with the following configuration:\n"
2835                            << "cantus firmus: ";
2836                printVector(cantusFirmus);
2837                std::cerr   << "solution array: ";
2838                printIntVarArray(problem->getSolutionArray());
2839                std::cerr   << "> The harmonic interval of the penultimate note must be either a minor third, a perfect fifth, a major sixth or an octave. " << std::endl;
2840            }
2841            delete problem;
2842        }
2843    }
2844    cout << "End test 1H7_3v_3sp." << endl;
2845 }
2846
2847 void FuxTest::test_1H7_2v_4sp() {
2848    cout << "Start test 1H7_2v_4sp..." << endl;
2849    int cpSize = cfSize*2-2; // size of a counterpoint
2850    // Test with cf as the lowest stratum
2851    spList = {FOURTH_SPECIES};
2852    v_type = {3};
2853    for (size_t interval = 0; interval < 12; interval++) {
2854        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
2855        if (interval == 9) { // correct interval
2856            rel(problem->getHome(), problem->getSolutionArray()[cpSize-2], IRT_EQ, cantusFirmus[cfSize-2]+12+interval); // fix the penultimate solution note as the major sixth with the cf
2857            if (!has_solution(problem)) {
2858                std::cerr   << "/!\\ ERROR test 1H7_2v_4sp: It doesn't exist a solution but it should with the following configuration:\n"
2859                            << "Cantus firmus: ";
2860                printVector(cantusFirmus);
2861                std::cerr   << "Solution array: ";
2862                printIntVarArray(problem->getSolutionArray());
2863                std::cerr   << "> The penultimate note should be correct" << std::endl;
2864            }
2865        } else {
2866            rel(problem->getHome(), problem->getSolutionArray()[cpSize-2], IRT_EQ, cantusFirmus[cfSize-2]+12+interval); // fix the penultimate solution note as a wrong interval with the cf
2867            if (has_solution(problem)) {
2868                std::cerr   << "/!\\ ERROR test 1H7_2v_4sp: It exists solution but it shouldn't with the following configuration:\n"
2869                            << "cantus firmus: ";
2870                printVector(cantusFirmus);
2871                std::cerr   << "solution array: ";
2872                printIntVarArray(problem->getSolutionArray());
2873                std::cerr   << "> The harmonic interval of the penultimate note must be a major sixth" << std::endl;
2874            }
2875        }
2876        delete problem;
```

```
2877     }
2878     cout << "End test 1H7_2v_4sp." << endl;
2879 }
2880
2881 void FuxTest::test_1H7_3v_4sp() {
2882     cout << "Start test 1H7_3v_4sp..." << endl;
2883     int cpSize = cfSize*2-2; // size of a counterpoint
2884     spList = {THIRD_SPECIES, THIRD_SPECIES};
2885     v_type = {3, 3};
2886     vector<int> intervals = {1, 2, 4, 5, 6, 8, 10, 11}; // forbidden intervals
2887     for (size_t i = 1; i < 3; i++) {
2888         for (int interval : intervals) {
2889             auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
2890             rel(problem->getHome(), problem->getSolutionArray()[cpSize*i-2], IRT_EQ,
     cantusFirmus[cfSize-2]+12+interval); // fix the penultimate solution note as a wrong
     interval with the cf
2891             if (has_solution(problem)) {
2892                 std::cerr   << "/!\\ ERROR test 1H7_3v_4sp: It exists solution but it shouldn't
     with the following configuration:\n"
2893                         << "cantus firmus: ";
2894                 printVector(cantusFirmus);
2895                 std::cerr   << "solution array: ";
2896                 printIntVarArray(problem->getSolutionArray());
2897                 std::cerr   << "> The harmonic interval of the penultimate note must be either a
      minor third, a perfect fifth, a major sixth or an octave. " << std::endl;
2898             }
2899             delete problem;
2900         }
2901     }
2902     cout << "End test 1H7_3v_4sp." << endl;
2903 }
2904
2905 // In three voice composition, tenths are prohibited in the last chord.
2906 void FuxTest::test_1H10(){
2907     cout << "Start test 1H10..." << endl;
2908     test_1H10_3v_1sp();
2909     test_1H10_3v_2sp();
2910     test_1H10_3v_3sp();
2911     test_1H10_3v_4sp();
2912     cout << "End test 1H10." << endl;
2913 }
2914
2915 void FuxTest::test_1H10_3v_1sp(){
2916     cout << "Start test_1H10_3v_1sp..." << endl;
2917     spList = {FIRST_SPECIES, FIRST_SPECIES};
2918     v_type = {3, 3};
2919     int intervals[] = {3, 4}; // forbidden intervals
2920     for (size_t i = 1; i < 3; i++) { // iterate over each counter point
2921         for (int interval: intervals) { // interval tested
2922             auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
2923             int note = cantusFirmus[cfSize-1] + 12 + interval;
2924             rel(problem->getHome(), problem->getSolutionArray()[(i*cfSize)-1], IRT_EQ, note); //
      fix the last note as a thens
2925             if (has_solution(problem)) {
2926                 std::cerr   << "/!\\ ERROR test 1H10_3v_1sp: It exists solution but it shouldn't
      with the following configuration:\n"
2927                             << "cantus firmus: ";
2928                 printVector(cantusFirmus);
2929                 std::cerr   << "solution array: ";
2930                 printIntVarArray(problem->getSolutionArray());
2931                 std::cerr   << "> A thenth is played in the last chord" << std::endl;
2932             }
2933             delete problem;
2934
2935         }
2936     }
2937     cout << "End test 1H10_3v_1sp" << endl;
2938 }
2939
2940 void FuxTest::test_1H10_3v_2sp() {
2941     cout << "Start test_1H10_3v_2sp..." << endl;
2942     int cpSize = cfSize*2-1; // size of a counterpoint
2943     spList = {SECOND_SPECIES, SECOND_SPECIES};
2944     v_type = {3, 3};
2945     int intervals[] = {3, 4}; // forbidden intervals
2946     for (size_t i = 1; i < 3; i++) { // iterate over each counter point
2947         for (int interval: intervals) { // interval tested
2948             auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
2949             int note = cantusFirmus[cfSize-1] + 12 + interval;
2950             rel(problem->getHome(), problem->getSolutionArray()[(i*cpSize)-1], IRT_EQ, note); //
      fix the last note as a thens
2951             if (has_solution(problem)) {
2952                 std::cerr   << "/!\\ ERROR test 1H10_3v_2sp: It exists solution but it shouldn't
      with the following configuration:\n"
2953                             << "cantus firmus: ";
2954                 printVector(cantusFirmus);
2955                 std::cerr   << "solution array: ";
```

```
2956                 printIntVarArray(problem->getSolutionArray());
2957                 std::cerr    << "> A thenth is played in the last chord" << std::endl;
2958             }
2959             delete problem;
2960         }
2961     }
2962     cout << "End test_1H10_3v_2sp" << endl;
2963 }
2964
2965 void FuxTest::test_1H10_3v_3sp() {
2966     cout << "Start test_1H10_3v_3sp..." << endl;
2967     int cpSize = cfSize*4-3; // size of a counterpoint
2968     spList = {THIRD_SPECIES, THIRD_SPECIES};
2969     v_type = {3, 3};
2970     int intervals[] = {3, 4}; // forbidden intervals
2971     for (size_t i = 1; i < 3; i++) { // iterate over each counter point
2972         for (int interval: intervals) { // interval tested
2973             auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
2974             int note = cantusFirmus[cfSize-1] + 12 + interval;
2975             rel(problem->getHome(), problem->getSolutionArray()[(i*cpSize)-1], IRT_EQ, note); //
      fix the last note as a thens
2976             if (has_solution(problem)) {
2977                 std::cerr    << "/!\\ ERROR test 1H10_3v_3sp: It exists solution but it shouldn't
      with the following configuration:\n"
2978                              << "cantus firmus: ";
2979                 printVector(cantusFirmus);
2980                 std::cerr    << "solution array: ";
2981                 printIntVarArray(problem->getSolutionArray());
2982                 std::cerr    << "> A thenth is played in the last chord" << std::endl;
2983             }
2984             delete problem;
2985         }
2986     }
2987     cout << "End test_1H10_3v_3sp" << endl;
2988 }
2989
2990 void FuxTest::test_1H10_3v_4sp() {
2991     cout << "Start test_1H10_3v_4sp..." << endl;
2992     int cpSize = cfSize*2-2; // size of a counterpoint
2993     spList = {FOURTH_SPECIES, FOURTH_SPECIES};
2994     v_type = {3, 3};
2995     int intervals[] = {3, 4}; // forbidden intervals
2996     for (size_t i = 1; i < 3; i++) { // iterate over each counter point
2997         for (int interval: intervals) { // interval tested
2998             auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
2999             int note = cantusFirmus[cfSize-1] + 12 + interval;
3000             rel(problem->getHome(), problem->getSolutionArray()[(i*cpSize)-1], IRT_EQ, note); //
      fix the last note as a thens
3001             if (has_solution(problem)) {
3002                 std::cerr    << "/!\\ ERROR test 1H10_3v_4sp: It exists solution but it shouldn't
      with the following configuration:\n"
3003                              << "cantus firmus: ";
3004                 printVector(cantusFirmus);
3005                 std::cerr    << "solution array: ";
3006                 printIntVarArray(problem->getSolutionArray());
3007                 std::cerr    << "> A thenth is played in the last chord" << std::endl;
3008             }
3009             delete problem;
3010         }
3011     }
3012     cout << "End test_1H10_3v_4sp" << endl;
3013 }
3014
3015 // Melodic intervals cannot exceed a minor sixth interval, but octave leaps are allowed in three
      and four voices.
3016 void FuxTest::test_1M2() {
3017     cout << "Start test 1M2..." << endl;
3018     test_1M2_2v_1sp();
3019     test_1M2_3v_1sp();
3020     cout << "End test 1M2." << endl;
3021 }
3022
3023 void FuxTest::test_1M2_2v_1sp() {
3024     cout << "Start test 1M2_2v_1sp" << endl;
3025     spList = {FIRST_SPECIES};
3026     v_type = {0};
3027     for (size_t i = 1; i < cfSize; i++) {
3028         auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
3029         rel(problem->getHome(), expr(problem->getHome(), abs(problem->getSolutionArray()[i] -
     problem->getSolutionArray()[i-1])), IRT_GR, 8); // fix the melodic interval between note i
     and i-1 greater than 8
3030         if (has_solution(problem)) {
3031             std::cerr    << "/!\\ ERROR test 1M2_2v_1sp: It exists solution but it shouldn't with
      the following configuration:\n"
3032                          << "cantus firmus: ";
3033             printVector(cantusFirmus);
3034             std::cerr    << "solution array: ";
```

```
3035             printIntVarArray(problem->getSolutionArray());
3036             std::cerr   << "> Same note played at the mesure: " << i << std::endl;
3037         }
3038         delete problem;
3039     }
3040     cout << "End test 1M2_2v_1sp" << endl;
3041 }
3042
3043 void FuxTest::test_1M2_3v_1sp() {
3044     cout << "Start test 1M2_3v_1sp..." << endl;
3045     spList = {FIRST_SPECIES, FIRST_SPECIES};
3046     v_type = {2, 2};
3047     // Add the constraint that the interval between note i and i-1 is greater than 8 and
     different from 12
3048     for (size_t i = 1; i < cfSize; i++) {
3049         auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
3050         cout << "Add constraint" << endl;
3051         IntVar absDiff = expr(problem->getHome(), abs(problem->getSolutionArray()[i] - problem->
     getSolutionArray()[i-1]));
3052         rel(problem->getHome(), absDiff, IRT_GR, 8);
3053         rel(problem->getHome(), absDiff, IRT_NQ, 12);
3054         cout << "Constraint added" << endl;
3055         if (has_solution(problem)) {
3056             std::cerr   << "/!\\ ERROR test 1M2_2v_1sp: It exists solution but it shouldn't with
      the following configuration:\n"
3057                         << "cantus firmus: ";
3058             printVector(cantusFirmus);
3059             std::cerr   << "solution array: ";
3060             printIntVarArray(problem->getSolutionArray());
3061             std::cerr   << "> Same note played at the mesure: " << i << std::endl;
3062         }
3063         delete problem;
3064     }
3065     cout << "End test 1M2_3v_1sp" << endl;
3066 }
3067
3068 // void FuxTest::test_1P1_2v_1sp() {
3069 //     cout << "Start test 1P1_2v_1sp..." << endl;
3070 //     spList = {FIRST_SPECIES};
3071 //     v_type = {0};
3072 //     int cons_intervals[] = {0, 3, 4, 7, 8, 9}; // consonant intervals
3073 //     for (size_t i = 1; i < cfSize; i++) {
3074 //         auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
3075 //         // Add the constraint that the note at index i is in cons_intervals
3076 //         Gecode::IntSet consSet(cons_intervals, sizeof(cons_intervals) / sizeof(cons_intervals
     [0]));
3077 //         dom(problem->getHome(), problem->getSolutionArray()[i], consSet);
3078
3079 //         BoolVar bothUp = expr(problem->getHome(), part1->getMelodicIntervals()[i] > 0 &&
     part2->getMelodicIntervals()[i] > 0);
3080 //         BoolVar bothDown = expr(home, part1->getMelodicIntervals()[i] < 0 && part2->
     getMelodicIntervals()[i] < 0);
3081 //         BoolVar sameDirection = expr(home, bothUp || bothDown);
3082
3083 //         // Constrain the motion to be direct
3084 //         rel(home, sameDirection == 1);
3085
3086 //         rel(problem->getHome(), problem->getSolutionArray()[i], IRT_EQ, cantusFirmus[i]); //
     fix the solution note as the same note as the cf
3087 //         if (has_solution(problem)) {
3088 //             std::cerr   << "/!\\ ERROR test 1P1_2v_1sp: It exists solution but it shouldn't
     with the following configuration:\n"
3089 //                         << "cantus firmus: ";
3090 //             printVector(cantusFirmus);
3091 //             std::cerr   << "solution array: ";
3092 //             printIntVarArray(problem->getSolutionArray());
3093 //             std::cerr   << "> Same note played at the mesure: " << i << std::endl;
3094 //         }
3095 //         problem->getSolutionArray().
3096 //         delete problem;
3097 //     }
3098 //     cout << "End test 1P1_2v_1sp" << endl;
3099 // }
3100
3101
3102 void FuxTest::test_2H2() {
3103     cout << "Start test 2H2..." << endl;
3104     test_2H2_2v_2sp();
3105     test_2H2_3v_2sp();
3106     test_2H2_4v_2sp();
3107     cout << "End test 2H2." << endl;
3108 }
3109
3110 void FuxTest::test_2H2_2v_2sp() {
3111     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
3112     int cons[] = {3, 4, 7, 8, 9}; // conssonant intervals without 0 because 1H5
3113     spList = {SECOND_SPECIES};
3114     v_type = {3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
```

```
3115     // Test that dissonant notes are forbidden
3116     for (int interval : dis) {
3117         //test first note
3118         CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
     melodic_params, general_params, specific_params, importance, borrowMode);
3119             int note = cantusFirmus[0] + 12 + interval; // note is dissonant
3120             rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); // fix the
     first note
3121             if (has_solution(problem)) {
3122                 std::cerr   << "/!\\ ERROR test 2H2_2v_2sp: It exists a solution but it shouldn'
     t with the following configuration:\n"
3123                             << "Cantus firmus: ";
3124                 printVector(cantusFirmus);
3125                 std::cerr   << "Solution array: ";
3126                 printIntVarArray(problem->getSolutionArray());
3127                 std::cerr   << "> Dissonant harmonic at the mesure " << 0 << std::endl;
3128             }
3129             delete problem;
3130         // test next notes
3131         for (size_t i = 1; i < cfSize; i++) {
3132             CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
     melodic_params, general_params, specific_params, importance, borrowMode);
3133             int note = cantusFirmus[i] + 12 + interval; // note is dissonant
3134             rel(problem->getHome(), problem->getSolutionArray()[i*2], IRT_EQ, note); // fix the
     note i
3135             if (has_solution(problem)) {
3136                 std::cerr   << "/!\\ ERROR test 2H2_2v_2sp: It exists a solution but it shouldn'
     t with the following configuration:\n"
3137                             << "Cantus firmus: ";
3138                 printVector(cantusFirmus);
3139                 std::cerr   << "Solution array: ";
3140                 printIntVarArray(problem->getSolutionArray());
3141                 std::cerr   << "> Dissonant harmonic at the mesure " << i << std::endl;
3142             }
3143             delete problem;
3144         }
3145     }
3146 }
3147
3148 void FuxTest::test_2H2_3v_2sp() {
3149     cout << "Start test 2H2_3v_2sp..." << endl;
3150     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
3151     int cons[] = {3, 4, 7, 8, 9}; // consonant intervals without 0 because 1H5
3152     spList = {SECOND_SPECIES, SECOND_SPECIES};
3153     v_type = {3, 3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
3154     // Test that dissonant notes are forbidden
3155     for (int interval : dis) {
3156         for (size_t j = 0; j < 2; j++) { // iterate over each solution line
3157             //test first note
3158             CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
     melodic_params, general_params, specific_params, importance, borrowMode);
3159             int note = cantusFirmus[0] + 12 + interval; // note is dissonant
3160             rel(problem->getHome(), problem->getSolutionArray()[j*((cfSize*2)-1)], IRT_EQ, note)
     ; // fix the first note
3161             if (has_solution(problem)) {
3162                 std::cerr   << "/!\\ ERROR test 2H2_3v_2sp: It exists a solution but it shouldn'
     t with the following configuration:\n"
3163                             << "Cantus firmus: ";
3164                 printVector(cantusFirmus);
3165                 std::cerr   << "Solution array: ";
3166                 printIntVarArray(problem->getSolutionArray());
3167                 std::cerr   << "> Dissonant harmonic at the mesure " << 0 << std::endl;
3168             }
3169             delete problem;
3170             // test next notes
3171             for (size_t i = 0; i < cfSize; i++) {
3172                 CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
     melodic_params, general_params, specific_params, importance, borrowMode);
3173                 int note = cantusFirmus[i] + 12 + interval; // note is dissonant
3174                 rel(problem->getHome(), problem->getSolutionArray()[j*cfSize+i], IRT_EQ, note);
     // fix the note i
3175                 if (has_solution(problem)) {
3176                     std::cerr   << "/!\\ ERROR test 2H2_3v_2sp: It exists a solution but it
     shouldn't with the following configuration:\n"
3177                                 << "Cantus firmus: ";
3178                     printVector(cantusFirmus);
3179                     std::cerr   << "Solution array: ";
3180                     printIntVarArray(problem->getSolutionArray());
3181                     std::cerr   << "> Dissonant harmonic at the measure " << i << std::endl;
3182                 }
3183                 delete problem;
3184             }
3185         }
3186     }
3187 }
3188
3189 void FuxTest::test_2H2_4v_2sp() {
3190     cout << "Start test 2H2_4v_2sp..." << endl;
3191     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
3192     int cons[] = {3, 4, 7, 8, 9}; // consonant intervals without 0 because 1H5
```

```
3193     spList = {SECOND_SPECIES, SECOND_SPECIES, SECOND_SPECIES};
3194     v_type = {3, 3, 3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
3195 
3196     // Test that dissonant notes are forbidden
3197     for (int interval : dis) {
3198         for (size_t j = 0; j < 3; j++) { // iterate over each solution line
3199             //test first note
3200             CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
     melodic_params, general_params, specific_params, importance, borrowMode);
3201             int note = cantusFirmus[0] + 12 + interval; // note is dissonant
3202             rel(problem->getHome(), problem->getSolutionArray()[j*((cfSize*2)-1)], IRT_EQ, note)
     ; // fix the first note
3203             if (has_solution(problem)) {
3204                 std::cerr   << "/!\\ ERROR test 2H2_4v_2sp: It exists a solution but it shouldn'
     t with the following configuration:\n"
3205                             << "Cantus firmus: ";
3206                 printVector(cantusFirmus);
3207                 std::cerr   << "Solution array: ";
3208                 printIntVarArray(problem->getSolutionArray());
3209                 std::cerr   << "> Dissonant harmonic at the mesure " << 0 << std::endl;
3210             }
3211             delete problem;
3212             // test next notes
3213             for (size_t i = 0; i < cfSize; i++) {
3214                 CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type,
     melodic_params, general_params, specific_params, importance, borrowMode);
3215                 int note = cantusFirmus[i] + 12 + interval; // note is dissonant
3216                 rel(problem->getHome(), problem->getSolutionArray()[j*cfSize+i], IRT_EQ, note);
     // fix the note i
3217                 if (has_solution(problem)) {
3218                     std::cerr   << "/!\\ ERROR test 2H2_4v_2sp: It exists a solution but it
     shouldn't with the following configuration:\n"
3219                                 << "Cantus firmus: ";
3220                     printVector(cantusFirmus);
3221                     std::cerr   << "Solution array: ";
3222                     printIntVarArray(problem->getSolutionArray());
3223                     std::cerr   << "> Dissonant harmonic at the measure " << i << std::endl;
3224                 }
3225                 delete problem;
3226             }
3227         }
3228     }
3229 }
3230 
3231 void FuxTest::test_2M2() {
3232     cout << "Start test 2M2..." << endl;
3233     test_2M2_2v_2sp();
3234     cout << "End test 2M2." << endl;
3235 }
3236 
3237 void FuxTest::test_2M2_2v_2sp() {
3238     cout << "Start test_2M2_2v_2sp" << endl;
3239     int cpSize = cfSize*2-1; // size of a counterpoint
3240     spList = {SECOND_SPECIES};
3241     v_type = {1}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
3242     for (size_t i = 1; i < cpSize; i++) {
3243         auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
3244         cout << "1" << endl;
3245         rel(problem->getHome(), problem->getSolutionArray()[i], IRT_EQ, problem->
     getSolutionArray()[i-1]); // fix the note i a the same than i-1
3246         cout << "2" << endl;
3247         if (has_solution(problem)) {
3248             std::cerr   << "/!\\ ERROR test 2M2_2v_2sp: It exists solution but it shouldn't with
      the following configuration:\n"
3249                         << "cantus firmus: ";
3250             printVector(cantusFirmus);
3251             std::cerr   << "solution array: ";
3252             printIntVarArray(problem->getSolutionArray());
3253             std::cerr   << "Same note is played consecutively" << std::endl;
3254         }
3255         cout << "3" << endl;
3256         delete problem;
3257         cout << "4" << endl;
3258     }
3259     cout << "End test 2M2_2v_2sp" << endl;
3260 }
3261 
3262 // ----- Figures tests -----
3263 
3264 void FuxTest::test_2v_1sp_fig22() {
3265     cout << "Start test_2v_1sp_fig22" << endl;
3266     spList = {FIRST_SPECIES};
3267     cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57};
3268     cp =           {69,64,67,65,64,72,69,71,71,69,68,69};
3269     v_type = {1};
3270     borrowMode = 1;
3271     test_configuration();
3272     cout << "End test_2v_1sp_fig22." << endl;
3273 }
```

```
3274
3275 void FuxTest::test_2v_1sp_fig23(){
3276     cout << "Start test_2v_1sp_fig23" << endl;
3277     spList = {FIRST_SPECIES};
3278     cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57};
3279     cp =           {69,69,67,65,64,64,62,60,67,69,68,69};
3280     v_type = {1};
3281     borrowMode = 1;
3282     test_configuration();
3283     cout << "End test_2v_1sp_fig23." << endl;
3284 }
3285
3286 void FuxTest::test_2v_2sp_fig38(){
3287     cout << "Start test_2v_2sp_fig38" << endl;
3288     spList = {SECOND_SPECIES};
3289     cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
3290     cp =           {-1,65,64,62,60,58,57,55,53,57,60,58,57,69,67,64,65,67,69,65,62,64,65};
3291     v_type = {0};
3292     borrowMode = 1;
3293     test_configuration();
3294     cout << "End test_2v_2sp_fig38." << endl;
3295 }
3296
3297 void FuxTest::test_2v_2sp_fig39(){
3298     cout << "Start test_2v_2sp_fig39" << endl;
3299     spList = {SECOND_SPECIES};
3300     cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
3301     cp =           {-1,53,52,48,53,52,50,48,46,58,55,60,57,53,52,48,53,41,45,50,48,52,53};
3302     v_type = {-2};
3303     borrowMode = 1;
3304     test_configuration();
3305     cout << "End test_2v_2sp_fig39." << endl;
3306 }
3307
3308 void FuxTest::test_2v_2sp_fig40(){
3309     cout << "Start test_2v_2sp_fig40" << endl;
3310     spList = {SECOND_SPECIES};
3311     cantusFirmus = {55,60,59,55,60,64,62,67,64,60,62,59,57,55};
3312     cp =           {-1,
    67,64,65,67,69,71,69,67,72,71,72,74,72,71,69,67,65,64,72,71,69,67,62,64,66,67};
3313     v_type = {2};
3314     borrowMode = 1;
3315     test_configuration();
3316     cout << "End test_2v_2sp_fig40." << endl;
3317 }
3318
3319 void FuxTest::test_2v_2sp_fig41(){
3320     cout << "Start test_2v_2sp_fig41" << endl;
3321     spList = {SECOND_SPECIES};
3322     cantusFirmus = {55,60,59,55,60,64,62,67,64,60,62,59,57,55};
3323     cp =
    {-1,67,64,65,67,65,64,62,60,64,60,72,71,69,67,71,72,71,69,67,66,62,67,59,62,66,67};
3324     v_type = {2};
3325     borrowMode = 1;
3326     test_configuration();
3327     cout << "End test_2v_2sp_fig41." << endl;
3328 }
3329
3330 void FuxTest::test_2v_2sp_fig42(){
3331     cout << "Start test_2v_2sp_fig42" << endl;
3332     spList = {SECOND_SPECIES};
3333     cantusFirmus = {57,60,59,60,64,65,64,62,60,59,57};
3334     cp =           {-1,69,64,65,67,62,64,76,72,71,69,65,67,71,74,69,72,64,66,68,69};
3335     v_type = {2};
3336     borrowMode = 1;
3337     test_configuration();
3338     cout << "End test_2v_2sp_fig42." << endl;
3339 }
3340
3341 void FuxTest::test_2v_2sp_fig43(){
3342     cout << "Start test_2v_2sp_fig43" << endl;
3343     spList = {SECOND_SPECIES};
3344     cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57};
3345     cp =           {-1,45,57,52,55,52,50,53,57,59,60,48,50,45,48,52,53,55,57,45,52,56,57};
3346     v_type = {-1};
3347     borrowMode = 1;
3348     test_configuration();
3349     cout << "End test_2v_2sp_fig43." << endl;
3350 }
3351
3352 void FuxTest::test_2v_2sp_fig44(){
3353     cout << "Start test_2v_2sp_fig44" << endl;
3354     spList = {SECOND_SPECIES};
3355     cantusFirmus = {60,64,65,67,64,69,67,64,65,64,62,60};
3356     cp =           {-1,67,72,71,69,74,71,69,67,71,72,74,76,74,72,71,69,71,72,67,69,71,72};
3357     v_type = {1};
3358     borrowMode = 1;
3359     test_configuration();
3360     cout << "End test_2v_2sp_fig44." << endl;
3361 }
```

```
3362
3363 void FuxTest::test_2v_2sp_fig45(){
3364     cout << "Start test_2v_2sp_fig45" << endl;
3365     spList = {SECOND_SPECIES};
3366     cantusFirmus = {60,64,65,67,64,69,67,64,65,64,62,60};
3367     cp =           {-1,60,72,71,69,74,71,67,72,71,69,72,76,74,72,69,74,69,72,60,67,71,72};
3368     v_type = {1};
3369     borrowMode = 1;
3370     test_configuration();
3371     cout << "End test_2v_2sp_fig45." << endl;
3372 }
3373
3374 void FuxTest::test_2v_3sp_fig55() {
3375     cout << "Start test_2v_3sp_fig55" << endl;
3376     spList = {THIRD_SPECIES};
3377     cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
3378     cp =
     {62,64,65,67,69,71,72,74,76,74,71,72,74,72,70,69,70,72,74,76,77,65,69,71,72,69,70,72,70,69,67,71,69,62,

3379     v_type = {2};
3380     borrowMode = 1;
3381     test_configuration();
3382     cout << "End test_2v_3sp_fig55." << endl;
3383 }
3384
3385 void FuxTest::test_2v_3sp_fig56(){
3386     cout << "Start test_2v_3sp_fig56" << endl;
3387     spList = {THIRD_SPECIES};
3388     cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
3389     cp =
     {62,64,65,67,69,62,69,71,72,71,67,69,71,69,67,65,64,76,71,72,74,69,62,64,65,67,69,71,72,74,76,72,74,69,

3390     v_type = {1};
3391     borrowMode = 1;
3392     test_configuration();
3393     cout << "End test_2v_3sp_fig56." << endl;
3394 }
3395
3396 void FuxTest::test_2v_3sp_fig57(){
3397     cout << "Start test_2v_3sp_fig57" << endl;
3398     spList = {THIRD_SPECIES};
3399     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
3400     cp =
     {71,67,69,71,72,71,69,67,65,67,69,71,72,64,65,67,69,72,76,74,72,71,69,72,71,74,71,69,67,71,72,71,69,71,

3401     v_type = {2};
3402     borrowMode = 1;
3403     test_configuration();
3404     cout << "End test_2v_3sp_fig57." << endl;
3405 }
3406
3407 void FuxTest::test_2v_3sp_fig58(){
3408     cout << "Start test_2v_3sp_fig58" << endl;
3409     spList = {THIRD_SPECIES};
3410     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
3411     cp =
     {64,65,67,64,69,67,65,64,62,64,65,67,69,64,69,67,65,64,62,64,65,67,69,71,72,74,76,74,72,60,72,71,69,74,

3412     v_type = {1};
3413     borrowMode = 1;
3414     test_configuration();
3415     cout << "End test_2v_3sp_fig58." << endl;
3416 }
3417
3418 void FuxTest::test_2v_3sp_fig59() {
3419     cout << "Start test_2v_3sp_fig59" << endl;
3420     spList = {THIRD_SPECIES};
3421     cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
3422     cp =
     {65,64,62,60,59,62,67,65,64,62,60,58,57,60,62,64,65,62,64,65,67,64,65,67,69,67,65,69,67,65,64,62,60,64,

3423     v_type = {-1};
3424     borrowMode = 1;
3425     test_configuration();
3426     cout << "End test_2v_3sp_fig59." << endl;
3427 }
3428
3429
3430 void FuxTest::test_2v_3sp_fig60() {
3431     cout << "Start test_2v_3sp_fig60" << endl;
3432     spList = {THIRD_SPECIES};
3433     cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
3434     cp =
     {62,74,72,71,72,69,71,72,74,72,71,69,71,69,66,65,66,74,78,77,76,69,81,78,77,76,74,71,72,69,71,72,74,72,

3435     v_type = {1};
3436     borrowMode = 1;
3437     test_configuration();
3438     cout << "End test_2v_3sp_fig60." << endl;
3439 }
```

```
3440
3441 void FuxTest::test_2v_4sp_fig74() {
3442     cout << "Start test_2v_4sp_fig74" << endl;
3443     spList = {FOURTH_SPECIES};
3444     cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
3445     cp =           {62,62,74,74,72,72,71,71,76,76,74,74,77,77,76,76,74,74,73,74};
3446     v_type = {1};
3447     borrowMode = 1;
3448     test_configuration();
3449     cout << "End test_2v_4sp_fig74." << endl;
3450 }
3451
3452 void FuxTest::test_2v_4sp_fig75() {
3453     cout << "Start test_2v_4sp_fig75" << endl;
3454     spList = {FOURTH_SPECIES};
3455     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
3456     cp =           {76,76,72,72,71,72,64,64,65,65,72,72,71,71,76,76,74,76};
3457     v_type = {2};
3458     borrowMode = 1;
3459     test_configuration();
3460     cout << "End test_2v_4sp_fig75." << endl;
3461 }
3462
3463 void FuxTest::test_2v_4sp_fig76() {
3464     cout << "Start test_2v_4sp_fig76" << endl;
3465     spList = {FOURTH_SPECIES};
3466     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
3467     cp =           {64,64,69,69,67,67,65,65,62,62,74,74,72,72,76,76,74,76};
3468     v_type = {0};
3469     borrowMode = 1;
3470     test_configuration();
3471     cout << "End test_2v_4sp_fig76." << endl;
3472 }
3473
3474 void FuxTest::test_2v_4sp_fig77() {
3475     cout << "Start test_2v_4sp_fig77" << endl;
3476     spList = {FOURTH_SPECIES};
3477     cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
3478     cp =           {65,65,64,64,60,60,65,65,69,69,67,67,65,65,64,64,69,69,65,65,64,65};
3479     v_type = {0};
3480     borrowMode = 1;
3481     test_configuration();
3482     cout << "End test_2v_4sp_fig77." << endl;
3483 }
3484
3485 void FuxTest::test_2v_4sp_fig78() {
3486     cout << "Start test_2v_4sp_fig78" << endl;
3487     spList = {FOURTH_SPECIES};
3488     cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
3489     cp =           {74,74,72,72,71,71,66,66,64,64,76,76,74,74,72,72,71,71,74,74,72,74};
3490     v_type = {0};
3491     borrowMode = 1;
3492     test_configuration();
3493     cout << "End test_2v_4sp_fig78." << endl;
3494 }
3495
3496 void FuxTest::test_3v_1sp_fig108(){
3497     cout << "Start test_3v_1sp_fig108" << endl;
3498     spList = {FIRST_SPECIES, FIRST_SPECIES};
3499     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
3500     cp =           {67,64,65,69,72,72,76,72,69,68,
3501                     64,69,62,65,65,65,72,72,74,76};
3502     v_type = {1, 0};
3503     borrowMode = 1;
3504     test_configuration();
3505     cout << "End test_3v_1sp_fig108." << endl;
3506 }
3507
3508 void FuxTest::test_3v_1sp_fig109(){
3509     cout << "Start test_3v_1sp_fig109" << endl;
3510     spList = {FIRST_SPECIES, FIRST_SPECIES};
3511     cantusFirmus = {52,48,50,48,45,57,55,52,53,52};
3512     cp =           {67,64,65,64,64,64,67,67,62,64,
3513                     71,72,69,69,72,72,71,71,69,71};
3514     v_type = {2, 3};
3515     borrowMode = 1;
3516     test_configuration();
3517     cout << "End test_3v_1sp_fig109." << endl;
3518 }
3519
3520 void FuxTest::test_3v_1sp_fig110(){
3521     cout << "Start test_3v_1sp_fig110" << endl;
3522     spList = {FIRST_SPECIES, FIRST_SPECIES};
3523     cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
3524     cp =           {60,60,60,62,58,59,57,69,65,62,64,65,
3525                     53,52,53,50,55,55,53,45,50,50,48,41};
3526     v_type = {-1, -3};
3527     borrowMode = 1;
3528     test_configuration();
3529     cout << "End test_3v_1sp_fig110." << endl;
```

```
}

void FuxTest::test_3v_1sp_fig111(){
    cout << "Start test_3v_1sp_fig111" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
    cp =           {69,72,72,74,74,70,69,67,65,77,76,77,
                    77,76,77,74,70,67,65,64,65,62,60,65};
    v_type = {1, 2};
    borrowMode = 1;
    test_configuration();
    cout << "End test_3v_1sp_fig111." << endl;
}

void FuxTest::test_3v_1sp_fig112(){
    cout << "Start test_3v_1sp_fig112" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
    cp =           {69,70,72,69,70,72,69,67,72,69,70,69,
                    65,62,60,60,65,67,60,64,64,65,64,65};
    v_type = {1, 0};
    borrowMode = 1;
    test_configuration();
    cout << "End test_3v_1sp_fig112." << endl;
}

void FuxTest::test_3v_1sp_fig113(){
    cout << "Start test_3v_1sp_fig113" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {55,60,59,55,60,64,62,67,64,60,62,59,57,55};
    cp =           {71,67,67,71,76,72,71,67,67,69,66,67,66,67,
                    55,52,52,52,48,48,55,52,48,45,47,43,50,43};
    v_type = {2, 0};
    borrowMode = 1;
    test_configuration();
    cout << "End test_3v_1sp_fig113." << endl;
}

void FuxTest::test_3v_1sp_fig114(){
    cout << "Start test_3v_1sp_fig114" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {55,60,59,55,60,64,62,67,64,60,62,59,57,55};
    cp =           {71,67,67,71,67,69,71,71,72,64,66,67,66,67,
                    55,52,52,52,52,48,55,52,48,48,47,43,50,43};
    v_type = {3, 0};
    borrowMode = 1;
    test_configuration();
    cout << "End test_3v_1sp_fig114." << endl;
}

void FuxTest::test_3v_1sp_fig115(){
    cout << "Start test_3v_1sp_fig115" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {69,72,71,74,72,76,77,76,74,72,71,69};
    cp =           {60,64,67,65,64,67,69,67,65,69,68,69,
                    69,69,76,74,69,67,65,72,74,69,76,69};
    v_type = {-1, 0};
    borrowMode = 1;
    test_configuration();
    cout << "End test_3v_1sp_fig115." << endl;
}

void FuxTest::test_3v_1sp_fig116(){
    cout << "Start test_3v_1sp_fig116" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57};
    cp =           {64,64,67,65,64,72,69,72,71,69,68,69,
                    45,45,52,50,57,57,50,48,55,57,52,45};
    v_type = {1, -1};
    borrowMode = 1;
    test_configuration();
    cout << "End test_3v_1sp_fig116." << endl;
}

void FuxTest::test_3v_1sp_fig117(){
    cout << "Start test_3v_1sp_fig117" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {45,48,47,50,48,52,53,52,50,48,47,45};
    cp =           {60,64,62,65,64,60,60,60,62,57,56,57,
                    69,69,71,71,72,67,69,67,65,64,62,64};
    v_type = {2, 3};
    borrowMode = 1;
    test_configuration();
    cout << "End test_3v_1sp_fig117." << endl;
}

void FuxTest::test_3v_1sp_fig118(){
    cout << "Start test test_3v_1sp_fig118" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {60, 64, 65, 67, 64, 69, 67, 64, 65, 64, 62, 60};
```

```
3620     cp =            {67, 72, 69, 67, 72, 72, 76, 72, 71, 72, 71, 72,
3621                      48, 48, 50, 52, 48, 53, 52, 57, 50, 48, 55, 48};
3622     v_type = {1, -1};
3623     borrowMode = 1;
3624    test_configuration();
3625     cout << "End test_3v_1sp_fig118." << endl;
3626 }
3627
3628 void FuxTest::test_3v_1sp_fig119(){
3629     cout << "Start test 3v_1sp_fig119" << endl;
3630     spList = {FIRST_SPECIES, FIRST_SPECIES};
3631     cantusFirmus = {60, 64, 65, 67, 64, 69, 67, 64, 65, 64, 62, 60};
3632     cp =            {64, 67, 69, 71, 72, 72, 76, 72, 69, 72, 71, 72,
3633                      72, 72, 69, 67, 69, 65, 72, 72, 74, 72, 67, 72};
3634     v_type = {1, 1};
3635     borrowMode = 1;
3636     test_configuration();
3637     cout << "End test_3v_1sp_fig119." << endl;
3638 }
3639
3640
3641 void FuxTest::test_3v_2sp_fig125(){
3642     cout << "Start test_3v_2sp_fig125" << endl;
3643     spList = {FIRST_SPECIES, SECOND_SPECIES};
3644     cantusFirmus = {52,   48,   50,   48,   45,   57,   55,   52,   53,   52};
3645     cp =            {64,   64,   65,   64,   65,   64,   67,   67,   69,   68,
3646                      -1,71,72,71,69,65,67,64,69,71,72,69,71,74,76,74,72,69,71};
3647     v_type = {2 ,3};
3648     borrowMode = 1;
3649     test_configuration();
3650     cout << "End test_3v_2sp_fig125." << endl;
3651 }
3652
3653 void FuxTest::test_3v_2sp_fig128(){
3654     cout << "Start test_3v_2sp_fig128" << endl;
3655     spList = {SECOND_SPECIES, FIRST_SPECIES};
3656     cantusFirmus = {65,   67,   69,   65,   62,   64,   65,   72,   69,   65,   67,   65};
3657     cp =            {-1,77,76,74,72,71,69,72,70,69,67,72,69,81,79,76,72,69,74,77,77,76,77,
3658                      65,   60,   65,   65,   67,   72,   74,   76,   77,   74,   72,   65};
3659     v_type = {1 ,0};
3660     borrowMode = 1;
3661     test_configuration();
3662     cout << "End test_3v_2sp_fig128." << endl;
3663 }
3664
3665 void FuxTest::test_3v_2sp_fig129(){
3666     cout << "Start test_3v_2sp_fig129" << endl;
3667     spList = {FIRST_SPECIES, SECOND_SPECIES};
3668     cantusFirmus =  {65,   67,   69,   65,   62,   64,   65,   72,   69,   65,   67,   65}; //1
        sp 2v cf
3669     cp =             {60,   60,   60,   62,   65,   67,   69,   67,   60,   62,   64,   65,
3670                       -1,53,52,48,53,52,50,48,46,45,43,48,41,53,52,48,53,52,50,46,43,48,41};
3671     v_type = {-1, -3};
3672     borrowMode = 1;
3673     test_configuration();
3674     cout << "End test_3v_2sp_fig129." << endl;
3675 }
3676
3677 void FuxTest::test_3v_3sp_fig132(){
3678     cout << "Start test_3v_3sp_fig132" << endl;
3679     spList = {FIRST_SPECIES, THIRD_SPECIES};
3680     cantusFirmus =  {62,65,64,62,67,65,69,67,65,64,62}; //1sp 2v cf
3681     cp =            {69,69,72,71,71,74,72,76,74,73,74,
3682
        -1,62,65,64,62,64,65,67,69,67,64,65,67,62,67,65,64,67,65,64,62,74,62,64,65,67,69,71,72,76,74,72,74,62,6

3683     v_type = {1, 0};
3684     borrowMode = 1;
3685     test_configuration();
3686     cout << "End test_3v_3sp_fig132." << endl;
3687 }
3688
3689 void FuxTest::test_3v_3sp_fig133(){
3690     cout << "Start test_3v_3sp_fig133" << endl;
3691     spList = {THIRD_SPECIES, FIRST_SPECIES};
3692     cantusFirmus =  {62,65,64,62,67,65,69,67,65,64,62}; //1sp 2v cf
3693     cp =
        {65,62,65,67,69,65,69,71,72,64,67,69,71,74,71,69,67,69,71,73,74,76,77,74,72,69,72,74,76,74,72,71,69,74,

3694                       50,50,48,55,52,50,53,48,50,57,50};
3695     v_type = {0, -3};
3696     borrowMode = 1;
3697     test_configuration();
3698     cout << "End test_3v_3sp_fig133." << endl;
3699 }
3700
3701 void FuxTest::test_3v_4sp_fig150(){
3702     cout << "Start test_3v_4sp_fig150" << endl;
3703     spList = {FOURTH_SPECIES, FIRST_SPECIES};
3704     cantusFirmus =  {65,67,69,65,62,64,65,72,69,65,67,65};
```

```
3705     cp =            {69,71,72,69,65,67,69,67,72,69,70,69,
3706                      65,65,64,64,60,60,57,57,62,62,60,60,65,65,64,64,60,60,65,65,64,65};
3707     v_type = {0, -3};
3708     borrowMode = 1;
3709     test_configuration();
3710     cout << "End test_3v_4sp_fig150." << endl;
3711 }
3712
3713 void FuxTest::test_3v_4sp_fig151(){
3714     cout << "Start test_3v_4sp_fig151" << endl;
3715     spList = {FIRST_SPECIES, FOURTH_SPECIES};
3716     cantusFirmus =  {65,67,69,65,62,64,65,72,69,65,67,65}; //1sp 2v cf
3717     cp =            {53,60,65,62,58,55,62,64,65,62,58,57,
3718                      53,53,52,52,50,50,46,46,43,-1,48,48,46,46,45,45,50,50,53,53,52,53};
3719     v_type = {0, -3};
3720     borrowMode = 1;
3721     test_configuration();
3722     cout << "End test_3v_4sp_fig151." << endl;
3723 }
3724
3725 void FuxTest::test_4v_1sp_fig171(){
3726     cout << "Start test_4v_1sp_fig171" << endl;
3727     spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
3728     cantusFirmus =  {65,67,69,65,62,64,65,72,69,65,67,65};
3729     cp =            {69,67,65,65,65,67,65,64,65,65,64,65,
3730                      60,60,60,62,62,58,57,57,53,57,60,57,
3731                      53,52,53,50,46,43,50,45,50,50,48,41};
3732     v_type = {3, 2, 3};
3733     borrowMode = 1;
3734     test_configuration();
3735     cout << "End test test_4v_1sp_fig171." << endl;
3736 }
3737
3738 void FuxTest::test_4v_1sp_fig172(){
3739     cout << "Start test_4v_1sp_fig172" << endl;
3740     spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
3741     cantusFirmus =  {65,67,69,65,62,64,65,72,69,65,67,65};
3742     cp =            {72,71,72,69,69,72,72,72,72,69,70,65,
3743                      69,67,64,65,65,67,69,67,65,65,64,65,
3744                      65,62,60,60,62,60,60,64,60,62,58,60};
3745     v_type = {3, 2, 3};
3746     borrowMode = 1;
3747     test_configuration();
3748     cout << "End test test_4v_1sp_fig172." << endl;
3749 }
3750
3751 void FuxTest::test_4v_2sp_fig175(){
3752     cout << "Start test_4v_2sp_fig175" << endl;
3753     spList = {FIRST_SPECIES, FIRST_SPECIES, SECOND_SPECIES};
3754     cantusFirmus =  {62,65,64,62,67,65,69,67,65,64,62};
3755     cp =            {74,74,72,74,76,77,77,76,74,73,74,
3756                      69,69,69,71,71,74,72,72,69,69,69,
3757                      -1,74,62,65,69,72,67,65,64,67,62,64,65,69,72,60,62,65,69,57,62};
3758     v_type = {3, 2, 3};
3759     borrowMode = 1;
3760     test_configuration();
3761     cout << "End test test_4v_2sp_fig175." << endl;
3762 }
3763
3764
3765 void FuxTest::test_4v_2sp_fig176(){
3766     cout << "Start test_4v_2sp_fig176" << endl;
3767     spList = {FIRST_SPECIES, SECOND_SPECIES, FIRST_SPECIES};
3768     cantusFirmus =  {50,   53,   52,   50,   55,   53,   57,   55,   53,   52,   50};
3769     cp =            {65,   69,   72,   74,   70,   69,   69,   71,   74,   73,   74,
3770                      -1,62,60,62,64,67,65,64,62,64,65,69,64,65,67,62,65,69,67,64,66,
3771                      69,   69,   67,   69,   70,   72,   72,   74,   69,   76,   69};
3772     v_type = {3, 2, 3};
3773     borrowMode = 1;
3774     test_configuration();
3775     cout << "End test test_4v_2sp_fig176." << endl;
3776 }
3777
3778 void FuxTest::test_4v_3sp_fig184(){
3779     cout << "Start test_4v_3sp_fig184" << endl;
3780     spList = {FIRST_SPECIES, THIRD_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
3781     cantusFirmus =  {76,72,74,72,69,81,79,76,77,76};
3782     cp =
    {68,71,64,67,69,72,69,67,65,62,64,65,67,65,64,62,60,72,69,67,65,67,69,71,72,74,76,71,72,69,71,72,69,74,

3783                      59,57,57,55,57,60,64,60,62,59,
3784                      64,65,62,64,65,65,64,69,62,64};
3785     v_type = {3, 2, 3};
3786     borrowMode = 1;
3787     test_configuration();
3788     cout << "End test test_4v_3sp_fig184." << endl;
3789 }
3790
3791 void FuxTest::test_4v_3sp_fig186(){
3792     cout << "Start test_4v_3sp_fig186" << endl;
```

```
3793     spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES, THIRD_SPECIES};
3794     cantusFirmus =  {64,60,62,60,57,69,67,64,65,64};
3795     cp =            {71,76,77,76,77,77,76,76,69,68,
3796                      68,69,69,72,72,72,72,72,74,71,
3797 
     76,74,72,71,69,67,65,64,62,64,65,62,69,72,69,67,65,67,69,67,65,67,69,71,72,64,65,67,69,67,65,64,62,65,6

3798     v_type = {3, 2, 3};
3799     borrowMode = 1;
3800     test_configuration();
3801     cout << "End test test_4v_3sp_fig186." << endl;
3802 }
3803 
3804 void FuxTest::test_4v_4sp_fig193(){
3805     cout << "Start test_4v_4sp_fig193" << endl;
3806     spList = {FOURTH_SPECIES, FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
3807     cantusFirmus =  {62,65,64,62,67,65,69,67,65,64,62};
3808     cp =            {69,69,74,74,72,72,71,71,74,74,69,69,77,77,76,76,74,74,73,74,
3809                      69,69,69,62,74,74,72,72,69,69,69,
3810                      50,50,45,47,43,50,53,48,50,45,50};
3811     v_type = {3, 2, 3};
3812     borrowMode = 1;
3813     test_configuration();
3814     cout << "End test test_4v_4sp_fig193." << endl;
3815 }
3816 
3817 void FuxTest::test_4v_4sp_fig196(){
3818     cout << "Start test_4v_4sp_fig196" << endl;
3819     spList = {FOURTH_SPECIES, FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
3820     cantusFirmus =  {62,65,64,62,67,65,69,67,65,64,62};
3821     cp =            {57,57,62,62,60,60,59,59,62,62,57,57,65,65,64,64,62,62,61,62,
3822                      65,69,69,62,74,74,72,72,69,69,69,
3823                      50,50,45,47,43,50,53,48,50,45,50};
3824     v_type = {3, 2, 3};
3825     borrowMode = 1;
3826     test_configuration();
3827     cout << "End test test_4v_4sp_fig196." << endl;
3828 }
3829 
3830 
3831 
3832 // ----- Util functions -----
3833 
3834 void printVector(const std::vector<int>& array) {
3835     for (size_t i = 0; i < array.size(); ++i) {
3836         std::cout << array[i] << " ";
3837     }
3838     std::cout << std::endl;
3839 }
3840 
3841 void printIntVarArray(const IntVarArray& array) {
3842     for (int i = 0; i < array.size(); ++i) {
3843         std::cout << array[i] << " ";
3844     }
3845     std::cout << std::endl;
3846 }
3847 
3848 std::vector<CounterpointProblem*> get_all_solutions(CounterpointProblem* problem) {
3849     std::vector<CounterpointProblem*> solutions;
3850     BAB<CounterpointProblem> e(problem);
3851     while (CounterpointProblem* pb = e.next()) {
3852         solutions.push_back(pb);
3853     }
3854     return solutions;
3855 }
3856 
3857 void FuxTest::test_configuration() {
3858     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
      specific_params, importance, borrowMode);
3859     for (size_t i = 0; i < cp.size(); i++) {
3860         int note = cp[i];
3861         if (note >  0) {
3862             rel(problem->getHome(), problem->getSolutionArray()[i], IRT_EQ, note);
3863         }
3864     }
3865     if (!has_solution(problem)) {
3866         std::cerr    << "/!\\ ERROR: It doesn't exist a solution but it should with the following
      configuration:" << endl;
3867     } else {
3868         cout << "\t Test passed with the following configuration:" << endl;
3869     }
3870     cout << "\t Cantus firmus: ";
3871     printVector(cantusFirmus);
3872     cout << "\t Solution array: ";
3873     printIntVarArray(problem->getSolutionArray());
3874     delete problem;
3875 }
3876 
3877 int FuxTest::get_motions(CounterpointProblem* problem, int i) {
3878     if (cp[i] > cp[i-1]) {
```

```
3879         if (problem->getSolutionArray()[i].val() > problem->getSolutionArray()[i-1].val()) {
3880             return 2;
3881         } else if (problem->getSolutionArray()[i].val() < problem->getSolutionArray()[i-1].val()
     ) {
3882             return 0;
3883         } else {
3884             return 1;
3885         }
3886     }
3887     if (cp[i] < cp[i-1]) {
3888         if (problem->getSolutionArray()[i].val() > problem->getSolutionArray()[i-1].val()) {
3889             return 0;
3890         } else if (problem->getSolutionArray()[i].val() < problem->getSolutionArray()[i-1].val()
     ) {
3891             return 2;
3892         } else {
3893             return 1;
3894         }
3895     }
3896     if (cp[i] == cp[i-1]) {
3897         if (problem->getSolutionArray()[i].val() == problem->getSolutionArray()[i-1].val()) {
3898             return 2;
3899         } else if (problem->getSolutionArray()[i].val() < problem->getSolutionArray()[i-1].val()
     ) {
3900             return 0;
3901         } else {
3902             return 1;
3903         }
3904     }
3905     return -1;
3906 }
3907
3908 // ================================================
3909
3910 CounterpointProblem* FuxTest::dispatcher(char* test){
3911     if(strcmp(test, "1H1")==0){
3912         return test_1sp_H1();
3913     } else if(strcmp(test, "1H2")==0){
3914         return test_1sp_H2();
3915     } else if(strcmp(test, "1H32")==0){
3916         return test_1sp_H3();
3917     } else if(strcmp(test, "1H33")==0){
3918         return test_1sp_H3_2();
3919     } else if(strcmp(test, "1H41")==0){
3920         return test_1sp_H4_1();
3921     } else if(strcmp(test, "1H42")==0){
3922         return test_1sp_H4_2();
3923     } else if(strcmp(test, "1H5")==0){
3924         return test_1sp_H5();
3925     } else if(strcmp(test, "1H7")==0){
3926         return test_1sp_H7();
3927     } else if(strcmp(test, "1H72")==0){
3928         return test_1sp_H7_2();
3929     } else if(strcmp(test, "2H2")==0){
3930         return test_2sp_H2();
3931     } else if(strcmp(test, "3H1")==0){
3932         return test_3sp_H1();
3933     } else if(strcmp(test, "4H2")==0){
3934         return test_4sp_H2();
3935     } else {
3936         std::invalid_argument("Test for constraint not found!");
3937         return nullptr;
3938     }
3939 }
3940
3941 FuxTest::FuxTest(int testNumber): FuxTest(testNumber, 0){
3942     idx = cp.size();
3943 }
3944
3945 FuxTest::FuxTest(int testNumber, int i){
3946     if(testNumber==22){
3947         test_2v_1sp_fig22_setter(i);
3948     } else if(testNumber==23){
3949         test_2v_1sp_fig23_setter(i);
3950     } else if(testNumber==38){
3951         test_2v_2sp_fig38_setter(i);
3952     } else if(testNumber==39){
3953         test_2v_2sp_fig39_setter(i);
3954     } else if(testNumber==40){
3955         test_2v_2sp_fig40_setter(i);
3956     } else if(testNumber==55){
3957         test_2v_3sp_fig55_setter(i);
3958     } else if(testNumber==74){
3959         test_2v_4sp_fig74_setter(i);
3960     }else if(testNumber==82){
3961         test_2v_5sp_fig82_setter(i);
3962     }else if(testNumber==108){
3963         test_3v_1sp_fig108_setter(i);
3964     } else if(testNumber==109){
3965         test_3v_1sp_fig109_setter(i);
```

```
3966     } else if(testNumber==110){
3967         test_3v_1sp_fig110_setter(i);
3968     } else if(testNumber==111){
3969         test_3v_1sp_fig111_setter(i);
3970     } else if(testNumber==125){
3971         test_3v_2sp_fig125_setter(i);
3972     } else if(testNumber==146){
3973         test_3v_4sp_fig146_setter(i);
3974     } else if(testNumber==166){
3975         test_4v_1sp_fig166_setter(i);
3976     } else if(testNumber==167){
3977         test_4v_1sp_fig167_setter(i);
3978     } else if(testNumber==176){
3979         test_4v_2sp_fig176_setter(i);
3980     }
3981     else {
3982         throw std::invalid_argument("This test number has not been implemented (yet)");
3983     }
3984 }
3985
3986 CounterpointProblem* FuxTest::test_1sp_H1(){
3987     spList = {FIRST_SPECIES};
3988     v_type = {2};
3989     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
          specific_params,
3990             importance, borrowMode);
3991     rel(problem->getHome(), problem->getSolutionArray()[1], IRT_EQ, 72); //does not work since
     it is not a consonant
3992     return problem;
3993 }
3994
3995 CounterpointProblem* FuxTest::test_1sp_H2(){
3996     spList = {FIRST_SPECIES};
3997     v_type = {2};
3998     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
          specific_params,
3999             importance, borrowMode);
4000     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 64);
4001     return problem;
4002 }
4003
4004 CounterpointProblem* FuxTest::test_1sp_H3(){
4005     spList = {FIRST_SPECIES};
4006     v_type = {2};
4007     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
          specific_params,
4008             importance, borrowMode);
4009     rel(problem->getHome(), problem->getSolutionArray()[problem->getSize()-1], IRT_EQ, 64);
4010     return problem;
4011 }
4012
4013 CounterpointProblem* FuxTest::test_1sp_H3_2(){
4014     spList = {FIRST_SPECIES, FIRST_SPECIES};
4015     v_type = {2, 1};
4016     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
          specific_params,
4017             importance, borrowMode);
4018     rel(problem->getHome(), problem->getSolutionArray()[(problem->getSize()/2)-1], IRT_EQ, 67);
4019     rel(problem->getHome(), problem->getSolutionArray()[(problem->getSize())-1], IRT_EQ, 65);
4020     return problem;
4021 }
4022
4023 CounterpointProblem* FuxTest::test_1sp_H4_1(){
4024     spList = {FIRST_SPECIES};
4025     v_type = {1};
4026     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
          specific_params,
4027             importance, borrowMode);
4028     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 67);
4029     return problem;
4030 }
4031
4032 CounterpointProblem* FuxTest::test_1sp_H4_2(){
4033     spList = {FIRST_SPECIES};
4034     v_type = {1};
4035     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
          specific_params,
4036             importance, borrowMode);
4037     rel(problem->getHome(), problem->getSolutionArray()[problem->getSize()-1], IRT_EQ, 67);
4038     return problem;
4039 }
4040
4041 CounterpointProblem* FuxTest::test_1sp_H5(){
4042     spList = {FIRST_SPECIES};
4043     v_type = {0};
4044     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
          specific_params,
4045             importance, borrowMode);
4046     rel(problem->getHome(), problem->getSolutionArray()[1], IRT_EQ, 62);
4047     return problem;
```

```
4048 }
4049
4050 CounterpointProblem* FuxTest::test_1sp_H7(){
4051     spList = {FIRST_SPECIES};
4052     v_type = {0};
4053     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
          specific_params,
4054             importance, borrowMode);
4055     rel(problem->getHome(), problem->getSolutionArray()[problem->getSize()-2], IRT_EQ, 69);
4056     return problem;
4057 }
4058
4059 CounterpointProblem* FuxTest::test_1sp_H7_2(){
4060     spList = {FIRST_SPECIES};
4061     v_type = {-1};
4062     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
          specific_params,
4063             importance, borrowMode);
4064     rel(problem->getHome(), problem->getSolutionArray()[problem->getSize()-2], IRT_EQ, 55);
4065     return problem;
4066 }
4067
4068 CounterpointProblem* FuxTest::test_2sp_H2(){
4069     spList = {SECOND_SPECIES};
4070     v_type = {2};
4071     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
          specific_params,
4072             importance, borrowMode);
4073     //these three lines are an example of the constraint allowing a dissonance
4074     //rel(problem->getHome(), problem->getSolutionArray()[2], IRT_EQ, 78);
4075     //rel(problem->getHome(), problem->getSolutionArray()[3], IRT_EQ, 79);
4076     //rel(problem->getHome(), problem->getSolutionArray()[4], IRT_EQ, 81);
4077     //these three lines are an example of the constraint blocking a dissonance
4078     rel(problem->getHome(), problem->getSolutionArray()[2], IRT_EQ, 74);
4079     rel(problem->getHome(), problem->getSolutionArray()[3], IRT_EQ, 79);
4080     rel(problem->getHome(), problem->getSolutionArray()[4], IRT_EQ, 81);
4081     return problem;
4082 }
4083
4084 CounterpointProblem* FuxTest::test_3sp_H1(){
4085     spList = {THIRD_SPECIES};
4086     v_type = {2};
4087     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
          specific_params,
4088             importance, borrowMode);
4089     rel(problem->getHome(), problem->getSolutionArray()[4], IRT_EQ, 66);
4090     rel(problem->getHome(), problem->getSolutionArray()[5], IRT_EQ, 66);
4091     rel(problem->getHome(), problem->getSolutionArray()[6], IRT_EQ, 67);
4092     rel(problem->getHome(), problem->getSolutionArray()[7], IRT_EQ, 69);
4093     rel(problem->getHome(), problem->getSolutionArray()[8], IRT_EQ, 69);
4094     return problem;
4095 }
4096
4097 CounterpointProblem* FuxTest::test_4sp_H2(){
4098     spList = {FOURTH_SPECIES};
4099     v_type = {-1};
4100     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
          specific_params,
4101             importance, borrowMode);
4102     rel(problem->getHome(), problem->getSolutionArray()[1], IRT_EQ, 52);
4103     rel(problem->getHome(), problem->getSolutionArray()[2], IRT_EQ, 52);
4104     return problem;
4105 }
4106
4107 vector<Species> FuxTest::getSpList(){
4108     return spList;
4109 }
4110
4111 vector<int> FuxTest::getCf(){
4112     return cantusFirmus;
4113 }
4114
4115 vector<int> FuxTest::getVType(){
4116     return v_type;
4117 }
4118
4119 int FuxTest::getBMode(){
4120     return borrowMode;
4121 }
4122
4123 vector<int> FuxTest::getCp(){
4124     return cp;
4125 }
4126
4127 int FuxTest::getIdx(){
4128     return idx;
4129 }
4130
4131 void FuxTest::test_2v_1sp_fig22_setter(int i){
4132     cout << "Fig. 22" << endl;
```

```
4133     spList = {FIRST_SPECIES};
4134     cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57}; //1sp 2v cf
4135     cp =           {69,64,67,65,64,72,69,71,71,69,68,69};
4136     v_type = {1};
4137     idx = i;
4138     borrowMode = 1;
4139 }
4140
4141 void FuxTest::test_2v_1sp_fig23_setter(int i){
4142     cout << "Fig. 23" << endl;
4143     spList = {FIRST_SPECIES};
4144     cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57}; //1sp 2v cf
4145     cp =           {57,57,55,53,52,52,50,48,55,57,56,57};
4146     v_type = {-1};
4147     idx = i;
4148     borrowMode = 1;
4149 }
4150
4151 void FuxTest::test_2v_2sp_fig38_setter(int i){
4152     cout << "Fig. 38" << endl;
4153     spList = {SECOND_SPECIES};
4154     cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53}; //1sp 2v cf
4155     cp =           {0,65,
4156                     64,62,
4157                     60,58,
4158                     57,55,
4159                     53,57,
4160                     60,58,
4161                     57,69,
4162                     67,64,
4163                     65,67,
4164                     69,65,
4165                     62,64,
4166                     65};
4167     v_type = {1};
4168     idx = i;
4169     borrowMode = 1;
4170 }
4171
4172 void FuxTest::test_2v_2sp_fig39_setter(int i){
4173     cout << "Fig. 39 (the voice types as defined previously don't work here)" << endl;
4174     spList = {SECOND_SPECIES};
4175     cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53}; //1sp 2v cf
4176     cp =           {0,53,52,48,53,52,50,48,46,58,55,60,57,53,52,48,53,41,45,50,48,52,53};
4177     v_type = {-1};
4178     idx = i;
4179     borrowMode = 1;
4180 }
4181
4182 void FuxTest::test_2v_2sp_fig40_setter(int i){
4183     cout << "Fig. 40" << endl;
4184     spList = {SECOND_SPECIES};
4185     cantusFirmus = {55,60,59,55,60,64,62,67,64,60,62,59,57,55}; //1sp 2v cf
4186     cp =
    {0,67,64,65,67,69,71,69,67,72,71,72,74,72,71,69,67,65,64,72,71,69,67,62,64,66,67};
4187     v_type = {2};
4188     idx = i;
4189     borrowMode = 1;
4190 }
4191
4192 void FuxTest::test_2v_3sp_fig55_setter(int i){
4193     cout << "Fig. 55" << endl;
4194     spList = {THIRD_SPECIES};
4195     cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62}; //1sp 2v cf
4196     cp =           {62,64,65,67,//ok
4197                     69,71,72,74,//ok
4198                     76,74,71,72,//ok
4199                     74,72,70,69,//ok
4200                     70,72,74,76,//ok
4201                     77,65,69,71,//ok, but check for maybe bemol on last note
4202                     72,69,70,72,//ok
4203                     70,69,67,70,//HERE FIRST NOTE ERROR
4204                     69,62,64,65,
4205                     67,69,71,73,
4206                     74};
4207     cout << "CP size : " << endl;
4208     cout << cp.size() << endl;
4209     v_type = {1};
4210     idx = i;
4211     borrowMode = 1;
4212 }
4213
4214 void FuxTest::test_2v_4sp_fig74_setter(int i){
4215     cout << "Fig. 74" << endl;
4216     spList = {FOURTH_SPECIES};
4217     cantusFirmus = {62,   65,   64,   62,   67,   65,   69,   67,   65,   64,   62}; //1sp 2v cf
4218     cp =           {   50,50,62,62,60,60,59,59,64,64,62,62,65,65,64,64,62,62,61,62};
4219     cout << cp.size() << endl;
4220     v_type = {-1};
4221     idx = i;
```

```
4222     borrowMode = 1;
4223 }
4224
4225 void FuxTest::test_2v_5sp_fig82_setter(int i){
4226     cout << "Fig. 82" << endl;
4227     spList = {FIFTH_SPECIES};
4228     cantusFirmus = {62,   65,   64,   62,   67,   65,   69,   67,   65,   64,   62}; //1sp 2v cf
4229     cp =           {0,0,69,69,69,62,64,65,67,65,64,67,65,62,
     74,74,74,72,70,67,69,71,72,72,72,72, 77,77,77,76,76,76,76,69, 74,74,74,74,73,73,74};
4230     cout << cp.size() << endl;
4231     v_type = {1};
4232     idx = i;
4233     borrowMode = 1;
4234 }
4235
4236 void FuxTest::test_3v_1sp_fig108_setter(int i){
4237     cout << "Fig. 108" << endl;
4238     spList = {FIRST_SPECIES, FIRST_SPECIES};
4239     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
4240     cp =           {67,64,65,69,72,72,76,72,69,68,
4241                     52,57,50,53,53,53,60,60,62,64};
4242     v_type = {1,-2};
4243     idx = i;
4244     borrowMode = 1;
4245 }
4246
4247 void FuxTest::test_3v_1sp_fig109_setter(int i){
4248     cout << "Fig. 109" << endl;
4249     spList = {FIRST_SPECIES, FIRST_SPECIES};
4250     cantusFirmus = {52,48,50,48,45,57,55,52,53,52};
4251     cp =           {67,64,65,64,64,64,67,67,62,64,
4252                     59,60,57,57,60,60,59,59,57,59};
4253     v_type = {2,0};
4254     idx = i;
4255     borrowMode = 1;
4256 }
4257
4258 void FuxTest::test_3v_1sp_fig110_setter(int i){
4259     cout << "Fig. 110" << endl;
4260     spList = {FIRST_SPECIES, FIRST_SPECIES};
4261     cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
4262     cp =           {60,60,60,62,58,59,57,69,65,62,64,65,
4263                     53,52,53,50,55,55,53,45,50,50,48,41};
4264     v_type = {-1 ,-3};
4265     idx = i;
4266     borrowMode = 1;
4267 }
4268
4269 void FuxTest::test_3v_1sp_fig111_setter(int i){
4270     cout << "Fig. 111" << endl;
4271     spList = {FIRST_SPECIES, FIRST_SPECIES};
4272     cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
4273     cp =           {69,72,72,74,74,70,69,67,65,77,76,77,
4274                     65,64,65,62,58,55,53,52,53,50,48,53};
4275     v_type = {1 ,-2};
4276     idx = i;
4277     borrowMode = 1;
4278 }
4279
4280 void FuxTest::test_3v_2sp_fig125_setter(int i){
4281     cout << "Fig. 125" << endl;
4282     spList = {FIRST_SPECIES, SECOND_SPECIES};
4283     cantusFirmus = {52,   48,   50,   48,   45,   57,   55,   52,   53,   52};
4284     cp =           {64,   64,   65,   64,   65,   64,   67,   67,   69,   68,
4285                     00,59,60,59,57,53,55,52,57,59,60,57,59,62,64,62,60,57,59};
4286     v_type = {2 ,1};
4287     idx = i;
4288     borrowMode = 1;
4289 }
4290
4291 void FuxTest::test_3v_4sp_fig146_setter(int i){
4292     cout << "Fig. 146" << endl;
4293     spList = {FOURTH_SPECIES, FIRST_SPECIES};
4294     cantusFirmus = {64,   60,   62,   60,   57,   69,   67,   64,   65,   64};
4295     cp =           {   76,76,72,72,71,71,69,69,72,72,74,74,72,72,69,69,71,68,
4296                     52,   57,   55,   57,   53,   53,   52,   48,   50,   52};
4297     v_type = {1 ,-2};
4298     idx = i;
4299     borrowMode = 1;
4300 }
4301
4302 void FuxTest::test_4v_1sp_fig166_setter(int i){
4303     cout << "Fig. 166" << endl;
4304     spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
4305     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
4306     cp =           {59,57,57,57,60,60,64,60,62,59,
4307                     56,57,53,52,52,53,60,60,57,56,
4308                     52,53,50,45,45,41,40,45,50,52};
4309     v_type = {-1 ,-1, -3};
4310     idx = i;
```

```
4311     borrowMode = 1;
4312 }
4313
4314 void FuxTest::test_4v_1sp_fig167_setter(int i){
4315     cout << "Fig. 167" << endl;
4316     spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
4317     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
4318     cp =           {68,69,65,64,64,65,72,72,69,68,
4319                     59,57,57,57,60,60,64,60,62,59,
4320                     52,53,50,45,45,53,52,57,50,52};
4321     v_type = {0 ,-1, -3};
4322     idx = i;
4323     borrowMode = 1;
4324 }
4325
4326 void FuxTest::test_4v_2sp_fig176_setter(int i){
4327     cout << "Fig. 173" << endl;
4328     spList = {SECOND_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
4329     cantusFirmus = {50,   53,   52,   50,   55,   53,   57,   55,   53,   52,   50};
4330     cp =           {00,62,60,62,64,67,65,64,62,64,65,69,64,65,67,62,65,69,67,64,66,
4331                     65,   69,   72,   74,   70,   69,   69,   71,   74,   73,   74,
4332                     57,   57,   55,   57,   58,   60,   60,   62,   57,   64,   57};
4333     v_type = {2, 3, 1};
4334     idx = i;
4335     borrowMode = 1;
4336 }
4337
4338
4339 //=============================================================================================
4340 //=============================================================================================
4341
4342 void FuxTest::test_gen_2v_1sp(){
4343     cout << "===== test_gen 2v 1sp =====" << endl;
4344     spList = {FIRST_SPECIES};
4345     v_type = {0};
4346
4347     std::fill(activeConstraints.begin(), activeConstraints.end(), true);
4348     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
         specific_params, importance, borrowMode);
4349
4350     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 60);
4351     auto home = problem->getHome();
4352
4353
4354     BAB<CounterpointProblem> e(problem);
4355     CounterpointProblem* best = nullptr;
4356     int myCount = 0;
4357     while (CounterpointProblem* s = e.next()) {
4358         delete best;      // keep only the best-so-far
4359         best = s;
4360         myCount++;
4361     }
4362
4363     if (best) {
4364         std::cout << "BEST (" << myCount << "): " << best->getSolutionArray() << std::endl;
4365         delete best;
4366     }
4367
4368     delete problem;
4369 }
4370
4371 void FuxTest::test_gen_2v_2sp(){
4372     cout << "===== test_gen 2v 2sp =====" << endl;
4373     spList = {SECOND_SPECIES};
4374     v_type = {1};
4375
4376     std::fill(activeConstraints.begin(), activeConstraints.end(), true);
4377     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
         specific_params, importance, borrowMode);
4378
4379     // Additional and necessary Constrains
4380     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 60);
4381     auto home = problem->getHome();
4382
4383
4384     // BAB
4385     BAB<CounterpointProblem> e(problem);
4386     CounterpointProblem* best = nullptr;
4387     int myCount = 0;
4388     while (CounterpointProblem* s = e.next()) {
4389         if (myCount%10 == 0){
4390             std::cout << "BAB " << myCount << " : " << s->getSolutionArray() << std::endl;
4391         }
4392         delete best;      // keep only the best-so-far
4393         best = s;
4394         myCount++;
4395     }
4396     if (best) {
4397         std::cout << "BEST (" << myCount << "): " << best->getSolutionArray() << std::endl;
4398         delete best;
```

```
4399     }
4400
4401     delete problem;
4402 }
4403
4404 void FuxTest::test_gen_2v_3sp(){
4405     cout << "===== test_gen 2v 3sp =====" << endl;
4406     spList = {THIRD_SPECIES};
4407     v_type = {1};
4408
4409     std::fill(activeConstraints.begin(), activeConstraints.end(), true);
4410     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
          specific_params, importance, borrowMode);
4411
4412     // Additional and necessary Constrains
4413     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 60);
4414     auto home = problem->getHome();
4415
4416
4417     // BAB
4418     BAB<CounterpointProblem> e(problem);
4419     CounterpointProblem* best = nullptr;
4420     int myCount = 0;
4421     while (CounterpointProblem* s = e.next()) {
4422         if (myCount%10 == 0){
4423             std::cout << "BAB " << myCount << " : " << s->getSolutionArray() << std::endl;
4424         }
4425         delete best;      // keep only the best-so-far
4426         best = s;
4427         myCount++;
4428     }
4429     if (best) {
4430         std::cout << "BEST (" << myCount << "): " << best->getSolutionArray() << std::endl;
4431         delete best;
4432     }
4433
4434     delete problem;
4435 }
4436
4437 void FuxTest::test_gen_2v_4sp(){
4438     cout << "===== test_gen 2v 4sp =====" << endl;
4439     spList = {FOURTH_SPECIES};
4440     v_type = {0};
4441
4442     std::fill(activeConstraints.begin(), activeConstraints.end(), true);
4443     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
          specific_params, importance, borrowMode);
4444
4445     // Additional and necessary Constrains
4446     auto home = problem->getHome();
4447
4448
4449     // BAB
4450     BAB<CounterpointProblem> e(problem);
4451     CounterpointProblem* best = nullptr;
4452     int myCount = 0;
4453     while (CounterpointProblem* s = e.next()) {
4454         if (myCount%5 == 0){
4455             std::cout << "BAB " << myCount << " : " << s->getSolutionArray() << std::endl;
4456         }
4457         delete best;      // keep only the best-so-far
4458         best = s;
4459         myCount++;
4460     }
4461     if (best) {
4462         std::cout << "BEST (" << myCount << "): " << best->getSolutionArray() << std::endl;
4463         delete best;
4464     }
4465
4466     delete problem;
4467 }
4468
4469 void FuxTest::test_gen_3v_1sp(){
4470     cout << "===== test_gen 3v 1sp =====" << endl;
4471     spList = {FIRST_SPECIES, FIRST_SPECIES};
4472     v_type = {0, 1};
4473
4474     std::fill(activeConstraints.begin(), activeConstraints.end(), true);
4475     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
          specific_params, importance, borrowMode);
4476
4477     // Additional and necessary Constrains
4478     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 60);
4479     auto home = problem->getHome();
4480
4481
4482     // BAB
4483     BAB<CounterpointProblem> e(problem);
4484     CounterpointProblem* best = nullptr;
4485     int myCount = 0;
```

```
4486     while (CounterpointProblem* s = e.next()) {
4487         if (myCount%100 == 0){
4488             std::cout << "BAB " << myCount << " : " << s->getSolutionArray() << std::endl;
4489         }
4490         delete best;      // keep only the best-so-far
4491         best = s;
4492         myCount++;
4493     }
4494 
4495     if (best) {
4496         std::cout << "BEST (" << myCount << "): " << best->getSolutionArray() << std::endl;
4497         delete best;
4498     }
4499 
4500     delete problem;
4501 }
4502 
4503 void FuxTest::test_gen_3v_2sp(){
4504     cout << "===== test_gen 3v 2sp =====" << endl;
4505     spList = {FIRST_SPECIES, SECOND_SPECIES};
4506     v_type = {0, 1};
4507 
4508     std::fill(activeConstraints.begin(), activeConstraints.end(), true);
4509     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
          specific_params, importance, borrowMode);
4510 
4511     // Additional and necessary Constrains
4512     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 60);
4513     auto home = problem->getHome();
4514 
4515 
4516     // BAB
4517     BAB<CounterpointProblem> e(problem);
4518     CounterpointProblem* best = nullptr;
4519     int myCount = 0;
4520     while (CounterpointProblem* s = e.next()) {
4521         if (myCount%100 == 0){
4522             std::cout << "BAB " << myCount << " : " << s->getSolutionArray() << std::endl;
4523         }
4524         delete best;      // keep only the best-so-far
4525         best = s;
4526         myCount++;
4527     }
4528 
4529     if (best) {
4530         std::cout << "BEST (" << myCount << "): " << best->getSolutionArray() << std::endl;
4531         delete best;
4532     }
4533 
4534     delete problem;
4535 }
4536 
4537 void FuxTest::test_gen_3v_2sp_2sp(){
4538     cout << "===== test_gen 3v 2sp 2sp =====" << endl;
4539     spList = {SECOND_SPECIES, SECOND_SPECIES};
4540     v_type = {0, 1};
4541 
4542     std::fill(activeConstraints.begin(), activeConstraints.end(), true);
4543     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
          specific_params, importance, borrowMode);
4544 
4545     // Additional and necessary Constrains
4546     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 60);
4547     auto home = problem->getHome();
4548 
4549 
4550     // BAB
4551     BAB<CounterpointProblem> e(problem);
4552     CounterpointProblem* best = nullptr;
4553     int myCount = 0;
4554     while (CounterpointProblem* s = e.next()) {
4555         if (myCount%100 == 0){
4556             std::cout << "BAB " << myCount << " : " << s->getSolutionArray() << std::endl;
4557         }
4558         delete best;      // keep only the best-so-far
4559         best = s;
4560         myCount++;
4561     }
4562 
4563     if (best) {
4564         std::cout << "BEST (" << myCount << "): " << best->getSolutionArray() << std::endl;
4565         delete best;
4566     }
4567 
4568     delete problem;
4569 }
4570 
4571 void FuxTest::test_gen_3v_3sp(){
4572     cout << "===== test_gen 3v 3sp =====" << endl;
4573     spList = {FIRST_SPECIES, THIRD_SPECIES};
```

```
4574     v_type = {0, 1};
4575
4576     std::fill(activeConstraints.begin(), activeConstraints.end(), true);
4577     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
          specific_params, importance, borrowMode);
4578
4579     // Additional and necessary Constrains
4580     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 60);
4581     auto home = problem->getHome();
4582
4583
4584     // BAB
4585     BAB<CounterpointProblem> e(problem);
4586     CounterpointProblem* best = nullptr;
4587     int myCount = 0;
4588     while (CounterpointProblem* s = e.next()) {
4589         if (myCount%1000 == 0){
4590             std::cout << "BAB " << myCount << " : " << s->getSolutionArray() << std::endl;
4591         }
4592         delete best;      // keep only the best-so-far
4593         best = s;
4594         myCount++;
4595     }
4596
4597     if (best) {
4598         std::cout << "BEST (" << myCount << "): " << best->getSolutionArray() << std::endl;
4599         delete best;
4600     }
4601
4602     delete problem;
4603 }
4604
4605 void FuxTest::test_gen_3v_3sp_3sp(){
4606     cout << "===== test_gen 3v 3sp 3sp =====" << endl;
4607     spList = {THIRD_SPECIES, THIRD_SPECIES};
4608     v_type = {0, 1};
4609
4610     std::fill(activeConstraints.begin(), activeConstraints.end(), true);
4611     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
          specific_params, importance, borrowMode);
4612
4613     // Additional and necessary Constrains
4614     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 60);
4615     auto home = problem->getHome();
4616
4617
4618     // BAB
4619     BAB<CounterpointProblem> e(problem);
4620     CounterpointProblem* best = nullptr;
4621     int myCount = 0;
4622     while (CounterpointProblem* s = e.next()) {
4623         if (myCount%100 == 0){
4624             std::cout << "BAB " << myCount << " : " << s->getSolutionArray() << std::endl;
4625         }
4626         delete best;      // keep only the best-so-far
4627         best = s;
4628         myCount++;
4629     }
4630
4631     if (best) {
4632         std::cout << "BEST (" << myCount << "): " << best->getSolutionArray() << std::endl;
4633         delete best;
4634     }
4635
4636     delete problem;
4637 }
4638
4639 void FuxTest::test_gen_3v_4sp(){
4640     cout << "===== test_gen 3v 4sp =====" << endl;
4641     spList = {FIRST_SPECIES, FOURTH_SPECIES};
4642     v_type = {0, 1};
4643
4644     std::fill(activeConstraints.begin(), activeConstraints.end(), true);
4645     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
          specific_params, importance, borrowMode);
4646
4647     // Additional and necessary Constrains
4648     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 60);
4649     auto home = problem->getHome();
4650
4651
4652     // BAB
4653     BAB<CounterpointProblem> e(problem);
4654     CounterpointProblem* best = nullptr;
4655     int myCount = 0;
4656     while (CounterpointProblem* s = e.next()) {
4657         if (myCount%10 == 0){
4658             std::cout << "BAB " << myCount << " : " << s->getSolutionArray() << std::endl;
4659         }
4660         delete best;      // keep only the best-so-far
```

```
4661         best = s;
4662         myCount++;
4663     }
4664 
4665     if (best) {
4666         std::cout << "BEST (" << myCount << "): " << best->getSolutionArray() << std::endl;
4667         delete best;
4668     }
4669 
4670     delete problem;
4671 }
4672 
4673 void FuxTest::test_gen_4v_1sp(){
4674     cout << "===== test_gen 4v 1sp =====" << endl;
4675     spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
4676     v_type = {0, 1, 2};
4677 
4678     std::fill(activeConstraints.begin(), activeConstraints.end(), true);
4679     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
         specific_params, importance, borrowMode);
4680 
4681     // Additional and necessary Constrains
4682     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 60);
4683     auto home = problem->getHome();
4684 
4685 
4686     // BAB
4687     BAB<CounterpointProblem> e(problem);
4688     CounterpointProblem* best = nullptr;
4689     int myCount = 0;
4690     while (CounterpointProblem* s = e.next()) {
4691         if (myCount%1000 == 0){
4692             std::cout << "BAB " << myCount << " : " << s->getSolutionArray() << std::endl;
4693         }
4694         delete best;      // keep only the best-so-far
4695         best = s;
4696         myCount++;
4697     }
4698 
4699     if (best) {
4700         std::cout << "BEST (" << myCount << "): " << best->getSolutionArray() << std::endl;
4701         delete best;
4702     }
4703 
4704     delete problem;
4705 }
4706 
4707 void FuxTest::test_gen_4v_2sp(){
4708     cout << "===== test_gen 4v 2sp =====" << endl;
4709     spList = {FIRST_SPECIES, FIRST_SPECIES, SECOND_SPECIES};
4710     v_type = {0, 1, 2};
4711 
4712     std::fill(activeConstraints.begin(), activeConstraints.end(), true);
4713     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
         specific_params, importance, borrowMode);
4714 
4715     // Additional and necessary Constrains
4716     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 60);
4717     auto home = problem->getHome();
4718 
4719 
4720     // BAB
4721     BAB<CounterpointProblem> e(problem);
4722     CounterpointProblem* best = nullptr;
4723     int myCount = 0;
4724     while (CounterpointProblem* s = e.next()) {
4725         if (myCount%10000== 0){
4726             std::cout << "BAB " << myCount << " : " << s->getSolutionArray() << std::endl;
4727         }
4728         delete best;      // keep only the best-so-far
4729         best = s;
4730         myCount++;
4731     }
4732 
4733     if (best) {
4734         std::cout << "BEST (" << myCount << "): " << best->getSolutionArray() << std::endl;
4735         delete best;
4736     }
4737 
4738     delete problem;
4739 }
4740 
4741 void FuxTest::test_gen_4v_2sp_2sp_2sp(){
4742     cout << "===== test_gen 4v 2sp full =====" << endl;
4743     spList = {SECOND_SPECIES, SECOND_SPECIES, SECOND_SPECIES};
4744     v_type = {0, 1, 2};
4745 
4746     std::fill(activeConstraints.begin(), activeConstraints.end(), true);
4747     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
         specific_params, importance, borrowMode);
```

```
4748
4749     // Additional and necessary Constrains
4750     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 60);
4751     auto home = problem->getHome();
4752
4753
4754     // BAB
4755     BAB<CounterpointProblem> e(problem);
4756     CounterpointProblem* best = nullptr;
4757     int myCount = 0;
4758     while (CounterpointProblem* s = e.next()) {
4759         if (myCount%1000 == 0){
4760             std::cout << "BAB " << myCount << " : " << s->getSolutionArray() << std::endl;
4761         }
4762         delete best;      // keep only the best-so-far
4763         best = s;
4764         myCount++;
4765     }
4766
4767     if (best) {
4768         std::cout << "BEST (" << myCount << "): " << best->getSolutionArray() << std::endl;
4769         delete best;
4770     }
4771
4772     delete problem;
4773 }
4774
4775 void FuxTest::test_gen_4v_3sp(){
4776     cout << "===== test_gen 4v 3sp =====" << endl;
4777     spList = {SECOND_SPECIES, SECOND_SPECIES, THIRD_SPECIES};
4778     v_type = {0, 1, 2};
4779
4780     std::fill(activeConstraints.begin(), activeConstraints.end(), true);
4781     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
      specific_params, importance, borrowMode);
4782
4783     // Additional and necessary Constrains
4784     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 60);
4785     auto home = problem->getHome();
4786
4787
4788     // BAB
4789     BAB<CounterpointProblem> e(problem);
4790     CounterpointProblem* best = nullptr;
4791     int myCount = 0;
4792     while (CounterpointProblem* s = e.next()) {
4793         if (myCount%100 == 0){
4794             std::cout << "BAB " << myCount << " : " << s->getSolutionArray() << std::endl;
4795         }
4796         delete best;      // keep only the best-so-far
4797         best = s;
4798         myCount++;
4799     }
4800
4801     if (best) {
4802         std::cout << "BEST (" << myCount << "): " << best->getSolutionArray() << std::endl;
4803         delete best;
4804     }
4805
4806     delete problem;
4807 }
4808
4809 void FuxTest::test_gen_4v_4sp(){
4810     cout << "===== test_gen 4v 4sp =====" << endl;
4811     spList = {SECOND_SPECIES, SECOND_SPECIES, FOURTH_SPECIES};
4812     v_type = {0, 1, 2};
4813
4814     std::fill(activeConstraints.begin(), activeConstraints.end(), true);
4815     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
      specific_params, importance, borrowMode);
4816
4817     // Additional and necessary Constrains
4818     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 60);
4819     auto home = problem->getHome();
4820
4821
4822     // BAB
4823     BAB<CounterpointProblem> e(problem);
4824     CounterpointProblem* best = nullptr;
4825     int myCount = 0;
4826     while (CounterpointProblem* s = e.next()) {
4827         if (myCount%100 == 0){
4828             std::cout << "BAB " << myCount << " : " << s->getSolutionArray() << std::endl;
4829         }
4830         delete best;      // keep only the best-so-far
4831         best = s;
4832         myCount++;
4833     }
4834
4835     if (best) {
```

```
4836         std::cout << "BEST (" << myCount << "): " << best->getSolutionArray() << std::endl;
4837         delete best;
4838     }
4839 
4840     delete problem;
4841 }
4842 
4843 //================================================================================
4844 
4845 void FuxTest::test_gen_constrains_check(Species species, int n_voices, vector<int> v_types,
     vector<int> v_types_1sp){
4846     cout << "===== test_gen constrains check " << n_voices << "v " << species+1 << "sp " << "
     =====" << endl;
4847     spList = {species};
4848     int n_types_4v = 1;
4849     int n_types_3v = 1;
4850     switch(n_voices) {
4851         case 4:
4852             spList = {FIRST_SPECIES, FIRST_SPECIES, species};
4853             n_types_3v = v_types_1sp.size();
4854             n_types_4v = v_types_1sp.size();
4855             break;
4856         case 3:
4857             spList = {FIRST_SPECIES, species};
4858             n_types_3v = v_types_1sp.size();
4859             break;
4860         default: // 2 voices
4861             break;;
4862     }
4863 
4864     int const_from = 52;
4865     int const_to = 61;
4866     switch(species) {
4867         case 3:
4868             const_from = 62;
4869             const_to = 76;
4870             break;
4871         case 4:
4872             const_from = 77;
4873             const_to = 91;
4874             break;
4875         case 5:
4876             const_from = 92;
4877             const_to = 113;
4878             break;
4879         default: // 2d species
4880             break;;
4881     }
4882 
4883     int timeout_value = 300000; // ms --> 5min
4884     for (int vidx_1sp_2 = 0; vidx_1sp_2 < n_types_4v; vidx_1sp_2++){
4885         for (int vidx_1sp_1 = 0; vidx_1sp_1 < n_types_3v; vidx_1sp_1++){
4886             for (int vidx_species = 0; vidx_species < v_types.size(); vidx_species++){
4887                 switch(n_voices) {
4888                     case 4:
4889                         v_type = {v_types_1sp.at(vidx_1sp_2), v_types_1sp.at(vidx_1sp_1),
     v_types.at(vidx_species)};
4890                         cout << "===== Testing v_type = " << v_types_1sp.at(vidx_1sp_2) << ","
     << v_types_1sp.at(vidx_1sp_1) << "," << v_types.at(vidx_species) << endl;
4891                         break;
4892                     case 3:
4893                         v_type = {v_types_1sp.at(vidx_1sp_1), v_types.at(vidx_species)};
4894                         cout << "===== Testing v_type = " << v_types_1sp.at(vidx_1sp_1) << ","
     << v_types.at(vidx_species) << endl;
4895                         break;
4896                     default: // 2 voices
4897                         v_type = {v_types.at(vidx_species)};
4898                         cout << "===== Testing v_type = " << v_types.at(vidx_species) << endl;
4899                         break;;
4900                 }
4901 
4902                 std::fill(activeConstraints.begin(), activeConstraints.end(), true);
4903                 cout << "=== Testing with all constrains" << endl;
4904 
4905                 auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
4906 
4907                 Search::Options opt;
4908                 Search::TimeStop ts(timeout_value);
4909                 opt.stop = &ts;
4910                 BAB<CounterpointProblem> e(problem, opt);
4911                 CounterpointProblem* best = nullptr;
4912 
4913                 while (CounterpointProblem* s = e.next()) {
4914                     cout << "BAB ok" << std::endl;
4915 
4916                     delete best; // keep only the best-so-far
4917                     best = s;
4918 
4919                     break;
```

```
4920                 }
4921                 delete problem;
4922                 if (best) continue; // If already ok, don't test constrains
4923 
4924                 for(int constrain=const_from; constrain <= const_to; constrain++){
4925                     std::fill(activeConstraints.begin(), activeConstraints.end(), true);
4926                     activeConstraints[constrain] = false;
4927                     cout << "=== Testing without constrain " << constrain << endl;
4928 
4929                     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
     general_params, specific_params, importance, borrowMode);
4930 
4931                     Search::Options opt;
4932                     Search::TimeStop ts(timeout_value);
4933                     opt.stop = &ts;
4934                     BAB<CounterpointProblem> e(problem, opt);
4935                     CounterpointProblem* best = nullptr;
4936 
4937                     while (CounterpointProblem* s = e.next()) {
4938                         cout << "BAB ok" << std::endl;
4939 
4940                         delete best; // keep only the best-so-far
4941                         best = s;
4942 
4943                         break;
4944                     }
4945 
4946                     delete problem;
4947                 }
4948             }
4949         }
4950     }
4951 }
4952 
4953 void FuxTest::test_gen_custom(){ // Generic tests
4954     cout << "===== test generation custom =====" << endl;
4955 
4956     // 2-voices
4957     for (int sp = 1; sp < 6; sp++){
4958         for (int v_type = 1; v_type < 4; v_type++){
4959             GenerationInput input;
4960             input.cf_notes = {48, 50, 53, 52, 55, 53, 52, 50, 48};
4961             input.species = {int_to_species(sp)};
4962             input.v_type = {v_type};
4963             input.verbose = true;
4964             input.stagnation_ms = 0;
4965             input.timeout_ms = 360000; // 6min
4966             input.output_subdir = "";
4967             input.preset_name = "default";
4968             generate_counterpoint(input);
4969         }
4970     }
4971 
4972     // 3-voices
4973     for (int sp = 1; sp < 6; sp++){
4974         for (int v_type = 1; v_type < 4; v_type++){
4975             for(int sec_type = 1; sec_type < 4; sec_type++){
4976                 GenerationInput input;
4977                 input.cf_notes = {48, 50, 53, 52, 55, 53, 52, 50, 48};
4978                 input.species = {FIRST_SPECIES, int_to_species(sp)};
4979                 input.v_type = {sec_type, v_type};
4980                 input.verbose = true;
4981                 input.stagnation_ms = 0;
4982                 input.timeout_ms = 360000; // 6min
4983                 input.output_subdir = "";
4984                 input.preset_name = "default";
4985                 generate_counterpoint(input);
4986             }
4987         }
4988     }
4989 
4990     // 4-voices
4991     vector<pair<int,int>> cp_types = {{1,1},{1,2},{1,3},{2,2},{2,3},{3,3}};
4992 
4993     for (int sp = 1; sp < 6; sp++){
4994         for (int main_type = 1; main_type < 4; main_type++){
4995             for(int cp_type_idx = 0; cp_type_idx < cp_types.size(); cp_type_idx++){
4996                 pair<int,int> local_types = cp_types[cp_type_idx];
4997 
4998                 GenerationInput input;
4999                 input.cf_notes = {48, 50, 53, 52, 55, 53, 52, 50, 48};
5000                 input.species = {FIRST_SPECIES, FIRST_SPECIES, int_to_species(sp)};
5001                 input.v_type = {local_types.first, local_types.second, main_type};
5002                 input.verbose = true;
5003                 input.stagnation_ms = 0;
5004                 input.timeout_ms = 360000; // 6min
5005                 input.output_subdir = "";
5006                 input.preset_name = "default";
5007                 generate_counterpoint(input);
5008             }
```

```
5009         }
5010     }
5011
5012 }
5013
5014 void FuxTest::test_gen_classic(){
5015     cout << "===== test generation classic =====" << endl;
5016     test_gen_2v_1sp();
5017     test_gen_2v_2sp();
5018     test_gen_2v_3sp();
5019     test_gen_3v_1sp();
5020     test_gen_3v_2sp();
5021     test_gen_3v_3sp();
5022     test_gen_4v_1sp();
5023     test_gen_4v_2sp();
5024 }
5025
5026 void FuxTest::test_gen_multi(){
5027     cout << "===== test generation full sp =====" << endl;
5028     test_gen_3v_2sp_2sp();
5029     test_gen_3v_3sp_3sp();
5030     test_gen_4v_2sp_2sp_2sp();
5031 }
5032
5033 void FuxTest::test_gen_all(){
5034     cout << "===== test generation all =====" << endl;
5035     test_gen_classic();
5036     test_gen_multi();
5037 }
5038
5039 void FuxTest::test_sacha(){
5040     cout << "===== sacha 1v 1sp =====" << endl;
5041     cantusFirmus = {60,   62,   65,   64,   67,   65,   64,   62,   60};
5042     cfSize = cantusFirmus.size();
5043     melodic_params = {0, 1, 2, 576, 5, 10, 25, 40};
5044     general_params = {4, 1, 1, 2, 2, 2, 8, 1};
5045     specific_params = {8 , 4 , 0 , 2 , 1 , 8 , 50};
5046     importance = {8,7,5,2,9,3,14,12,6,11,4,10,1,13};
5047     borrowMode = 1;
5048
5049     spList = {FIRST_SPECIES};
5050     v_type = {1};
5051
5052     std::fill(activeConstraints.begin(), activeConstraints.end(), true);
5053     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params,
      specific_params, importance, borrowMode);
5054
5055     cout << "- Problem defined" << endl;
5056
5057     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 60);
5058
5059     cout << "- Getting solutions" << endl;
5060
5061     BAB<CounterpointProblem> e(problem);
5062     CounterpointProblem* best = nullptr;
5063     while (CounterpointProblem* s = e.next()) {
5064         delete best;      // keep only the best-so-far
5065         best = s;
5066     }
5067
5068     if (best) {
5069         auto solutionArray = best->getSolutionArray();
5070         std::vector<int> solVec;
5071         for (int i = 0; i < solutionArray.size(); ++i) {
5072             solVec.push_back(solutionArray[i].val());
5073         }
5074         std::vector<int> cfVec;
5075         for (int i = 0; i < cantusFirmus.size(); ++i) {
5076             cfVec.push_back(cantusFirmus[i]);
5077         }
5078
5079         // 4. Appeler saveMidi avec les vecteurs
5080         saveMidi("test_sacha.mid", cfVec, solVec, FIRST_SPECIES);
5081         std::cout << "BEST:\n" << best->to_string() << std::endl;
5082         delete best;
5083     }
5084
5085     delete problem;
5086
5087
5088    cout << "===== test_sacha 4v 3sp =====" << endl;
5089    //test_sacha_4v_3sp();
5090
5091    // test_gen_4v_3sp();
5092 }
```

A.10.2 figureTests.cpp

```cpp
#include "../../headers/Tests/figureTests.hpp"
#include <iostream>
#include <queue>

using namespace std;

FigureTests::FigureTests() {
    borrowMode = 1;
    melodic_params = {0, 1, 1, 576, 2, 2, 2, 1};
    general_params = {4, 1, 1, 2, 2, 2, 8, 1};
    specific_params = {8, 4, 0, 2, 1, 8, 50};
    importance = {8,7,5,2,9,3,14,12,6,11,4,10,1,13};
    notesSpeciesFor5sp = {};
}

CounterpointProblem* FigureTests::set_configuration() {
    // create a new problem
    auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
    // set the solution array
    for (size_t i = 0; i < cp.size(); i++) {
        int note = cp[i];
        if (note >  0) {
            rel(problem->getHome(), problem->getSolutionArray()[i], IRT_EQ, note);
        }
    }
    // set the species array for the 5th sepcies counterpoint
    if (spList[0] == FIFTH_SPECIES) {
        auto* part = problem->getCounterpoint_1();
        for (int i = 0; i < notesSpeciesFor5sp.size(); i++) {
            int sp = notesSpeciesFor5sp[i];
            if (sp > -1) {
                rel(problem->getHome(), part->getSpeciesArray()[i], IRT_EQ, sp);
            }

        }
    }
    if (spList.size() > 1 && spList[1] == FIFTH_SPECIES) {
        auto* part = problem->getCounterpoint_2();
        for (int i = 0; i < notesSpeciesFor5sp.size(); i++) {
            int sp = notesSpeciesFor5sp[i];
            if (sp > -1) {
                rel(problem->getHome(), part->getSpeciesArray()[i], IRT_EQ, sp);
            }
        }
    }
    if (spList.size() > 2 && spList[2] == FIFTH_SPECIES) {
        auto* part = problem->getCounterpoint_3();
        for (int i = 0; i < notesSpeciesFor5sp.size(); i++) {
            int sp = notesSpeciesFor5sp[i];
            if (sp > -1) {
                rel(problem->getHome(), part->getSpeciesArray()[i], IRT_EQ, sp);
            }
        }
    }
    return problem;
}

std::vector<CounterpointProblem*> FigureTests::get_all_solutions() {
    CounterpointProblem* problem = set_configuration();
    std::vector<CounterpointProblem*> solutions;
    BAB<CounterpointProblem> e(problem);
    while (CounterpointProblem* pb = e.next()) {
        cout << "Solution found" << endl;
        solutions.push_back(pb);
        cout << pb->getSolutionArray() << endl;
        cout << pb->getCounterpoint_1()-> getSpeciesArray() << endl;
        cout << "Solution added" << endl;
    }
    return solutions;
}

/*
* Check if the problem is unsatisfiable
* @return true if the problem is unsatisfiable, false otherwise
*/
bool FigureTests::is_unsat(const set<int>& cons_set) {
    // set active cons_set
    activeConstraints = std::vector<bool>(activeConstraints.size(), false);
    for (int cons : cons_set) {
        activeConstraints[cons] = true;
    }
    // create a new problem and set the solution array
    CounterpointProblem* problem = set_configuration();
    // check if the problem is unsatisfiable
    bool unsat = !has_solution(problem);
    delete problem;
```

```
87      return unsat;
88  }
89
90  /**
91   * Minimize the set of constraints by removing one constraint at a time
92   * until the problem becomes satisfiable.
93   * @param cons_set the set of constraints to minimize
94   * @return the minimized set of constraints
95   */
96  std::set<int> FigureTests::minimize(const std::set<int>& cons_set) {
97      std::set<int> mus = cons_set;
98      for (auto it = mus.begin(); it != mus.end(); ) {
99          set<int> trial = mus;
100         trial.erase(*it);
101         if (is_unsat(trial)) {
102             mus = trial;
103             it = mus.begin(); // restart
104         } else {
105             it++;
106         }
107     }
108     return mus;
109 }
110
111 /**
112  * Find all minimal unsatisfiable subsets (MUSes) of the active constraints
113  * using a breadth-first search algorithm.
114  */
115 void FigureTests::findAllMUSes() {
116     // Initialization
117     set<int> cons_set; // all constraint indices
118     for (int i = 0; i < activeConstraints.size(); i++) {
119         cons_set.insert(i);
120     }
121
122     set<set<int>> mus_found;
123     set<set<int>> visited;
124     queue<set<int>> queue;
125
126     queue.push(cons_set);
127     visited.insert(cons_set);
128
129     while (!queue.empty()) {
130         std::set<int> current = queue.front();
131         queue.pop();
132         // Check if the current set is a minimal unsatisfiable subset
133         if (!is_unsat(current))  {
134             continue;
135         }
136         std::set<int> mus = minimize(current);
137         if (!mus_found.count(mus)) {
138             mus_found.insert(mus);
139             // Remove one constraint at a time and add the new set to the queue to explore
    further
140             for (int c : mus) {
141                 set<int> next = current;
142                 next.erase(c);
143                 if (!visited.count(next)) {
144                     visited.insert(next);
145                     queue.push(next);
146                 }
147             }
148         }
149     }
150     // Print the MUSes found
151     for (set<int> mus : mus_found) {
152         cout << "\t MUS found: ";
153         for (int cons : mus) {
154             cout << get_constraint_name(cons) << " ";
155         }
156         cout << endl;
157     }
158 }
159
160 void FigureTests::find_unsat_constraints() {
161     // activeConstraints = std::vector<bool>(activeConstraints.size(), false);
162     // cout << has_solution(set_configuration()) << endl;
163     for (int i = 0; i < consSize; i++) {
164         set<int> cons_set; // all constraint indices
165         cons_set.insert(i);
166         if (is_unsat(cons_set))  {
167             cout << "\t" << get_constraint_name(i) << " is unsatisfiable." << endl;
168         }
169     }
170 }
171
172 /* ============================================================================
173                             FIGURES
174 ============================================================================ */
175
```

```
176 void FigureTests::test_2v_1sp_fig22() {
177     cout << "Start test_2v_1sp_fig22" << endl;
178     spList = {FIRST_SPECIES};
179     cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57};
180     cp =           {69,64,67,65,64,72,69,71,71,69,68,69};
181     v_type = {1};
182     findAllMUSes();
183 }
184
185 void FigureTests::test_2v_1sp_fig23() {
186     cout << "Start test_2v_1sp_fig23" << endl;
187     spList = {FIRST_SPECIES};
188     cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57};
189     cp =           {57,57,55,53,52,52,50,48,55,57,56,57};
190     v_type = {-1};
191     findAllMUSes();
192 }
193
194 void FigureTests::test_2v_2sp_fig38() {
195     cout << "Start test_2v_2sp_fig38" << endl;
196     spList = {SECOND_SPECIES};
197     cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
198     cp =           {-1,65,64,62,60,58,57,55,53,57,60,58,57,69,67,64,65,67,69,65,62,64,65};
199     v_type = {2};
200     findAllMUSes();
201 }
202
203 void FigureTests::test_2v_2sp_fig39() {
204     cout << "Start test_2v_2sp_fig39" << endl;
205     spList = {SECOND_SPECIES};
206     cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
207     cp =        {-1,53,52,48,53,52,50,48,46,58,55,60,57,53,52,48,53,41,45,50,48,52,53};
208     v_type = {0};
209     findAllMUSes();
210 }
211
212 void FigureTests::test_2v_2sp_fig40() {
213     cout << "Start test_2v_2sp_fig40" << endl;
214     spList = {SECOND_SPECIES};
215     cantusFirmus = {55,60,59,55,60,64,62,67,64,60,62,59,57,55};
216     cp =           {-1,
    67,64,65,67,69,71,69,67,72,71,72,74,72,71,69,67,65,64,72,71,69,67,62,64,66,67};
217     v_type = {2};
218     findAllMUSes();
219 }
220
221 void FigureTests::test_2v_2sp_fig41() {
222     cout << "Start test_2v_2sp_fig41" << endl;
223     spList = {SECOND_SPECIES};
224     cantusFirmus = {55,60,59,55,60,64,62,67,64,60,62,59,57,55};
225     cp =
    {-1,55,52,53,55,53,52,50,48,52,48,60,59,57,55,59,60,59,57,55,54,50,55,47,50,54,55};
226     v_type = {0};
227     findAllMUSes();
228 }
229
230 void FigureTests::test_2v_2sp_fig42() {
231     cout << "Start test_2v_2sp_fig42" << endl;
232     spList = {SECOND_SPECIES};
233     cantusFirmus = {57,60,59,60,64,65,64,62,60,59,57};
234     cp =           {-1,69,64,65,67,62,64,76,72,71,69,65,67,71,74,69,72,64,66,68,69};
235     v_type = {2};
236     findAllMUSes();
237 }
238
239 void FigureTests::test_2v_2sp_fig43() {
240     cout << "Start test_2v_2sp_fig43" << endl;
241     spList = {SECOND_SPECIES};
242     cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57};
243     cp =           {-1,45,57,52,55,52,50,53,57,59,60,48,50,45,48,52,53,55,57,45,52,56,57};
244     v_type = {-1};
245     findAllMUSes();
246 }
247
248 void FigureTests::test_2v_2sp_fig44() {
249     cout << "Start test_2v_2sp_fig44" << endl;
250     spList = {SECOND_SPECIES};
251     cantusFirmus = {60,64,65,67,64,69,67,64,65,64,62,60};
252     cp =           {-1,67,72,71,69,74,71,69,67,71,72,74,76,74,72,71,69,71,72,67,69,71,72};
253     v_type = {1};
254     findAllMUSes();
255 }
256
257 void FigureTests::test_2v_2sp_fig45() {
258     cout << "Start test_2v_2sp_fig45" << endl;
259     spList = {SECOND_SPECIES};
260     cantusFirmus = {60,64,65,67,64,69,67,64,65,64,62,60};
261     cp =           {-1,48,60,59,57,62,59,55,60,59,57,60,64,62,60,57,62,57,60,48,55,59,60};
262     v_type = {-1};
263     findAllMUSes();
```

```
}

void FigureTests::test_2v_3sp_fig55() {
    cout << "Start test_2v_3sp_fig55" << endl;
    spList = {THIRD_SPECIES};
    cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
    cp =
    {62,64,65,67,69,71,72,74,76,74,71,72,74,72,70,69,70,72,74,76,77,65,69,71,72,69,70,72,70,69,67,71,69,62,

    v_type = {2};
    findAllMUSes();
}

void FigureTests::test_2v_3sp_fig56() {
    cout << "Start test_2v_3sp_fig56" << endl;
    spList = {THIRD_SPECIES};
    cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
    cp =
    {50,52,53,55,57,50,57,59,60,59,55,57,59,57,55,53,52,64,59,60,62,57,50,52,53,55,57,59,60,62,64,60,62,57,

    v_type = {-1};
    findAllMUSes();
}

void FigureTests::test_2v_3sp_fig57() {
    cout << "Start test_2v_3sp_fig57" << endl;
    spList = {THIRD_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =
    {71,67,69,71,72,71,69,67,65,67,69,71,72,64,65,67,69,72,76,74,72,71,69,72,71,74,71,69,67,71,72,71,69,71,

    v_type = {2};
    findAllMUSes();
}

void FigureTests::test_2v_3sp_fig58() {
    cout << "Start test_2v_3sp_fig58" << endl;
    spList = {THIRD_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =
    {52,53,55,52,57,55,53,52,50,52,53,55,57,52,57,55,53,52,50,52,53,55,57,59,60,62,64,62,60,48,60,59,57,62,

    v_type = {-1};
    findAllMUSes();
}

void FigureTests::test_2v_3sp_fig59() {
    cout << "Start test_2v_3sp_fig59" << endl;
    spList = {THIRD_SPECIES};
    cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
    cp =
    {65,64,62,60,59,62,67,65,64,62,60,58,57,60,62,64,65,62,64,65,67,64,65,67,69,67,65,69,67,65,64,62,60,64,

    v_type = {1};
    findAllMUSes();
}

void FigureTests::test_2v_3sp_fig60() {
    cout << "Start test_2v_3sp_fig60" << endl;
    spList = {THIRD_SPECIES};
    cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
    cp =
    {41,53,52,50,52,48,50,52,53,52,50,48,50,48,46,45,46,53,58,57,55,48,60,58,57,55,53,50,52,48,50,52,53,52,

    v_type = {0};
    findAllMUSes();
}

void FigureTests::test_2v_4sp_fig74() {
    cout << "Start test_2v_4sp_fig74" << endl;
    spList = {FOURTH_SPECIES};
    cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
    cp =           {50,50,62,62,60,60,59,59,64,64,62,62,65,65,64,64,62,62,61,62};
    v_type = {-1};
    findAllMUSes();
}

void FigureTests::test_2v_4sp_fig75() {
    cout << "Start test_2v_4sp_fig75" << endl;
    spList = {FOURTH_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =           {76,76,72,72,71,72,64,64,65,65,72,72,71,71,76,76,74,76};
    v_type = {2};
    findAllMUSes();
}

void FigureTests::test_2v_4sp_fig76() {
    cout << "Start test_2v_4sp_fig76" << endl;
    spList = {FOURTH_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
```

```
342     cp =            {52,52,57,57,55,55,53,53,50,50,62,62,60,60,64,64,62,64};
343     v_type = {-2};
344     findAllMUSes();
345 }
346
347 void FigureTests::test_2v_4sp_fig77() {
348     cout << "Start test_2v_4sp_fig77" << endl;
349     spList = {FOURTH_SPECIES};
350     cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
351     cp =            {65,65,64,64,60,60,65,65,69,69,67,67,65,65,64,64,69,69,65,65,64,65};
352     v_type = {2};
353     findAllMUSes();
354 }
355
356 void FigureTests::test_2v_4sp_fig78() {
357     cout << "Start test_2v_4sp_fig78" << endl;
358     spList = {FOURTH_SPECIES};
359     cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
360     cp =            {53,53,52,52,50,50,46,46,43,43,55,55,53,53,52,52,50,50,53,53,52,53};
361     v_type = {-1};
362     findAllMUSes();
363 }
364
365 void FigureTests::test_2v_5sp_fig82() {
366     cout << "Start test_2v_5sp_fig82" << endl;
367     spList = {FIFTH_SPECIES};
368     cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
369     cp =            {-1,-1,69,-1,
370                      69,-1, 64,65,
371                      67,65,64,67,
372                      65,62,74,-1,
373                      74,-1,70,67,
374                      69,71,72,-1,
375                      72,-1,77,-1,
376                      77,-1,76,-1,
377                      76,-1,74,-1,
378                      74,-1,73,-1,
379                      74};
380     notesSpeciesFor5sp = {
381                      -1, -1, FOURTH_SPECIES, -1,
382                      FOURTH_SPECIES,-1, THIRD_SPECIES, THIRD_SPECIES,
383                      THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES,
384                      THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
385                      FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
386                      THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
387                      FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
388                      FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
389                      FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
390                      FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
391                      FOURTH_SPECIES
392                  };
393     v_type = {2};
394     findAllMUSes();
395 }
396
397 void FigureTests::test_2v_5sp_fig83() {
398     cout << "Start test_2v_5sp_fig83" << endl;
399     spList = {FIFTH_SPECIES};
400     cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
401     cp =            {-1,-1,62,-1,
402                      62,-1,57,59,
403                      60,55,60,-1,
404                      60,-1,59,57,
405                      55,57,59,60,
406                      62,57,62,-1,
407                      62,-1,65,-1,
408                      65,-1,64,-1,
409                      64,-1,62,-1,
410                      62,-1,61,-1,
411                      62};
412     notesSpeciesFor5sp = {
413                      -1, -1, FOURTH_SPECIES, -1,
414                      FOURTH_SPECIES,-1, THIRD_SPECIES, THIRD_SPECIES,
415                      THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
416                      FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
417                      THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES,
418                      THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
419                      FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
420                      FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
421                      FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
422                      FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
423                      FOURTH_SPECIES
424     };
425     v_type = {0};
426     findAllMUSes();
427 }
428
429 void FigureTests::test_2v_5sp_fig87_1() {
430     cout << "Start test_2v_5sp_fig87_1" << endl;
431     spList = {FIFTH_SPECIES};
```

```
    cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57};
    cp =           {-1,-1,69,-1,
                    69,-1,67,69,
                    71,67,71,-1,
                    71,-1,69,71,
                    72,67,72,-1,
                    72,-1,71,72,
                    74,69,74,-1,
                    74,-1,72,-1,
                    72,-1,71,-1,
                    71,-1,69,-1,
                    69,-1,68,-1,
                    69};
    notesSpeciesFor5sp = {
                    -1, -1, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
                    THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
                    THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
                    THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES
                };
    v_type = {2};
    findAllMUSes();
}

void FigureTests::test_3v_1sp_fig108(){
    cout << "Start test_3v_1sp_fig108" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =           {67,64,65,69,72,72,76,72,69,68,
                    52,57,50,53,53,53,60,60,62,64};
    v_type = {1, -2};
    findAllMUSes();
}

void FigureTests::test_3v_1sp_fig109(){
    cout << "Start test_3v_1sp_fig109" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {52,48,50,48,45,57,55,52,53,52};
    cp =           {67,64,65,64,64,64,67,67,62,64,
                    59,60,57,57,60,60,59,59,57,59};
    v_type = {2, 1};
    findAllMUSes();
}

void FigureTests::test_3v_1sp_fig110(){
    cout << "Start test_3v_1sp_fig110" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
    cp =           {60,60,60,62,58,59,57,69,65,62,64,65,
                    53,52,53,50,55,55,53,45,50,50,48,41};
    v_type = {-1, -3};
    findAllMUSes();
}

void FigureTests::test_3v_1sp_fig111(){
    cout << "Start test_3v_1sp_fig111" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
    cp =           {69,72,72,74,74,70,69,67,65,77,76,77,
                    65,64,65,62,58,55,53,52,53,50,48,53};
    v_type = {1, -2};
    findAllMUSes();
}

void FigureTests::test_3v_1sp_fig112(){
    cout << "Start test_3v_1sp_fig112" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
    cp =           {69,70,72,69,70,72,69,67,72,69,70,69,
                    65,62,60,60,65,67,60,64,64,65,64,65};
    v_type = {3, 2};
    findAllMUSes();
}

void FigureTests::test_3v_1sp_fig113(){
    cout << "Start test_3v_1sp_fig113" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {55,60,59,55,60,64,62,67,64,60,62,59,57,55};
    cp =           {59,55,55,59,64,60,59,55,55,57,54,55,54,55,
                    55,52,52,52,48,48,55,52,48,45,47,43,50,43};
    v_type = {0, -1};
    findAllMUSes();
}
```

```
522
523 void FigureTests::test_3v_1sp_fig114(){
524     cout << "Start test_3v_1sp_fig114" << endl;
525     spList = {FIRST_SPECIES, FIRST_SPECIES};
526     cantusFirmus = {55,60,59,55,60,64,62,67,64,60,62,59,57,55};
527     cp =           {71,67,67,71,67,69,71,71,72,64,66,67,66,67,
528                     55,52,52,52,52,48,55,52,48,48,47,43,50,43};
529     v_type = {2, -1};
530     findAllMUSes();
531 }
532
533 void FigureTests::test_3v_1sp_fig115(){
534     cout << "Start test_3v_1sp_fig115" << endl;
535     spList = {FIRST_SPECIES, FIRST_SPECIES};
536     cantusFirmus = {69,72,71,74,72,76,77,76,74,72,71,69};
537     cp =           {60,64,67,65,64,67,69,67,65,69,68,69,
538                     57, 57, 64, 62, 57, 55, 53, 60, 62, 57, 64, 57};
539     v_type = {-1, -2};
540     findAllMUSes();
541 }
542
543 void FigureTests::test_3v_1sp_fig116(){
544     cout << "Start test_3v_1sp_fig116" << endl;
545     spList = {FIRST_SPECIES, FIRST_SPECIES};
546     cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57};
547     cp =           {64,64,67,65,64,72,69,72,71,69,68,69,
548                     45,45,52,50,57,57,50,48,55,57,52,45};
549     v_type = {1, -1};
550     findAllMUSes();
551 }
552
553 void FigureTests::test_3v_1sp_fig117(){
554     cout << "Start test_3v_1sp_fig117" << endl;
555     spList = {FIRST_SPECIES, FIRST_SPECIES};
556     cantusFirmus = {45,48,47,50,48,52,53,52,50,48,47,45};
557     cp =           {60,64,62,65,64,60,60,60,62,57,56,57,
558                     57,57,59,59,60,55,57,55,53,52,50,52};
559     v_type = {2, 1};
560     findAllMUSes();
561 }
562
563 void FigureTests::test_3v_1sp_fig118(){
564     cout << "Start test test_3v_1sp_fig118" << endl;
565     spList = {FIRST_SPECIES, FIRST_SPECIES};
566     cantusFirmus = {60, 64, 65, 67, 64, 69, 67, 64, 65, 64, 62, 60};
567     cp =           {55, 60, 57, 55, 60, 60, 64, 60, 59, 60, 59, 60,
568                     48, 48, 50, 52, 48, 53, 52, 57, 50, 48, 55, 48};
569     v_type = {-1, -1};
570    findAllMUSes();
571 }
572
573 void FigureTests::test_3v_1sp_fig119(){
574     cout << "Start test 3v_1sp_fig119" << endl;
575     spList = {FIRST_SPECIES, FIRST_SPECIES};
576     cantusFirmus = {60, 64, 65, 67, 64, 69, 67, 64, 65, 64, 62, 60};
577     cp =           {64, 67, 69, 71, 72, 72, 76, 72, 69, 72, 71, 72,
578                     60, 60, 57, 55, 57, 53, 60, 60, 62, 60, 55, 60};
579     v_type = {1, -1};
580     findAllMUSes();
581 }
582
583 void FigureTests::test_3v_2sp_fig125(){
584     cout << "Start test_3v_2sp_fig125" << endl;
585     spList = {FIRST_SPECIES, SECOND_SPECIES};
586     cantusFirmus = {52,   48,   50,   48,   45,   57,   55,   52,   53,   52};
587     cp =           {64,   64,   65,   64,   65,   64,   67,   67,   69,   68,
588                     -1,59,60,59,57,53,55,52,57,59,60,57,59,62,64,62,60,57,59};
589     v_type = {2 ,1};
590     findAllMUSes();
591 }
592
593 void FigureTests::test_3v_2sp_fig126(){
594     cout << "Start test_3v_2sp_fig126" << endl;
595     spList = {FIRST_SPECIES, SECOND_SPECIES};
596     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
597     cp =           {52,57,53,57,53,65,64,60,57,56,
598                     -1,52,53,52,50,52,53,48,50,52,53,57,60,55,57,52,52,50,52};
599     v_type = {-1, -2};
600     findAllMUSes();
601 }
602
603 void FigureTests::test_3v_2sp_fig127(){
604     cout << "Start test_3v_2sp_fig127" << endl;
605     spList = {SECOND_SPECIES, FIRST_SPECIES};
606     cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
607     cp =           {-1,65,64,62,60,57,62,57,58,57,55,60,57,69,67,64,60,57,62,65,65,64,65,
608                     53,48,53,50,46,48,50,52,53,50,48,41};
609     v_type = {-1, -3};
610     findAllMUSes();
611 }
```

```
612
613
614
615 void FigureTests::test_3v_2sp_fig128(){
616     cout << "Start test_3v_2sp_fig128" << endl;
617     spList = {SECOND_SPECIES, FIRST_SPECIES};
618     cantusFirmus = {65,   67,   69,   65,   62,   64,   65,   72,   69,   65,   67,   65};
619     cp =           {-1,77,76,74,72,71,69,72,70,69,67,72,69,81,79,76,72,69,74,77,77,76,77,
620                     53,   48,   53,   53,   55,   60,   62,   64,   65,   62,   60,   53};
621     v_type = {1 ,-2};
622     findAllMUSes();
623 }
624
625 void FigureTests::test_3v_2sp_fig129(){
626     cout << "Start test_3v_2sp_fig129" << endl;
627     spList = {FIRST_SPECIES, SECOND_SPECIES};
628     cantusFirmus =  {53, 55, 57, 53, 50, 52, 53, 60, 57, 53, 55, 53};
629     cp =            {60,   60,   60,   62,   65,   67,   69,   67,   60,   62,   64,   65,
630                      -1,53,52,48,53,52,50,48,46,45,43,48,41,53,52,48,53,52,50,46,43,48,41};
631     v_type = {1, -1};
632     findAllMUSes();
633 }
634
635 void FigureTests::test_3v_3sp_fig130(){
636     cout << "Start test_3v_3sp_fig130" << endl;
637     spList = {THIRD_SPECIES, FIRST_SPECIES};
638     cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
639     cp =
    {69,62,65,67,69,65,69,71,72,71,67,69,71,72,74,71,76,74,71,73,74,77,76,74,72,69,72,74,76,74,72,71,69,62,

640                     50,50,48,55,52,50,53,48,50,57,50};
641     v_type = {1, -2};
642     findAllMUSes();
643 }
644
645 void FigureTests::test_3v_3sp_fig131(){
646     cout << "Start test_3v_3sp_fig131" << endl;
647     spList = {THIRD_SPECIES, FIRST_SPECIES};
648     cantusFirmus = {50,53,52,50,55,53,57,55,53,52,50};
649     cp =
    {62,64,65,67,69,57,60,62,64,65,67,64,65,69,65,64,62,58,62,64,65,69,67,65,64,60,64,65,67,64,65,67,69,57,

650                     57,57,61,62,58,57,60,58,57,55,54};
651     v_type = {2, 1};
652     findAllMUSes();
653 }
654
655 void FigureTests::test_3v_3sp_fig132(){
656     cout << "Start test_3v_3sp_fig132" << endl;
657     spList = {FIRST_SPECIES, THIRD_SPECIES};
658     cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
659     cp =           {69,69,72,71,71,74,72,76,74,73,74,
660
    -1,50,53,52,50,52,53,55,57,55,52,53,55,50,55,53,52,55,53,52,50,62,50,52,53,55,57,59,60,64,62,60,62,50,5

661     v_type = {1, -1};
662     findAllMUSes();
663 }
664
665 void FigureTests::test_3v_3sp_fig133(){
666     cout << "Start test_3v_3sp_fig133" << endl;
667     spList = {THIRD_SPECIES, FIRST_SPECIES};
668     cantusFirmus =  {62,65,64,62,67,65,69,67,65,64,62};
669     cp =
    {53,50,53,55,57,53,57,59,60,52,55,57,59,62,59,57,55,57,59,61,62,64,65,62,60,57,60,62,64,62,60,59,57,62,

670                      50,50,48,55,52,50,53,48,50,57,50};
671     v_type = {-1, -2};
672     findAllMUSes();
673 }
674
675 void FigureTests::test_3v_4sp_fig146() {
676     cout << "Start test_3v_4sp_fig146" << endl;
677     spList = {FOURTH_SPECIES, FIRST_SPECIES};
678     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
679     cp =           {76,76,72,72,71,71,69,69,72,72,74,74,72,72,69,69,71,68,
680                     52,57,55,57,53,53,52,48,50,52};
681     v_type = {2,-2};
682     findAllMUSes();
683 }
684
685 void FigureTests::test_3v_4sp_fig147() {
686     cout << "Start test_3v_4sp_fig147" << endl;
687     spList = {FOURTH_SPECIES, FIRST_SPECIES};
688     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
689     cp =           {64,64,60,60,59,59,57,57,62,62,60,60,59,59,57,57,59,56,
690                     52,57,55,57,53,53,52,48,50,52};
691     v_type = {0,-2};
692     findAllMUSes();
693 }
```

```
694
695 void FigureTests::test_3v_4sp_fig148() {
696     cout << "Start test_3v_4sp_fig148" << endl;
697     spList = {FIRST_SPECIES, FOURTH_SPECIES};
698     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
699     cp =           {67,67,67,64,72,74,76,72,69,68,
700                     52,52,48,48,47,47,45,45,53,53,50,50,48,48,52,52,50,52};
701     v_type = {2,-2};
702     findAllMUSes();
703 }
704
705 void FigureTests::test_3v_4sp_fig149() {
706     cout << "Start test_3v_4sp_fig149" << endl;
707     spList = {FOURTH_SPECIES, FIRST_SPECIES};
708     cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
709     cp =           {65,65,64,64,62,62,65,65,67,67,69,69,65,65,64,64,62,62,65,65,64,65,
710                     53,48,53,50,46,48,50,57,53,50,48,41};
711     v_type = {2,0};
712     findAllMUSes();
713 }
714
715 void FigureTests::test_3v_4sp_fig150(){
716     cout << "Start test_3v_4sp_fig150" << endl;
717     spList = {FIRST_SPECIES, FOURTH_SPECIES};
718     cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
719     cp =           {69,71,72,69,65,67,69,67,72,69,70,69,
720                     65,65,64,64,60,60,57,57,62,62,60,60,65,65,64,64,60,60,65,65,64,65};
721     v_type = {3, 2};
722     findAllMUSes();
723 }
724
725 void FigureTests::test_3v_4sp_fig151(){
726     cout << "Start test_3v_4sp_fig151" << endl;
727     spList = {FIRST_SPECIES, FOURTH_SPECIES};
728     cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
729     cp =           {53,60,65,62,58,55,62,64,65,62,58,57,
730                     53,53,52,52,50,50,46,46,43,-1,48,48,46,46,45,45,50,50,53,53,52,53};
731     v_type = {-2, -2};
732     findAllMUSes();
733 }
734
735 void FigureTests::test_3v_5sp_fig154() {
736     cout << "Start test_3v_5sp_fig154" << endl;
737     spList = {FIFTH_SPECIES, FIRST_SPECIES};
738     cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
739     cp =           {-1,-1,69,-1,
740                     69,-1,69,71,
741                     72,67,72,-1,
742                     72,-1,71,-1,
743                     71,-1,71,73,
744                     74,77,76,74,
745                     72,65,77,-1,
746                     77,-1,76,-1,
747                     76,-1,74,-1,
748                     74,-1,73,-1,
749                     74,
750
751                     50,62,60,55,52,50,53,60,62,57,50};
752     notesSpeciesFor5sp = {
753                         -1, -1, FOURTH_SPECIES, -1,
754                         FOURTH_SPECIES,-1, THIRD_SPECIES, THIRD_SPECIES,
755                         THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
756                         FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
757                         FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
758                         THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES,
759                         THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
760                         FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
761                         FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
762                         FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
763                         FOURTH_SPECIES,
764     };
765     v_type = {2, -2};
766     findAllMUSes();
767 }
768
769 void FigureTests::test_3v_5sp_fig155() {
770     cout << "Start_test_3v_5sp_fig155" << endl;
771     spList = {FIFTH_SPECIES, FIRST_SPECIES};
772     cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
773     cp =           {-1,-1,57,-1,
774                     57,-1,57,59,
775                     60,55,60,-1,
776                     60,-1,59,-1,
777                     59,-1,59,61,
778                     62,57,62,-1,
779                     62,-1,65,-1,
780                     65,-1,64,-1,
781                     64,-1,62,-1,
782                     62,-1,61,-1,
783                     62,
```

```
784
785                     50,50,48,55,52,50,53,48,50,57,50};
786     notesSpeciesFor5sp = {
787                          -1, -1, FOURTH_SPECIES, -1,
788                          FOURTH_SPECIES,-1, THIRD_SPECIES, THIRD_SPECIES,
789                          THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
790                          FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
791                          FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
792                          THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
793                          FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
794                          FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
795                          FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
796                          FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
797                          FOURTH_SPECIES,
798     };
799     v_type = {0, -2};
800     findAllMUSes();
801 }
802
803 void FigureTests::test_3v_5sp_fig156() {
804     cout << "Start test_3v_5sp_fig156" << endl;
805     spList = {FIRST_SPECIES, FIFTH_SPECIES};
806     cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
807     cp =           {69,69,72,71,71,74,72,72,69,67,66,
808
809                     -1,-1,50,-1,
810                     50,-1,53,55,
811                     57,52,57,-1,
812                     57,-1,55,53,
813                     52,55,53,52,
814                     50,52,53,-1,
815                     53,-1,53,-1,
816                     53,-1,52,-1,
817                     52,-1,50,-1,
818                     50,-1,49,-1,
819                     50};
820     notesSpeciesFor5sp = {
821                          -1, -1, FOURTH_SPECIES, -1,
822                          FOURTH_SPECIES,-1, THIRD_SPECIES, THIRD_SPECIES,
823                          THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
824                          FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
825                          THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES,
826                          THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
827                          FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
828                          FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
829                          FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
830                          FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
831                          FOURTH_SPECIES
832     };
833     v_type = {2,-2};
834     findAllMUSes();
835 }
836
837 void FigureTests::test_3v_5sp_fig157() {
838     cout << "Start test_3v_5sp_fig157" << endl;
839     spList = {FIFTH_SPECIES, FIRST_SPECIES};
840     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
841     cp =           {-1,-1,76,-1,
842                     76,-1,72,71,
843                     -1,-1,67,65,
844                     -1,-1,69,-1,
845                     69,-1,72,-1,
846                     72,-1,77,-1,
847                     77,-1,76,74,
848                     72,67,72,-1,
849                     72,-1,71,69,
850                     68,
851
852                     52,57,53,57,53,53,60,60,62,64};
853     notesSpeciesFor5sp = {
854                          -1, -1, FOURTH_SPECIES, -1,
855                          FOURTH_SPECIES,-1, THIRD_SPECIES, THIRD_SPECIES,
856                          -1,  -1, THIRD_SPECIES, THIRD_SPECIES,
857                          -1, -1, FOURTH_SPECIES, -1,
858                          FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
859                          FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
860                          FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
861                          THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
862                          FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
863                          FOURTH_SPECIES
864     };
865     v_type = {2,0};
866     findAllMUSes();
867 }
868
869 void FigureTests::test_4v_1sp_fig166() {
870     cout << "Start test_4v_1sp_fig166" << endl;
871     spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
872     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
873     cp =           {59,57,57,57,60,60,64,60,62,59,
```

```
                    56,57,53,52,52,53,60,60,57,56,
                    52,53,50,45,45,41,40,45,50,52};
    v_type = {-1,-1,-3};
    findAllMUSes();
}

void FigureTests::test_4v_1sp_fig167() {
    cout << "Start test_4v_1sp_fig167" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =           {68,69,65,64,64,65,72,72,69,68,
                    59,57,57,57,60,60,64,60,62,59,
                    52,53,50,45,45,53,52,57,50,52};
    v_type = {1,-1,-3};
    findAllMUSes();
}

void FigureTests::test_4v_1sp_fig168() {
    cout << "Start test_4v_1sp_fig168" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =           {71,69,69,72,72,72,76,76,74,71,
                    68,69,65,67,69,65,72,72,69,68,
                    52,53,50,52,53,53,52,57,50,52};
    v_type = {1,1,-2};
    findAllMUSes();
}

void FigureTests::test_4v_1sp_fig169() {
    cout << "Start test_4v_1sp_fig169" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
    cp =           {69,67,65,69,70,71,69,69,65,65,64,65,
                    60,60,60,62,65,67,65,64,62,57,60,57,
                    53,52,53,50,46,43,50,45,50,50,48,41};
    v_type = {0,-1,-3};
    findAllMUSes();
}

void FigureTests::test_4v_1sp_fig170() {
    cout << "Start test_4v_1sp_fig170" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
    cp =           {60,60,60,60,58,59,57,69,65,65,64,65,
                    57,55,53,57,53,55,53,52,53,57,60,57,
                    53,52,53,41,46,43,50,45,50,50,48,41};
    v_type = {-1,-2,-3};
    findAllMUSes();
}

void FigureTests::test_4v_1sp_fig171() {
    cout << "Start test_4v_1sp_fig171" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
    cp =           {69,67,65,65,65,67,65,64,65,65,64,65,
                    60,60,60,62,62,58,57,57,53,57,60,57,
                    53,52,53,50,46,43,50,45,50,50,48,41};
    v_type = {2,1,-1};
    findAllMUSes();
}

void FigureTests::test_4v_1sp_fig172() {
    cout << "Start test_4v_1sp_fig172" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
    cp =           {72,71,72,69,69,72,72,72,72,69,70,65,
                    69,67,64,65,65,67,69,67,65,65,64,65,
                    65,62,60,60,62,60,60,64,60,62,58,60};
    v_type = {3,2,1};
    findAllMUSes();
}

void FigureTests::test_4v_2sp_fig173() {
    cout << "Start test_4v_2sp_fig173" << endl;
    spList = {FIRST_SPECIES, SECOND_SPECIES, FIRST_SPECIES};
    cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
    cp =           {57,62,55,55,59,62,60,64,62,61,62,
                    -1,53,57,59,60,55,59,57,55,52,57,53,52,53,52,60,57,53,57,57,57,
                    50,50,48,55,52,50,45,48,50,45,50};
    v_type = {-1,-1,-2};
    findAllMUSes();
}

void FigureTests::test_4v_2sp_fig174() {
    cout << "Start test_4v_2sp_fig174" << endl;
    spList = {SECOND_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
    cp =           {-1,65,69,71,72,67,71,74,76,71,74,77,76,72,76,72,69,74,73,73,74,
                    57,53,55,55,55,57,60,60,53,57,57,
                    50,50,48,55,52,50,45,48,50,45,50};
```

```
964     v_type = {1,-1,-2};
965     findAllMUSes();
966 }
967
968 void FigureTests::test_4v_2sp_fig175() {
969     cout << "Start test_4v_2sp_fig175" << endl;
970     spList = {FIRST_SPECIES, FIRST_SPECIES, SECOND_SPECIES};
971     cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
972     cp =           {74,74,72,74,76,77,77,76,74,73,74,
973                     69,69,69,71,71,74,72,72,69,69,69,
974                     -1,62,50,53,57,60,55,53,52,55,50,52,53,57,60,48,50,53,57,45,50};
975     v_type = {2,1,-2};
976     findAllMUSes();
977 }
978
979 void FigureTests::test_4v_2sp_fig176() {
980     cout << "Start test_4v_2sp_fig176" << endl;
981     spList = {FIRST_SPECIES, SECOND_SPECIES, FIRST_SPECIES};
982     cantusFirmus = {50,53,52,50,55,53,57,55,53,52,50};
983     cp =            {65,   69,   72,   74,   70,   69,   69,   71,   74,   73,   74,
984                      -1,62,60,62,64,67,65,64,62,64,65,69,64,65,67,62,65,69,67,64,66,
985                      57,   57,   55,   57,   58,   60,   60,   62,   57,   64,   57};
986     v_type = {3, 2, 1};
987     findAllMUSes();
988 }
989
990 void FigureTests::test_4v_3sp_fig183() {
991     cout << "Start test_4v_3sp_fig183" << endl;
992     spList = {THIRD_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
993     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
994     cp =
    {71,64,71,72,69,72,69,67,65,62,64,65,67,64,65,67,69,65,69,71,72,69,72,74,76,71,76,74,72,76,72,71,69,74,

995                     56,57,57,55,60,57,59,60,62,59,
996                     52,53,50,52,53,53,52,57,50,52};
997     v_type = {0, -1, -2};
998     findAllMUSes();
999 }
1000
1001 void FigureTests::test_4v_3sp_fig184(){
1002     cout << "Start test_4v_3sp_fig184" << endl;
1003     spList = {THIRD_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
1004     cantusFirmus = {76,72,74,72,69,81,79,76,77,76};
1005     cp =
    {68,71,64,67,69,72,69,67,65,62,64,65,67,65,64,62,60,72,69,67,65,67,69,71,72,74,76,71,72,69,71,72,69,74,

1006                     59,57,57,55,57,60,64,60,62,59,
1007                     52,53,50,52,53,53,52,57,50,52};
1008     v_type = {-2, -4, -5};
1009     findAllMUSes();
1010 }
1011
1012 void FigureTests::test_4v_3sp_fig185(){
1013     cout << "Start test_4v_3sp_fig185" << endl;
1014     spList = {FIRST_SPECIES, THIRD_SPECIES, FIRST_SPECIES};
1015     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
1016     cp =           {71,69,69,72,72,72,76,69,69,68,
1017
    56,59,52,55,57,60,57,55,53,57,55,53,55,52,64,62,60,59,57,55,53,55,57,59,60,62,64,62,60,57,60,59,57,50,6

1018                     52,53,50,52,53,53,48,48,50,52};
1019     v_type = {1, -2, -2};
1020     findAllMUSes();
1021 }
1022
1023 void FigureTests::test_4v_3sp_fig186(){
1024     cout << "Start test_4v_3sp_fig186" << endl;
1025     spList = {FIRST_SPECIES, FIRST_SPECIES, THIRD_SPECIES};
1026     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
1027     cp =           {71,76,77,76,77,77,76,76,69,68,
1028                     68,69,69,72,72,72,72,72,74,71,
1029                     64, 62, 60, 59, 57, 55, 53, 52, 50, 52, 53, 50, 57, 60, 57, 55, 53, 55, 57,
    55, 53, 55, 57, 59, 60, 52, 53, 55, 57, 55, 53, 52, 50, 53, 52, 50, 52};
1030     v_type = {1, 1, -2};
1031     findAllMUSes();
1032 }
1033
1034 void FigureTests::test_4v_4sp_fig196(){
1035     cout << "Start test_4v_4sp_fig196" << endl;
1036     spList = {FOURTH_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
1037     cantusFirmus =  {62,65,64,62,67,65,69,67,65,64,62};
1038     cp =            {57,57,62,62,60,60,59,59,62,62,57,57,65,65,64,64,62,62,61,62,
1039                     53, 57, 57, 50, 62, 62, 60, 60, 57, 57, 57,
1040                      50,50,45,47,43,50,53,48,50,45,50};
1041     v_type = {-1, -1, -3};
1042     findAllMUSes();
1043 }
1044
1045 void FigureTests::test_4v_5sp_fig200(){
1046     cout << "Start test_4v_5sp_fig200" << endl;
```

```
    spList = {FIFTH_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus =  {62,65,64,62,67,65,69,67,65,64,62};
    cp =            {
                    -1,-1,69,-1,
                    69,-1,69,71,
                    72,67,72,-1,
                    72,-1,71,-1,
                    71,-1,71,73,
                    74,69,74,-1,
                    74,-1,77,-1,
                    77,-1,76,-1,
                    76,-1,74,-1,
                    74,-1,73,-1,
                    74,

                    57, 53, 55, 55, 55, 57, 57, 60, 57, 57, 57,
                    50,50,48,55,52,50,53,48,50,45,50};
    notesSpeciesFor5sp = {
                        -1, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES,-1, THIRD_SPECIES, THIRD_SPECIES,
                        THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
                        THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES,
    };
    v_type = {1, -1, -2};
    findAllMUSes();
}

void FigureTests::test_4v_5sp_fig201() {
    cout << "Start test_4v_5sp_fig201" << endl;
    spList = {FIFTH_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus =  {62,65,64,62,67,65,69,67,65,64,62};
    cp =            {
                    -1, -1, 57, -1,
                    57, -1, 57, 59,
                    60, 55, 60, -1,
                    60, -1, 59, -1,
                    59, -1, 59, 61,
                    62, 50, 62, -1,
                    62, -1, 65, -1,
                    65, -1, 64, -1,
                    64, -1, 62, -1,
                    62, -1, 61, -1,
                    62,

                    57, 53, 55, 55, 55, 57, 57, 60, 57, 57, 57,
                    50,50,48,55,52,50,53,48,50,45,50};
    notesSpeciesFor5sp = {
                        -1, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES,-1, THIRD_SPECIES, THIRD_SPECIES,
                        THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
                        THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES,
    };
    v_type = {-1, -1, -2};
    findAllMUSes();
}

void FigureTests::test_4v_Xsp_fig204() {
    cout << "Start test_4v_Xsp_fig204" << endl;
    spList = {SECOND_SPECIES, THIRD_SPECIES, FOURTH_SPECIES};
    cantusFirmus =  {50,53,52,50,55,53,57,55,53,52,50};
    cp = {  69,74,72,69,71,72,74,62,70,67,69,65,72,69,70,67,69,65,67,64,66,

    62,64,65,67,69,67,65,69,67,65,64,67,65,62,65,64,62,60,58,62,65,67,69,65,64,62,60,57,62,60,58,55,60,59,5

            57, 57, 62, 62, 60, 60, 58, 58, 62, 62, 60, 60, 65, 65, 64, 64, 62, 62, 61, 62,
        };
    v_type = {2,2,1};
    findAllMUSes();
}

void FigureTests::run_twoVoice_tests() {
    cout << "Running two voice tests..." << endl;
    // Two voice first species figures
    test_2v_1sp_fig22();
    test_2v_1sp_fig23();
    // Two voice second species figures
```

```
1135     test_2v_2sp_fig38();
1136     test_2v_2sp_fig39();
1137     test_2v_2sp_fig40();
1138     test_2v_2sp_fig41();
1139     test_2v_2sp_fig42();
1140     test_2v_2sp_fig43();
1141     test_2v_2sp_fig44();
1142     test_2v_2sp_fig45();
1143     // Two voice third species figures
1144     test_2v_3sp_fig55();
1145     test_2v_3sp_fig56();
1146     test_2v_3sp_fig57();
1147     test_2v_3sp_fig58();
1148     test_2v_3sp_fig59();
1149     test_2v_3sp_fig60();
1150     // Two voice fourth species figures
1151     test_2v_4sp_fig74();
1152     test_2v_4sp_fig75();
1153     test_2v_4sp_fig76();
1154     test_2v_4sp_fig77();
1155     test_2v_4sp_fig78();
1156     // Two voice fifth species figures
1157     test_2v_5sp_fig82();
1158     test_2v_5sp_fig83();
1159     test_2v_5sp_fig87_1();
1160     test_2v_5sp_fig82();
1161     test_2v_5sp_fig83();
1162     test_2v_5sp_fig87_1();
1163 }
1164
1165 void FigureTests::run_threeVoice_tests() {
1166     cout << "Running three voice tests..." << endl;
1167     // Three voice first species figures
1168     test_3v_1sp_fig108();
1169     test_3v_1sp_fig109();
1170     test_3v_1sp_fig110();
1171     test_3v_1sp_fig111();
1172     test_3v_1sp_fig112();
1173     test_3v_1sp_fig113();
1174     test_3v_1sp_fig114();
1175     test_3v_1sp_fig115();
1176     test_3v_1sp_fig116();
1177     test_3v_1sp_fig117();
1178     test_3v_1sp_fig118();
1179     test_3v_1sp_fig119();
1180     // Three voice second species figures
1181     test_3v_2sp_fig125();
1182     test_3v_2sp_fig126();
1183     test_3v_2sp_fig127();
1184     test_3v_2sp_fig128();
1185     test_3v_2sp_fig129();
1186     // Three voice third species figures
1187     test_3v_3sp_fig130();
1188     test_3v_3sp_fig131();
1189     test_3v_3sp_fig132();
1190     test_3v_3sp_fig133();
1191     // Three voice fourth species figures
1192     test_3v_4sp_fig146();
1193     test_3v_4sp_fig147();
1194     test_3v_4sp_fig148();
1195     test_3v_4sp_fig149();
1196     test_3v_4sp_fig150();
1197     test_3v_4sp_fig151();
1198     // Three voice fifth species figures
1199     test_3v_5sp_fig154();
1200     test_3v_5sp_fig155();
1201     test_3v_5sp_fig156();
1202 }
1203
1204 void FigureTests::run_fourVoice_tests() {
1205     cout << "Running four voice tests..." << endl;
1206     // Four voice first species figures
1207     test_4v_1sp_fig166();
1208     test_4v_1sp_fig167();
1209     test_4v_1sp_fig168();
1210     test_4v_1sp_fig169();
1211     test_4v_1sp_fig170();
1212     test_4v_1sp_fig171();
1213     test_4v_1sp_fig172();
1214     // Four voice second species figures
1215     test_4v_2sp_fig173();
1216     test_4v_2sp_fig174();
1217     test_4v_2sp_fig175();
1218     test_4v_2sp_fig176();
1219     // Four voice third species figures
1220     test_4v_3sp_fig183();
1221     test_4v_3sp_fig184();
1222     test_4v_3sp_fig185();
1223     test_4v_3sp_fig186();
1224     // Four voice fourth species figures
```

```
1225     test_4v_4sp_fig196();
1226     // Four voice fifth species figures
1227     test_4v_5sp_fig201();
1228 }
1229
1230 void FigureTests:: run_thirdSpecies_tests() {
1231     cout << "Running third species tests..." << endl;
1232     // Two voice third species figures
1233     test_2v_3sp_fig55();
1234     test_2v_3sp_fig56();
1235     test_2v_3sp_fig57();
1236     test_2v_3sp_fig58();
1237     test_2v_3sp_fig59();
1238     test_2v_3sp_fig60();
1239     // Three voice third species figures
1240     test_3v_3sp_fig130();
1241     test_3v_3sp_fig131();
1242     test_3v_3sp_fig132();
1243     test_3v_3sp_fig133();
1244     // Four voice third species figures
1245     test_4v_3sp_fig183();
1246     test_4v_3sp_fig184();
1247     test_4v_3sp_fig185();
1248     test_4v_3sp_fig186();
1249     // Four voice fifth species figures
1250     test_4v_5sp_fig201();
1251 }
1252
1253 void FigureTests::run_fourthSpecies_tests() {
1254     cout << "Running four species tests..." << endl;
1255     // Two voice fourth species figures
1256     test_2v_4sp_fig74();
1257     test_2v_4sp_fig75();
1258     test_2v_4sp_fig76();
1259     test_2v_4sp_fig77();
1260     test_2v_4sp_fig78();
1261     // Three voice fourth species figures
1262     test_3v_4sp_fig146();
1263     test_3v_4sp_fig147();
1264     test_3v_4sp_fig148();
1265     test_3v_4sp_fig149();
1266     test_3v_4sp_fig150();
1267     test_3v_4sp_fig151();
1268     // Four voice fourth species figures
1269     test_4v_4sp_fig196();
1270 }
1271
1272 void FigureTests::run_fifthSpecies_tests() {
1273     cout << "Running fifth species tests..." << endl;
1274     test_2v_5sp_fig82();
1275     test_2v_5sp_fig83();
1276     test_2v_5sp_fig87_1();
1277     test_3v_5sp_fig154();
1278     test_3v_5sp_fig155();
1279     test_3v_5sp_fig156();
1280     test_4v_5sp_fig201();
1281 }
1282
1283 void FigureTests::run_all_tests() {
1284     cout << "Running all figure tests..." << endl;
1285     run_twoVoice_tests();
1286     run_threeVoice_tests();
1287     run_fourVoice_tests();
1288     test_4v_Xsp_fig204();
1289     cout << "All figure tests completed." << endl;
1290 }
1291
1292 // Add to the MUSTest method to include our new test
1293 void FigureTests::MUSTest() {
1294     cout << "Running MUSTest..." << endl;
1295     run_all_tests();
1296
1297     cout << "MUSTest completed." << endl;
1298 }
1299
1300 void FigureTests::quickTest() {
1301     // test_2v_1sp_fig22();
1302     // test_2v_2sp_fig39();
1303     // test_2v_2sp_fig40();
1304     // test_2v_2sp_fig41();
1305     // test_2v_2sp_fig42();
1306     // test_2v_2sp_fig43();
1307     // test_2v_2sp_fig44();
1308     // test_2v_2sp_fig45();
1309     // test_2v_3sp_fig55();
1310     // test_2v_3sp_fig56();
1311     // test_2v_3sp_fig57();
1312     // test_2v_3sp_fig58();
1313     // test_2v_4sp_fig74();
1314     // test_2v_4sp_fig76();
```

```
    // test_2v_4sp_fig77();
    // test_2v_4sp_fig78();
    // test_2v_5sp_fig82();
    // test_2v_5sp_fig83();
    // test_3v_1sp_fig108();
    // test_3v_1sp_fig109();
    // test_3v_1sp_fig110();
    test_3v_2sp_fig126();
    // test_3v_2sp_fig127();
    // test_3v_2sp_fig128();
    // test_3v_3sp_fig130();
    // test_3v_3sp_fig133();
    // test_3v_4sp_fig146();
    // test_3v_4sp_fig147();
    // test_3v_4sp_fig148();
    // test_3v_4sp_fig151();
    // test_3v_5sp_fig154();
    // test_3v_5sp_fig156();
    // test_4v_1sp_fig169();
    // test_4v_1sp_fig171();
    // test_4v_2sp_fig173();
    // test_4v_3sp_fig183();
    // test_4v_3sp_fig185();
    // test_4v_3sp_fig186();
    // test_4v_3sp_fig184();
    // test_4v_3sp_fig186();
    // test_4v_4sp_fig196();
    // test_4v_5sp_fig201();
    // test_4v_Xsp_fig204();
}
```

A.10.3 main.cpp

```
//
// Created by Luc Cleenewerk and Diego de Patoul. Modified by Bryce Burignat
//

#include "../../headers/Tests/fuxTest.hpp"
#include "../../headers/Tests/figureTests.hpp"
#include "../../headers/GenerationsAPI.hpp"

using namespace Gecode;
using namespace std;

int main(int argc, char* argv[]) {
    if (argc < 2) {
        std::cout << "Warning :no test type, number, or 'figs' as argument. Doing default API check." << std::endl;
    }
    if (consSize != constraintNames.size()) {
        cout << "Error: constraintNames size does not match the number of constraints." << endl;
        cout << "consSize = " << consSize << endl;
        cout << "constraintNames.size() = " << constraintNames.size() << endl;
        return 1;
    }

    if(argc==1){ // QUICK TEST on API
        // EXAMPLE of single
        cout << "------------- API TEST -------------" << endl;

        GenerationInput gi;
        gi.cf_notes = {60,   62,   65,   64,   67,   65,   64,   62,   60}; // The cantus firmus notes
        gi.species = {THIRD_SPECIES, FIRST_SPECIES};
        gi.v_type = {2 , 1};
        gi.verbose = true;
        gi.stagnation_ms = 0;
        gi.output_subdir = "api_basic_test";
        gi.timeout_ms = 2000;

        GenerationResult result = generate_counterpoint(gi);

        // ---------- DISABLED BY DEFAULT ----------
        // EXAMPLE of generations of many many configurations of counterpoints :
        //      - every species/voices
        //      - every v_type configuration (with values from 1 to 3)
        //
        // DISABLED by default because 150 generations is a lot
        //
        if (false){
            // 2-voices (15 cases)
            for (int sp = 1; sp < 6; sp++){
                for (int v_type = 1; v_type < 4; v_type++){
                    GenerationInput input;
                    input.cf_notes = {48, 50, 53, 52, 55, 53, 52, 50, 48};
                    input.species = {int_to_species(sp)};
                    input.v_type = {v_type};
```

```
                    input.verbose = true;
                    input.stagnation_ms = 0;
                    input.timeout_ms = 360000; // 6min
                    input.output_subdir = "test_multi";
                    input.preset_name = "default";
                    generate_counterpoint(input);
                }
            }

            // 3-voices (45 cases)
            for (int sp = 1; sp < 6; sp++){
                for (int v_type = 1; v_type < 4; v_type++){
                    for(int sec_type = 1; sec_type < 4; sec_type++){
                        GenerationInput input;
                        input.cf_notes = {48, 50, 53, 52, 55, 53, 52, 50, 48};
                        input.species = {FIRST_SPECIES, int_to_species(sp)};
                        input.v_type = {sec_type, v_type};
                        input.verbose = true;
                        input.stagnation_ms = 0;
                        input.timeout_ms = 360000; // 6min
                        input.output_subdir = "test_multi";
                        input.preset_name = "default";
                        generate_counterpoint(input);
                    }
                }
            }

            // 4-voices (90 cases)
            vector<pair<int,int>> cp_types = {{1,1},{1,2},{1,3},{2,2},{2,3},{3,3}};

            for (int sp = 1; sp < 6; sp++){
                for (int main_type = 1; main_type < 4; main_type++){
                    for(int cp_type_idx = 0; cp_type_idx < cp_types.size(); cp_type_idx++){
                        pair<int,int> local_types = cp_types[cp_type_idx];

                        GenerationInput input;
                        input.cf_notes = {48, 50, 53, 52, 55, 53, 52, 50, 48};
                        input.species = {FIRST_SPECIES, FIRST_SPECIES, int_to_species(sp)};
                        input.v_type = {local_types.first, local_types.second, main_type};
                        input.verbose = true;
                        input.stagnation_ms = 0;
                        input.timeout_ms = 360000; // 6min
                        input.output_subdir = "test_multi";
                        input.preset_name = "default";
                        generate_counterpoint(input);
                    }
                }
            }
        }

    } else if(argc==2){ // FIGURE / FUX TESTS
        string arg1 = argv[1];
        char* str = argv[1];
        if (arg1 == "figs") {
            FigureTests figureTests;
            if (argc > 2) {
                string arg2 = argv[2];
                if (arg2 == "2v") {
                    figureTests.run_twoVoice_tests();
                } else if (arg2 == "3v") {
                    figureTests.run_threeVoice_tests();
                } else if (arg2 == "4v") {
                    figureTests.run_fourVoice_tests();
                } else if (arg2 == "4sp") {
                    figureTests.run_fourthSpecies_tests();
                } else if (arg2 == "5sp") {
                    figureTests.run_fifthSpecies_tests();
                } else if (arg2 == "MUS") {
                    figureTests.MUSTest();
                } else if (arg2 == "quick") {
                    figureTests.quickTest();
                }
                else {
                    std::cout << "Invalid argument: " << arg2 << std::endl;
                    return 1;
                }
            } else {
                figureTests.run_all_tests();
            }
            return 0;
        } else if(!notInt(str)){ // Run test targetting a figure example
            FuxTest* test = new FuxTest(atoi(argv[1]));
            vector<Species> species = test->getSpList();
            //la do si re do mi fa mi re do si la
            //57 60 59 62 60 64 65 64 62 60 59 57
            vector<int> cantusFirmus = test->getCf();

            int size = cantusFirmus.size();
            vector<int> v_type = test->getVType();

```

```
143             vector<int> melodic_params = {0, 1, 1, 576, 2, 2, 2, 1};
144             //borrow, h-5th, h-octave, succ, variety, triad, direct move, penult rule check
145             vector<int> general_params = {4, 1, 1, 2, 2, 2, 8, 1};
146 
147             //penult sixth, non-ciambata, con m after skip, h triad 3rd species, m2 eq zero, no
    syncopation, pref species slider
148             vector<int> specific_params = {8 , 4 , 0 , 2 , 1 , 8 , 50};
149 
150             vector<int> importance = {8,7,5,2,9,3,14,12,6,11,4,10,1,13};
151 
152             int borrowMode = test->getBMode();
153 
154             // create a new problem
155             // auto* problem = new TwoVoiceCounterpoint(cantusFirmus, species[0], C,
    lower_bound_domain, upper_bound_domain);
156             auto* problem = create_problem(cantusFirmus, species, v_type, melodic_params,
    general_params, specific_params,
157                 importance, borrowMode);
158             //cout << problem->to_string() << endl;
159             // create a new search engine
160             if(atoi(argv[1])==125){
161                 cout << "HERE" << endl;
162                 for(int j = 0; j < problem->getSize(); j++){
163                     if(j!=10){
164                         rel(problem->getHome(), problem->getSolutionArray()[j], IRT_EQ, test->
    getCp()[j]);
165                     }
166                 }
167             }
168             else{
169                 if(species[0]==FIRST_SPECIES || species[0]==THIRD_SPECIES){
170                     for(int j = 0; j < problem->getSize(); j++){
171                         rel(problem->getHome(), problem->getSolutionArray()[j], IRT_EQ, test->
    getCp()[j]);
172                     }
173                 } else if(species[0]==SECOND_SPECIES || species[0]==FOURTH_SPECIES){
174                     for(int j = 1; j < problem->getSize(); j++){
175                         rel(problem->getHome(), problem->getSolutionArray()[j], IRT_EQ, test->
    getCp()[j]);
176                     }
177                 }
178             }
179 
180             BAB<CounterpointProblem> e(problem);
181 
182             int nb_sol = 0;
183             while(CounterpointProblem* pb = e.next()){
184                 nb_sol++;
185 
186                 // cout << int_vector_to_string(cantusFirmus) << endl;
187                 cout << "Solution " << nb_sol << ": " << endl;
188                 cout << pb->to_string() << endl;
189                 delete pb;
190                 if (nb_sol >= 1)
191                     break;
192             }
193             if(nb_sol!=1){
194                 cout << "This test did not pass. An error was found!" << endl;
195             } else {
196                 cout << "This test passed successfully!" << endl;
197             }
198         } else { // Run test targetting a specific constraint
199             FuxTest* test = new FuxTest(argv[1]); // FUXTESTS
200         }
201     } else if(argc==3){ // Run test targetting a figure example by fixing the fisrt n measures
202         FuxTest* test = new FuxTest(atoi(argv[1]), atoi(argv[2]));
203         vector<Species> species = test->getSpList();
204         //la do si re do mi fa mi re do si la
205         //57 60 59 62 60 64 65 64 62 60 59 57
206         vector<int> cantusFirmus = test->getCf();
207 
208         int size = cantusFirmus.size();
209         vector<int> v_type = test->getVType();
210 
211         vector<int> melodic_params = {0, 1, 1, 576, 2, 2, 2, 1};
212         //borrow, h-5th, h-octave, succ, variety, triad, direct move, penult rule check
213         vector<int> general_params = {4, 1, 1, 2, 2, 2, 8, 1};
214 
215         //penult sixth, non-ciambata, con m after skip, h triad 3rd species, m2 eq zero, no
    syncopation, pref species slider
216         vector<int> specific_params = {8 , 4 , 0 , 2 , 1 , 8 , 50};
217 
218         vector<int> importance = {8,7,5,2,9,3,14,12,6,11,4,10,1,13};
219 
220         int borrowMode = test->getBMode();
221 
222         // create a new problem
223         // auto* problem = new TwoVoiceCounterpoint(cantusFirmus, species[0], C,
    lower_bound_domain, upper_bound_domain);
224         auto* problem = create_problem(cantusFirmus, species, v_type, melodic_params,
```

```
    general_params, specific_params,
225             importance, borrowMode);
226         //cout << problem->to_string() << endl;
227         // create a new search engine
228 
229         if(atoi(argv[1])==125){
230             for(int j = 0; j < problem->getSize(); j++){
231                 if(j!=10){
232                     rel(problem->getHome(), problem->getSolutionArray()[j], IRT_EQ, test->getCp
    ()[j]);
233                 }
234             }
235         }
236         else if(atoi(argv[1])==74){
237             for(int j = 0; j < test->getIdx(); j++){
238                 if(j!=0){
239                     rel(problem->getHome(), problem->getSolutionArray()[j], IRT_EQ, test->getCp
    ()[j]);
240                 }
241             }
242         }
243         else if(atoi(argv[1])==82){
244             vector<int> speciesArray =
    {-1,-1,3,-1,3,2,2,2,2,2,2,2,2,2,3,-1,3,2,2,2,2,2,3,-1,3,-1,3,-1,3,2,3,-1,3,2,3,-1,3,-1,3,-1,3};
245             for(int j = 0; j < test->getIdx(); j++){
246                 if(j>1){
247                     rel(problem->getHome(), problem->getSolutionArray()[j], IRT_EQ, test->getCp
    ()[j]);
248                     //rel(problem->getHome(), problem->get_species_array_5sp(0)[j], IRT_EQ, test
    ->getCp()[j]);
249                 }
250             }
251         }
252         else{
253             if(species[0]==FIRST_SPECIES || species[0]==THIRD_SPECIES || species[0]==
    FOURTH_SPECIES){
254                 for(int j = 0; j < test->getIdx(); j++){
255                     rel(problem->getHome(), problem->getSolutionArray()[j], IRT_EQ, test->getCp
    ()[j]);
256                 }
257             } else if(species[0]==SECOND_SPECIES){
258                 for(int j = 1; j < test->getIdx(); j++){
259                     rel(problem->getHome(), problem->getSolutionArray()[j], IRT_EQ, test->getCp
    ()[j]);
260                 }
261             }
262         }
263 
264         BAB<CounterpointProblem> e(problem);
265 
266         int nb_sol = 0;
267         while(CounterpointProblem* pb = e.next()){
268             nb_sol++;
269             cout << "Solution " << nb_sol << ": " << endl;
270             cout << pb->to_string() << endl;
271             // cout << int_vector_to_string(cantusFirmus) << endl;
272 
273             delete pb;
274             if (nb_sol >= 1)
275                 break;
276         }
277         cout << "No (more) solutions." << endl;
278     } else {
279         throw std::invalid_argument("Wrong amount of arguments!");
280     }
281 
282     return 0;
283 }
```

A.11 Build system and command-line interface

A.11.1 Makefile

```
1  # ============================================================
2  # Makefile     FuxCP : g n ration de contrepoints par contraintes
3  # ============================================================
4  # Cibles principales :
5  #   make                                        compile GenerateCounterpoint
6  #   make generate COMBO="2 1"                   g n re un contrepoint (CF=1 par d faut)
7  #   make generate COMBO="2 1" CF=3              avec cantus firmus n 3
8  #   make generate COMBO="3 2 5" CF=2 TIMEOUT=300000 STAGNATION=120000
9  #   make so                                     compile la lib dynamique libgecode.so
10 #   make clean                                  supprime les binaires compil s
```

```
#   make clean_results                          supprime r sultats (2voices/3voices/4voices)
#   make clean_all                              tout supprimer (binaires + r sultats)
#
# Pour lancer des campagnes ou utiliser l'interface interactive :
#   ../fuxcp                                    menu interactif (depuis FuxCP/)
#   ../fuxcp campaign <id>                      lance une campagne (config/campaigns.csv)
# ============================================================

# === DOSSIERS ===
COMPILED_DIR = ../compiled
RESULTS_DIR  = ../../results
OUT_DIR      = ../sources

# === CANTUS FIRMUS (d faut : 1) ===
CF ?= 1

$(shell mkdir -p $(COMPILED_DIR) $(RESULTS_DIR))

# === COMPILATEUR ===
CXX            = g++
CXXFLAGS       = -std=c++11 -O3 -ffast-math -fPIC
CXXFLAGS_DEBUG = -g -std=c++11
LDFLAGS        = -shared

LIBS = -lgecodedriver -lgecodeflatzinc -lgecodefloat -lgecodeint \
       -lgecodekernel -lgecodeminimodel -lgecodesearch -lgecodeset -lgecodesupport

LIBS_MINIMAL = -lgecodekernel -lgecodesearch -lgecodeint \
               -lgecodedriver -lgecodeminimodel -lgecodesupport

# === SOURCES PARTAG ES ===
SHARED_SOURCES = \
  src/Utilities.cpp src/Parts/Voice.cpp src/Parts/Stratum.cpp src/CounterpointProblems/
    CounterpointUtils.cpp \
  src/ConfigLoader.cpp \
  src/CounterpointProblems/CounterpointProblem.cpp \
  src/Parts/Part.cpp \
  src/Parts/FirstSpeciesCounterpoint.cpp \
  src/Parts/SecondSpeciesCounterpoint.cpp \
  src/Parts/ThirdSpeciesCounterpoint.cpp \
  src/Parts/FourthSpeciesCounterpoint.cpp \
  src/Parts/FifthSpeciesCounterpoint.cpp \
  src/Parts/CantusFirmus.cpp \
  src/CounterpointProblems/TwoVoiceCounterpoint.cpp \
  src/CounterpointProblems/ThreeVoiceCounterpoint.cpp \
  src/CounterpointProblems/FourVoiceCounterpoint.cpp \
  src/Parts/constraints.cpp \
  src/Midi.cpp src/Generations.cpp src/GenerationsAPI.cpp

# ============================================================
# CIBLE PAR D FAUT
# ============================================================

all: compile_generate_counterpoint

# ============================================================
# BIBLIOTH QUE DYNAMIQUE LINUX (.so)
# ============================================================

SRC_DIR = src
SRCS    = $(shell find $(SRC_DIR) -name '*.cpp')
OBJS    = $(patsubst $(SRC_DIR)/%.cpp, $(COMPILED_DIR)/%.o, $(SRCS))
TARGET  = $(OUT_DIR)/libgecode.so

$(COMPILED_DIR)/%.o: $(SRC_DIR)/%.cpp
  @mkdir -p $(dir $@)
  $(CXX) $(CXXFLAGS) -c $< -o $@

so: $(TARGET)

$(TARGET): $(OBJS)
  @mkdir -p $(OUT_DIR)
  $(CXX) $(CXXFLAGS) -o $(TARGET) $(OBJS) $(LDFLAGS) $(LIBS)

# ============================================================
# G N RATEUR DE CONTREPOINTS (binaire principal)
# ============================================================

compile_generate_counterpoint:
  @mkdir -p $(COMPILED_DIR)
  $(CXX) $(CXXFLAGS_DEBUG) src/main.cpp $(SHARED_SOURCES) \
      -o $(COMPILED_DIR)/Generations $(LIBS_MINIMAL)

# --- G n ration d'un contrepoint pr cis ---
# Usage : make generate COMBO="<nb_voix> <especes...>" [CF=<n>] [TIMEOUT=<ms>] [STAGNATION=<ms>]
# Exemple : make generate COMBO="2 1"
# Exemple : make generate COMBO="4 1 2 3" CF=2 TIMEOUT=300000 STAGNATION=120000
generate: compile_generate_counterpoint
  $(COMPILED_DIR)/Generations $(COMBO) -c $(CF) $(if $(TIMEOUT),-t $(TIMEOUT),) $(if $(
    STAGNATION),-s $(STAGNATION),) $(if $(VTYPE),-v $(VTYPE),) $(if $(PRESET),--preset $(PRESET)
```

```
    ,)

# ============================================================
# TESTING SUITES
# ============================================================

compile_for_debug:
  @mkdir -p $(COMPILED_DIR)
  $(CXX) $(CXXFLAGS_DEBUG) src/Tests/main.cpp src/Tests/fuxTest.cpp src/Tests/figureTests.cpp $(
    SHARED_SOURCES) -o $(COMPILED_DIR)/Main $(LIBS_MINIMAL)

test: compile_for_debug
  $(COMPILED_DIR)/Main $(name)

test_all: compile_for_debug
  $(COMPILED_DIR)/Main all

# ============================================================
# NETTOYAGE
# ============================================================

clean:
  rm -f $(COMPILED_DIR)/Generations
  rm -rf $(COMPILED_DIR)/CounterpointProblems $(COMPILED_DIR)/Parts
  rm -f $(COMPILED_DIR)/*.o
  rm -f $(TARGET)

clean_results:
  rm -rf $(RESULTS_DIR)/2voices $(RESULTS_DIR)/3voices $(RESULTS_DIR)/4voices

clean_all: clean clean_results
  rm -rf $(COMPILED_DIR)

.PHONY: all so compile_generate_counterpoint generate \
  clean clean_results clean_all
```

A.11.2 fuxcp

```
#!/usr/bin/env bash
# [Claude Code]
# ============================================================
# fuxcp     interface CLI interactive pour la g n ration de contrepoints.
#
# Usage : ./fuxcp                    (menu interactif)
#         ./fuxcp generate <args>    (forward direct au binaire C++)
#         ./fuxcp tail               (suit la derni re g n ration en arri re-plan)
#
# Tous les param tres (CF, presets) sont lus depuis FuxCP/config/*.csv.
# ============================================================

set -uo pipefail

# === Localisation ===
SCRIPT_DIR="$(cd "$(dirname "${BASH_SOURCE[0]}")" && pwd)"
ROOT_DIR="$SCRIPT_DIR"
REPO_DIR="$(cd "$ROOT_DIR/.." && pwd)"
CPP_DIR="$ROOT_DIR/c++"
CONFIG_DIR="$ROOT_DIR/config"
RESULTS_DIR="$REPO_DIR/results"
LOGS_DIR="$RESULTS_DIR/cli_logs"
BIN="$ROOT_DIR/compiled/Generations"

mkdir -p "$LOGS_DIR"

# === Couleurs ===
if [[ -t 1 ]]; then
    C_BOLD=$'\e[1m'; C_DIM=$'\e[2m'; C_RED=$'\e[31m'; C_GRN=$'\e[32m'
    C_YEL=$'\e[33m'; C_BLU=$'\e[34m'; C_MAG=$'\e[35m'; C_CYN=$'\e[36m'; C_OFF=$'\e[0m'
else
    C_BOLD=""; C_DIM=""; C_RED=""; C_GRN=""; C_YEL=""; C_BLU=""; C_MAG=""; C_CYN=""; C_OFF=""
fi

banner() {
    echo "${C_BOLD}${C_CYN}============================================${C_OFF}"
    echo "${C_BOLD}${C_CYN}  $1${C_OFF}"
    echo "${C_BOLD}${C_CYN}============================================${C_OFF}"
}

err() { echo "${C_RED}[ERREUR]${C_OFF} $*" >&2; }
ok()  { echo "${C_GRN}[OK]${C_OFF} $*"; }
inf() { echo "${C_BLU}[INFO]${C_OFF} $*"; }

# === V rifications ===
ensure_binary() {
    if [[ ! -x "$BIN" ]]; then
        err "Binaire absent : $BIN"
        echo "  Compilation en cours..."
```

```
        (cd "$CPP_DIR" && make compile_generate_counterpoint) || { err "Compilation    choue   ";
    exit 1; }
    fi
}

ensure_config() {
    for f in cantus_firmus.csv presets.csv; do
        if [[ ! -f "$CONFIG_DIR/$f" ]]; then
            err "Config manquante : $CONFIG_DIR/$f"; exit 1
        fi
    done
}

# === Lister CF / presets ===
list_cf() {
    banner "Cantus Firmus disponibles"
    "$BIN" --list-cf
}

list_presets() {
    banner "Presets disponibles"
    "$BIN" --list-presets
}

# === G n ration pr cise (interactive) ===
run_precise() {
    banner "G n ration pr cise"
    "$BIN" --list-cf
    echo
    read -rp "ID du cantus firmus [1] : " cf_id
    cf_id="${cf_id:-1}"

    read -rp "Nombre de voix (2/3/4) [2] : " nv
    nv="${nv:-2}"
    if [[ "$nv" != "2" && "$nv" != "3" && "$nv" != "4" ]]; then
        err "nb_voix invalide : $nv"; return 1
    fi

    n_sp=$((nv - 1))
    echo "Esp ces des $n_sp voix de contrepoint (1-5, s par es par espaces)."
    echo "Convention 3v : '1 X' (1 re  esp ce + cible). Convention 4v : '1 1 X'."
    read -rp "Esp ces : " sp_input
    sp_input="${sp_input:-1}"

    "$BIN" --list-presets
    read -rp "Preset [default] : " preset
    preset="${preset:-default}"

    read -rp "Timeout total ms [300000] : " tmo
    tmo="${tmo:-300000}"
    read -rp "Stagnation ms [120000] : " stg
    stg="${stg:-120000}"
    read -rp "Sous-dossier de sortie [precise] : " sub
    sub="${sub:-precise}"

    echo "Mode d'objectif :"
    echo "  lex   = lexicographique par priorit  (d faut , recommand )"
    echo "  total = somme globale (toutes contraintes au m me poids)"
    echo "  mixed = lex pond r  + somme globale"
    read -rp "Mode (lex/total/mixed) [lex] : " obj
    obj="${obj:-lex}"

    log_file="$LOGS_DIR/precise_$(date +%Y%m%d_%H%M%S).log"
    inf "Log : $log_file"

    cmd=("$BIN" "$nv" $sp_input -c "$cf_id" --preset "$preset" -t "$tmo" -s "$stg" -m "$obj" --
    subdir "$sub")
    echo "${C_DIM}Commande :${C_OFF} ${cmd[*]}"
    "${cmd[@]}" 2>&1 | tee "$log_file"
}

# === Surveiller une g n ration en cours ===
monitor() {
    banner "Surveillance des g n rations"
    echo "${C_BOLD}Processus GenerateCounterpoint actifs :${C_OFF}"
    if pgrep -af GenerateCounterpoint > /dev/null; then
        pgrep -af GenerateCounterpoint
    else
        echo "  ${C_DIM}(aucun)${C_OFF}"
    fi
    echo
    echo "${C_BOLD}Dernier log :${C_OFF}"
    last_log="$(ls -1t "$LOGS_DIR"/*.log 2>/dev/null | head -n 1 || true)"
    if [[ -z "$last_log" ]]; then
        echo "  ${C_DIM}(aucun log dans $LOGS_DIR)${C_OFF}"
        return 0
    fi
    echo "  $last_log"
    echo
    echo "Choix :"
```

```
    echo "  1) tail -f en direct (Ctrl+C pour quitter)"
    echo "  2) Voir les 30 derni res lignes"
    echo "  3) Filtrer solutions trouv es ([Sol ...])"
    echo "  4) Filtrer it rations (It ration ...)"
    echo "  5) Filtrer final (STAGNATION/TIMEOUT/EXHAUSTIVE)"
    echo "  6) Compter g n rations termin es"
    echo "  0) Retour"
    read -rp "Choix [1] : " m
    m="${m:-1}"
    case "$m" in
        1) tail -f "$last_log" ;;
        2) tail -n 30 "$last_log" ;;
        3) grep -E "\[Sol " --color=always "$last_log" | tail -n 30 ;;
        4) grep -E "It ration [0-9]+ \|" --color=always "$last_log" | tail -n 50 ;;
        5) grep -E "STAGNATION|TIMEOUT|EXHAUSTIVE|AUCUNE SOLUTION" --color=always "$last_log" |
    tail -n 50 ;;
        6) echo "Solutions trouv es   : $(grep -c '\[Sol ' "$last_log")"
           echo "It rations logg es   : $(grep -c 'It ration [0-9]' "$last_log")"
           echo "Terminaisons         : $(grep -cE 'STAGNATION|TIMEOUT|EXHAUSTIVE' "$last_log")"
     ;;
        *) ;;
    esac
}

# === Explorer les r sultats ===
explore_results() {
    banner "Exploration des r sultats"
    echo "Sous-dossiers de $RESULTS_DIR :"
    mapfile -t subs < <(find "$RESULTS_DIR" -mindepth 1 -maxdepth 1 -type d 2>/dev/null | sort)
    if [[ "${#subs[@]}" -eq 0 ]]; then
        echo "  ${C_DIM}(vide)${C_OFF}"; return 0
    fi
    for i in "${!subs[@]}"; do
        d="${subs[$i]}"
        n_txt=$(find "$d" -name '*.txt' ! -name 'cantus_firmus.txt' 2>/dev/null | wc -l)
        n_err=$(find "$d" -name 'error_*.txt' 2>/dev/null | wc -l)
        n_mid=$(find "$d" -name '*.mid' 2>/dev/null | wc -l)
        printf "  %2d) %-30s  (txt=%d  erreurs=%d  midi=%d)\n" $((i+1)) "$(basename "$d")" "
    $n_txt" "$n_err" "$n_mid"
    done
    echo "  0) Retour"
    read -rp "Choix : " ch
    [[ "$ch" == "0" || -z "$ch" ]] && return 0
    sel="${subs[$((ch-1))]}"
    [[ -z "$sel" || ! -d "$sel" ]] && { err "Choix invalide"; return 1; }
    echo "${C_BOLD}Contenu de $sel :${C_OFF}"
    find "$sel" -name '*.txt' ! -name 'cantus_firmus.txt' | sort | head -n 40
    echo
    read -rp "Fichier txt    afficher (chemin complet, vide=skip) : " f
    [[ -n "$f" && -f "$f" ]] && less "$f"
}

# === Inspecter logs ===
inspect_logs() {
    banner "Inspection des logs"
    mapfile -t logs < <(ls -1t "$LOGS_DIR"/*.log 2>/dev/null)
    if [[ "${#logs[@]}" -eq 0 ]]; then
        echo "  ${C_DIM}(aucun log)${C_OFF}"; return 0
    fi
    for i in "${!logs[@]}"; do
        size=$(du -h "${logs[$i]}" | cut -f1)
        printf "  %2d) %-50s  %s\n" $((i+1)) "$(basename "${logs[$i]}")" "$size"
    done
    echo "  0) Retour"
    read -rp "Choix : " ch
    [[ "$ch" == "0" || -z "$ch" ]] && return 0
    sel="${logs[$((ch-1))]}"
    [[ -z "$sel" || ! -f "$sel" ]] && { err "Choix invalide"; return 1; }
    echo "Action :"
    echo "  1) less"
    echo "  2) tail -f"
    echo "  3) Solutions trouv es ([Sol ...])"
    echo "  4) It rations progressives (It ration ...)"
    echo "  5) Terminaisons (STAGNATION/TIMEOUT/EXHAUSTIVE/AUCUNE)"
    read -rp "Choix [1] : " a
    a="${a:-1}"
    case "$a" in
        1) less "$sel" ;;
        2) tail -f "$sel" ;;
        3) grep -E "\[Sol " --color=always "$sel" | less -R ;;
        4) grep -E "It ration [0-9]+ \|" --color=always "$sel" | less -R ;;
        5) grep -E "STAGNATION|TIMEOUT|EXHAUSTIVE|AUCUNE|EXHAUSTIVE" --color=always "$sel" |
    less -R ;;
    esac
}

# === Menu principal ===
main_menu() {
    while true; do
        clear
```

```
        banner "FuxCP     interface CLI"
        echo "Configuration : ${C_DIM}$CONFIG_DIR${C_OFF}"
        echo "R sultats     : ${C_DIM}$RESULTS_DIR${C_OFF}"
        echo "Binaire       : ${C_DIM}$BIN${C_OFF}"
        echo
        echo "  1) G n ration pr cise (CF + voix + esp ces + preset interactif)"
        echo "  2) Surveiller une g n ration en cours (tail/grep/processus)"
        echo "  3) Explorer les r sultats (txt/csv/midi)"
        echo "  4) Inspecter les logs"
        echo "  5) Lister CF disponibles"
        echo "  6) Lister presets"
        echo "  7) Recompiler le binaire"
        echo "  0) Quitter"
        echo
        read -rp "Choix : " ch
        case "$ch" in
            1)  run_precise ;;
            2)  monitor ;;
            3)  explore_results ;;
            4)  inspect_logs ;;
            5)  list_cf ;;
            6)  list_presets ;;
            7)  (cd "$CPP_DIR" && make compile_generate_counterpoint) ;;
            0)  exit 0 ;;
            *)  err "Choix invalide" ;;
        esac
        echo
        read -rp "Appuyer sur Entr e pour revenir au menu..." _dummy
    done
}

# === Entr e principale ===
ensure_config

case "${1:-}" in
    "")           ensure_binary; main_menu ;;
    generate)     ensure_binary; shift; "$BIN" "$@" ;;
    tail)         last="$(ls -1t "$LOGS_DIR"/*.log 2>/dev/null | head -n1 || true)"
                  [[ -n "$last" ]] && tail -f "$last" || err "Aucun log" ;;
    list-cf)      ensure_binary; list_cf ;;
    list-presets) ensure_binary; list_presets ;;
    -h|--help|help)
        echo "Usage : $0 [generate <args> | tail | list-cf | list-presets]"
        echo "        (sans argument = menu interactif)" ;;
    *)            err "Commande inconnue : $1"; exit 1 ;;
esac
```

A.12 Configuration data

A.12.1 cantus*firmus.csv*

```
# Registre des cantus firmus disponibles. Source unique de v rit .
#
# Colonnes :
#   id          : identifiant entier (1..6)
#   notes       : notes MIDI s par es par des espaces (toute la liste dans une seule cellule)
#   name        : libell  court affich  dans la CLI et utilis  comme  tiquette
#   scale       : indicateur (informatif) de la gamme attendue
#                 Note : la gamme effective est toujours d tect e automatiquement
#                 par le moteur (detect_scale_for_cf), cette colonne sert uniquement
#                 d'aide au choix du CF dans la CLI.
#   description : commentaire libre
#
# S lection : 6 CF de r f rence couvrant les six modes principaux.
# Convention de saisie utilis e :
#   - sans suffixe = octave de base de la cl  de sol (do=60, r  =62, ..., si=71)
#   - suffixe 2    = octave en dessous (la2=57, si2=59, sol2=55)
#   - suffixe 3    = meme octave que sans suffixe (do3=60, la3=69), notation alternative
id,notes,name,scale,description
1,62 65 64 62 67 65 69 67 65 64 62,Re Dorien,Dorien,CF 11 notes mode dorien sur re
2,64 60 62 60 57 69 67 64 65 64,Mi Phrygien,Phrygien,CF 10 notes mode phrygien sur mi (cadence phrygienne fa vers mi)
3,65 67 69 65 62 64 65 60 69 65 67 65,Fa Lydien,Lydien,CF 12 notes mode lydien sur fa (triton fa-si gestion musica ficta)
4,55 60 59 55 60 64 62 67 64 60 62 59 57 55,Sol Mixolydien,Mixolydien,CF 14 notes mode mixolydien sur sol
5,57 60 59 62 60 64 65 64 62 60 59 57,La Eolien,Mineur naturel,CF 12 notes mode eolien sur la (mineur naturel)
6,60 64 65 67 64 69 67 64 65 64 62 60,Do Ionien,Majeur,CF 12 notes mode ionien sur do (majeur moderne)
7,60 62 65 64 67 65 64 62 60,Do Reference Court,Majeur,CF 9 notes reference utilisateur (do re fa mi sol fa mi re do)
8,48 50 53 52 55 53 52 50 48,Do Reference Grave,Majeur,CF 9 notes reference utilisateur (do re fa mi sol fa mi re do)
```

A.12.2 presets.csv

```
# Presets de param tres pour la g n ration de contrepoints.
#
# Chaque preset d finit l'ensemble des poids/co ts utilis s par le solveur.
# Les listes sont s par es par des espaces. Toutes les colonnes sont obligatoires.
#
# Colonnes :
#   name             : identifiant texte du preset (utilis  via --preset NAME)
#   melodic_params   : 8 entiers - [unisson, m2, M2, sauts, m3, M3, P4, sup4]
#   general_params   : 8 entiers - [octave, P5, m6, M6, P8, mvtCommun, variete, triade]
#   specific_params  : 7 entiers - [parfaite, faiblet, mvtDirect, syncope, cambiata, melodique, fux2]
#   importance       : 14 entiers - ordre lexicographique des co ts
#                      [borrow, fifth, octave, succ, variety, triad, direct, motion,
#                       penult, cambiata, triad3, m2, syncopation, melodic]
#   borrow_mode      : 0=interdit emprunts | 1=autorise emprunts (mode courant) | 2=mineur emprunt
#   description      : texte libre (affichage uniquement, pas de virgule)
name,melodic_params,general_params,specific_params,importance,borrow_mode,description
default,0 1 2 576 5 10 25 40,4 1 1 2 2 2 8 1,8 4 0 2 1 8 50,8 7 5 2 9 3 14 12 6 11 4 10 1 13,0,Preset par defaut equilibre
borrowed,0 1 1 576 2 2 2 1,4 1 1 2 2 2 8 1,8 4 0 2 1 8 50,8 7 5 2 9 3 14 12 6 11 4 10 1 13,1,Preset par defaut avec borrow mode
strict,0 1 2 576 5 10 25 40,4 1 1 2 2 2 8 1,8 4 0 2 1 8 50,8 7 5 2 9 3 14 12 6 11 4 10 1 13,0,Sans emprunts melodiques contraints
musique,0 1 1 576 2 2 2 1,4 1 1 2 2 2 8 1,8 4 0 2 1 8 50,4 3 2 2 9 3 7 6 3 11 4 10 1 13,1,Importance accrue mvts/octaves/melodique
variete_haute,0 1 1 576 2 2 2 1,4 1 1 2 2 2 16 1,8 4 0 2 1 8 50,8 7 5 2 9 3 14 12 6 11 4 10 1 13,1,Co t de variete double pour eviter repetitions
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**