"Melodizer 2.0 : A Constraint Programming Tool For Computer-aided Musical Composition"

Chardon, Clément ; Diels, Amaury ; Gobbi, Federico

ABSTRACT

This master's thesis presents the design of a tool destined to assist musical composers in the creation of their next masterpiece. The composers state the musical ideas that they want to include in their themes which are translated into a Constraint Satisfaction Problem. This thesis develops two already ex- isting master's thesis. The first one, written by Baptiste Lapière, was more rhythm-oriented [10]. While the second one, written by Damien Sprockeels, was more focused on pitch-oriented scenarios [28]. Therefore, we combined both works to create a tool that allows to play with pitches and rhythms simultaneously. On the one hand, Gecode is a powerful C++ toolkit that is used in order to model and solve Constraint Optimization Problems. While, on the other hand; OpenMusic, based on Lisp, serves as the visual programming and composition environment where Melodizer 2.0 is employed. GiL works as the bridge between Gecode and Lisp that allows us to solve Constraint Satisfaction Problems in Openmusic. Melodizer 2.0 provides an intuitive interactive interface that works as a melody synthesizer with many knobs and buttons to tweak in search of inspiring results. We do not pretend to replace musician's creativity nor come up with a full master- piece when launched. Nevertheless, it stimulates songwriters in their production process. If you are an inspired compositor that is eager to use Melodizer 2.0 we recommend you to go directly to chapters 6 and 7 where we explain how to manipulate the interface, and, provide plenty of musical scenarios to picture the different uses a...

CITE THIS VERSION

UCLouvain

École polytechnique de Louvain

Melodizer 2.0 : A Constraint Programming Tool For Computer-aided Musical Composition

Authors: **Clément Chardon, Amaury Diels , Federico Gobbi**
Supervisor: **Peter Van Roy**
Readers: **Augustin Delecluse, Karim Haddad**
Academic year 2021–2022
Master [120] in Computer Science and Engineering

Abstract

This master's thesis presents the design of a tool destined to assist musical composers in the creation of their next masterpiece. The composers state the musical ideas that they want to include in their themes which are translated into a Constraint Satisfaction Problem. This thesis develops two already existing master's thesis. The first one, written by Baptiste Lapière, was more rhythm-oriented [10]. While the second one, written by Damien Sprockeels, was more focused on pitch-oriented scenarios [28]. Therefore, we combined both works to create a tool that allows to play with pitches and rhythms simultaneously.

On the one hand, Gecode is a powerful C++ toolkit that is used in order to model and solve Constraint Optimization Problems. While, on the other hand; OpenMusic, based on Lisp, serves as the visual programming and composition environment where Melodizer 2.0 is employed. GiL works as the bridge between Gecode and Lisp that allows us to solve Constraint Satisfaction Problems in Openmusic.

Melodizer 2.0 provides an intuitive interactive interface that works as a melody synthesizer with many knobs and buttons to tweak in search of inspiring results. We do not pretend to replace musician's creativity nor come up with a full masterpiece when launched. Nevertheless, it stimulates songwriters in their production process.

If you are an inspired compositor that is eager to use Melodizer 2.0 we recommend you to go directly to chapters 6 and 7 where we explain how to manipulate the interface, and, provide plenty of musical scenarios to picture the different uses and the musical relevance of our instrument.

We would like to express gratitude to
Peter Van Roy,
Karim Haddad from IRCAM,
Augustin Delecluse,
Vanessa Maons and the INGI System Team,
Damien Sprockeels,
Baptiste Lapière
The Organizers and participants of the IRCAM Forum
for the help they provided throughout the production of this master's thesis.

Contents

1 Introduction **1**
- 1.1 Context and motivation . . . 1
- 1.2 Previous work and Melodizer's main upgrades . . . 2
- 1.3 Implementation procedure . . . 3
- 1.4 Playing with Melodizer 2.0 as a composer . . . 4
- 1.5 Contributions . . . 5
  - 1.5.1 System contributions . . . 5
  - 1.5.2 Musical contributions . . . 7
- 1.6 Melodizer 2.0 example . . . 8
- 1.7 Roadmap . . . 10

2 Theoretical framework **12**
- 2.1 What is music after all ? . . . 12
- 2.2 Music Theory . . . 13
  - 2.2.1 Music terminology . . . 14
  - 2.2.2 Rhythm . . . 17
  - 2.2.3 Melody . . . 17
  - 2.2.4 Harmony . . . 18
  - 2.2.5 Musical Forms to structure your piece . . . 20
- 2.3 Music Composition . . . 21
  - 2.3.1 Tip 1 : Let the listener rest . . . 21
  - 2.3.2 Tip 2 : Tension and release . . . 21
  - 2.3.3 Tip 3 : Coming back home . . . 22
  - 2.3.4 Tip 4 : Popular chord progressions . . . 22
  - 2.3.5 Tip 5 : Repetition with a twist . . . 22
  - 2.3.6 Tip 6 : Arpeggio . . . 22
- 2.4 Constraint Programming . . . 22
  - 2.4.1 Definitions . . . 23
  - 2.4.2 Example 1 : Solving a sudoku . . . 24
  - 2.4.3 Constraint Propagation . . . 27
  - 2.4.4 Branching heuristics . . . 28
  - 2.4.5 Exploration and search engines . . . 30
  - 2.4.6 Branch-and-Bound (BAB) . . . 33

3 Tools **36**
3.1 Gecode . . . 36
3.1.1 Search Space . . . 36
3.1.2 Variables . . . 37
3.1.3 Constraints . . . 38
3.1.4 Propagators . . . 41
3.1.5 Branching . . . 42
3.1.6 Search . . . 44
3.2 OpenMusic . . . 45
3.2.1 Boxes within Patches . . . 45
3.2.2 How to represent score sheets in OM . . . 46
3.2.3 Box evaluation . . . 51

4 GiL **53**
4.1 How does it work ... briefly . . . 53
4.1.1 Lisp Wrapper . . . 53
4.1.2 C Wrapper . . . 54
4.2 New features . . . 54
4.3 How to use GiL . . . 58
4.4 How to improve GiL yourself . . . 61

5 Melodizer 2.0 **62**
5.1 What is Melodizer ? . . . 62
5.1.1 New features . . . 62
5.2 Variable structure . . . 63
5.3 Blocks . . . 65
5.3.1 Block definition . . . 65
5.3.2 Blocks connection . . . 66
5.4 Musical constraints . . . 69
5.4.1 Blocks' general constraints . . . 69
5.4.2 Rhythm constraints . . . 72
5.4.3 Pitch constraints . . . 76
5.5 Branch and bound . . . 81
5.6 Solver . . . 83
5.6.1 Branching heuristics . . . 84
5.7 Implementation structure . . . 86
5.7.1 block.lisp . . . 87
5.7.2 melodizer-csp.lisp . . . 87
5.7.3 melodizer-csts.lisp . . . 88
5.7.4 melodizer-utils.lisp . . . 88

6 User Manual **90**
6.1 Block object . . . 90
6.1.1 Block constraint panel . . . 91
6.1.2 Time constraint panel : . . . 92
6.1.3 Pitch constraint panel . . . 93

6.2 Search object . . . 95
6.3 Connecting blocks to form a structured piece . . . 95

7 Making music with Melodizer 98
7.1 Scenario 1 : Playing with a chord . . . 98
7.1.1 Description . . . 98
7.1.2 Patch set up . . . 98
7.1.3 Modus operandi . . . 99
7.2 Scenario 2 : Playing with two chords . . . 100
7.2.1 Description . . . 100
7.2.2 Patch set up . . . 100
7.2.3 Modus operandi . . . 101
7.3 Scenario 3 : Melody on top of chords . . . 103
7.3.1 Description . . . 103
7.3.2 Patch set up . . . 103
7.3.3 Modus operandi . . . 103
7.4 Scenario 4 : Blues in C Major . . . 105
7.4.1 Description . . . 105
7.4.2 Patch set up . . . 106
7.4.3 Modus operandi . . . 107
7.5 Scenario 5 : The strumming effect . . . 107
7.5.1 Description . . . 107
7.5.2 Patch set up . . . 108
7.5.3 Modus operandi . . . 108
7.6 Scenario 6 : Unexpected results . . . 109
7.6.1 Description . . . 109
7.6.2 Patch set up . . . 109
7.6.3 Modus operandi . . . 109

8 Conclusion 111
8.1 Melodizer 2.0 major achievements . . . 111
8.1.1 Necessary steps to develop Melodizer 2.0 . . . 111
8.2 Further improvements and using Melodizer 2.0 as a cornerstone . 112
8.2.1 Some general ideas . . . 112
8.2.2 Extending the block structure . . . 113
8.2.3 A final word about musical constraints . . . 115

Bibliography 116

A How to install Melodizer 2.0 118
A.1 Download and install . . . 118
A.2 Loading the libraries to OpenMusic . . . 118

B Gecode source code 121
B.1 Sudoku propagation example . . . 121

C Gil source code **122**
C.1 C Wrapper . . . 122
C.1.1 space_wrapper.hpp . . . 122
C.1.2 space_wrapper.cpp . . . 130
C.1.3 gecode_wrapper.hpp . . . 142
C.1.4 gecode_wrapper.cpp . . . 148
C.2 Lisp Wrapper . . . 156
C.2.1 gecode-wrapper.lisp . . . 156
C.2.2 gecode-wrapper-ui.lisp . . . 169

D Melodizer source code **176**
D.1 block.lisp . . . 176
D.2 melodizer-csp.lisp . . . 188
D.3 melodizer-csts.lisp . . . 195
D.4 melodizer-utils.lisp . . . 198

Chapter 1

Introduction

1.1 Context and motivation

Digital revolution began in the latter half of the 20th century. It is not a secret that, computers, Internet and mobile phone devices became increasingly common whether we like it or not. Computers not only became an imperative tool in the work environment but also became widely used for entertainment purposes. Music was not an exception [13].

Even the most classical composers have to use a computer at some given point. Whether it is to provide various sounds to work with, print out parts quickly and neatly, record and polish musical theme, or even for uploading the final masterpiece [9]. These days, you can even create and "perform" a symphony without touching a musical instrument. One of the applications of computers in music is Computer-Assisted Composition (CAC). Our thesis focuses on this particular field of music informatics that aims at generating scores from computer programs.

Therefore, the main goal of this master thesis is to provide a tool destined to assist composers by giving them the computational power of constraint programming. The name of this tool as you could have guessed by the name of the thesis' title is Melodizer 2.0. This tool succeeds Melodizer implemented by Damien Sprockeels [28].

Constraint programming is one of the closest paradigms to what would be considered as the “holy grail of programming" where the only task for the programmer is to state the problem and the computer will find a solution. For instance, the composer simply has to state the musical constraints they wants to include, depending on his preferred genre and compositing style.

Gecode[1], a powerful C++ toolkit used to model and solve Constraint Opti-

[1]https://www.gecode.org/doc/6.2.0/reference/index.html

mization Problems, serves as the backbone of Melodizer 2.0 . OpenMusic[2] (OM), developed at IRCAM[3], is the visual programming and composition environment where Melodizer 2.0 is used. Finally, GiL[4] works as the bridge between Gecode and Lisp that allows us to solve Constraint Optimization Problems in Openmusic.

1.2 Previous work and Melodizer's main upgrades

This thesis was built on top of two previously written masters' thesis. The first one, written by Baptiste Lapière, conceived GiL and provided the Rhythm-Box tool that, as the name indicates, generates rhythms. The second one, written by Damien Sprockeels, improved GiL and created Melodizer; a tool that generates pitch variations with an interactive interface in Openmusic. Therefore, Melodizer 2.0 combines both works in an attempt to generate scores that take into account both pitch and rhythm constraints. This was a major objective since a brilliant pitch sequence without a good rhythm can sound dull. And inversely, a catchy rhythm without pitch won't make a melodious song. Composers don't separate rhythm and pitch. On the contrary, they try to marry them together so as to express the musical piece they had in their head. It was thus essential for Melodizer 2.0 to allow composers to specify rhythmic and melodic constraint simultaneously.

The second major improvement that Melodizer experienced was its new capacity to generate polyphonic themes. In fact, the older version could only originate simple melodies and was hence exclusive for voice and monophonic instruments representation. This restricts considerably the amount of different applicable scenarios. Melodizer 2.0 is considerably more multi-functional. It can represent polyphonic instruments such as piano and guitar, several monophonic instruments playing simultaneously, or even a melody accompanied by a harmonic part.

Additionally, a common strategy used by composers is to introduce some variation in their songs not to bore the listeners with a repetitive sound. This is why composers like to alternate tempos, moving from a slow-paced melody to a faster-paced one and inversely. As well as to change from one mode or key to another. There are many practices producers use to introduce contrast and surprise to the audience. It was thus crucial that Melodizer 2.0 allowed to add different musical constraints to different fragments of the song. Thanks to the Block structure we defined and implemented, the user can couple different musically constrained segments into a whole masterpiece. To be more precise, one Block represents a constrained musical segment where its length is decided by the composer. The developed Blocks take advantage of the visual environment

[2]http://repmus.ircam.fr/openmusic/home
[3]https://www.ircam.fr
[4]https://github.com/sprockeelsd/GiLv2.0

provided by OpenMusic which allows to easily connect “boxes".

Furthermore, when composing for an orchestra, a band or a choir, we have to combine different instruments and voices. Each of these has its own specificities (such as its tessitura or whether it is a diatonic or chromatic instrument) that can be represented by musical constraints. Moreover, a composer could, for example, consider to integrate to his piece an harmonic part that follows a genre-specific chord progression, a melody with a given direction and a counter-melody with a different direction. For this reason, it was important that Melodizer 2.0 could generate different musical ideas, each with its specific constraints, played simultaneously into a whole symphonic piece. Thereby, the introduced Blocks can also represent musical constrained segments that are going to be played synchronously.

Finally, it was important that the solutions provided were diverse enough. Depending on pleasant the solution generated by the tool, the composer can decide what percentage they would like to change from one solution to another. This is why we chose the Branch-and-Bound exploration strategy. The Branch-and-Bound allows to add constraints whenever a feasible solution is found. As a consequence, this exploration strategy not only can be used to generate diverse solutions but can also recreate more musical scenarios. This exploration strategy along with some applicable musical scenarios is explained in details in section 2.4.6.

1.3 Implementation procedure

The implementation of Melodizer 2.0 follows the logical cycle path presented hereinafter:

- First of all, as we explain in detail in chapter 5, we conceived an entire new model in Gecode that allowed us to state a Constraint Satisfaction Problem to generate musical polyphonic solutions with the composer's pitch and rhythm constraints. Within this framework, we translated the musical general rules[5] and ideas into mathematical constraints. Furthermore, we implemented the base structure that allowed block connection to combine musical phrases to be played sequentially or simultaneously. Also, we introduced Branch and Bound to generate diverse solutions efficiently.

- Second of all, we introduced our model, all the constraints and Branch and Bound to GiL, the interface between Gecode and Lisp.

- Third of all, we created an interactive user-friendly interface in OpenMusic that allows composers to easily choose amongst the implemented musical constraints, the ones that they would like to incorporate into his theme. We also developed the Block objects that can eventually be connected to

[5]Even though rules in music are actually meant to be broken

recreate a structured piece with different constrained parts played together or successively.

It may seem like a sequential implementation procedure to follow, but it is crucial to notice that we used the term “cycle", as shown in figure 1.1. Despite the fact that there is a sequence we must follow, the origin of our ideas did not necessarily follow these specific steps. If we eventually came up with the idea to add a button to Melodizer 2.0 that represents a given musical constraint (only increasing pitch melody, for example); then, in order to implement this, we had to translate this musical constraint into a mathematical constraint using Gecode. Next, we had to introduce it to GiL so as to create the bridge between Gecode and Lisp, for the button to be finally functional.

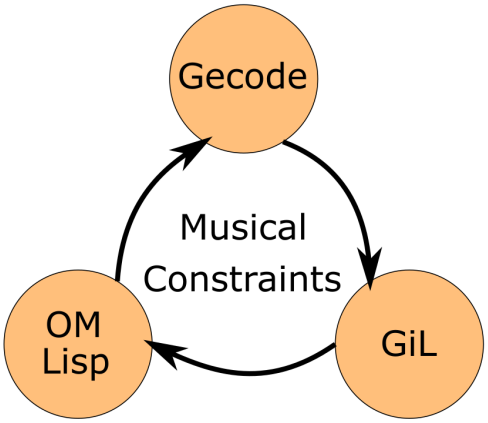


Figure 1.1: Implementation cycle

1.4 Playing with Melodizer 2.0 as a composer

Once we had a base model with some musical constraints, it was finally time for the most expected part: testing Melodizer 2.0 by creating our own music as shown in chapter 7. This was the most creative part which allowed us to discover more musical constraints to add and improve the overall interface to provide a more comfortable user experience. This phase can be pictured with figure 1.2.

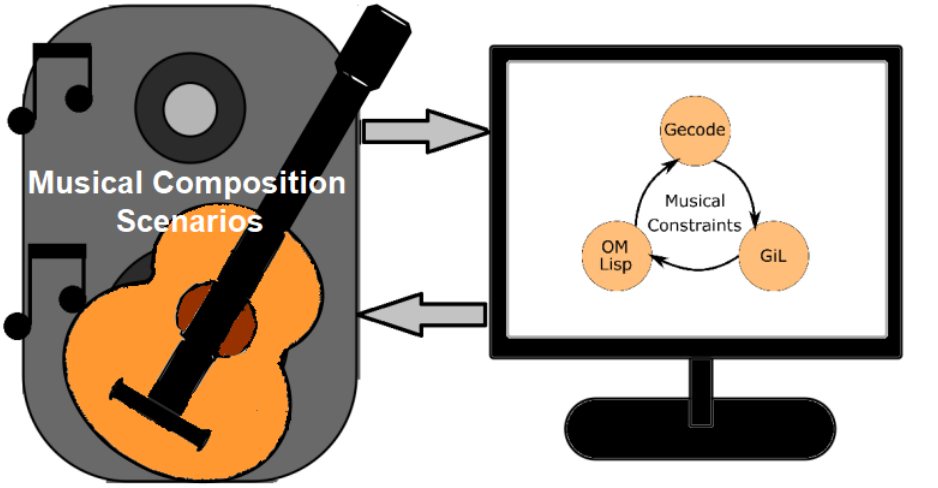


Figure 1.2: From music creation to Melodizer 2.0 development

1.5 Contributions

1.5.1 System contributions

The system contributions that allowed us to develop Melodizer 2.0 can be grouped into three categories :

Create a Constraint Optimization Problem with Gecode as a foundation:

- A new model that allowed us to generate pitch and rhythmic constrained polyphonic themes was developed. This model makes it easy to incorporate different constrained parts throughout the piece in order to surprise the listener. For these reasons, since arrays can easily be concatenated, and you can combine two sets with the union operator, we chose integer set variables arrays to represent the musical partitions. Then, it is also possible to decide the degree of polyphony by constraining the cardinality of the sets. This is explained in detail in chapter 5.

- The Block structure was defined and implemented (explained in section 5.3). Blocks represent constrained parts of the theme. Blocks can be juxtaposed to create variation throughout the piece. For example, we can use two different Blocks to pass from a C Major tonality to an E minor. Blocks can also be superposed to play several instrumental parts, each with its own constraints, simultaneously. Furthermore, Blocks can be repeated throughout the piece. For instance, in the classical ragtime song form AA BB A C, the part A can be represented by a Block which is repeated three times.

- General, rhythmic and pitch constraints were translated into mathematical set constraints as explained in section 5.4.

- By comparing the different search engines provided by Gecode, we concluded that the branch-and-bound is more performant and multi-functional. It is thus the exploration strategy used by the Search. As a result, it helped us to provide more diverse solutions to the users and to represent more musical scenarios. Refer to section 2.4.6 and 5.6 for more information.

Extend GiL, the interface that allows to use Gecode in Lisp (the list presented hereunder is explained in details in chapter 4 entirely dedicated to GiL) :

- Previously, GiL was only used for integer variable problems. Because of that, we not only added integer set variables and integer set variables arrays to the library, but we also included the constraint and branching strategies that are exclusive to set variables.

- Alongside, we have also incorporated to GiL the useful constraints for set variables and set variables arrays enumerated in section 4.2.

- We have also broadened the constraint's catalogue by adding reified constraints. This was useful, to constrain only sets that weren't empty for example. Reified constraints allows to perform if-clauses on variables in constraint programming.

- We introduced the variable and value selection strategies for set variable arrays enumerated in section 3.1.5.

- As only depth-first-search was accepted by GiL, we had to integrate branch-and-bound to allow constraint addition whenever a solution is found. This not only enlarges the achievable musical scenarios but also allows to have more diversity from one solution to another.

- Previously, GiL would only work on MacOS. To introduce GiL and thus Melodizer 2.0 to the Open Source world we also made GiL Linux compatible.

OpenMusic interface, musical environment for Melodizer 2.0 :

- We have created an intuitive interface where the compose can easily state the characteristics of the desired piece. The different musical constraints can be selected by using buttons, check-boxes, sliders and list boxes. This is described in detail in chapter 6.

- We have also allowed to connect different Block object boxes throughout their inlets and outlets to generate a structured piece with different sections, each with its specificities and own constraints. Many examples are shown in chapter 7.

- We have proposed different exploration options through the search box object as explained in section 6.2.

1.5.2 Musical contributions

The main musical contributions of Melodizer 2.0 compared to its previous version are :

- The new faculty of generating polyphonic pieces as shown in section 7. Contrarily to the previous version that would only generate monophonic melodies.
- Its ability to combine both rhythm and pitch constraints. The previous tool would only present pitch related constraints. This is explained in section 5.4 and 6.1.
- The definition and implementation of Block and Search objects that allows you to structure your piece. The composer can now combine Blocks, each one with its specific constraints, to be played simultaneously or consecutively. This opens the door to many new musical scenarios compared to the previous version where the same constraints were applied to the whole piece. Blocks, similarly to musical phrases, can be repeated. For instance, in the basic ternary form ABA, parts A and B can be represented by two different Blocks and we juxtapose Block A, Block B and then repeat Block A. You can find more information about blocks in section 5.3.
- Melodizer 2.0 guarantees that the solver provides diverse solutions. In fact, the users can now decide what is the percentage that they would like to change from one solution to another. This is a big improvement compared to the previous version of Melodizer that provided solutions where only one note would change. In order to achieve this, the Branch-and-Bound search engine was introduced to Melodizer 2.0. This is developed in section 5.6.

We have also produced some musical pieces in order to test Melodizer 2.0. This also serve as an example for producers that are eager to test our tool. Please refer to chapter 7 for more information. The introduced scenarios are :

- The first scenario in section 7.1 is mainly used as a basic example where it shows what can be done with a single Block representing a chord and a Search object.
- The second scenario in section 7.2 serves as an example of how two Blocks can be concatenated to form a chord progression.
- The third scenario in section 7.3 shows how to generate a melody accompanied by a chord progression. This is the first example that combines both Blocks juxtaposed, to form a chord progression, and Blocks superposed to simultaneously the melody and chord progression.
- The fourth scenario in section 7.4 illustrates how the 12-bar-blues can be recreated using the same Block for several parts of the piece.

- The fifth scenario in section 7.5 recreates a guitar strumming chords.
- The sixth scenario in section 7.6 presents how to generate original solutions with the help of the Search Block that utilizes Branch-and-Bound.

1.6 Melodizer 2.0 example

In this section we consider the example illustrated in figure 1.3, to give a taste of how to use Melodizer 2.0 and what it can do. You can find more examples in chapter 7. In this scenario the composer wants to generate a melody on top of a chord progression. Figure 1.3 shows how the Blocks and the Search object should be connected in the OpenMusic's patch environment. The Blocks, illustrated by the boxes that have four inlets and four outlets, represent constrained parts of the theme. The Blocks can be parts that will be played sequentially such as the 4 chords Blocks that are passed to the chord progression Block. Or rather, they can also represent parts that will be played simultaneously as for the chord progression and the melody Blocks. The Search object (illustrated by the box that has three inlets and outlets) is in charge of searching the solution of the Constraint Optimization Problem. Figure 1.4 presents the solution provided and figure 1.5 displays the editor interface of a Block with the available musical constraints. This example is explained in detail in section 7.3.

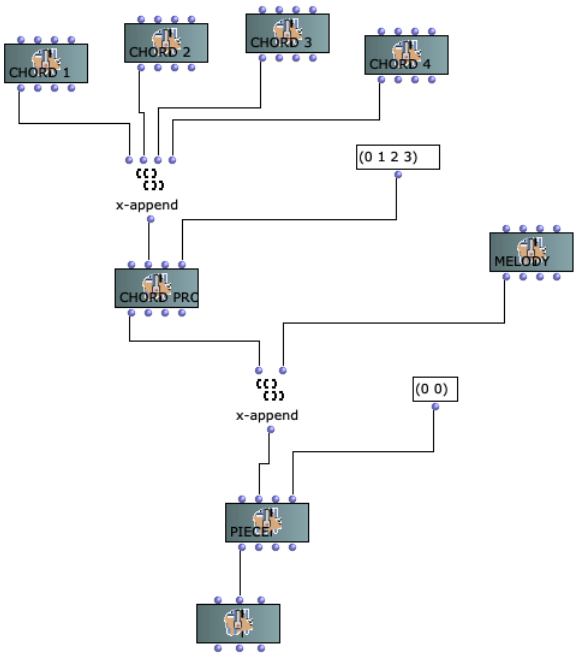


Figure 1.3: Patch setup with the Block and Search objects connected

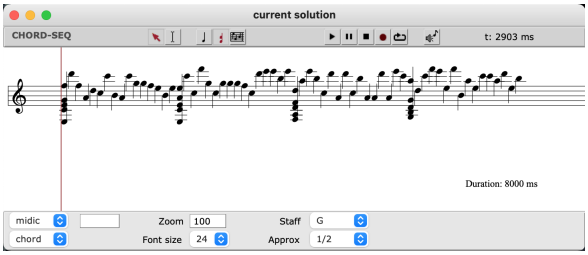


Figure 1.4: Solution provided by the Search object

Figure 1.5: Available musical constraints in the Block's editor interface

1.7 Roadmap

This thesis covers the following topics :

- Chapter 2 captures the two theoretical frameworks that are used throughout the entire thesis. The first part of the chapter gives an insight about the three main pillars of music theory, which are rhythm, melody, and harmony. It also provides some useful tips that could be used by composers. The second part of the chapter focuses on Constraint Satisfaction Problem and describes some useful notions such as constraint propagation, branching heuristics and exploration algorithms such as Depth-First Search and Branch-and-Bound.

- Chapter 3 describes the two main programming tools that allowed us to build Melodizer 2.0, namely Gecode and OpenMusic. On the one hand Gecode, as we already explained, is a powerful C++ toolkit that solves Constraint Satisfaction Problems. We explain the different types of variables, constraints, propagators, branching strategies, and search engines supported by Gecode. On the other hand, OpenMusic is a visual programming environment for Computer Assisted Composition. We demonstrate which are the different objects available to represent score sheets and how to generate them by using midicent lists and rhythm trees. Furthermore, we explain how to utilize inlets and outlets and describe how the evaluation of an object is performed, or, in other words, how the class is calling its inner function and parameters.

- Chapter 4 covers how to bring the constraint solver from Gecode to Lisp by using the interface GiL. We reveal how to use GiL and which features had to be added to the previous GiL versions done by Baptiste Lapière and Damien Sprockeels [10] [28].

- Chapter 5 covers the implementation architecture of Melodizer 2.0 . There are several points to be explained: the modeling choices, how the musical constraints were translated to mathematical constraints, how the solver is exploited; and how the connection of the Blocks and the Search works.

- Chapter 6 serves as a user manual. This is definitely the first chapter that a composer with little interest in programming should read. It explains what each button, sliders and other features of the interface do. Moreover, we provide a detailed explanation on how to interconnect the different modules and what arguments (such as lists and objects) can be passed as inlets.

- Chapter 7 shows musical composition scenarios with Melodizer 2.0 . We explain what was the intention of the composer and how did they use our tool to make his musical idea come true. Again, we recommend for composers that aren't computer savvy to read directly chapter 6 as well as this one.

- Chapter 8 summarizes the master's thesis work and provides some further improvement ideas that could eventually be added to Melodizer 3.0 .

Chapter 2

Theoretical framework

2.1 What is music after all ?

In a video released in 2019 by the media The Daily Wire, the very controversial American personality Ben Shapiro declared that Rap/Hip-Hop music could in fact not be considered as music. Whereas, in 2017, two years earlier; according to a study done by Nielsen Music on trends in the music industry, Hip-Hop/R&B was the dominant musical genre in the US. This raises a very interesting question: how can people have such different opinions on whether a certain three-ish minute sound should be considered as music?

To answer that question, it might be useful to come up with a definition of music that everyone agrees on. Music is an art that uses sound as a channel for inducing an emotion to the listener. This definition is correct but also too broad. Poetry, sound effects in movies or even ASMR both fall within this definition [1] and they are still not quite like music. Narrowing down the scope of the definition is actually quite difficult as it will quickly exclude some ancient, actual or even future musical genres. Still, there seems to be a consensus on three major components of music, three essential pillars without which a sounding creation will not stand as "music" : rhythm, melody and harmony.

Rhythm is the involvement of time in music. If we think of time as a one-dimension line, then rhythm is the set of positions of musical events, that is, notes and silences, in that timeline. More precisely, rhythm is the relative position in time of all components of a piece of music.

What melody is to pitch or frequency is quite similar to what rhythm is to time. Melody is the set of frequencies of notes and interval of frequencies between those. We can think of the combination of rhythm and melody as a 2-D graph where the X-axis represents time and the Y-axis the frequency domain. Figure 2.1 shows what a simple melody could look like in a 2-D space.

[1] Although saying ASMR is an art is quite a bold statement

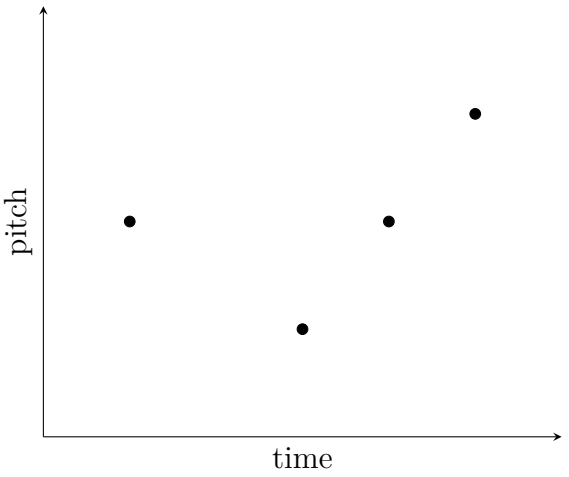


Figure 2.1: A simple melody in pitch-time space.

Finally, harmony is the relation between frequencies or notes playing together. Most instruments playing one note will actually play more than one frequency simultaneously. Those multiple played frequencies harmonize together and shape the sound that we hear from them. Similarly, when an instrument plays multiple notes at the same time, those notes will harmonize together.

Now that we understand the three essential pillars of music, it is easy to notice that every piece of music has different levels of complexity of rhythm, melody and harmony. Rock music usually has lower rhythmic and harmonic complexity, and mainly focuses on melody. An EDM [2] song will probably have lower rhythmic, melodic and harmonic complexity than a jazz song. This brings us back to the initial question of this section : how can people disagree on whether something is or is not music ? Well, they will have different tolerance regarding the complexity needed for something to be considered as music. With all that said, the most important thing to remember is that music doesn't need at all to be complex to move people.

2.2 Music Theory

The concept of music theory can be a bit misleading. Associating the artistic nature of music with such a scientific, even mathematical idea of a theory may seem odd. Is music theory considered a set of rules or principles to tell musicians what to do and not to do when playing music ? Definitely not. Music theory is a tool, a language that musicians can use to communicate about music. In this section, we are going to be looking at the basic principles of music theory through the three pillars of music.

[2]Electro Dance Music

In order to write the Music Theory and Music Composition sections, we were particularly inspired by :

- The book *Théorie de la musique* written by Adolphe Danhauser [4] .
- The book *Vingt leçons d'harmonie pour comprendre et composer la musique* written by Jean-Louis Foucart [6].
- The book *Music theory for dummies* written by Michael Pilhofer and Holly Day [16].
- The book *Music composition for dummies* written by Scott Jarrett and Holly Day [9].
- The solfeggio classes we have had to follow as a complement to our instrumental formation.
- Our musical experiences and previously learned lessons by composing on computer or playing around with our instruments.

2.2.1 Music terminology

Let's begin by providing some essential music terminology that is used throughout the rest of this thesis.

Beat : Basic unit of time. One of a series of repeating consistent pulsations. Following the beat allow to interpret appropriately the intended pace of the song.

Tempo : Rate or speed of the beat of a musical piece generally expressed as beats per minute (bpm). This unit of measurement is rather self-explanatory, where 60bpm would mean that a beat lasts one second.

Rhythm : Music's regular or irregular pattern in time. Indispensable element of music since rhythm can exist without melody while the inverse is false.

Pitch : Frequency of vibration of a sound. There exist two predominant notations. The French notation that represents pitches by Do-Re-Mi-Fa-Sol-La-Si. And the English notation that uses the first alphabetical letters as A-B-C-D-E-F-G. The relation between both notations is presented in figure 2.2.

Interval : Difference between two musical pitches.

Semitone : Also known as **half-step**. In Western Music, it is the smallest interval between two pitches. On a guitar, you can play two pitches one semitone apart by playing one string and pressing from one fret to the next one . On a piano, if you play a key and then play the key at the right (or left), either a black

French notation	Do	Re	Mi	Fa	Sol	La	Si	Do
English notation	C	D	E	F	G	A	B	C

Figure 2.2: English and French notation correspondence

or white key, then you're playing a semitone higher (or lower).

Tone : Also known as **whole-step**. It corresponds to an interval of two semitones.

MIDI : Is the acronym for Musical Instrument Digital Interface. It is a communication protocol between virtual instruments, controllers and software. In MIDI, the middle C or C4 is represented by the value 60, C4# by 61, D4 by 62 and so on. To increase a note by a semitone, you simply add 1 to its value.

Midicent : Pitch unit measure allowing micro-tonal representation where the MIDI pitch value is multiplied by one hundred.

Note : Musical notation used to represent the duration and the pitch of a sound.

Note value : Relative duration of a note defined by the **note-head** and whether it has a **stem** or **flags/beams**. Figure 2.3 shows an eight note characterized to have a full note-head, a stem and one flag.

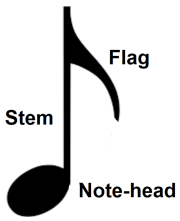


Figure 2.3: Eighth note

Staff : Five separated parallel lines with four spaces in between upon which notes drawn.

Score : Musical notation containing staves that can represent a single part for a solo work or all the parts for an ensemble.

Clef : Symbol found at the beginning of the staff that indicates the pitches of the notes situated in or in between the staff's lines. There are two predominant clefs, the treble clef for pitches higher than the middle C and the bass clef for

pitches lower than the middle C.

Rest : Symbol used to represent a time interval of silence, where no note is being played.

Melody : Also called tune, voice, or line, correspond to a succession of musical notes and rests.

Octave : Correspond to an interval of twelve semitones. From a C to the next C there is a separation of 12 semitones or an octave.

Chord : Two or more notes played simultaneously.

Harmony : Notes played simultaneously forming chords and chord progression that usually accompanies a melody.

Scale : Series of notes in ascending or descending order that presents the pitches of a key, beginning and ending at the tonic's key [16].

Measure : Also called **bar**. In a score, it corresponds to the segment of music delimited by two bars.

Time signature : Fraction placed at the beginning of the staff determining the duration of each measure. The numerator indicates the quantity, and the denominator indicates the note value of the beats of the measure. For example, the most used time signature 4/4 means that in a measure there are 4 quarter notes as beats.

Key note : Principal and lowest note of the scale in which a piece of music is set.

Mode : Series of notes into which the octave is divided. The difference between a scale and a mode is that in the scale the notes are ordered, while the mode can be seen as the set of possible notes.

Tonality : Organization of a musical piece based on a tonic note (or keynote) and a mode.

Ties : It connects equally pitched notes so to create one sustained note instead of two notes separated.

Dotted note : A dotted note is increased by one half of its original duration.

Alterations : Symbols that change the pitch by one semitone. The **flat** (♭) decreases the pitch by one semitone. The **sharp** (♯) increases the pitch by one semitone. And, the **natural** (♮) cancels previous alterations.

2.2.2 Rhythm

Rhythm can be described by using notes and rests inside measures that have a time signature and a tempo. It is hence important to understand how to represent the relative duration of the notes. Figure 2.4 shows the relationship between note values. As we can observe, one whole note lasts the same as two half notes. One half note lasts the same as two quarter notes and so on. Figure 2.5 demonstrates how the relative duration of rests can be represented. The composers can play with note values depending on the feeling they want to convey. Many consecutive notes with a short duration will probably be associated to an animated or frenetic sensation. While, few long duration notes will likely correspond to a more peaceful feeling.

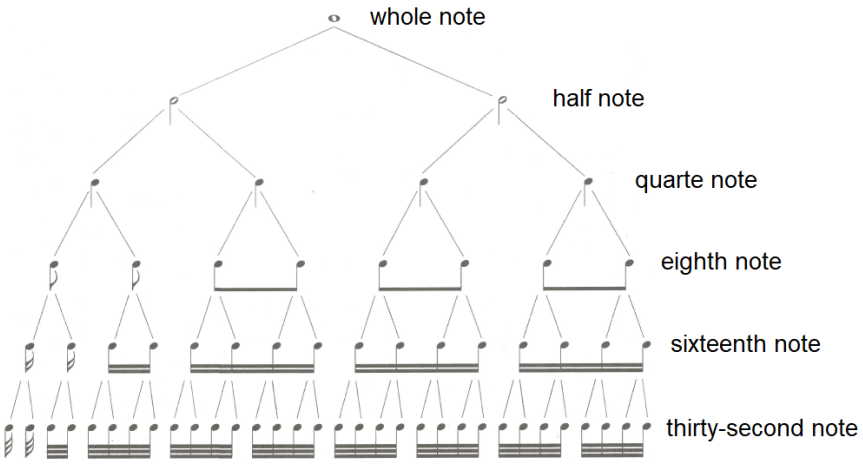


Figure 2.4: Relationship between note values [4]

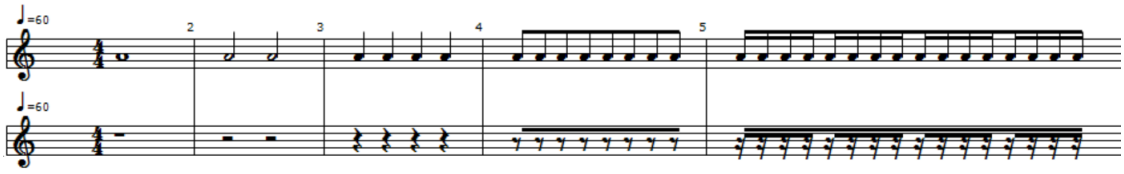


Figure 2.5: Rest values' respective note value duration

2.2.3 Melody

If you are lacking inspiration when composing, a good way to start would be to write your melody out of a given scale that can be shaped by suppressing notes, adding passing notes or repeating notes. It is thus important to know the main scales used in music. These can be expressed as a list of semitones. For example, the C major scale in figure 2.6 can be represented by the list (2 2 1 2 2 2 1). The nth element of the list represents the pitch interval, expressed in

semitones, between the nth note and the next note of the scale.

Apart from the major scales, the minor scales are also very important in music. Figure 2.7 shows a B natural minor scale (2 1 2 2 1 2 2). Figure 2.8 shows a B harmonic minor scale (2 1 2 2 1 3 1) where the seventh note of the natural minor is sharpened. As a rule of thumbs, minor scales express a sad feeling while major scales express happiness.

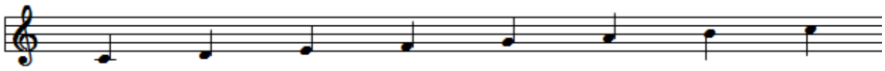


Figure 2.6: C Major scale

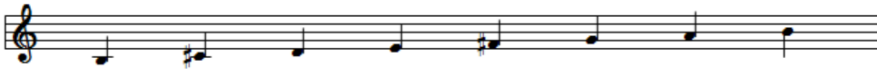


Figure 2.7: B natural minor scale

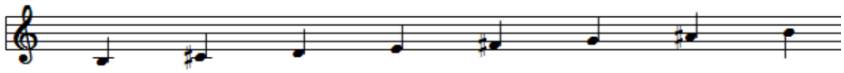


Figure 2.8: B harmonic minor scale

In the major and minor scales the first note is often referred as the **tonic** note, the fifth note as **dominant** and the fourth as **sub-dominant**. These are considered to be the most important notes, it is thus likely that they appear more frequently in the melody.

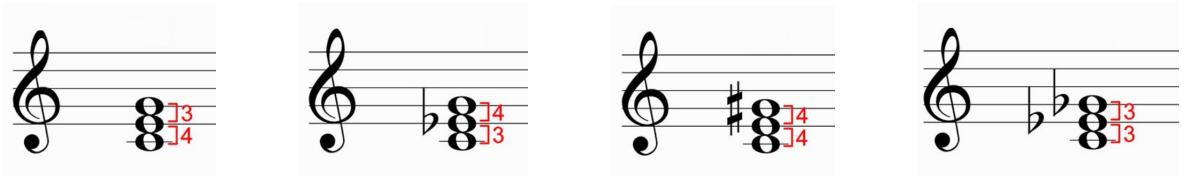
2.2.4 Harmony

Chord progressions can be seen as the basis of harmonies. This is why we will introduce the different chords and then analyze how to combine them in order to form chord progressions.

Chords

A chord is defined by a root note and a quality that determines the intervals between the notes. The principal chords are :

- **Major chords** are composed by the root note, the major third (4 semitones above the root) and the perfect fifth (7 semitones above the root) (figure 2.9a).
- **Minor chords** are composed by the root note, the minor third (3 semitones above the root) and the perfect fifth (7 semitones above the root) (figure 2.9b).
- **Augmented chords** are composed by the root note, the minor third (4 semitones above the root) and the augmented fourth (8 semitones above the root) (figure 2.9c).
- **Diminished chords** are composed by the root note, the major third (3 semitones above the root) and the augmented fifth (6 semitones above the root) (figure 2.9d).



(a) C major chord (b) C minor chord (c) C augmented chord (d) C diminished chord

Figure 2.9: Different types of chords with C as a root note [29]

Inversion : Chords can be found in root position, first inversion or second inversion. Consider the C major example: the root position is the one presented in figure 2.10a. In the first inversion, the major third will be the lowest note of the chord as presented in figure 2.10b. While, in the second inversion, the perfect fifth will be the lowest note of the chord as presented in figure 2.10c.

(a) Root position (b) First inversion (c) Second inversion

Figure 2.10: C major chord inversions

Diatonic chords : The diatonic term means that the notes of the chords belong to a given key. For instance, the diatonic chords in C Major are only composed by notes belonging to the C Major scale. Figure 2.11 presents the diatonic chords in C Major. As we can see, there are as many diatonic chords as different notes. Each note of the scale is the root note of a diatonic chord and the

other two notes correspond to the second and fourth note of the scale that comes after the root note. For example, the first diatonic chord is composed by the first, third and fifth notes of the scales. The chords are represented by a roman number that indicates which note of the scale is the root note. If the number is in capital letters, it means that it is a major chord. If it is in lower case, it represents a minor chord. And, if it has a ^{o} as superscript, it represents a diminished chord.

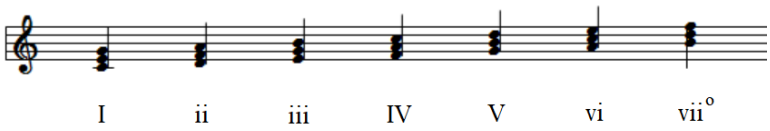


Figure 2.11: Diatonic chords in C Major

Chord progression

In order to create pleasant chord progressions, there exists some widely used tools. For instance, you can create a melodious chord progression with the diagram showed in figure 2.12. You can perhaps start from the I chord and try to find your way back by using any available route. For example, the I-iii-IV-ii-I can be an acceptable chord progression. This diagram is by no means a rule that must be followed always. However, it assures you that your chord progression will sound melodious and natural to the listener [9].

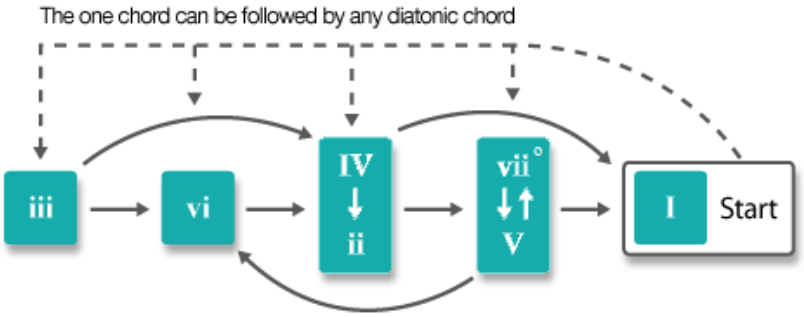


Figure 2.12: Chord motion in a Major key [15]

2.2.5 Musical Forms to structure your piece

Composers often arrange their pieces by using different musical parts usually represented by capital letters such as A, B, C, D and so on. This is a common practice in almost every genre, going from Classical music, to Rock and Pop. Here are some of the most common musical forms [9]:

- **One-part form A :** Most primitive song structure, it presents slight to no changes in each successive verse.
- **Binary form AB :** It consists of two contrasting parts. This form can also be expanded to AABB.
- **Ternary form ABA :** In here, B represents the bridge between the two A parts. It can be expanded to AABBA or AABA.
- **Arch form ABCBA :** The song starts with a part A then moves to a part B, then C, then comes back to B to finish with the beginning part A.
- **Rondo ABACADA :** The song revolves around part A and the parts B, C and D are tying the parts A of the song.

With the musical forms, we can observe the importance of having Blocks with different constraints that can be juxtaposed. Chapter 7 provide musical scenarios where different or same Blocks are juxtaposed.

2.3 Music Composition

It is true that there are no unbreakable rules in music. However, the probability of creating a pleasant piece of music without following some kind of guidelines is really low. Throughout history, humans have discovered and learned concepts, patterns, sets of notes, chord progressions that “work well" and that are free for musicians to pick without having to discover it all over again. We do not pretend to give a complete music composition support course. Nevertheless, we are going to list some of those tips that create a good melody.

2.3.1 Tip 1 : Let the listener rest

Imagine reading a story. If there is nothing happening, you will most probably become bored and stop reading. Meanwhile, if the story is too packed with action and does not let you rest, then you will probably lose attention as well. As a matter of fact, writing music is similar to writing a story. In order to keep the listener interested, it is a good idea to alternate between action and rest in your melody. So instead of having evenly distributed notes, try to have different densities throughout it.

2.3.2 Tip 2 : Tension and release

Similarly to action and rest, a good story should not provoke the same emotion all along the narration. Building up tension has more sense if it is followed by some kind of release and vice versa. This tip can apply to rhythm as well as melody and harmony or even in the arrangement of the song.

2.3.3 Tip 3 : Coming back home

A good way to imagine a melody is to see it as an adventure were the further you get from the fundamental note, the further from home you are. You will eventually come back home to the fundamental and be ready for a new adventure. The distance from home and the duration of the adventure provokes different kinds of feelings to the listener.

2.3.4 Tip 4 : Popular chord progressions

If you are lacking inspiration and can't come up with anything, you can begin by considering some of the popular chord progressions that are used in the musical genre you are willing to compose. To mention some of them, we have the I-IV-V chord used to write many hits and the 12-bar blues, for example. Once you have found the popular chord progression that pleases you, you can change it in your style and start developing the melody that interlaces the best with the harmony.

2.3.5 Tip 5 : Repetition with a twist

Once you have composed a catchy musical phrase, how should you continue your song ? One widely used technique is to replay the phrase but giving it an interesting unexpected twist to captivate the audience. This makes the phrase stick more easily into the listener head and with the introduced twist you don't make it sound repetitive. It is also an intelligent way to transition smoothly into a new musical part.

2.3.6 Tip 6 : Arpeggio

It's not always sufficient to have a melody accompanied by a chord progression. In some cases, you can create much more interest by having more than one musical idea moving in a melodic way. It is therefore not a bad idea to consider instead of striking the notes of the chords simultaneously playing them in an arpeggio fashion. This not only allows for the harmonic part to "dialog" with the melodic part, but it also adds a second layer to play with rhythm.

2.4 Constraint Programming

As we have already mentioned, music can be represented or translated into a mathematical language. You can picture this by considering the time signature presented as a fraction, the tempo expressed in beats per minute, or the pitch represented by its frequency or by its MIDI value.

Likewise, musical composition can be expressed as a Constraint Satisfaction Problem. For example, where the composer constraints the notes to follow a given tonality and the rhythm to a certain genre specific rhythm. The constraints

can also depend on the mood that the composer wants to convey. For instance; if they wants to transmit happiness, they will probably constraint the notes to be short, so that the pace of the piece is fast, and to follow a major mode, known to be the brighter version of its minor counterpart. While, if the composer wants to express sadness, they will probably constraint the notes to be long and to follow a minor mode.

It is important to understand that we do not want to replace musician's creativity with Constraint Programming. On the contrary, it should be used as an aid to enhance its inspiration.

2.4.1 Definitions

Constraint programming (CP) is a programming paradigm that aims at solving combinatorial problems by narrowing down the domains of the variables that specify it using mathematical constraints. Constraint programming is one of the closest paradigms to what would be considered as the "holy grail of programming" where the only task for the programmer is to state the problem and the computer will find a solution. Does this mean that we only have to focus on the modeling part and blindly trust Gecode to find a solution? Technically we could. But if the problem is large, we won't know if the performance of the solver is poor because the stated model has too many variables with complex constraint or simply because we have chosen an inefficient **Branching heuristic** or an unsuitable **Search engine** (these terms are explained in detail later). This is why we can resume the Constraint programming mantra into the following equation :

$$\text{Constraint programming} = \text{Model} + (\text{Search})$$

Although the modeling part is considered to be the most important; the search part that appears in parenthesis implies that it isn't mandatory to know specific algorithmic details about how the search tree is computed, how the constraints are propagated, and how the search engine explores the tree. However, having a solid foundation understanding how these three aspects work and how they interact is crucial especially when dealing with complex models with many constraints.

A **Constraint Satisfaction Problem (CSP)** is an application of Constraint Programming for solving problems arising in artificial intelligence tasks. A CSP $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ is a tuple of a set $\mathcal{X} = \{i, j, ...\}$ of n variables, $\mathcal{D} = \{D_i, D_j, ...\}$ a set of n domains for the variables and a set of constraints $\mathcal{C}$ imposing logical, arithmetic or combinatorial relation on one or more variables of $\mathcal{X}$. A solution for P is a set of values $\{I_j\}_{j \in \mathcal{X}}$ s.t. $\forall j \in \mathcal{X}, I_j \in D_j$ that satisfies all the constraints in $\mathcal{C}$ [11].

Constraint Optimisation Problem (COP) is a CSP where the quality of a solution is estimated by an objective function that the algorithm tries to maximize or minimize.

The **Search** is the organized review of combinations of values for the variables. The **Search Space** is the set of all possible combinations or values for all variables. A search is said to be complete when it reviews the entire search space.

Backtracking Search is a very popular complete search algorithm for constraint programming. It organizes the search space as a tree that it runs through with Depth First Search. Every node of the tree represents a subset of the initial $\mathcal{D} = \{D_i, D_j, ...\}$. Every branch of the tree represents a reduction of domain for one or more variables. The subset of domains of any node is the updated set of domain of its parent node with regard to the change in domain that the branch imposes. The domains are updated so that all the constraints remain true. The update of the domains due to a change in some other variable's domain is called **propagation**. If after the propagation all the domains are empty, then the algorithm must backtrack to the parent node and choose another branch [5] [7].

A common way to organize the tree is as a binary tree where the first branch of a node sets a variable to a certain value in its domain and the second branch removes it from the domain. Surely, there are other strategies such as removing half of the domain in the first branch and the other half in the second.

2.4.2 Example 1 : Solving a sudoku

The task of solving a sudoku is an excellent example for understanding the concepts of constraints, backtracking and propagation. It happens that the way most humans solve a sudoku is really similar to how a computer does it using constraint programming. The rules of this very popular American game (and not Japanese as its name might suggest !) are very simple and probably well-known by any reader of this text. Still, a quick reminder makes sure that everyone is on the same page.

The game takes place in a square divided in 9 boxes of equal dimensions, each of which is also divided in 9 equal squares. There are only 4 rules in this game :

Rule 1. Every square has to be given a number between 1 and 9.
Rule 2. Every row must have distinct numbers.
Rule 3. Every column must have distinct numbers.
Rule 4. Every 3x3 box must have distinct numbers.

The game starts with some of the squares fixed to some values that makes it more or less difficult for the player to fill in the rest of the squares while respecting the rules.

As explained earlier, a CSP is defined by a tuple of three sets of variable, respective domains and constraints. For this problem, we can use a matrix of variables $x_{i,j}$ with i as row index and j as column index (see Figure 2.13). The

$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	$x_{1,4}$	$x_{1,5}$	$x_{1,6}$	$x_{1,7}$	$x_{1,8}$	$x_{1,9}$
$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	$x_{2,4}$	$x_{2,5}$	$x_{2,6}$	$x_{2,7}$	$x_{2,8}$	$x_{2,9}$
$x_{3,1}$	$x_{3,2}$	$x_{3,3}$	$x_{3,4}$	$x_{3,5}$	$x_{3,6}$	$x_{3,7}$	$x_{3,8}$	$x_{3,9}$
$x_{4,1}$	$x_{4,2}$	$x_{4,3}$	$x_{4,4}$	$x_{4,5}$	$x_{4,6}$	$x_{4,7}$	$x_{4,8}$	$x_{4,9}$
$x_{5,1}$	$x_{5,2}$	$x_{5,3}$	$x_{5,4}$	$x_{5,5}$	$x_{5,6}$	$x_{5,7}$	$x_{5,8}$	$x_{5,9}$
$x_{6,1}$	$x_{6,2}$	$x_{6,3}$	$x_{6,4}$	$x_{6,5}$	$x_{6,6}$	$x_{6,7}$	$x_{6,8}$	$x_{6,9}$
$x_{7,1}$	$x_{7,2}$	$x_{7,3}$	$x_{7,4}$	$x_{7,5}$	$x_{7,6}$	$x_{7,7}$	$x_{7,8}$	$x_{7,9}$
$x_{8,1}$	$x_{8,2}$	$x_{8,3}$	$x_{8,4}$	$x_{8,5}$	$x_{8,6}$	$x_{8,7}$	$x_{8,8}$	$x_{8,9}$
$x_{9,1}$	$x_{9,2}$	$x_{9,3}$	$x_{9,4}$	$x_{9,5}$	$x_{9,6}$	$x_{9,7}$	$x_{9,8}$	$x_{9,9}$

Figure 2.13: Variable distribution for sudoku puzzle

initial domain of these variables would be integers between 1 and 9 for every variable. The constraints would be distinct rows, columns and boxes. Equations (2.1), (2.2), (2.3) and (2.4) show the mathematical definition of the sudoku CSP [20].

$$P = (\mathcal{X}, \mathcal{D}, \mathcal{C}) \tag{2.1}$$
$$\mathcal{X} = \{x_{i,j} | i, j \in \mathbb{N} \wedge 1 \leq i, j \leq 9\} \tag{2.2}$$
$$\mathcal{D} = \{D_{i,j} = \{1, 2, ..., 9\} | i, j \in \mathbb{N} \wedge 1 \leq i, j \leq 9\} \tag{2.3}$$
$$\mathcal{C} = \{\forall i : distinct(row_i), \forall j : distinct(column_j), \forall box : distinct(box)\} \tag{2.4}$$

Figure 2.14a shows an instance of a sudoku puzzle. Some of the variables have been fixed to their initial value. The others still have their domain untouched. The first intuition one might have when implementing the *distinct* constraint for sudoku is to simply remove from the other variables' domain the value of all fixed variables from the row, column or box. For instance, variable $x_{1,1}$ in figure 2.14a would have a domain of {1,2,4,6,7,8}. Propagating this constraint on every variable would reveal that $x_{8,5}$ is a 7 as it is the only value remaining in its domain (figure 2.14b).

At this point, all variables have more than one value left in their domain. The computer has to branch on the value of a variable. Let $x_{7,4}$ be the selected variable for branching. Its domain is currently {3, 5, 6}. Let now 5 be the chosen value for branching (figure 2.14c. Propagating this change in domain leaves {6} as domain for both $x_{9,5}$ and $x_{9,6}$, which violates the *distinct* constraint. The computer has to backtrack and remove 5 from the domain of $x_{7,4}$ which becomes {3,6}. Let 3 be the chosen value for branching. Then, 3 is removed from $x_{9,5}$ and $x_{9,6}$ domain and the program can keep running.

If, in addition to the previous intuition on the *distinct* constraint implementation, we fix a variable to a certain value if this value is absent from all other variables' domain from the row, column or box, then the very first propagation

		5		3				9
					8		6	
			7					
5				2		9	1	
3			9	1		6		
	9				3	2		
		1		4	9			
9		6	8		1			5
	7		2					1

(a) Initial sudoku puzzle

		5		3				9
					8		6	
			7					
5				2		9	1	
3			9	1		6		
	9				3	2		
		1		4	9			
9		6	8	**7**	1			5
	7		2					1

(b) Weak constraint implementation (1) : Propagation

		5		3				9
					8		6	
			7					
5				2		9	1	
3			9	1		6		
	9				3	2		
		1	**5**	4	9			
9		6	8	7	1			5
	7		2					1

(c) Weak constraint implementation (2) : Branch

		5		3				9
					8		6	
			7					
5				2		9	1	
3			9	1		6		
	9				3	2		
		1	**3**	4	9			
9		6	8	7	1			5
	7		2					1

(d) Weak constraint implementation (3) : Backtrack and branch

Figure 2.14: Sudoku puzzle CSP - Weak constraint propagation

		5		3				9
					8		6	
			7					
5				2		9	1	**3**
3			9	1		6		
	9			**8**	3	2		
	5	1	**3**	4	9			
9		6	8	**7**	1			5
	7		2				**9**	1

Figure 2.15: Sudoku puzzle CSP - Strong constraint implementation

gives what you can see on figure 2.15. With this implementation, we avoid wrong guesses on $x_{7,4}$, but also on many other variables.

This example is a good illustration of how significant of an impact can the implementation of a constraint have on the execution of the algorithm. While being complete and sound, the first implementation is weaker than its improved version because of its lack of foresight into what choices of values are doomed to fail. On the other hand, the second implementation is more computationally expensive. This is a very typical situation in constraint programming where there is a trade off to be done between constraint strength and computational price the propagation is executed.

2.4.3 Constraint Propagation

We already had a quick insight in constraint propagation with the sudoku example and we concluded that it wasn't trivial to choose between a weak constraints propagation and a strong one. A strong propagation prunes more values from the domain of the variables and it will probably lead to a smaller search tree. But, at each node of the tree, the propagation is more computationally expensive. While a weak propagation prunes less values from the domain of the variables and that will probably lead to a bigger search tree. Yet at each node of the tree the propagation is less computationally expensive [21]. In order to illustrate this better, we consider a single line of the sudoku example using the distinct constraint with the three propagation levels proposed by Gecode (which are the most known in CP) :

- **Value propagation** : In the distinct constraint, it naively waits a variable to be bound in order to prune it from the domain of the other variables.
- **Bound propagation** : Achieves bound consistency by mainly considering the minimal and maximal values of the variables domain during propagation [3].
- **Domain propagation** : Achieves domain consistency, therefore, it is a stronger propagation than the bound propagation. And that is because

when propagating it takes into account all the variables' domain's values and not only its minimum and maximum value.

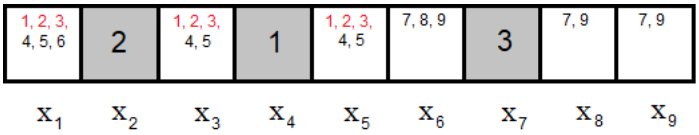


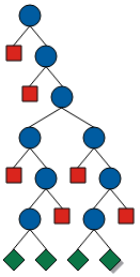
Figure 2.16: Sudoku line

We therefore know that domain propagation is stronger than bound propagation, which is stronger than value propagation as well. By stronger we mean that the constraint propagation prunes more values from the domain of the variables. If we consider the sudoku line at figure 2.16 where variables x_2, x_4 and x_7 are bound, we can observe that the values in red will be pruned by all three propagators. If we consider the values in black, a decent sudoku player would notice that x_3 and x_5 can only take values 4 and 5 which means that we can prune these values from x_1 and that x_1 will be bound to 6. Similarly, since x_8 and x_9 can only take values 7 and 9, we can prune these values from x_1 and that x_6 will be bound to 8. Thus, without further information, this sudoku line could have four possible different solutions. Testing these three types of propagation led to the following expected results: Firstly, all three propagations pruned the red values 1, 2 and 3, and the naive value propagation could only pruned these three values. Secondly, the domain propagation, which is the strongest, functioned as a decent sudoku player that could prune 4 and 5 from x_1 and 7 and 9 from x_6. Thirdly, bound propagation, since it works mainly with the minimum and maximum values of the domains, it could prune 4 and 5 from x_1 but couldn't prune 7 and 9 from x_6.

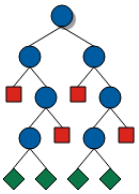
We can observe in figure 2.17 that the stronger the propagation applied is, the smaller the search tree will be. We can also notice that the propagation does not modify the stated model and thus the same four solutions are found. It could consequently seem that the propagation should be chosen in a case by case basis but there are actually some level of propagation that work better with certain type of constraints. However, we won't go into details since this topic would be outside of the scope of this master thesis.

2.4.4 Branching heuristics

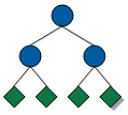
The branching is a two step decision that defines the shape and size of the search tree. In this two step decision, first we have to choose which variable we are going to branch on and secondly to which values we are going to bound the variable at each branch [14]. We can observe in figure 2.18 how some different branching strategies influences the ramification at a given node and thus forge the entire tree. We mainly focus on binary trees since they benefit more from propagation than the n-ary tree as in figure 2.18c [22].



(a) Search tree with value propagation



(b) Search tree with bound propagation



(c) Search tree with domain propagation

Figure 2.17: Search tree applying three different propagation levels

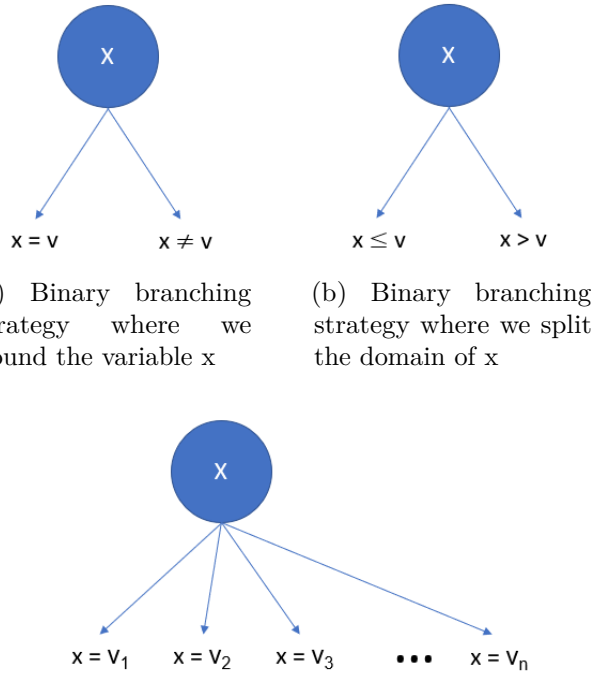
It is important to make good branching decisions considering that it may have a strong impact on the size of the decision tree. This said, there are two logical branching heuristics for the variable selection and for the value selection. These are widely used in Constraint programming and it helped us choose a good branching strategy that reduces the size of the search tree and "provides" a quickly solution to the search engine. These two heuristics are :

- **First-fail** for variable selection : If there are no solutions under a node (failure), we prefer to discover this quickly, not to waste too much time exploring the subtree under the node [7] [11].
- **First-success** for value and partition selection : Once a variable x is selected, if there is a solution under the node, we want to find it as soon as possible. Therefore, we would want to first inspect the most promising value v of the domain of x by bounding x to v into the left branch of the node [11].

Gecode provides many implemented variable selection strategies that the user only has to introduce, equivalent to a parameter when choosing the search engine. Some of them follow the first-fail principle such as :

- choosing the variable that has the smallest domain size.
- choosing the variable that has the most propagators (approximated measure of how much the variable is constrained).
- choosing the variable with the most accumulated failure count.

- choosing the variable with the higher propagators to domain size ratio.



(a) Binary branching strategy where we bound the variable x

(b) Binary branching strategy where we split the domain of x

(c) N-ary branching strategy where we bound x to each value of its domain

Figure 2.18: Search tree ramification with three different branching strategy

We won't revisit all the variable selection strategies into detail since it would be outside of the scope of this thesis. However, it is important **not** to choose a variable selection proposed by Gecode that goes against the first-fail principle, they might be there for instructional purpose rather than for actual application utility.

As far as the value selection is concerned, choosing a strategy that follows the first-success principle is a more subtle task that requires more specific knowledge about the problem. As an illustration, if we desire to have a decreasing pitch melody, where each note is a variable and the variable selection branching strategy naively branches the notes in the staff from left to right; then, the most promising value to bound the variables in the left branch is to choose the maximum value of its domain. Nevertheless, if there isn't a clear value selection strategy that follows the first-success principle and we also want the solver to surprise us by generating original melodies, a simple random value selection strategy would work well.

2.4.5 Exploration and search engines

After having analyzed the different types of propagators: how they prune values from the variable's domain at each node of the search tree, and the different

branching that defines the shape of the tree, we now tackle the strategy adopted to explore the tree. We thus present in this section the three search engines proposed by Gecode: Depth-First Search, Limited Discrepancy Search, and Branch and Bound.There is also the Large Neighborhood Search (LNS) that inspired us on how to use Branch and Bound intending to solve the problem of not having solutions that resemble too much and that only change by one note, as an example. At each time, we not only discuss strictly about constraint programming aspects but we also observe the possible advantages and disadvantages of these explorations with the eyes of a composer. Furthermore, we examine parallel search and under which scenarios it can be useful or not.

Depth-First Search (DFS)

This is probably the most popular exploration strategy. Firstly it inspects, as the name indicates, the depth-first left-most leaf node of the tree, and then it visits all the leaf nodes from left to right as shown in figure 2.19.

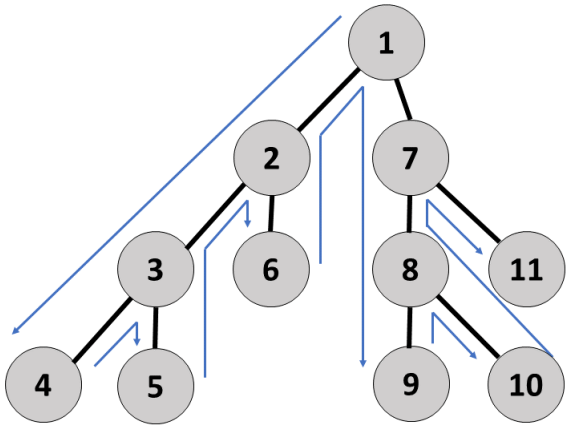


Figure 2.19: Depth-First Search of a tree from left-most leaf to right-most leaf

The first issue we have with this search engine is that it can only be used for a Constraint Satisfaction problem and thus we cannot have an objective function to maximize or minimize. The second main problem we get as a composer's viewpoint, is that if the solver finds a solution, then the next solution the solver provides to the composer will probably be very similar. For example, if each note is a variable, it could indeed differ by only one note from one solution to another.

Moreover, if the branching strategy is not optimal and hence the first left branch does not lead to any solution, then the search engine wastes a lot of time exploring the left-hand side subtree before exploring the subtree from the right.

One interesting tool that we can easily exploit, thanks to Gecode, is the use of multiple threads for this search engine. This does not only improve search performance but it also provides more diversity from one solution to the next one. In fact, the first solution found could eventually not be the left-most one. Note that parallel search is beneficial in large search trees; in smaller ones it won't make much difference compared to the classic DFS using only one thread.

Limited Discrepancy Search (LDS)

This exploration strategy proceed as follows:

- in its first iteration, the search engine starts as in DFS with the left-most node,
- in its second iteration, it visits all the leaf nodes where we arrive by taking at most one right branch and all the other are left branches,
- in its third iteration, it visits all the leaf nodes where we arrive by taking at most two right branch and all the other are left branches

and so on [8].

If we consider a perfectly balanced depth three the LDS engine will visit the nodes as shown in figure 2.20.

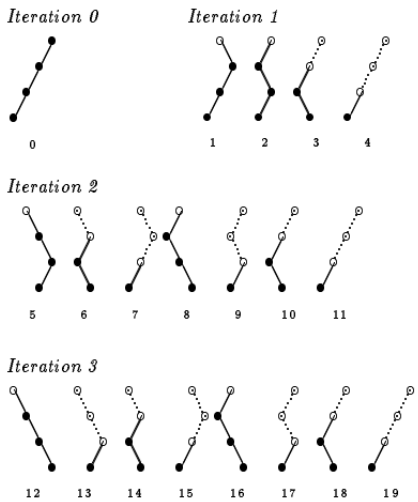


Figure 2.20: Iterations of the Limited Discrepancy Search [8]

As with DFS, in Gecode, this search engine can only be used for a Constraint Satisfaction problem and thus we cannot have an objective function to maximize or minimize. As seen in figure 2.20, the exploration is much less sequential than the classic DFS that explores each leaf node from left to right. This has two main benefits:

- The first one is that we have an improvement in terms of diversity in solutions provided by the solver compared to DFS, as we can observe in figure 2.20.
- Another advantage of LDS compared to DFS is, that if the first left branch doesn't lead to any solutions the engine won't explore all the left subtree before exploring the right one since we saw that the exploration was more dispersed throughout the tree and not as a sequential.

However, the major drawback of the Limited Discrepancy Search is that Gecode doesn't provide the use of multiple threads with this search engine.

Large Neighborhood Search (LNS)

We wont get into much details for this exploration strategy since we are introducing it because it inspired us on how to use Branch-and-Bound to solve the problem of not having similar solutions. Nevertheless, the algorithm of LNS proceeds roughly as follows [17] [12]:

1. It finds a first solution S.
2. Randomly relaxes S and searches for a better solution with a search limit. Relaxing S means to fix some variables to their values in S and then proceed to find a better solution with the non-fixed variables.
3. Replaces S by the best solution found and repeats step 2.

In the scope of our thesis we prefer to use Branch-and-Bound (BAB) rather that Large Neighborhood Search (LNS) because it is much more straightforward to use in Gecode (LNS is considered as meta-search engine in Gecode) and also because BAB can benefit from the use of multiple threads in Gecode.

2.4.6 Branch-and-Bound (BAB)

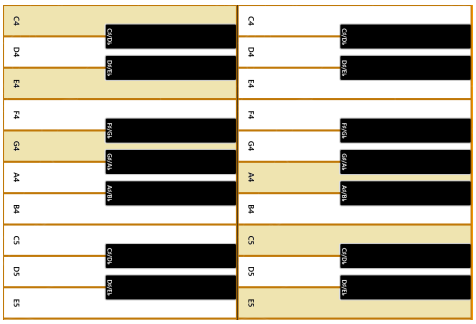
Branch and Bound should normally belong to the previous subsection just like DFS, LDS and LNS. However, due to its special importance detailed hereinafter, we decided to dedicate an entire subsection for this exploration strategy.

This exploration strategy works similarly as the depth-first search algorithm (from depth-first left-most leaf to the right-most leaf) with the difference that it can maximize (or minimize) an objective function. In other words, branch-and-bound can solve constraint optimization problems while the depth-first search and limited discrepancy search can only solve constraint satisfaction problems.

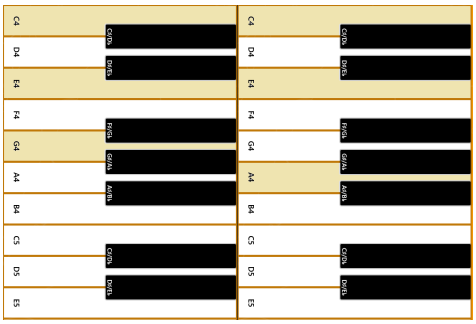
To be more specific, the exploration works as the depth first search (same path order) except that at each time the solver finds a solution, it adds a new constraint to impose to have an objective function to the next solution found. It will be smaller in the case of a minimization problem or larger in the case of a

maximization problem. [2]. Therefore, the last solution found is the best solution with a maximum/minimum objective function. Since at each time a solution is found, a constraint is added and thus more values from the variables domain will be pruned. Then, the tree explored by BAB is smaller than the tree explored by DFS of the analogous problem without an objective function.

This exploration strategy can expand to a broader extent the possible modeling scenarios. For instance, if there are two different instruments that play at different scales (e.g., two friends where one has a harmonica in the scale of C and the other one has a harmonica in the scale of A), therefore, the compositor would probably like to minimize the dissonance of the two melodies played simultaneously. Furthermore, another case scenario where the compositor would need a COP instead of a CSP is if the compositor has a chord progression played in its root position but it sounds too disjointed. Then, they can opt to inverse its chords such that the transition is smoother. To do so, us, as compositors, could indeed minimize the span (maximum pitch minus minimum pith) of the chord progression. If you have a keyboard by your side (or a MIDI piano keyboard software [27] as we used for figure 2.21), you can hear (and also observe) the difference of playing a C Major followed by a A Minor chord both played in root position as in figure 2.21a rather than its much smoother C Major in root position followed by a second inverted A Minor chord as in figure 2.21b.



(a) C Major - A Minor played in root position



(b) C Major in root position - second inverted A Minor

Figure 2.21: Making a smoother transition by inverting chords

Moreover, BAB can handle two issues that are more CSP related that DFS and LDS cannot manage:
The first one is related to the diversity of the solutions provided. Although LDS gives better solution diversity than DFS, it is still not optimal since we can't really control the diversity that the solver provides from one solution to another. Thanks to the flexibility that Gecode provides with BAB, we can use this search engine in more manners than a simple minimization/maximization of an objective function.

This said, when a solution is found, we can add as many constraints as we intend, so we can ask for the solver to have at least a $X\%$ of diversity from one solution to another. We go more into details about the model we use for Melodizer 2.0 in Chapter 5. But it can be useful to have a first glance in the difference in terms of solution diversification that we can obtain with DFS in figure 2.22 compared to BAB in figure 2.23. Moreover, this flexibility allows us also to minimize two objective functions. For example, if we want the least amount of possible dissonance and the least amount of span in the harmonic part of the piece.

The second issue that BAB can handle is that if we have a CSP, where the solution space is empty, we could indeed relax the problem so that the solution space contains some solution vectors. We can do this with BAB by minimizing the number of constraints that are not satisfied (we tackle this topic more in details in Chapter 3 with the reified constraints).

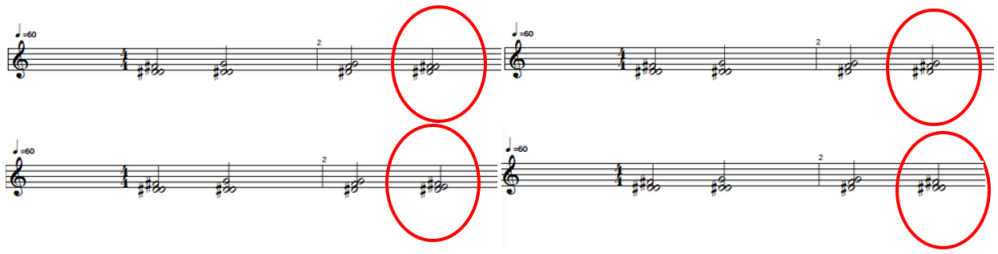


Figure 2.22: First four solutions with DFS where only the circled chords have changed

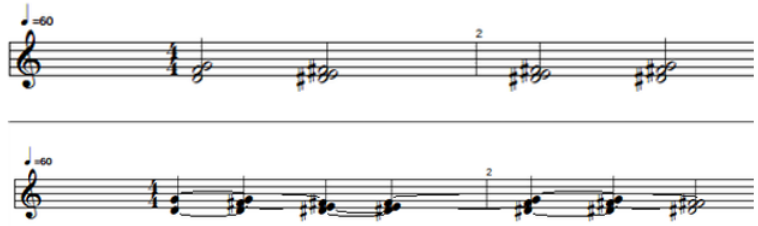


Figure 2.23: First two solutions imposing diversity with BAB

Chapter 3

Tools

In this chapter we explain the main features and the usage of the two most important tools for the realization of this project, namely Gecode and OpenMusic. Gecode is an open source C++ toolkit used to develop constraint-based systems and applications [1] and OpenMusic is a visual programming environment designed to help composers in their work [2]. If you are more interested in Gecode, section 3.1 is for you, if you prefer to read about OpenMusic, you can jump to section 3.2.

3.1 Gecode

Gecode [1] is an open, free, accessible, and efficient environment responsible for developing constraint-based systems and applications. It is implemented in C++ and offers a great deal of powerful and efficiently implemented constraints [23], branching strategies and search engines. It also allows the programmer to craft its custom constraints, branching strategies and search engines, which can be really useful for some uncommon tasks.

Since integer sets are the main foundation for Melodizer's 2.0 model (posteriorly explained in section 5.2), this chapter revolves around set variables, constraints and branching.

3.1.1 Search Space

In Gecode, CSPs are typically modelled by creating a class extending the `Space` class. The `Space` class serves as a *home* for *variables*, *propagators*(constraints' implementations), *branchers* and a facultative *objective function* to determine a best solution during the search [23]. Once our model is created, we ought to create a search engine for that model that is responsible for finding the possible solutions.

[1] https://www.gecode.org/

[2] https://openmusic-project.github.io/

3.1.2 Variables

Gecode provides four types of variables : `integer`, `boolean`, `float` and `set`. The following code lines shows how to declare each of these.

```
IntVar i(home, -4, 20); // creates an integer variable i
   and sets its domain to {-4,..., 20}
BoolVar b(home, 0, 1); // creates a boolean variable b and
   sets its domain to {0, 1}
FloatVar f(home, -1.0, 1.0); // creates a float variable f
   and sets its domain to [-1.0 ... 1.0]
SetVar s(home, IntSet::empty, IntSet(1,3), 1, 2); //
   creates a set variable s and sets its domain to [{} ...
   {1,2,3}] and its cardinality domain to [1 ... 2].
   Therefore, considering the cardinality (set's size
   measure), the actual domain will be [{1} {2} {3} {1,2}
   {1,3} {2,3}]
```

In these examples, `home` is the variable pointing to the Space class containing the whole CSP.

Gecode also provides arrays structures for each type of variable. Thereby, an array of integer variables can be declared with the following code line.

```
IntVarArray x(home, 4, -10, 10); // creates an array of 4
   integer variables with domain {-10, ..., 10}
```

Similarly, arrays of sets can be declared with the following code line.

```
SetVarArray x(home, 10, IntSet::empty, IntSet(1,3), 1, 2);
   // creates an array of 10 integer set variables with
   domain [{} ... {1,2,3}] and its cardinality domain to [1
    ... 2].
```

Dynamic arguments arrays

Useful variable array type, that can grow dynamically by adding elements or whole arrays with the operator "<<" and two arrays can be concatenated by using the "+" operator. Practical for when we don't want to fix the size of the array when creating it as demonstrated in the following lines of code.

```
SetVarArgs x;
x << SetVar(home, IntSet::empty, IntSet(1,3), 1, 2);
x << SetVar(home, IntSet::empty, IntSet(0,5), 1, 2);
SetVarArgs y;
y << SetVar(home, IntSet::empty, IntSet(0,3), 0, 3);
y << x; // y thus has three integer set variables elements,
    the first one with domain[{}...{1,2,3}] and cardinality
    [1...2], the second one with domain
   [{}...{0,1,2,3,4,5}] and cardinality [1...2] and the
   third one  domain[{}...{1,2,3}] and cardinality [0...3]
```

3.1.3 Constraints

Gecode has many sorts of constraints for every type of variable. The following code lines give a few examples of typical constraints for integer variables.

```
rel(home, x, IRT_LE, y); // x < y
rel(home, x, IRT_NQ, 4); // x != 4
dom(home, x, 2, 12); // 2 <= x <= 12
```

There are also constraints designed for constraining arrays of integers. Here are a few examples.

```
distinct(home, x); // all values in x are different
nvalues(home, x, IRT_EQ, 4); // 4 different values in x
```

Some constraints also create a relation between variables of different types. For example, the following constrain z (integer variable) to be equal to the number of elements of x (integer variable array) that are equal to y (integer value).

```
count(home, x, y, IRT_EQ, z);
```

Note that the data type of the arguments passed defines the behavior of the constraint. If we consider the count() function previously introduced, but instead of passing an integer y as third variable we pass an array of integers c as follows.

```
count(home, x, c, IRT_EQ, z);// where x is an integer
   variable array, c is an array of integers of the same
   size of x and z is an integer variable
```

Then, z is constrained to be equal to how often $x_i = c_i$. Or, alternatively, in mathematical notation [21]:

$$z = \#\{i \in \{0, 1, ..., |x| - 1\} \mid x_i = c_i\}$$

Gecode also allows a user-friendlier manner to write constraints by including the MiniModel library header to you program. The first three shown constraints could be rewritten more comprehensibly as follows.

```
rel(home, x < y); // x < y
rel(home, x != 4); // x != 4
rel(home, 2 <= x <= 12); // 2 <= x <= 12
rel(home, (2 <= x) && (x <= 12)); // Another form for 2 <=
   x and x <= 12
```

Set constraints

Relation constraints are the mainly functions used to constraint set variables and set variable arrays, by using the classical set operators and relations presented in figure 3.1. Again, the arguments you pass to the function define its behavior. For example,

```
rel(home, x, SOT_INTER, y, SRT_EQ, z);
```

Identifier	Relation
SRT_EQ	Equality ($=$)
SRT_NQ	Disequality ($\neq$)
SRT_SUB	Subset ($\subseteq$)
SRT_SUP	Superset ($\supseteq$)
SRT_DISJ	Disjoint ($\|\|$)
SRT_CMPL	Complement
SRT_LQ	Less or Equal ($\leq$)
SRT_LE	Less ($<$)
SRT_GQ	Greater or equals ($\geq$)
SRT_GR	Greater ($>$)

Identifier	Operation
SOT_UNION	Union ($\cup$)
SOT_DUNION	Disjoint union
SOT_INTER	Intersection ($\cap$)
SOT_MINUS	Difference ($\setminus$)

Figure 3.1: Operation and relations types on SetVar [24]

where x, y and z are set variables, constrains $x \cap y = z$.

```
rel(home, x, SRT_SUP, y);
```

Where x and y are set variables, constrains $x \supseteq y$.

```
rel(home, SOT_UNION, x, y);
```

Where x is a set variable array and y is a set variable, constrains $x_0 \cup x_1 \cup ... \cup x_{|x|-1} = y$.

As mentioned before, by adding the Minimodel library header, these three constraints can be rewritten more comprehensibly as presented hereinafter.

```
rel(home, x & y == z);// x intersection y equals z
rel(home, x >= y);// x is a superset of y
rel(home, setunion(x) == y);// the union of all the sets
   variables of the array x is equal to y
```

Domain constraints, as the name indicates, define the domains of set variable and set variable arrays as shown below.

```
dom(home, x, SRT_SUB, 1, 5); // Constrains the domain of x
   to be a subset of the set {1, 2, ..., 5}. Note that x
   can either be a set variable or a set variable array.
dom(home, x, SRT_SUB, IntSet(1, 5));// Same constraint
   differently written.
```

Cardinality constraints, are also quiet self-explanatory, imposes the number of elements a set variable can have, as follows.

```
cardinality(home, x, 2, 4); // Imposes the cardinality of x
    to be between 2 and 4. In other words, x must have
   minimum two elements and maximum four. Again, x can
   either be a set variable or a set variable array.
```

Notice that the cardinality and domain for set variables and set variable arrays can also be directly specified in its constructor.

Different constrains functions with different arguments can actually express the same constraint as demonstrated hereunder.

```
dom(home, x, SRT_SUB, 1, 5);
rel(home, (min(x)>=1) && (max(x)<=5));
```

Channel constraints can link arrays of booleans, integers and sets variables. Its conduct can notably differ depending on the arguments passed. This said, we provide the two most significant examples in the context of our master thesis.

```
channel(home, x, y);
```

For two set variable arrays x and y, the channel function posts the constraints :

$$j \in x_i \Leftrightarrow i \in y_j \qquad \text{for } \; 0 \leq i \leq |x| - 1 \qquad 0 \leq j \leq |y| - 1$$

This constraint allows us to have a "dual"[3] variable structure that allows us to set some constraints more comfortably. In practice, it allows us to represent a set variable array where each index correspond to a specific time period; the sets are the pitches being played into another set variable array where each index correspond to a pitch and the sets are the time periods when this pitch is being played. A simple example is shown in figure 3.2. This is explained in more details in section 5.2.

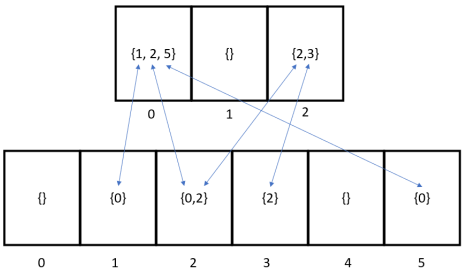


Figure 3.2: Channel example with two set variables arrays passed as argument

Another interesting behavior of the channel constraint is when a boolean variable array b and a set array x are passed as arguments as shown below.

```
channel(home, b, x);
```

Enforces the constraint

$$b_i = 1 \Leftrightarrow i \in x \qquad \text{for } \; 0 \leq i \leq |b| - 1$$

[3]We put dual inside quote-marks because it is not the correct terminology used in optimization problems

Finally, the **element constraint** presented hereunder

```
element(home, SOT_UNION,x, y, z);
```

constrains the set variable array x and the set variables y and z as follows

$$z = \bigcup_{i \in y} x_i$$

We have just presented the useful set constraints that were used for the musical constraints without entering too much into the details. For an exhaustive explanation on how these constraints functions were used to model the musical constraints, please refer to section 5.4

Reified constraints

Many constraints exist in their reified form where a boolean control variable is passed as the last variable. This boolean eventually inform us if the constraint was satisfied or not. For example,

```
rel(home, x, SRT_SUP, y);
rel(home, x, SRT_SUP, y, b); // fully reified version
```

the reified version constrain

$$x \supseteq y \iff b = 1$$

We expand the different types of reification and how their propagation occurs in section 4.2.

3.1.4 Propagators

Gecode provides the following propagation levels which are passed as an optional argument to the constraint functions :

- IPL_VAL to perform value propagation.
- IPL_BND to perform bound propagation.
- IPL_DOM to perform domain propagation.
- IPL_DEF to perform the default propagation.
- IPL_BASIC to optimize the execution performance at the expense of having a weaker propagation.
- IPL_ADVANCED to optimize the propagation strength at the expense of the execution performance.

Note that IPL_DEF can be omitted since it is the one performed if we don't pass any argument. In addition, we can request two propagation levels. The particular combination IPL_BASIC | IPL_ADVANCED is the most relevant since we would be requesting both advanced and basic propagation.

3.1.5 Branching

Gecode proposes different variable and value selection strategies that are passed to the function branch() which is in charge of performing the branching. If we pass a single variable to branch(), then we only need to pass a value selection strategy as an argument. Whilst, if we pass a variable array to the branch function, we then need to specify the variable and value selection strategy as extra arguments. For an n-sized array of integer set variables x, we can consider the two following fragments of code, which provide the same branching, as an example.

```
for(int i =0 ; i<n; i++)
    branch(home, x[i], SET_VAL_RND_INC(r)); // x[i] is a
   set variable and thus we only have to pass a value
   selection strategy as extra argument
```

```
branch(home, x, SET_VAR_NONE(), SET_VAL_RND_INC(r));// x is
    an array of set variable and thus we have to pass a
   value and a variable selection strategy as extra
   argument. In this case the variable selection strategy
   simply chooses the first unassigned variable of the
   array as in the for loop implemented above
```

Let's recall that the branching doesn't affect the solution set of the problem. Nevertheless, it affects the tree shape, the execution time to find solutions and which are the firsts solutions provided.

Variable selection strategy

When choosing our variable selection strategy we should make sure that it follows the first-fail principle stated in 2.4.4 or that at least that it doesn't oppose this principle. For example, the SET_VAR_RND(r) strategy doesn't follow the first-fail principle. However, it doesn't go against this principle and can actually provide very original rhythmic solutions. We won't state all the variable selection strategies proposed by Gecode since many of them go against the first-fail principle. Let's thus mention in figure 3.3 some of the strategies (out of the 26 proposed by Gecode) that could be reasonable to use.

We won't dive into a deep comparison analysis of the different variable selection strategies since it would lie outside of the scope of this thesis. However, we wanted to make a point about the importance of choosing a reasonable variable selection strategy. And not choosing an inadequate strategy such as SET_VAR_DEGREE_SIZE_MIN() where the branching would first be performed in variables that are less constrained and with a large domain. Which would imply a time-consuming execution to find solutions.

SET_VAR_NONE()	first unassigned
SET_VAR_RND(r)	randomly
SET_VAR_DEGREE_MAX()	most number of propagators depending on the variable
SET_VAR_AFC_MAX()	largest accumulated failure count of all propagators depending on the variable
SET_VAR_ACTION_MAX()	variables whose domain were pruned more often
SET_VAR_SIZE_MIN()	smallest unknown set
SET_VAR_DEGREE_SIZE_MAX()	largest number of propagators depending on the variable divided by domain's size
SET_VAR_AFC_SIZE_MAX()	largest accumulated failure count divided by domain's size
SET_VAR_ACTION_SIZE_MAX()	variables whose domain were pruned more often by their domain's size

Figure 3.3: Set variable selection strategies [22]

SET_VAL_RND_INC(r)	include random element
SET_VAL_RND_EXC(r)	exclude random element

Figure 3.4: Set value selection strategies [22]

Value selection strategy

Once we have decided which variable selection strategy to use, choosing a value selection strategy that follows the first-success principle is a significantly more subtle task that requires both a broader knowledge of the problem's constraints and of the selected variable selection strategy. For instance, if we want an increasing pitch melody and that we have a naive variable selection strategy where we branch from the left-hand side of the musical staff to its right-hand side, then a value selection strategy that follows the first-success principle would be to choose the smallest value of the domain as the value for the left branch. However, if we choose a more performant variable selection strategy such as to branch on the most constrained variable compared to its domain's size, then choosing a value selection strategy that follows the first-success principle becomes more complicated.

Hence, since choosing a value selection strategy that follows the first-success principle is a complex exercise for our model, we decided to employ the random set value selection proposed by Gecode. Its main advantage is that it provides more original and thus inspiring score solutions to the composer. Gecode gives the two random set value selection presented in figure 3.4.

If we want the first musical scores provided by the solver to be fuller, usually associated to a more fast chaotic pace sensation, we use SET_VAL_RND_INC(r). Conversely, if we want the first musical scores provided by the solver to be

emptier, usually associated to a more slow peaceful pace sensation, we use SET_VAL_RND_EXC(r).

We would like to reiterate that the chosen branching heuristic won't change the solution set, but it definitely influences the order in which solutions are presented. If there are hundreds or even thousands of solutions, the users of Melodizer 2.0 won't revisit all of them but only the first proposed ones. Therefore, branching should not be neglected.

3.1.6 Search

In figure 3.5 we can observe Gecode's proposed search engines. The theory behind each search engine has already been covered in section 2.4.5 . We have decided to use branch-and-bound for two reasons. Firstly, the possibility to maximize or minimize an objective function allowed us to recreate more musical scenarios. Secondly, the flexibility of the constraint() function provided by Gecode allowed us to do much more than simply maximizing or minimizing an objective function. It allowed us to provide different solutions to the composer contrarily to DFS that provided us solutions where only one note changed.

engine	shortcut	exploration	best solution	parallel
DFS	dfs	depth-first left-most		✓
LDS	lds	limited discrepancy		
BAB	bab	branch-and-bound	✓	✓

Figure 3.5: Available search engines in Gecode [25]

Search options

Search options are passed as an argument to the selected search engines. These can include : the number of solutions solicited, the maximum acceptable execution time before stopping the search, the maximum number of explored nodes to stop the search and the number of threads between others. In the following lines of code, we demonstrate an example of how to instruct the branch-and-bound search engine to look for the first ten solutions by using four threads.

```
1 SizeOptions opt("Problem"); // option object created
2 opt.solutions(10); // first ten solutions
3 opt.threads(4); // four threads solicited
4 Script::run<Problem,BAB,SizeOptions>(opt); // solve the
    problem modelled with BAB and the specified options
```

3.2 OpenMusic

OpenMusic (OM) is a visual programming language, based on Common Lisp and CLOS [4], designed for music composition [18]. Since OM is a musical extension of Lisp (abbreviation of List processing), you might expect to have parenthesized lists with many parenthesized sub-lists inside, in order to represent rhythms or chords progressions, for example.

When OM is launched a *workspace* and a *Lisp listener* are opened. The workspace is the main interface which can contain *maquettes*, lisp functions and *patches*. The Lisp listener shows the results of program evaluations and error messages, among others. Most of the visual programming is done inside the patch editor, which opens by double clicking on the patch icon [18]. Patches can communicate between them through their respective inputs and outputs.

3.2.1 Boxes within Patches

Boxes are the main components of patches. Boxes communicate between them through their respective inlets and outlets. A given box receives information through its inlets which are represented by blue dots on top of the box; then it transfers information through its inlets which are represented by blue dots on the bottom part of the box. Notice that boxes can have multiple inlets and outlets and that the type of information they receive and transfer depends from box to box, inlet to inlet and outlet to outlet. To connect two boxes simply click the outlet of a box and drag the line into the inlet of another box.

Data boxes represent primitive Lisp types, such as integers, floats, lists or strings. They do not have inlets and have only one outlet that transfers its value.

Function boxes works exactly like programming functions where we pass the arguments through the inlets and the returned values are communicated through the outlets. Note that the output of a function box can depend of the inlets' arguments type. For instance, if we attach two integers to the multiplication function of figure 3.6, the output that the outlet communicates will be an integer equals to the product of the two arguments. Whereas, if we pass an integer and a list of integers to the multiplication function, it returns a list of same length where each element correspond to the product of the integer argument times element at the same index of the argument list.

Object boxes produce instances of objects which are represented by classes, an important concept deriving from object-oriented programming. The first inlet and outlet reference the object itself, while the other inlets and outlets act as

[4] subset of Common Lisp dealing with object-oriented programming

setters and getters respectively.

Figure 3.6 provides an example of the OpenMusic environment and how we can connect objects, functions and data boxes through their inlets and outlets. We can observe how the text box only serves to take a peek into the list that is passed from the "mktree" function's outlet to the voice object's second inlet.

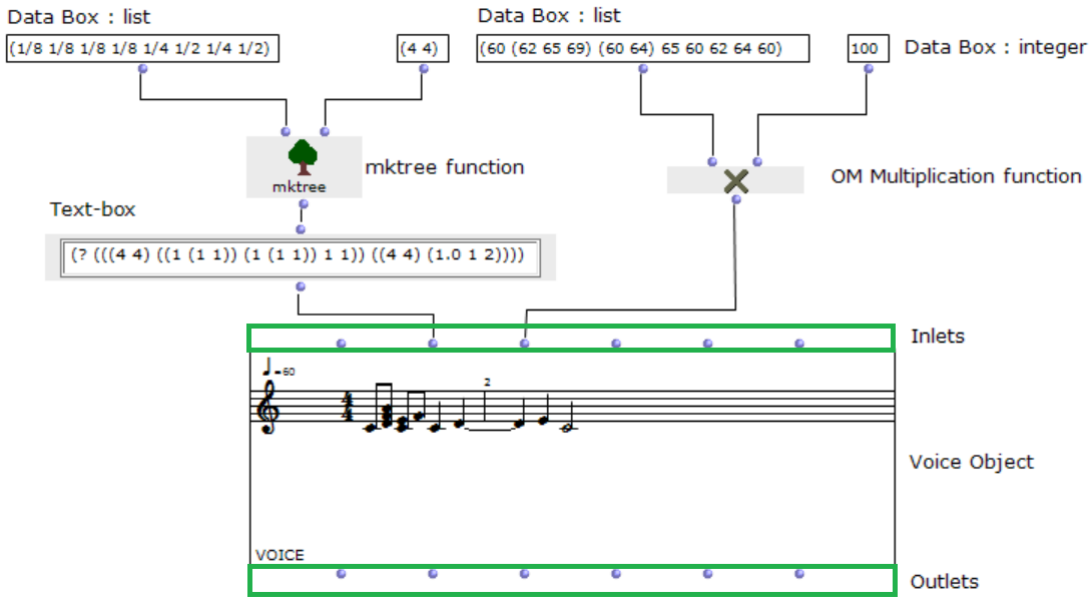


Figure 3.6: Example of the patch environment where data, functions and object boxes are connected

We won't explain all the features of OM nor explain OM in detail since there is already a complete documentation [18]. However, we look through the objects and functions that we considered to be relevant for the use of Melodizer 2.0 . If you have any doubts about the OM patch environment you can visit the OM documentation. Or, you can also type SHIFT+CTRL+H or click "Help->Editor Command Keys" for a window to pop with all the commands that might be useful as shown in figure 3.7.

3.2.2 How to represent score sheets in OM

Score boxes are a group of objects used to represent notes, chords and partitions. These objects can be instantiated by connecting data boxes through the inlets or by using its editor (double-click on the box) where you can modify and play the score. Figure 3.8 shows the different score objects proposed by OM.

Figure 3.9 presents the hierarchy of score objects. Multiple notes compose a chord as shown in figure 3.10, as well as a chord-seq and voice objects as illustrated in figure 3.11 and 3.12. Multiple voice objects compose a poly object

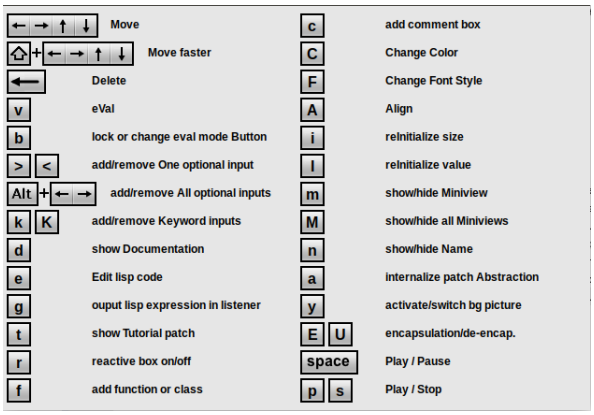


Figure 3.7: Useful OM patch's command keys

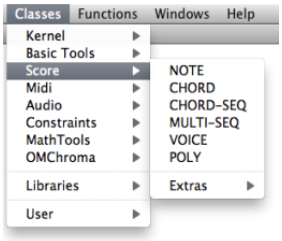


Figure 3.8: Different score objects proposed by OM

as demonstrated in figure 3.13 and multiple chord-seq objects compose a multi-seq object as illustrated in figure 3.14.

Notice that in figure 3.10 we could have rather attached this (6000 6400 6700) MIDICENT list to the corresponding inlet and it would have given the same chord. Similarly, for the examples in figures 3.11 and 3.12, we could have attached this ((6200 6500 6900) (6700 7100 7400) (6000 6400 6700)) list with MIDICENT sub-lists, representing a chord progression, to the corresponding inlets instead.

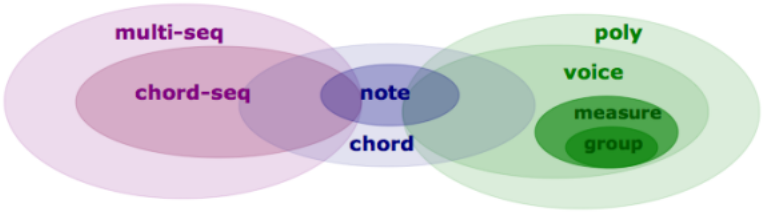


Figure 3.9: Score objects hierarchy [18]

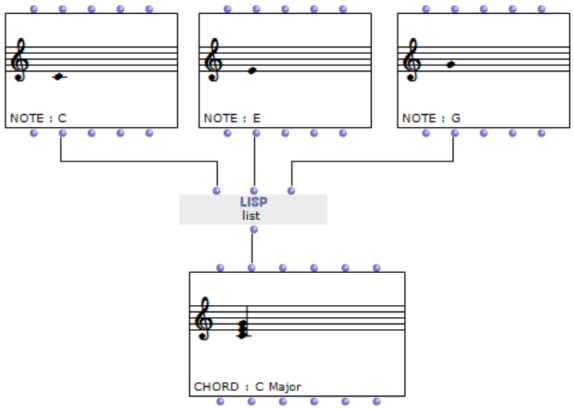


Figure 3.10: C Major chord made of C, E and G notes

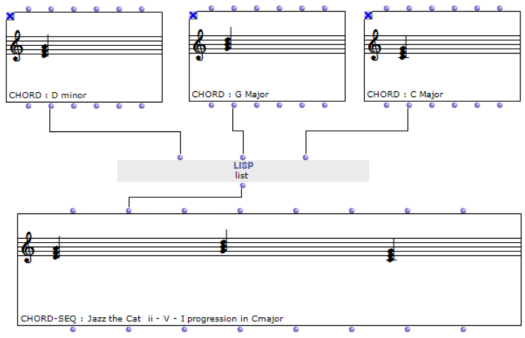


Figure 3.11: Chord-seq object representing
the Jazz Cat ii - V - I progression in C Major

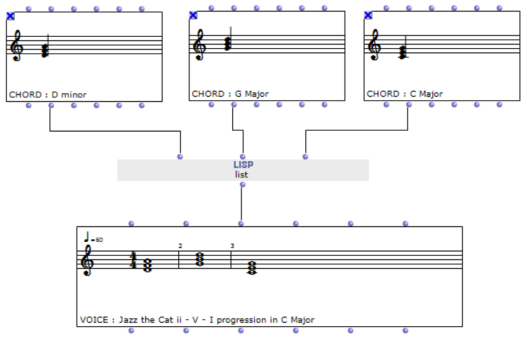


Figure 3.12: Voice object representing the Jazz Cat ii - V - I progression in C Major

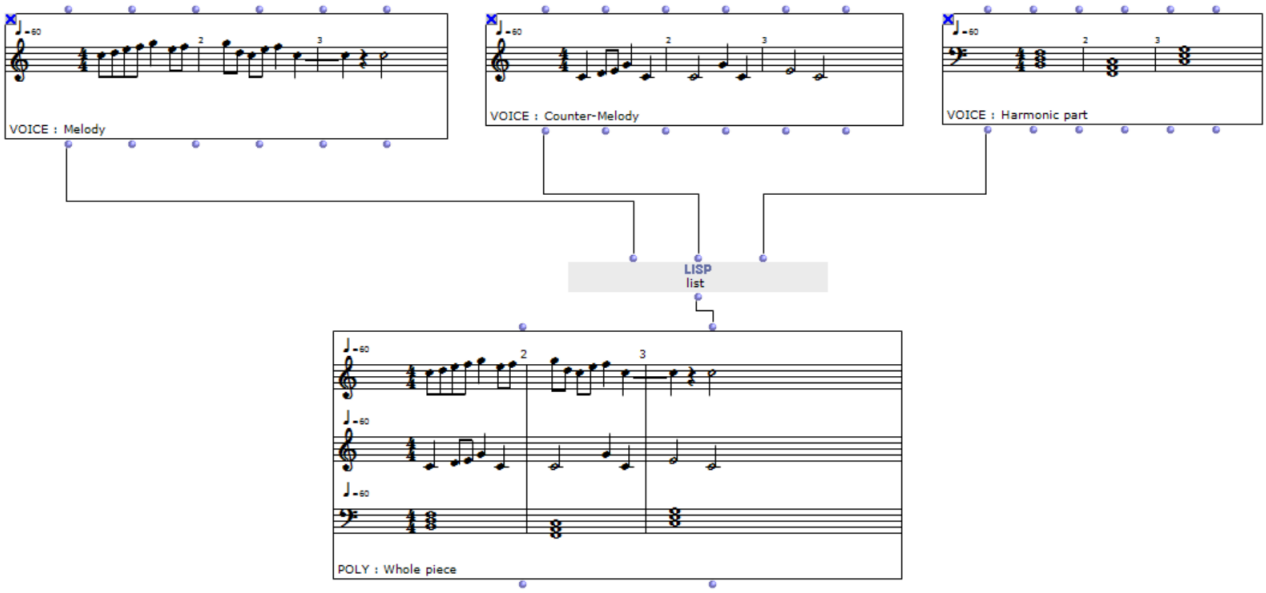


Figure 3.13: Poly object representing a piece made of a melody, a counter-melody and an harmonic part each represented by a voice object

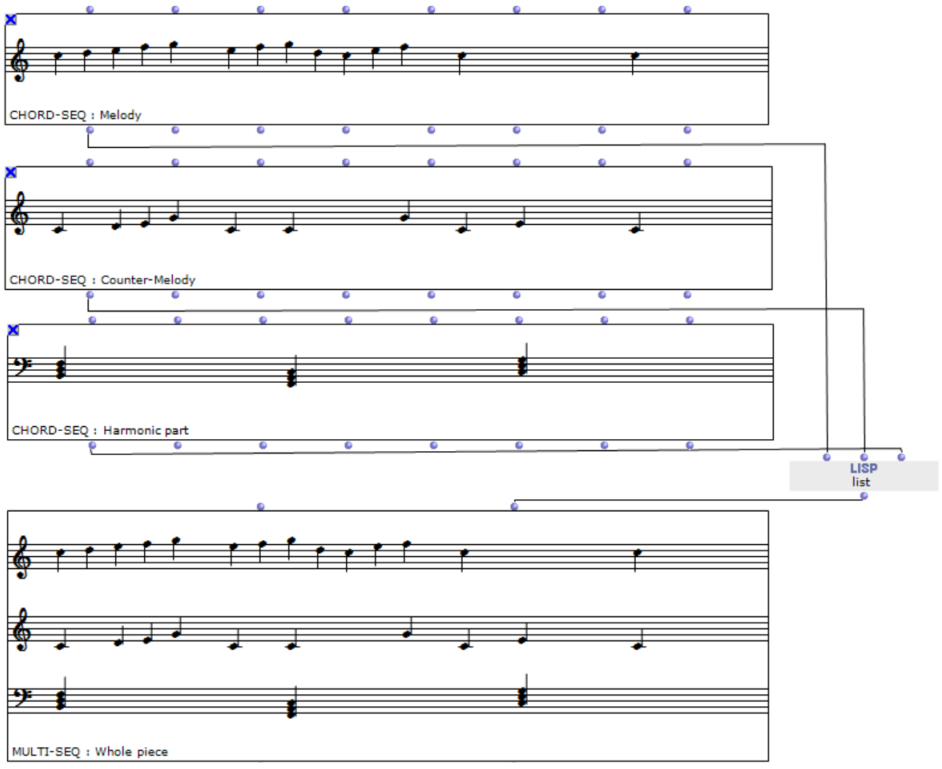
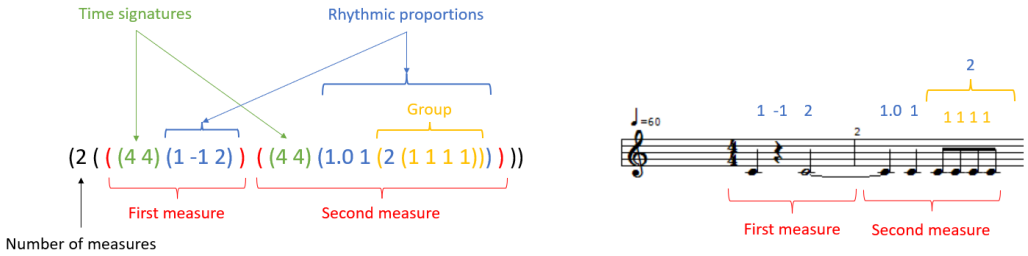


Figure 3.14: Multi-seq object representing the same piece as in figure 3.13 piece but the melody, the counter-melody and the harmonic part are now represented by a chord-seq objects.

Rhythm tree is a list that represents a rhythmic structure. It is mainly used for voice objects. The first element is the total number of measures that the rhythm has, but if there is an interrogation mark "?" it is OM's task to compute it. The second element is a list that contains as many sub-lists as measures. Each sub-list represents a bar and also has two sub-list as elements. The first one indicates the time signature while the second one represents the rhythmic proportion of the bar.
The rhythmic proportion's size list indicates how many notes and rests we have where the length of each note depends on its value accordingly to the total sum of the bar. Note that equal proportional structures, such as (1 2 1), (4 8 4) and (30 60 30), produce the same rhythmic result. There can be groups of notes that are again represented by a sub-list of two elements where the first one indicates the length duration of the group and the second is a sub-list indicating the rhythmic proportion of the group. Positive values represent notes, negative values represent rests and values followed by a ".0" are tied to the previous note. Let's observe figure's 3.15 example to have a better understanding.

In practice, it can be a little overwhelming to use as many sub-lists since you can easily get lost amongst this jungle of parenthesis. This is why you can easily generate rhythmic trees with the box function "mktree" as shown in figure 3.6.



(a) Rhythm tree

(b) Resulting voice object

Figure 3.15: Rhythm tree example

Voice and chord-seq objects

Let's begin by presenting the differences between chord-seq and voice objects and compare their main advantages. Voice objects can be instantiate by connecting a rhythm tree to the second inlet and a MIDICENT list into the third inlet for the pitch related part. While, chord-seq objects can be instantiate by connecting a MIDICENT list into the second inlet; the third inlet is used for specifying at what time expressed in milliseconds the note or chord starts to be played and its duration (also expressed in milliseconds) is specified through the fourth inlet.

It can thus be seen that voice objects respect classical score sheets representation as introduced in 2.2.2, which explains theoretically rhythm principles. Voice objects are thus preferred by instrument performers and by composers that are very comfortable with Music theory. While, on the other hand, chord-seq are more familiar to composers that are used to music creation softwares. Furthermore, it is significantly more intuitive to employ chord-seq with the variable structure of Melodizer 2.0 introduced in 5.2 .

Notice that chord-seq and voice objects can also be edited by double-clicking on its box and then using the editor's available commands shown in figures 3.16 and 3.17. We won't dive into details since the interface is pretty intuitive to use. To prove this with an example, you can change the pitch of a note or chord by selecting it with your mouse and dragging it up or down. Similarly, in the chord-seq object, you can change the beginning time that a note or chord is played by selecting it with your mouse and dragging it left or right. Furthermore, there is an entire section of the OM's documentation dedicated to Score objects [19].

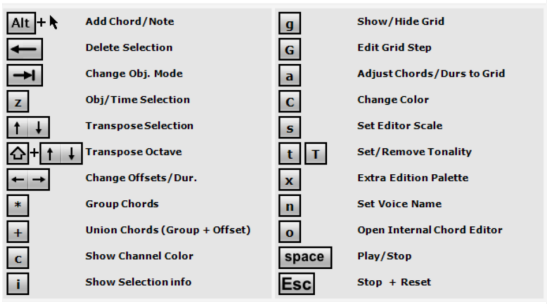


Figure 3.16: Commands for CHORD-SEQ Editor

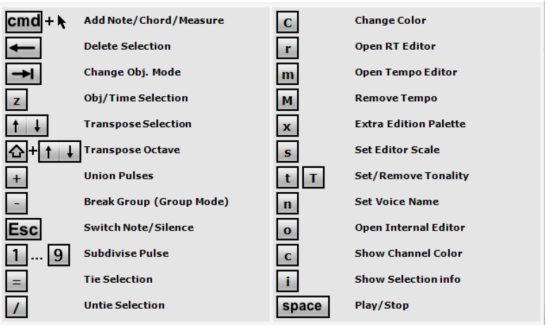


Figure 3.17: Commands for VOICE Editor

3.2.3 Box evaluation

Once we have visually structured our program by connecting the different boxes in order to run it, we have to evaluate the desired box by clicking on it and pressing it. If we don't evaluate, boxes are set to their default value; as you can compare between figure 3.18 and its evaluated counterpart 3.6.

When evaluating a box, all the upstream boxes connected directly or indirectly to the inlets of the box are also evaluated one by one, unless they are locked. This evaluation chain is performed bottom-up and left to right.

If you edit a voice object, for example through his editor window, and don't want to re-evaluate the box and lose your modifications. Then, you should make sure that the box is locked with a little blue cross on its top-left corner. To lock and unlock boxes you simply have to click on it and press b.

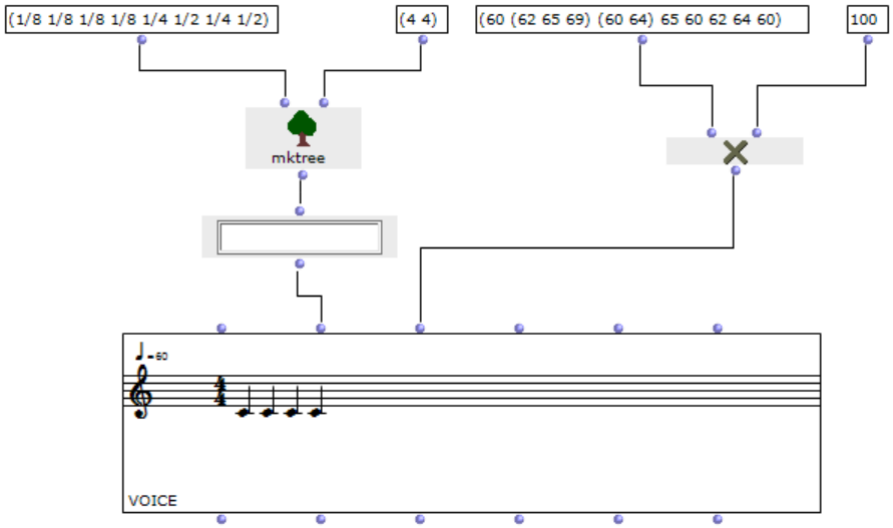


Figure 3.18: Figure's 3.6 example before evaluation

Chapter 4

GiL

When trying to build a constraint based musical composition tool into OpenMusic the first challenge is unsurprisingly bringing the constraint solver to Lisp. This task was done by Baptiste Lapière [10]when he created the GiL interface between Lisp and Gecode. We used his work to create our own composition tool, but we also improved Gil to handle more cases and realize specific task that were not thought of before.

In this section we give a brief explanation of the functioning of GiL without going in much details as it was already thoroughly detailed in another master thesis that the interested reader can find here[10]. Instead, we focus on the improvement that were brought to GiL and how you can use it or modify it yourself.

4.1 How does it work ... briefly

GiL is composed of 4 main files that can easily be divided in 2 distinct parts: the C wrapper and the Lisp Wrapper. Those two parts are linked thanks to the "Common Foreign Function Interface (CFFI)". As its name suggests, this interface allows us to call function and access variables in another programming language, in this case C. Figure 4.1 shows the structure of Gil, each rectangle represents a file and arrows show the direction of function calls.

4.1.1 Lisp Wrapper

The Lisp wrapper is there to wrap our C library, which is explained just below, using CFFI. The main part is the definition of the foreign function to link C

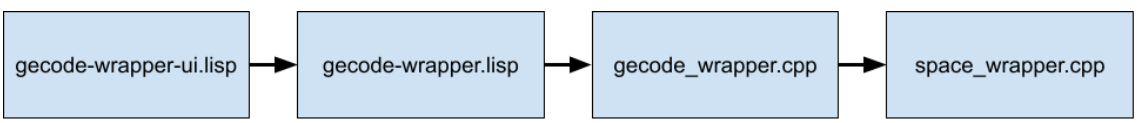


Figure 4.1: Gil file structure

and Lisp, each foreign function is used to call a specific method of the Gecode Wrapper. The second part is a simple wrapper to this first part to make the signature of the function more readable and streamlined, in order to facilitate the usage of GiL and make it as similar as possible to using Gecode.

4.1.2 C Wrapper

As said just above CFFI allows us to call C function from Lisp code, but our constraint solver Gecode is a C++ toolkit so we first had to build a C library capable of executing Gecode functions while being called from Lisp code.

The C wrapper is also made of two parts: the Gecode Wrapper and the Space Wrapper. The first one is the C library that used to call the methods from the Space Wrapper while being called from the Lisp Wrapper explained before. The second one, contains all the calls to the Gecode functions and the definition of the WSpace (for wrapper space) class which is the translation of the Gecode's space. It also represents a problem and contains all the methods to add variables and post constraints as well as branching. In order to use WSpace with CFFI, that we already know it doesn't support this kind of data structure, we have to cast it to a void pointer and inversely cast it back to a WSpace object when we want to use in the space Wrapper.

Variables are instances of C++ class, so we can not reference them from the C code as they are not compatible. To overcome this problem, we store them in vectors. There is one vector per variable type namely IntVar, BoolVar, and SetVar. When creating a new variable, its index in the vector is returned and is the value we need to pass when calling a function to post a new constraint.

Constraints are simply called using one or more methods wrapped in the WSpace.

Search engines are a bit more complex as they have their own wrapper containing a reference to an existing Gecode search engine. They have to provide methods to search the next solution which return a new space that holds the solution to the CSP. Ideally, they should also provide methods to stop the search and set the options of the search, the number of threads and the timeout, for example.

4.2 New features

Improving GiL was not the main objective of this master thesis but it was a necessary step in order for our music composition tool to achieve all the things we planned it to. So all the following features were added with a specific usage in mind. Still, their implementation was made as general as possible to be usable

Identifier	Relation
SRT_EQ	Equality (=)
SRT_NQ	Disequality ($\neq$)
SRT_SUB	Subset ($\subseteq$)
SRT_SUP	Superset ($\supseteq$)
SRT_DISJ	Disjoint ($\|\|$)
SRT_CMPL	Complement
SRT_LQ	Less or Equal ($\leq$)
SRT_LE	Less ($<$)
SRT_GQ	Greater or equals ($\geq$)
SRT_GR	Greater ($>$)

Identifier	Operation
SOT_UNION	Union ($\cup$)
SOT_DUNION	Disjoint union
SOT_INTER	Intersection ($\cap$)
SOT_MINUS	Difference ($\setminus$)

Figure 4.2: Operation and relation types on SetVar [24]

by other people in other projects and in a way that makes it easy to develop over them or add new features without conflicts.

- **Linux compatibility** [1] : Before, this work Gil was only usable on MacOS, making it inaccessible for an important number of potentially interested users. In this objective a new compilation method was added to create a *.so* file compatible with Linux. The loader file of Melodizer 2.0 used by Open Music to load the library automatically chooses the correct file to use depending on the operating system.

- **SetVar** : GiL was limited to use only a few basic Gecode variable types, namely IntVar and BoolVar, with their respective arrays variation: IntVarArray and BoolVarArray. When dealing with music, this representation seemed a bit weak as music is usually the combination of multiple tones playing at the same time and having specific relations between them, which is hard to represent with the variable types previously cited. The SetVar variable is used to represent a set of int values which can easily be transferred to a chord of notes being played at the same time. This new variable type is the core of the new features that were added, and most of the following additions were made to allow us to interact more with those SetVars.

- **Operation and relation constraints** : Two essential constraints to work with a new variable type. These two constraints, as their name suggests, allow us to execute operations and ensure some relation between SetVars. In figure 4.2 is located the list of possible relation types and operations on SetVar along with their Gecode variable names that can also be used through GiL.

- **Cardinality constraint** : It constrains the size of the domain of a SetVar. Given a SetVar x and two integer i and j, the constraint ensures that the

[1] the compatibility was only tested on Debian distribution

domain size is larger or equal to i and smaller or equal to j.

$$i \leq |x| \leq j \tag{4.1}$$

- **Domain constraint** : It is very similar to the relation constraint but instead of having a relation between two SetVars, we have a relation between a SetVar and a domain that can be represented in many ways. The two ways of representing a domain in the domain constraint we have added to GiL are :

  1. a full domain represented by its lower and upper bound. For a SetVar x, a relation r and a domain with bound i and j we constrain that :

  $$x \sim_r \{i, ..., j\} \tag{4.2}$$

  2. a domain represented by a SetVar. For two SetVars x and d we can constrain domain of x according to domain of d.

  Using a SetVar in the domain constraint can seem surprising as we have just written that the relation constraint can do that. Yet, constraining two SetVar using the domain constraint actually creates between them a superset/subset relation in a more efficient and easy way to use than the relation constraint.

- **Empty constraint** : This constraint directly inherits from the domain constraint but in the specific case in which we want the domain of a SetVar to be empty. For a SetVar x we ensure that :

$$x = \emptyset \tag{4.3}$$

- **Channel constraint** : This constraint creates a channel between two arrays of SetVars. For two arrays of SetVars x and d, it ensures that if i is an element of the domain of the SetVar at index j in x, then j is an element of the SetVar at index i in d.

$$j \in x_i \iff i \in d_j \tag{4.4}$$

- **Minimum and Maximum constraint** : They create a new IntVar constrained to the minimum/maximum value of a SetVar. For a SetVar s and an IntVar x, they ensure that x is the minimal/maximal element of s, and therefore that s is not empty.

- **Reification** : This was added as optional to multiple constraints. Reification is a way to control the validity of a constraint through the use of a boolean variable. For example, the following constraint posts that the int variable x should be equal to the int variable y.

```
1 rel(home, x, IRT_EQ, y);
```

Reification mode	identifier	propagation
Equivalence (full)	RM_EQV	$b = 1 \leftrightarrow c$
Implication	RM_IMP	$b = 1 \leftarrow c$
inverse implication	RM_PMI	$b = 1 \rightarrow c$

Figure 4.3: Reification modes with their propagation's direction [21]

Next is the reified version of this constraint with the boolean variable b

```
rel(home, x, IRT_EQ, y, b);
```

The propagation associated to this constraint function as follows :

1. if b is assigned to 1, the constraint $x = y$ is propagated.
2. if b is assigned to 0, the constraint $x \neq y$ is propagated.
3. if $x = y$ holds, $b = 1$ is propagated.
4. if $x \neq y$ holds, $b = 0$ is propagated.

In this case we are presenting full reification, however, half of it was also added to GiL. The implication reification only propagates according to 1. and 4. . While inverse implication reification only propagates according to 2. and 3. . Figure 4.3 presents a list of the various possible modes for reification and how they modify the propagation of a constraint c.

- **BAB search constraint** : Branch and bound was already present in Gil, but it was not possible yet to add new constraint after finding a solution to influence the next solution found, in other words using the BAB search was equivalent to using a DFS. The constrain function, which is called every time the user request the next solution was implemented into Gil the same way as a new constraint. It takes as argument the space of the previous best solution and is executed in the space of the next solution. For the moment, this function ensures that a certain percentage of the next solution is different from the previous one as explained in section 5.6. Below is the code in Gil for this constrain function with the current constraints used in Melodizer.

```
void WSpace::constrain(const Space& _b) {
    const WSpace& b = static_cast<const WSpace&>(_b);
    //getting variables of the previous solution
    SetVarArgs bvars(b.var_sol_size);
    for(int i = 0; i < b.var_sol_size; i++)
        bvars[i] = (b.set_vars).at((b.
   solution_variable_indexes)[i]);
    //getting variables for the next solution
    SetVarArgs vars(b.var_sol_size);
    for(int i = 0; i < b.var_sol_size; i++)
        vars[i] = (set_vars).at((b.
   solution_variable_indexes)[i]);
```

```
11      //Constraints on the variables, should be modified
        according to the use
12      for(int i=0; i<b.var_sol_size; i++){
13          if((rand()%100)< b.percent_diff){
14            SetVar tmp(bvars[i]);
15            rel(*this, (tmp!=IntSet::empty) >> (vars[i]
        != tmp) );
16          }
17      }
18  }
```

As seen in the code above we use *solution_variable_indexes* to specify on which variables the constrain function should add constraints. This is necessary as the variables in Gil are stored in arrays without information on their uses and in most applications we don't want to apply the constrain function on all the variables. The variables to be used in the constrain function can be specified through the *g-specify-sol-variables (sp vids)* method.

4.3 How to use GiL

In this section we show the usage of GiL through an example comparing how to solve a problem in C++ using Gecode and in Lisp using GiL. The problem we are trying to solve is finding correct Golomb rulers of a specific size. A Golomb ruler is a set of marks at integer positions along a ruler such that there are no pairs of marks that are at the same distance apart. You can find in listing 4.1 the C++ implementation which was taken from the official Gecode examples [2] and slightly modified, and in listing 4.2 the lisp code. In this example, we have a distinct constraint on an IntVarArray and multiple operations and relations on IntVar. The branching strategy we use selects the first unassigned value and the smallest value of the domain first. We then use a depth first search engine to find the solutions.

```
1  class GolombRuler : public IntMinimizeScript {
2  protected:
3    IntVarArray m;
4  public:
5    GolombRuler(const SizeOptions& opt)
6      : IntMinimizeScript(opt),
7        m(*this,opt.size(),0,(1 << (opt.size()-1))-1) {
8
9      // Assume first mark to be zero
10     rel(*this, m[0], IRT_EQ, 0);
11
12     // Order marks
13     rel(*this, m, IRT_LE);
```

[2] https://www.gecode.org/doc/6.2.0/reference/classGolombRuler.html

```
14
15      // Number of marks and differences
16      const int n = m.size();
17      const int n_d = (n*n-n)/2;
18
19      // Array of differences
20      IntVarArgs d(*this, n_d, 0, (1 << (m.size()-1))-1);
21
22      // Setup difference constraints
23      for (int k=0, i=0; i<n-1; i++)
24        for (int j=i+1; j<n; j++, k++)
25          // d[k] is m[j]-m[i] and must be at least sum of
    first j-i integers
26          rel(*this, d[k] = expr(*this, m[j]-m[i]),
27                      IRT_GQ, (j-i)*(j-i+1)/2);
28
29      distinct(*this, d);
30
31      // Symmetry breaking
32      if (n > 2)
33        rel(*this, d[0], IRT_LE, d[n_d-1]);
34
35      branch(*this, m, INT_VAR_NONE(), INT_VAL_MIN());
36    }
37
38    // Code that doesn't help comparing Gecode and GiL
39
40  int main(int argc, char* argv[]) {
41    SizeOptions opt("GolombRuler");
42    opt.solutions(0);
43    opt.size(10);
44    opt.parse(argc,argv);
45    if (opt.size() > 0)
46      IntMinimizeScript::run<GolombRuler,DFS,SizeOptions>(opt
     );
47    return 0;
48  }
```

Listing 4.1: Golomb ruler using C++

```
1  (defun golomb-ruler (size)
2      (let ((sp (gil::new-space))
3            m d se sopts size-d k)
4          ; initializing the IntVarArray
5          (setq m (gil::add-int-var-array sp size 0
6          (- (expt 2 (- size 1)) 1)))
7
8          ; Assuming first mark to be zero
9          (gil::g-rel sp (nth 0 m) gil::IRT_EQ 0)
```

```
10
11          ; Order marks
12          (gil::g-rel sp m gil::IRT_LE nil)
13
14          ; Number of differences
15          (setf size-d (/ (- (* size size) size) 2))
16
17          ; array of differences
18          (setq d (gil::add-int-var-array sp size-d 0
19          (- (expt 2 (- size 1)) 1)))
20
21          ; Setup difference constraints
22          (setf k 0)
23          (loop :for i :from 0 :below (- size 1) :by 1 :do
24             (loop :for j :from (+ i 1) :below size :by 1 :
    do
25                 (progn
26                     (gil::g-linear sp '(1 -1) (list (nth j
    m) (nth i m))
27                      gil::IRT_EQ (nth k d))
28                     (gil::g-rel sp (nth k d)
29                      gil::IRT_GQ (* (- j i) (/ ( + (- j i)
    1) 2)))
30                     (setf k (+ k 1)))))
31
32          (gil::g-distinct sp d)
33
34          ; Symmetry breaking
35          (if (> size 2)
36             (gil::g-rel sp (nth 0 d) gil::IRT_LE (nth (-
    size-d 1) d)))
37
38          (gil::g-branch sp m gil::INT_VAR_NONE gil::
    INT_VAL_MIN)
39
40          (setq sopts (gil::search-opts))
41          (gil::init-search-opts sopts)
42
43          (setq se (gil::search-engine sp (gil::opts sopts)
    gil::DFS))
44          (list se m sopts)))
45
46 (defun search-next-golomb-ruler (l)
47     (let ((se (first l)) (mark* (second l)) (sopts (third l
    )) sol marks)
48          (setq sol (gil::search-next se))
49          (if (null sol)
50              (error "No more solution"))
```

```
51          (setq marks (gil::g-values sol mark*))))
```

Listing 4.2: Golomb ruler using GiL

4.4 How to improve GiL yourself

Gil is open source and any improvements are most welcome. If you feel like adding some constraints or more, here is the procedure to follow:

To add to GiL you first have to get the source code [3], then you can wrap your gecode code in the space_wrapper.cpp file. Most usage only requires you to wrap your code in a single C++ function. However, some special addition, like a complete search engine, needs to be wrapped using a class. After that, you have to create one or more functions in the gecode_wrapper.cpp file to call the functions you have just created in the space wrapper. Don't forget to complete the headers files with the signature of all the functions you created in the corresponding files.

Now we are going to write some Lisp. Head to the gecodewrapper.lisp file and call the C function you've just created using CFFI. Also, add more user friendly call to those lisp function in the gecodewrapperui.lisp file and all the coding is done.

Now head to the C++ folder where lies the make-file that helps you compile your new Gil version. Open a terminal and, depending on your OS, execute *make so* if you are using Linux or *make dylib* if you are on MacOS. Now everything should be setup. Note that if you compile Gil on Linux and switch to MacOS you will have to recompile it in order to use it, and reciprocally from MacOS to Linux.

[3] https://github.com/sprockeelsd/GiLv2.0

Chapter 5

Melodizer 2.0

In this chapter is presented the center part of our work, the OpenMusic' library Melodizer 2.0. The next sections contain information about the new features of the library, the new structure of variables used, all the available musical constraints, how the solver works and all of the implementation to it.

5.1 What is Melodizer ?

Melodizer is a tool for musicians to create melodies in the non-traditional way of describing music with mathematical constraints, rather than regular music theory. It has been created by Damien Sprockeels and released in January 2022 as a part of his master thesis. It is coded in Common Lisp and runs in Open Music as an external library. Since its release, we have been developing a new version in which we wanted to improve its completeness, its usability and its efficiency.

It is important to understand that we do not want Melodizer to replace musician's creativity nor come up with a full masterpiece when launched. Instead, we like to think of Melodizer as a melody synthesizer with many knobs and buttons to tweak in search of inspiring results. In fact, the main purpose of the tool is to amplify the composer's creativity.

5.1.1 New features

There is a lot of novelty to this version of Melodizer, first off the structure of the variables has been completely modified, with new a variable type and a representation that allow to create the rhythm of the melody, more about that in section 5.2. New Open Music objects were also added to give more control on the constraints, and more complexity to the solution. You can find more on this subject in section 5.3. As expected new musical constraints are also available, to give more options of composition, and also to control the rhythm which was not editable previously, all those constraints are listed and explained in section

5.4. Finally a new and improved solving algorithm was implemented, with new features and more control given to the user, this part is developed in section 5.6

5.2 Variable structure

In this new representation, we decided to use SetVarArrays, instead of IntVarArrays, in order to represent multiple notes at the same time. Each entry of the array is not a note but a specific time (where we discretized the beats). The size of the arrays depends on how much the composers want to discretize a measure and how many measures does his melody have. For example, if they want to have 4 measures and discretize each measure 16 possible notes, then Push and Pull will be of size $4 \times 16 + 1 = 65$. We added one extra element to the array in order to pull all the notes that were played at the end of the array and finish the piece. At each entry, we can push (start playing) or pull (stop playing) a set of notes (expressed as MIDI pitches) as in figure 5.1.

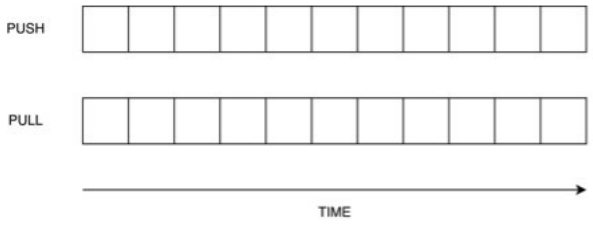


Figure 5.1: New representation

Hence, it can be seen that with this new representation, pushing chords and playing several notes at the same time is much easier and intuitive. Let's consider the following example where we want to make a 4 bar C-Am-G-G chord progression with a 1 beat quantification. We just have to constrain the variables as in figure 5.2. Figure 5.3 provides the score representation of the C-Am-G-G chord progression.

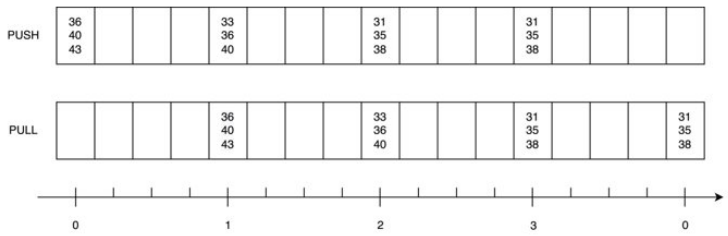


Figure 5.2: C-Am-G-G chord progression example

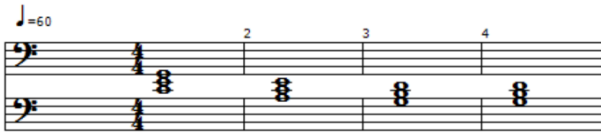


Figure 5.3: C-Am-G-G chord progression score representation

In this new model, we decided to create some redundant variables that considerably ease the task of modeling some musical constraints. By redundant variables we mean variables that aren't independent to Push and Pull variables and thus that we don't have to branch on. The principal redundant Set Variable Array is Playing, a link between Push and Pull, with each entry still being a time and the value inserted represents a note being played at this specific time, meaning the note has already been pushed but not pulled yet. Figure 5.4 illustrates the Playing array of C-Am-G-G chord progression example given at figure 5.2 .

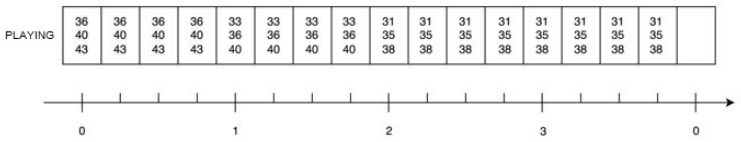


Figure 5.4: C-Am-G-G chord progression example Playing variable

In order for our model to work, we had to impose some structure constraints such that it had mathematical sense:

- Pull[0] has to be empty. Logical since it's the beginning of the piece and we haven't pushed any keys yet. $Pull_0 = \emptyset$
- Similarly, Push[end] has to be empty. $Push_{end} = \emptyset$
- Cannot pull a note that is not being played. $Pull_i \subseteq Playing_i$
- Cannot push a note that is played and isn't pulled. $Push_i \cap (Playing_{i-1} - Pull_i) = \emptyset$

PushMap and PullMap are also important redundant variable arrays of sets of integer created using the **channel** constraint where each entry of the array corresponds to a pitch and the sets are the times where those pitches are pushed or pulled as shown in figure 5.5. These arrays are considered as an alternative approach to represent Push and Pull that help us when modeling some musical constraints. Figure 5.6 provides the C-Am-G-G chord progression example of figure 5.2 represented by pushMap and pullMap. Notice how the pushMap and pullMap representation resemble more to the MIDI Piano representation shown in figure 5.7. While the Push and Pull representation resemble more to the classical music score as illustrated in figure 5.3.

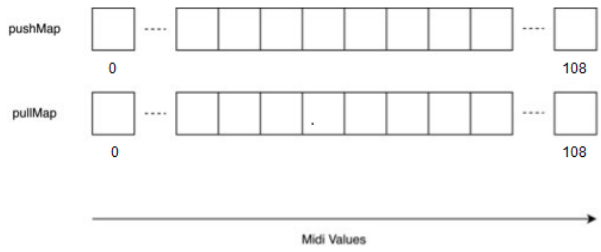


Figure 5.5: Representation of pushMap and pullMap

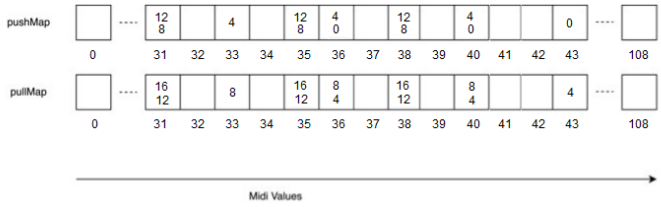


Figure 5.6: pushMap and pullMap : C-Am-G-G chord progression example

5.3 Blocks

Blocks are one of the most important new feature of Melodizer 2.0. It was introduced to allow the creation of more complex and interesting melodies. In practice, blocks are a new class in Open Music that can be linked together to mix and juxtapose constraints. The next section gives a detailed explanation of what are blocks and how they work individually, while the following section goes over how to connect blocks and what this does.

5.3.1 Block definition

A block is an abstraction that can be used to represent a part or the totality of a melody of various length using constraints. Before connections every block is an independent instance of a lisp class with its own set of variables as described above (push, pull, pushMap, ...) and constraints selected by the user, it therefore represents a full constraint solving problem that can be individually solved.
In Open Music blocks are embodied by an object with multiple inputs and outputs which is discussed in the next subsection. Every block object added by the user are independent before linking and can be used with a search object (see subsection 5.6 Solver) to find a solution to a CSP representing a melody. In order to control the constraints contained in the problem, blocks come with an interface allowing to add and modify multiple musical constraints listed and explained in subsection 5.4.
In figure 5.8 is a visual representation of the operation of a simple individual block, here the rectangle represent a block with its constraints chosen by the user, the blue squircle represents the CSP contained in the block, this CSP can be

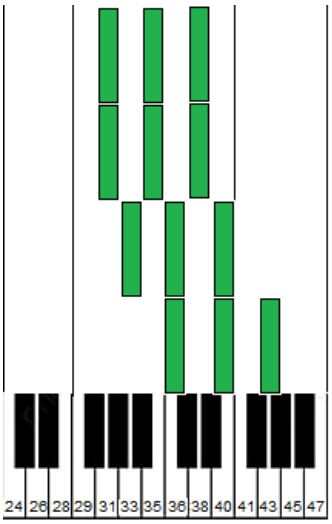


Figure 5.7: MIDI Piano : C-Am-G-G chord progression example

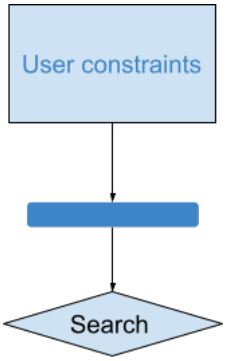


Figure 5.8: A simple block structure representation

fed into a search block which takes care of finding solutions. It is important to understand that the block doesn't return a fixed melody, but a CSP representing a melody with a defined length.

5.3.2 Blocks connection

Blocks used individually don't give much more option than the old Melodizer implementation, except for the new variety of musical constraints. But as you might expect blocks actually have much more to offer through the way we can connect them together. When composing music it is hardly possible to create a whole piece with every notes bound to the same sets of constraints, it is usually necessary to apply specific constraints on specific part of a score and individual voice. All of this is possible by connecting blocks together.

The main principle is that a block, that we will call parent from now on, can take one or more blocks as input, that we will call children, with a specific position as seen in figures 5.9 and 5.10. When doing so the parent block considers

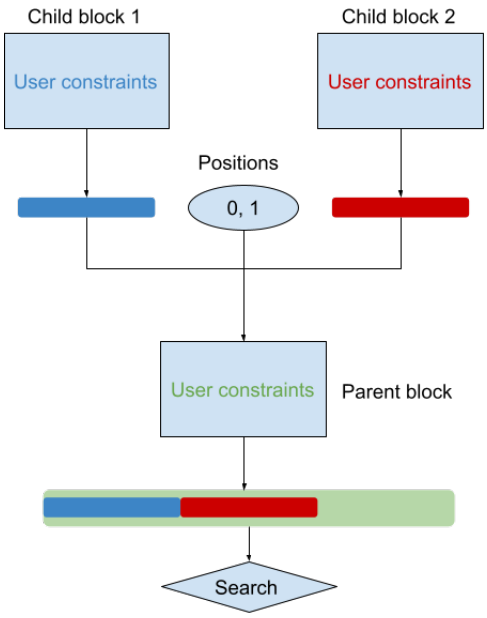


Figure 5.9: Blocks structure representation with juxtaposition of children constraints

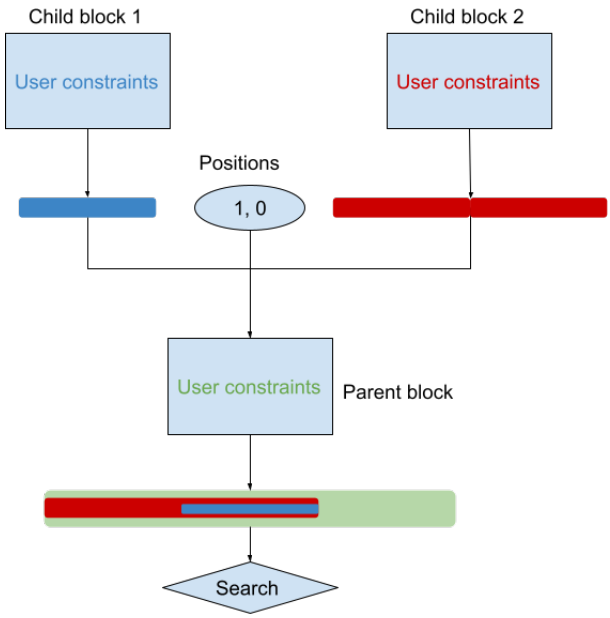


Figure 5.10: Another block structure representation with superposition of children constraints

the constraint of the child block in addition to its own constraints, this layering of constraints takes place from the specified position and for the length of the child block, both can be sets by the composer. using this mechanic in a tree like structure allow to enforce very specific constraints on precise note of a full piece. This mechanic help composer to mix various melody or simply put them one after the other.

In fig 5.9 is a more visual representation of how the interaction between the blocks work in the same simple example as in figure 5.11. Here a parent block has two child, each one creating a one bar melody with their own constraints, the parent block create a 3 bars melody and place the child blocks constraint respectively at positions 0 and 1 in bar. As seen in the figure 5.9, the final 3 bars melody is constraint on its whole length by the constraints of the parent block, but in addition we have the constraints of the first child block on the first bar, and the constraints of the second child block on the second bar.

This example shows the juxtaposition of the two child blocks constraints and superposition with the parent block constraints, but some combinations of block length and positions can lead to superposition of the child blocks in addition to the superposition with the parent blocks as seen in figure 5.10 where the first child still create a one bar melody but the second one generate a two bars melody, creating an overlap between the constraints of the two blocks in the final CSP. Layering multiple blocks constraints in this fashion is useful to create different voices in the melody, separating chords, bass and lead for example, while

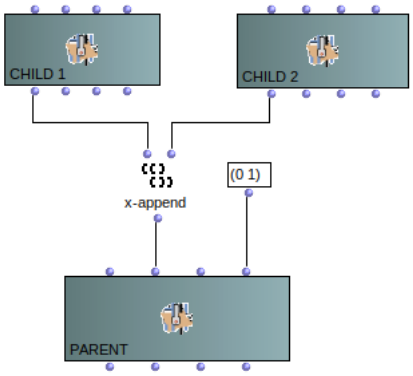


Figure 5.11: Connection between blocks in Open Music

juxtaposition of constraints as shown previously is more meant to create evolution throughout the composition, like it is done in classical ragtime with the famous AA BB A C melody pattern.

To achieve this in practice we constrain that the various variables described above in the parent block, at the positions specified by the user, have to be a superset of the same variables in the child block. When doing so we ensure that the constraint of the child are respected at the specified location and also the constraint of the parent block that applies on the whole melody described by the block.

After having build our inheritance tree and specified all our constraints we need to get a score from all those blocks, in order to do that we have to connect the root of the tree to a search object, which handles the call to the solver. In fig 5.11 is an example of how to connect blocks and the search object with a parent blocks and two children, more information on how to practically connect the block is given in chapter 6 User manual.

Voice object input

Sometimes a composer want to add an already existing melody to his piece, to do that it is possible to use a voice object as an input of a block in addition to eventual child blocks as they both use different inputs. When using this possibility the content of the voice object is added to the CSP and is therefore part of the constraints allowing to more easily create melody or chords to go with an existing piece of music.

To achieve this in practice we create a fake block in which the variables push and pull are already set to represent the same melody as the voice object, this fake block is then considered in the same way a child block would be.

5.4 Musical constraints

When opening a block object in Open Music you are met with a lot of button slider, all modifying various parameters of constraints in the background. In this section we describe their actions in a more mathematical and computational way, for information on their musical usage we recommend reading chapter 6 "User Manual".

Below is the list of all available options sorted as they are in the Open Music interface, with the constraints they use, their mathematical explanation and a basic implementation using Gecode. The implementation is shown using Gecode as it is more compact and easier to read than Lisp using Gil, the implementation might not seem optimal in some cases as we tried to keep the same idea as the final implementation used in Melodizer, which is bound to other restriction than Gecode. Also notes that some small elements like adding one to the size of array and other equivalent operations have been neglected in the math representation and shown implementation as they are not important for the understanding of the constraint and would hide the essential.

5.4.1 Blocks' general constraints

- **Bar length** : This option is used to set the total length of the melody created by the current block. It determines the number of bars of the melody, each bars containing four beats. The implementation is pretty straightforward as we simply have to multiply the value of the length entered by the user (denoted barLength) by a predefined quantification factor (denoted quant) and use this value as the size of the arrays described above that represent our score.

$$|Push| = |Pull| = \text{barLength} * \text{quant} \tag{5.1}$$

  In Gecode constraint programming this translate to the creation of two set variable array

```
1 SetVarArray push(*this, barLength * quant, IntSet::
     empty, 0, max_pitch);
2 SetVarArray pull(*this, barLength * quant, IntSet::
     empty, 0, max_pitch);
```

  with the range [0, max_pitch] being a range of acceptable pitch.

- **Voices** : Determine the maximum number of notes that can be played at the same time. To achieve our goal we simply constrain that the cardinality

of each elements of the playing array should be between 0 and the number entered by the user (denoted voices).

$$0 \leq |playing_i| \leq \text{voices} \qquad \text{for} \quad 0 \leq i < \text{barLength} * \text{quant} \tag{5.2}$$

To do so we use the cardinality constraint of a set variable which enforce the minimum and maximum number of elements contained by a set variable.

```
cardinality(*this, playing, 0, voices);
```

- **Minimum/maximum pushed notes** : Set the minimum and maximum number of notes that can start playing at the same time. As the push array represent the time at which we start playing a note, we ensure that the elements of this array have a cardinality below the maximum value (denoted MaxPushed) and above the minimum value (denoted minPushed).

$$minPushed \leq |Push_i| \leq maxPushed \qquad \text{for} \quad 0 \leq i < \text{barLength} * \text{quant} \tag{5.3}$$

To achieve that using Gecode we have to create an array of int variable representing the cardinality of each elements of push

```
IntVarArray notes_array(*this, barLength * quant, 0,
   max_pitch);
for(int i = 0; i < barLength  * quant; i++)
    cardinality(*this, push[i], notes_array[i]);
```

Once we have this array we can add some relation constraint with reification to take account of the element of push that should not add any notes and therefore should keep a cardinality of 0 despite the minimum pushed notes value.

```
for(int i = 0; i < barLength * quant; i++){
    Boolvar isEmpty(*this, 0, 1);
    BoolVar isNotEmpty(*this, 0, 1);
    rel(*this, isEmpty == (notes_array[i] == 0));
    rel(*this, isNotEmpty == (minPushed <= notes_array[
   i] <= maxPushed));
}
```

- **Minimum/Maximum notes** : Determines the total number of notes that are in the melody at the exit of this block, also counting the notes that were eventually added by sub blocks. This is helpful to control the total number of notes or simply to mute a section of the tree in a complex structure. to do that we have to limits between the minimum value minNotes and the maximum value maxNotes the sum of the cardinality of every elements of the push array.

$$minNotes \leq \Sigma_{i=0}^{n} |push_i| \leq maxNotes \qquad \text{for} \quad n = \text{barLength} * \text{quant} \tag{5.4}$$

In practice we take advantage of the array of cardinality notes_array we have created just before to get the sum of the cardinality and then use simple relation constraints.

```
1            IntVar notes(*this, 0, barLength * quant *
   max_pitch);
2            rel(*this, notes == expr(*this, sum(
   notes_array)));
3
4            rel(*this, notes >= minNotes) ;
5            rel(*this, notes <= maxNotes) ;
```

- **Minimum/Maximum added notes** : using these options allows to control the number of notes added by this specific bloc in addition to the one eventually coming from any sub blocks, if the maximum is set to zero this block only return notes coming from its sub blocks. To do that we have to set a maximum and minimum value to the cardinality of the push variable from this block, which depends on the value selected by the user and the cardinality of the sub-blocks, but we can't simply take the sum of the cardinality of the push variable of the sub blocks as a limit because two sub blocks can generate the same value adding one to the cardinality of the final push variable but two to the sum of the cardinalities of the push variables of the sub blocks. To get around this problem we create a new variable (pushUnion) which is the union of the push variables of the sub blocks and use its cardinality plus the minimum or maximum value chosen as the limits for the considered block push variable cardinality.
  Below, in the mathematical representation, $subPush_{i,j}$ represent the element at index j in the push variable of sub block i, maxAddedNotes is the maximum number of notes chosen by the user to be added by this block and minAddedNotes is the minimum number of notes to be added.

$$\begin{cases} pushUnion_j = \cup_i^n \, subPush_{i,j} \\ |pushUnion_j| + minAddedNotes \leq |push_j| \leq |pushUnion_j| + maxAddedNotes \end{cases}$$

$$\text{for} \quad \begin{cases} \text{n the number of sub blocks} \\ 0 \leq j < \text{barLength * quant} \end{cases} \tag{5.5}$$

  The implementation add the allPush variable which contains at index i a SetVarArray containing all the set at index i in the push variable of each sub blocks. To create pushUnion we use an union relation over the element of allPush, we then create an array of cardinality of the elements of pushUnion to set the constraint using a relation constraints between the cardinality of each elements of push and the corresponding elements of pushUnion.

```
1 SetVarArray allPush[barLength * quant];
2 SetVarArray pushUnion(*this, barLength * quant, 0,
     max_pitch, 0, max_pitch);
```

```
IntVarArray pushUnion_card(*this, barLength * quant, 0,
    127);
//n is the number of sub blocks
for(int j = 0; j < barLength * quant; j++){
    allPush[j] = new SetVarArray(*this, n, 0, max_pitch
   , 0, max_pitch);
    for(int i = 0; i < n; i++){
        allPush[j][i] = subPush[i][j]
    }
    rel(*this, pushUnion[j], SRT_EQ, expr(*this,
   setunion(allPush[j])));
    cardinality(*this, pushUnion[j], pushUnion_card[j])
   ;
    //Getting cardinality of the push element
    IntVar pushCard(*this, 0, 127);
    cardinality(*this, push[j], pushCard);
    rel(*this, pushUnion_card + minAddedNotes <=
   pushCard <= pushUnion_card  + maxAddedNotes);
}
```

5.4.2 Rhythm constraints

- **Minimum note length** : Set the minimum length of all the notes being played. This is done by ensuring that a note appearing at an index i of the push array does not appear in the pull array before index i + minLength is reached, minLength being the value chosen by the user.

$$Push_i \cap Pull_{i+j} = \emptyset \qquad \text{for} \quad \begin{cases} 0 < j < \text{minLength} \\ 0 \leq i < \text{barLength * quant} \end{cases} \tag{5.6}$$

  In order to do that we use a disjoint relation between each push element and the minLength following pull elements that should not contain the pushed notes.

```
for (int i = 0; i <= barLength*quant; i++){
    for (int j = 1; j < minlength && i+j <= barLength*
   quant ; j++){
       rel(*this, pull[i+j] || push[i]);
    }
}
```

- **Maximum note length** : Set the maximum length of all the notes being played to a value chosen by the user (denoted maxLength). this is equivalent to imposing that a note that starts to play at time i, which mean it appears at index i in the push array, is pulled before time i + maxLength, meaning it does appear in the pull array before index i + maxLength.

$$push_i \in \cup_{j=0}^{maxLength} pull_{i+j} \qquad \text{for} \quad 0 < i < \text{barLength * quant - maxLength} \tag{5.7}$$

To do that we create a set variable containing the union of all the sets contained in pull from index i to i + maxLength, then we use a relation constraint to ensure that the set at index i of push is a subset of this union.

```
for(int i = 0; i < barLength * quant - maxLength; i++){
    SetVarArray l_pull(*this, maxLength, 0, max_pitch,
   0, max_pitch) ;
    SetVar l_pull_union(*this, 0, max_pitch, 0,
   max_pitch);
    for(int j = 0; j < maxLength; j++){
        rel(*this, l_pull[j] == pull[i + j]);
    }
    rel(*this, SOT_UNION, l_pull, l_pull_union);
    rel(*this, push[i], SRT_SUB, l_pull_union);
}

```

- **Quantification** : Select the smallest beat fraction allowed in the melody. The easy way to do that would be to change the quantification factor (quant) we have been using in many constraint shown until now, but in doing so we would lose compatibility between blocks with different quantification as some variables would represent time that don't exist in variables with other quantification. To get around this problem we use the same global quantification through all the blocks and we enforce the set that are not on time fraction compatible with the quantification chosen by the user (denoted userQuant) to be empty.

$$Push_i = Pull_i = \emptyset \qquad \text{for} \quad \begin{cases} 0 \leq i < \text{barLength} * \text{quant} \\ \text{i mod(userQuant)} \neq 0 \end{cases} \tag{5.8}$$

In Gecode this is done with a simple empty domain constraint on each set variables not accepted by the quantification.

```
for(int i = 0; i < barLength * quant; i++){
    if(j % userQuant != 0){
        dom(*this, push[i], STR_EQ, IntSet::Empty);
        dom(*this, pull[i], STR_EQ, IntSet::Empty);
    }
}
```

- **Note repartition :** Constrains the distribution of the notes throughout the measures. The user defines how the note are spread with a distribution percentage. A 0% constrains to play all the notes simultaneously. Whereas 100% constrains to play the notes as distributed as possible across the measures.

```
if (percentDist == 0.0){
```

```
    SetVar unionPush(*this, IntSet::empty, IntSet(0,
   bars*quantification), 0, bars*quantification+1);
    rel(*this, SOT_UNION, pushMap, unionPush);
    cardinality(*this, unionPush, 1, 1);
}else{
    int pushEvery = int(minLength/percentDist);
    for(int i=0;  i<bars*quantification; i++){
      if(i%pushEvery==0){
          rel(*this, cardinality(push[i])>=1);
      }else{
          cardinality(*this, push[i], IRT_EQ, 0);
      }
    }
}
```

- **Rhythm repetition** : Forces some rhythm patterns of a length chosen by the user to be repeated throughout the melody, we consider that the rhythm is defined by the cardinality of a sequence of set variables, so repeating rhythm is equivalent to repeating the same sequence of cardinality as much as needed on the duration of the melody. In this case the user choose a value of length (denoted len) that represent the length of the rhythm pattern that will be repeated throughout the melody as described right after.

$$|push_i| = |push_{i+(j*len)}| \qquad \text{for} \quad \begin{cases} 0 \leq i < \text{len} \\ 1 \leq j < \frac{\text{barLength * quant}}{\text{len}} - i \end{cases} \tag{5.9}$$

  In practice we create an array of int variables the same size as push, each elements being constrained to the cardinality of the push element at the same index. We then post equality relations between elements of this array at fixed intervals.

```
IntVarArray notes_array(*this, barLength * quant, 0,
   127);
for(int i = 0; i < barLength * quant; i++){
    cardinality(*this, push[i], notes_array[i]);
}
for(int i = 0; i < len; i++){
    for(int j = 1; j < barLength * quant && i + (j *
   len) < barLength * quant){
        rel(*this, notes_array[i] == notes_array[i + (j
    * len)]) ;
    }
}
```

- **Pause quantity** : Fixes the number of rest in the melody, a rest is equivalent to an empty set in the array of set variables Playing. In this case the user can fixes the quantity of rests (denoted pauseQuantity) they

desire, from none to a melody full of pause.

$$\begin{cases} G = \{i \; s.t. \; Playing_i = \emptyset\} \\ |G| = pauseQuantity \end{cases} \quad \text{for} \quad 0 \leq i < \text{barLength} * \text{quant} \tag{5.10}$$

We implemented this by first creating an array of int variables each constrained to the cardinality of elements of the push array which are not already set to empty by the quantification constraint as those are not necessarily rest but might just be time during which notes are being held down and played. We use this array with a sequence constraint to ensure that there is the right number of zero value, in other words the right number of empty set variables. You might be wondering why we don't use the playing array as suggested above, that's because playing is not constrained by the quantification constraint, meaning that using it to enforce the number of rests might lead to smaller rest than the quantification should accept and they might be on unwanted beats.

```
1  //q-push is the array of element of push compatible
       with the quantification of the user
2  SetVarArray q_push(*this, barLength * userQuant, 0,
       max_pitch, 0, max_pitch);
3  for(int i = 0; i < barLength * userQuant; i++){
4      rel(*this, q_push[i], SRT_EQ, push[i * \frac{quant
       }{userQuant}]);
5  }
6  IntVarArray q_push_card(*this, barLength * userQuant,
       0, 127);
7  for(int i = 0; i < barLength * userQuant; i++){
8      cardinality(*this, q_push[i], q_push_card[i]);
9  }
10 //number of pause to add, pauseQuantity is in
       percentage and goes from 1 to 100
11 int pause = pauseQuantity * (barLength * userQuant) /
       100 ;
12 count(this, q_push_card, 0, IRT_EQ, pause) ;
```

- **Pause repartition** : this enforces that any sub sequence of a given length from the melody contains at least one rest, the length (denoted pauseLength) is a value chosen by the user. As previously mentioned a rest is equivalent to an empty set in the playing array, so we want any sub-list of length pauseLength from the playing array to contains an empty set.

$$\begin{cases} G = \{playing_i \; s.t. \; k \leq i < pauseLength + k\} \\ \emptyset \in G \end{cases} \quad \text{for} \quad 0 \leq k < (\text{barLength} * \text{quant}) - \text{p} \tag{5.11}$$

To do that we reuse the array of cardinality created just before but this time we use a sequence constraint which post that the number of repetition

modes	intervals
Ionian (major)	2 2 1 2 2 2 1
Dorian	2 1 2 2 2 1 2
Phrygian	1 2 2 2 1 2 2
Lydian	2 2 2 1 2 2 1
Mixolydian	2 2 1 2 2 1 2
Aeolian (natural minor)	2 1 2 2 1 2 2
Locrian	1 2 2 1 2 2 2
Harmonic minor	2 1 2 2 1 3 1
Pentatonic	2 2 3 2 3
Chromatic	1 1 1 1 1 1 1 1 1 1 1 1

Figure 5.12: scale modes and the associated intervals between notes

of a given sequence in every sub-sequence of a given length must be between two chosen values.

```
int length = ((barLength * userQuant) * (192 -
    pauseLength)) / 192 ;
//sub-sequence of q_push_card of length length must
    have at least 1 and maximum length zero.
sequence(*this, q_push_card, IntSet(0, 0), length, 1,
    length);
```

5.4.3 Pitch constraints

- **Key and mode selection** : Determine in what key and mode the melody should be written. Those are two different buttons in the interface but are strongly linked when creating the constraints. Following a key and mode is simply ensuring that the notes played are taken from a specific sets of acceptable pitch (denoted scaleSet) build according to those two information. in fig 5.12 is the list of possible modes and the associated list of intervals between notes, a value of 1 correspond to a semitone, 2 to a tone and so on.

$$Push_i \subset \text{scaleSet} \qquad \text{for} \quad 0 \leq i < \text{barLength} * \text{quant} \tag{5.12}$$

In Gecode this translate to a subset relation constraint between each element of push and the scaleSet.

```
for(int i = 0; i < barLength*quant; i++){
    rel(*this, push[i] <= scaleSet);
}
```

When selecting a key but no mode we assume that the last one is major, but when selecting a mode but no key we can't really assume anything, in this case we create the scaleSet for every key with this mode and ensure

modes	intervals
Major	4 3 5
Minor	3 4 5
Augmented	4 4 4
Diminished	3 3 6
Major 7	4 3 4 1
Minor 7	3 4 3 2
Dominant 7	4 3 3 2
Minor 7 flat 5	3 3 4 2
Diminished 7	3 3 3 3
Minor-major 7	3 4 4 1

Figure 5.13: chord modes and the associated intervals between notes

that the melody is composed of notes contained in at list one of the scaleSet. To do that in Gecode we use reification and enforce that at least one of the boolean used for reification is True.

```
//12 possible keys identified by a int in [0, 12[
reification BoolVarArray(*this, 12, 0, 1);
for(int key = 0; key < 12; key++){
    scaleset = build_scaleset(key, mode);
    for(int i = 0; i < barLength*quant; i++){
        rel(*this, push[i], SRT_SUB, scaleSet,
   reification[key], RM_IMP);
    }
}
//1 is equivalent to True
rel(*this, BOT_OR, reification, 1)
```

- **Chord key and quality** : This constraint is identical to the previous one in the way it works, except that the sets of acceptable pitch is build differently. In fig 5.13 is the list of possible chord modes and their associated list of pitch intervals, obviously those values lead to different results than using the values in 5.12. Each combination of mode and key as multiple sets of possible pitch, representing the same chord on different octave.

- **Minimum/maximum pitch** : set the respectively the minimum and maximum pitch value any note from this block can have. Once again for compatibility between blocks reasons we can't simply change the range of the set variable when creating them, so we have to make each set variable in push a subset of a full domain between the minimum pitch (minPitch) and the maximum pitch (maxPitch).

$$Push_i \subset [minPitch, maxPitch] \qquad \text{for} \quad 0 \leq i < \text{barLength} * \text{quant} \tag{5.13}$$

Concretely this translate to a relation constraint to enforce that each element of push is a subset of a domain.

```
for(int i = 0; i < barLength * quant; i++)
    dom(*this, push[i], SRT_SUB, minPitch, maxPitch);
}
```

- **Note repetition and repetition type** : determine approximately how much the same notes should be repeated throughout the melody. The user use slider to set a percentage (repeatPercentage) of the notes that should be a repetition of another note, 0% means all the notes will be different from each other and 100% that the melody only use one note. in the mathematical form we take a random subset of the possible index, the size of this subset is determined by the percentage chosen by the user, we then ensure that the notes at index contained in the subset are repetition of another note, while note at index not contained in the subset are not a repetition.

$$\begin{cases} Push_i \in Push \setminus Push_i \\ Push_j \notin Push \setminus Push_j \end{cases} \quad \text{for} \quad \begin{cases} i \in G \\ j \notin G \\ G \subseteq [0, \text{barLength * quant}] \\ |G| = \frac{\text{repeatPercentage * barLength * quant}}{100} \end{cases} \tag{5.14}$$

In practice, we also use the repetition type option, which determine how to enforce the repetition constraint among three strategies. The three strategies give different results and have different impact on the solver performance. For the random selection we shuffle the list of possible notes index and randomly use relation constraint to make set variable equivalent or disjoint two by two depending on the percentage requested by the user.

```
//range create an array from 0 to barLength * quant by
   step of userQuant
int[] index = shuffle(range(0, barLength * quant,
   userQuant)) ;
for(int i = 0; i < (sizeof(index)/sizeof(index[0])) -
   1; i++){
    if (rand() % 100 < repeatPercentage)
        rel(*this, push[index[i]] == push[index[i +
   1]]);
    else
        rel(*this, push[index[i]] || push[index[i +
   1]]);
}
```

For the soft and hard option we use the cardinality of the pushMap array, which represent how much times a given appear in the melody, below is the implementation of the pushMap_card array containing the cardinality of every pushMap elements.

```
IntVarArray pushMap_card(*this, max_pitch, 0, barLength
    * quant);
```

```
for(int i = 0; i < max_pitch; i++){
    cardinality(*this, pushMap[i], pushMap_card[i]);
}
```

Then the soft option enforces that a certain percentage of the notes are not in the melody, thus forcing remaining notes to repeat themselves. This is done with a count constraint that ensure that a minimum number of the pushMap_card value are set to zero.

```
//repeatPercentage goes from 0 to 100
int c = repeatPercentage * (max_pitch - 1) / 100 ;
count(*this, pushMap_card, 0, IRT_GQ, c)
```

The hard option simply make sure that one note repeat at least a minimum number of times in the whole melody. Once again this is done using a count constraint.

```
int repetition = percent * max-repetition / 100 ;
count(*this, pushMap_card, repetition, IRT_GQ, 1);
```

You have probably noticed that these 3 methods have a very different interpretation of what note repetition is, either repeating one note a lot of times, ensuring that multiple notes are repeated a few times or an in-between that let the choice to the solver. This is the objective of this option, to give more possibilities to the composer.

- **Pitch direction** : set how the melody should evolve through time, either with increasing or decreasing pitch. Increasing pitch is equivalent to making sure that the minimum value of all the sets after index i are larger than the minimum value of the set at index i, for decreasing pitch the idea is similar except that we take the maximum values that have to be smaller further in the list than the maximum value at index i.
  Increasing pitch :

$$min(Push_i) \leq min(Push_j) \qquad \text{for} \quad \begin{cases} 0 \leq i < \text{barLength * quant} \\ i < j < \text{barLength * quant} \end{cases} \tag{5.15}$$

  Decreasing pitch :

$$max(Push_i) \geq max(Push_j) \qquad \text{for} \quad \begin{cases} 0 \leq i < \text{barLength * quant} \\ i < j < \text{barLength * quant} \end{cases} \tag{5.16}$$

  These first constraints make sure that the melody globally goes up or down in pitch, but if we want to be more constraining we can use strictly increasing and strictly decreasing constraints that respectively enforce that after playing a note, all the following notes are of greater pitch or of smaller pitch.

Strictly increasing pitch :

$$max(Push_i) < min(Push_j) \qquad \text{for} \quad \begin{cases} 0 \leq i < \text{barLength} * \text{quant} \\ i < j < \text{barLength} * \text{quant} \end{cases} \tag{5.17}$$

Strictly decreasing pitch :

$$min(Push_i) > max(Push_j) \qquad \text{for} \quad \begin{cases} 0 \leq i < \text{barLength} * \text{quant} \\ i < j < \text{barLength} * \text{quant} \end{cases} \tag{5.18}$$

This is done using relation constraints with reification. We need reification to avoid adding the constraint on empty set variable as they don't add notes. First we create the boolean variables isPlayed linked to push that hold true if the element of push at the same index contains a note and false if this element is empty.

```
//[0, max_pitch] is the range of acceptable pitch
SetVar allPlayed(*this, IntSet::empty, 0, max_pitch, 0,
    max_pitch);
BoolVarArray isPlayed(*this, max_pitch, 0, 1);
rel(*this, SOT_UNION, push, allPlayed);
channel(*this,  isPlayed, allPlayed);
```

Now we can create the four constraints.

```
// increasing pitch
for (int i = 0; i < max_pitch - 1; i++){
    for (int j=i+1; j< max_pitch; j++){
      rel(*this, (isPlayed[i] && isPlayed[j]) >> (min(
   push[i]) <= min(push[j])));
    }
}

//decreasing pitch
for (int i = 0; i < max_pitch - 1; i++){
    for (int j = i+1; j < max_pitch; j++){
      rel(*this, (isPlayed[i] && isPlayed[j]) >> (max(
   push[i]) >= max(push[j])));
    }
}

// strictly increasing pitch
for (int i = 0; i < max_pitch-1; i++){
    for (int j = i+1; j < max_pitch; j++){
  rel(*this, (isPlayed[i] && isPlayed[j]) >> (max(push[
   i]) < min(push[j])));
    }
}
```

```
21
22 // strictly decreasing pitch
23 for (int i = 0; i < max_pitch-1; i++){
24     for (int j = i+1; j < max_pitch; j++){
25       rel(*this, (isPlayed[i] && isPlayed[j]) >> (min(
   push[i]) > max(push[j])));
26     }
27 }
```

- **Golomb ruler size** : set as much note as chosen by the user at the beginning of the melody to form a Golomb ruler. First off a Golomb ruler is a set of marks at integer positions along a ruler such that no two pairs of marks are the same distance apart [26]. In other words this constraint allow to create melody with varying intervals between notes. this constraint works only when playing one note at the time, and we have that the difference between any two notes is unique among all the other possible differences.

$$push_i - push_j \neq push_k - push_l \qquad \text{for} \quad \begin{cases} 0 \leq i < \text{barLength} * \text{quant} \\ i \leq j < \text{barLength} * \text{quant} \\ 0 \leq k < \text{barLength} * \text{quant} \\ k \leq l < \text{barLength} * \text{quant} \end{cases} \tag{5.19}$$

  In Gecode we create an array of int variable and we constrain each element to the difference of two element of push, we then enforce that all elements of this array are different.

```
1 //size of the difference array based on n, the size
     chosen by the user
2 const int size_d = (n*n-n)/2;
3
4 // Array of differences
5 IntVarArgs d(*this, n_d, 0, max_pitch);
6
7 // Setup difference constraints
8 for (int k=0, i=0; i<n-1; i++)
9     for (int j=i+1; j<n; j++, k++)
10         rel(*this, d[k] = expr(*this, m[j]-m[i]));
11
12 distinct(*this, d);
```

5.5 Branch and bound

One major upgrade of Melodizer 2.0 is the addition of the branch and bound search algorithm which allows to find way more interesting solutions. You can find more information on the branch and bound in section 2.4.6, in this section we discuss its implementation in Melodizer 2.0 using Gil. Adding the BAB algorithm

in itself is an easy task, as easy as adding any other search engine, the difficulty appears when trying to use the full capacity of branch and bound.

When using the branch and bound search algorithm we can add new constraints every time we look for a new solution in order to conduct our results in a certain direction. We have decided to use this opportunity to ensure variety among the returned melody. More practically this is done by constraining that a certain percentage of values in the next solution have to be different from the values of the previous solution. This percentage has a huge influence on the results of the search so we let the user chose it. If the composer likes the first proposition they can set the percentage to a small value and get just a few modification, otherwise if they do not like it, they can set the percentage to 100% and get a totally different solution. With "Next" representing the array of notes in the next solution, "Prev" the array of notes in the previous solution and "diffPercentage" the percentage of difference chosen by the user, the mathematical representation is :

$$Next_i \neq Prev_i \qquad \text{for} \quad \begin{cases} i \in G \\ G \subseteq [0, \text{barLength} * \text{quant}] \\ |G| = \frac{\text{diffPercentage} * \text{barLength} * \text{quant}}{100} \end{cases} \tag{5.20}$$

In Gecode we add these constraints by using the constrain function, this function is called inside the space of the next solution and take as argument the space of the previous best solution, having access to these two spaces allow us to add constraint between the variables of the two solutions. Below is the code we can use in Gecode to get solution with a percentage of varying elements using an inequality relation between equivalent elements of the two spaces. Note that we constraint the inequality only if the previous solution variable is not empty, this is because the quantification chosen by the user force some sets to be empty and this doesn't change between solutions, an inequality relation between two set constrained to be empty would immediately lead to no solutions, which wouldn't be a good thing.

```
virtual void constrain(const Space& _b) {
    const Melody& b = static_cast<const Melody&>(_b);
    for(int i = 0; i <= barLength * quant; i=i+1){
        if((rand()%100) < diffPercentage){
            //get the previous solution variables
          SetVar tmp(b.push[i]);
          //inequality relation only if the set is not
   empty
          rel(*this, (tmp != IntSet::empty) >> (push[i] !=
   tmp) );
        }
    }
}
```

5.6 Solver

The solver is embodied by the search object in Open Music as it is the interface that allow the user to interact with the solving algorithm. The search object has to be connected to a block or a tree structure of blocks which represents a CSP as described in subsection 5.3 to find solution to this problem.

Below is the code used by Melodizer to create a new search engine object, we use the Gecode option to stop the search after a certain amount of time if no solution have been found to prevent search taking too much time. If the timeout is reached and the search is stopped, the user is informed that no solution have been found. If needed the search can also be stopped by the click of a button in the search object interface.

```
(setq tstop (gil::t-stop)); create the time stop object
(gil::time-stop-init tstop 500); initialize it (time is
   expressed in ms)

;search options
(setq sopts (gil::search-opts)); create the search options
   object
(gil::init-search-opts sopts); initialize it
(gil::set-n-threads sopts 1); set the number of threads to
   be used during the search (default is 1, 0 means as many
    as available)
(gil::set-time-stop sopts tstop); set the timestop object
   to stop the search if it takes too long

; search engine
(setq se (gil::search-engine sp (gil::opts sopts) gil::BAB)
   )
```

The search is done in a separated thread than the rest of the execution to avoid blocking the execution of Open Music during the search. The thread is created using mp:process-run-function in the Lisp code of Melodizer when the next solution is requested by the user. The code used can be found below, new-search-next is the call to the function that interact with Gil to get a solution and return an Open Music object representing the melody found by the solver.

```
(mp:process-run-function ; start a new thread for the
   execution of the next method
    "next thread" ; name of the thread, not necessary but
   useful for debugging
    nil ; process initialization keywords, not needed here
    (lambda () ; function to call
        (setf (solution (om::object editor)) (
   new-search-next (result (om::object editor)) (om::object
    editor)))
```

```
        (om::openeditorframe ; open a voice window displaying the solution
            (om::omNG-make-new-instance (solution (om::object editor)) "current solution")
        )
    )
)
```

As explained in previous section we use the branch and bound search algorithm and the user can choose a percentage of modification they want to see in the next solutions, but that's not all, one important element of the searching algorithm is the branching strategy, the next subsection explains the options available to the user and then we compare the result of the solver for different inputs.

5.6.1 Branching heuristics

The branching strategy is an important aspect as it determines our path through the search tree and thus the order of the solutions provided by the search engine. Our solver branch over the two arrays of set variables push and pull as they are the main variables describing the final solution, all other variables are redundant but helpful for some constraints.

How to branch

As the solver can be dealing we some complex mix of constraints over a wide range of variables we want to favor performance in order to get a result in a decent amount of time. We have to be aware of how the branch function works in Gecode, for example if we branch on the push and pull arrays as follows:

```
branch(*this, push, SET_VAR_SIZE_MIN(), SET_VAL_RND_INC(r1));
branch(*this, pull, SET_VAR_SIZE_MIN(), SET_VAL_RND_INC(r2));
```

It might seem that we have chosen a good variable selection heuristic since it follows the first-fail principle. However, it is not a very efficient branching strategy since it firstly branches through all the push variable array before branching the pull variable array. There is multiple ways we can create better branching that gives different and interesting results. To give more diversity of solutions to the user we have decided to implement 3 different variables branching. The first one called "Top Down" first branch on the push and pull variables of the root block, then on the same variables of its child blocks and so on. Going from the top, the root, to the leaf of the tree structure. To do so we have access to push_list and pull_list, respectively the list of push and pull variables of all the blocks in the structure, variables of the root being at the last position in the lists.

```
for(int i = push_list_size - 1; i >= 0; i--){
    SetVarArgs pushPull ;
```

```
    pushPull < push_list[i] ;
    pushPull < pull_list[i] ;
    branch(*this, pushPull, SET_VAR_SIZE_MIN(),
   SET_VAL_RND_INC(Rnd(3U))) ;
}
```

The second option is called "Full" as it branches on all the push and pull elements of all the blocks in the tree structure at the same time, this is implemented in the code below using the same variables as before.

```
SetVarArgs fullPush ;
SetVarArgs fullPull ;
for(int i = 0; i < push_list_size; i++){
    fullPush << push_list[i] ;
    fullPull << pull_list[i] ;
}
//concatenate all push and pull variables into fullPush
fullPush << fullPull ;
branch(*this, fullPush, SET_VAR_SIZE_MIN(), SET_VAL_RND_INC
   (Rnd(3U))) ;
```

In those two example we have used SET_VAR_SIZE_MIN() as the variable selection heuristic and SET_VAL_RND_INC() as the value selection heuristic. We have decided to branch on the variable with the smallest domain because it's an easy way to set variables faster and get a solution in a decent amount of time, then choosing a random value seemed logic as there is not really a mathematically better solution than another when dealing with music, so randomness is a great way to give a chance to every possibilities. In every examples we use Rnd() to set a random number generator with a specific seed, this is useful to guarantee reproducibility of the research.
This said the final option for branching is similar to the first one "top down" with the difference that we use a random variable selection heuristic this time, hence its name "Top down random". This method can be less efficient, but it can also yield more interesting and unpredictable results, which makes it an attractive method.

```
for(int i = push_list_size - 1; i >= 0; i--){
    SetVarArgs pushPull ;
    pushPull << push_list[i] ;
    pushPull << pull_list[i] ;
    branch(*this, pushPull, SET_VAR_RND(Rnd(1U)),
   SET_VAL_RND_INC(Rnd(3U))) ;
}
```

Comparison

Now that we have shown our three branching heuristics let's compare them. In figure 5.14 are the execution time to find the next solution of the different branching for different difficulty of constraints and blocks structure, by that we

difficulty	branching	1	2	3	4	5	Mean
Easy	Top down	242	90	147	92	163	147
Easy	Full	253	148	103	96	140	148
Easy	Top down random	265	146	163	110	107	158
Medium	Top down	290	153	155	141	139	176
Medium	Full	261	319	232	250	246	262
Medium	Top down random	202	149	121	139	143	151
Hard	Top down	265	231	304	233	239	254
Hard	Full	235	185	164	174	180	188
Hard	Top down random	TO	TO	TO	TO	TO	TO

Figure 5.14: Comparison of execution time of different branching

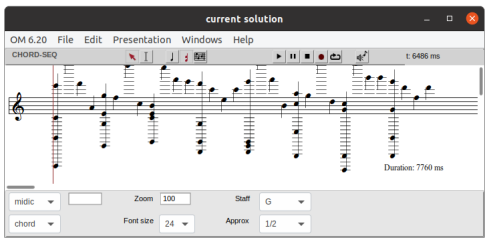


Figure 5.15: Second solutions found with "Top down" strategy

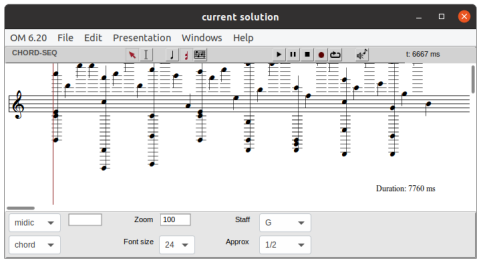


Figure 5.16: Second solution found with "Full" strategy

mean that more blocks and more constraints is harder to solve, some constraints like pitch direction are also harder than other, like quantification. The time is expressed in ms and was computed on 5 successive solutions for each case. In this table we can see that the "top down" and "full" strategies have quite similar result, with slightly better performance for the "full" option on a hard scenario. On the other hand the "random" strategy timed out with difficult constraints, which was expected, the fact that it has quite similar execution time as the two other in easier structure is due to a bit of luck and quite easy constraint being used, adding a constraint like "pitch direction" almost always results in a time out of the random branching. You can also find in figure 5.15 and 5.16 a comparison of the second solution found by the two branching strategy "Top down" and "Full" in the hard scenario, this is not meant to compare the quality of the two solutions, but more to show the differences between them.

5.7 Implementation structure

The implementation of Melodizer 2.0 is split in 4 files to make the code easier to read. In figure 5.17 is a diagram of the relation between the files, an arrow going from file A to file B show that a function from B is called inside A. Below is a description of each file as well as the list of the main features they each contain. The complete code for all these files can be found in Appendix D.

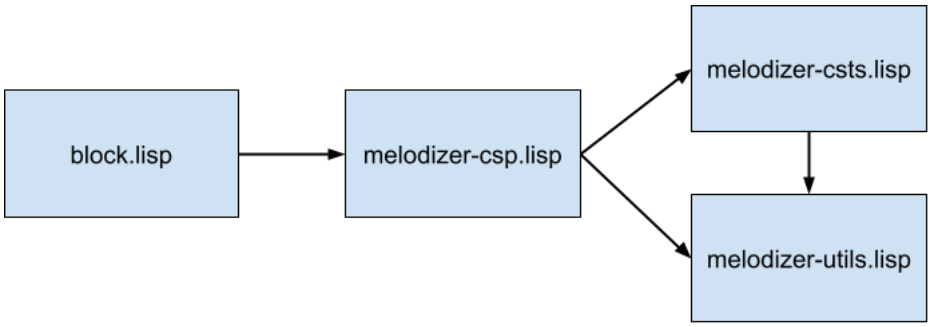


Figure 5.17: File structure of Melodizer 2.0

5.7.1 block.lisp

This file contains the code for the two objects of the library: the block and search objects. It includes the code for the interface of both objects and the call to all the needed functions from other files. The main parts are :

- Declaration of the block class with its attributes.
- Declaration of the search class with its attributes.
- Creation of the 3 panels composing the interface of the block object.
- Creation of the single panel composing the interface of the search object.
- Creation of the thread for searching the next solution

5.7.2 melodizer-csp.lisp

This file is the main part of creating the csp, it creates all the variables and post constraints or call function to post them. The main functions are :

- *new-melodizer* which creates the csp by setting variables, posting the constraints and creating the search engine.
- *get-sub-block-values* which adds the eventual constraints brought by sub-blocks.
- *post-optional-constraints* which posts optional constraints according to the user's will.
- *new-search-next* which looks for the next solution found by the solver and translates this solution to musical representation.
- *stopped-or-ended* which allows the user to stop the search early

5.7.3 melodizer-csts.lisp

This files is used to post more complex constraints that would have congest other files, or simply constraints that can be used in multiple situations to avoid repetition and make them easier to call. the main constraints are :

- *scale-follow* and *scale-follow-reify* that make all elements of a SetVarArray subset of a given set, used to follow a specific scale or chords.
- *chordprog-follow* which is similar to the previous but the SetVarArray elements are subset of different sets for different index, used to follow a chord progression.
- *pitch-range* which limits the maximum and minimum possible pitch.
- *note-min-length* which constraint the minimum length of the notes.
- *chords-rhythm* which forces some beats to contain a chords and other a single note
- *chords-length* which constraints the minimum length of a chord, obviously has to be greater than the minimum length of a note.
- *num-added-note* which limits the number of notes added
- *set-quantification* which constrains the variables to use the quantification chose by the user by setting out of quantification beats to be empty
- *set-rhythm-repetition* which creates repetition in the rhythm of the melody
- *set-pause-quantity* which sets the quantity of rests in the melody.
- *set-pause-repartition* which distributes the rest throughout the melody according to some distribution value.
- 4 pitch direction functions that sets the pitch to be either increasing, strictly increasing, decreasing, strictly decreasing.
- *golomb-rule* which sets a certain number of the notes pitch to be acceptable value for Golomb ruler
- *repeat-note* which makes sure that a certain percentage of the note pitch are duplicated throughout the melody.

5.7.4 melodizer-utils.lisp

This file provides multiple functions useful to manipulate some data, as there is quite a lot of function in there we only give the list of the most important ones.

- Conversion function to change MIDI value to MIDICent, or keys Letter to their pitch values, etc

- List manipulation function to get the maximum/minimum values of list, make a list from a range, etc
- Functions to change from the Gecode variable to Open Music Object and vice versa.
- Function to build set that is used to follow a scale, a chord or a chord progression.
- Small utility function of various usage.

Chapter 6

User Manual

Melodizer's 2.0 presents two different objects. On the one hand, the Block object allows to select the constraints for a specific part or for the whole piece. While, on the other hand, the Search object, whose name is is quite self-explanatory, looks for solutions with some characteristic defined by the user.

Search objects receive through his second inlet a Block. Each musical phrase can have distinct musical ideas inside. This is why we introduced the possibility for the Block objects to also receive a list of Blocks, through his second inlet as well. Some real musical scenarios are provided as examples in chapter 7.

6.1 Block object

When we create a Block object inside an OpenMusic patch, the box in figure 6.1 appears. We can observe that it has four inlets and four outlets. The inlets represent the setter of the object while the outlets are the getters. The first inlet/outlet, also called 'self', represents the object itself. The 'self' outlet is used to communicate to other Block objects or Search objects the computed object itself. The second and fourth inlets are respectively used to receive a list of Block objects and a list with its starting positions expressed in bars as shown in figure 6.2. The second inlet can also be simply used to receive a Block as shown in figure 6.3 . The third inlet can receive a voice object as illustrated in figure 6.4.

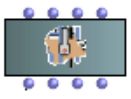


Figure 6.1: Block object's box

By double-clicking on the Block object box, its interface editor pops up as portrayed in figure 6.5. We can notice that there three panels, the block constraints panel (1), the time constraints panel (2) and the pitch constraints panel

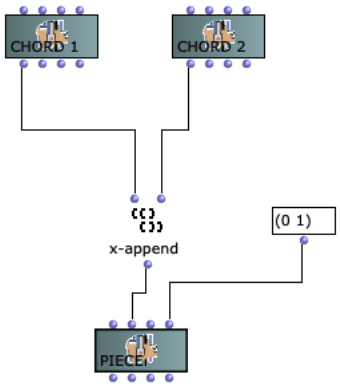


Figure 6.2: List of Block objects with its starting position connected to a Block object

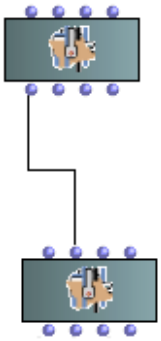


Figure 6.3: Block object connected to a Block object

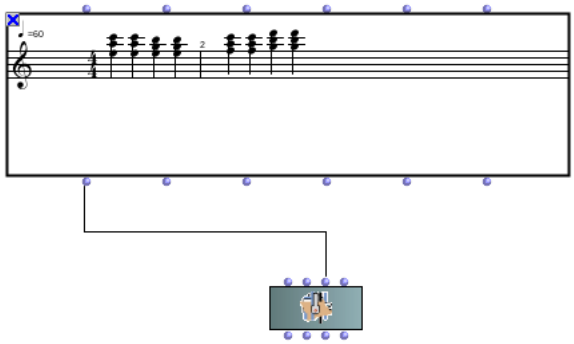


Figure 6.4: Voice object connected to a Block object

(3), each with its own different check-boxes, pop-up menus and sliders.

6.1.1 Block constraint panel

This panel serves for general constraints concerning the block. We present the different pop-up menus from top to bottom.

Bar length : The pop-up menu (4) allows the user to choose the number of measures that the block encompasses. Each bar contains four beats.

Voices : The pop-up menu (5) determines the number of notes that can be played simultaneously. For example, you want to model a harmonic part with seventh chords accompanied by a monophonic melody, you would set the voices to 5. For monophonic instruments representation such as the trumpet, you would set the voices to 1. And for a five stringed guitar you would set the voices to 5.

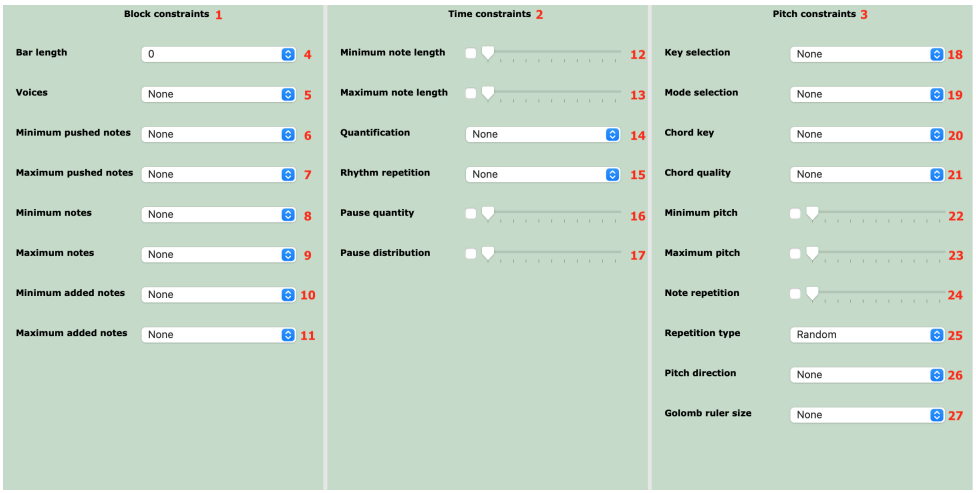


Figure 6.5: Block object's interface

Minimum/Maximum pushed notes : The pop-up menus (6) and (7) constrains the minimum and maximum notes that can be pushed. In other words, it defines the cardinality domain of non-empty pushed sets of notes. For example if we want to represent a guitar that strums all the strings at the same time we would set the minimum pushed notes to five. While, if we want to represent a guitar playing an arpeggio, then we set the maximum pushed notes to one.

Minimum/Maximum notes : The pop-up menus (8) and (9) defines the minimum and maximum number of notes that are played within the block. For instance, to represent a french horn since it is not an instrument built for speed the maximum notes is relatively low [9]. While, when composing for a violinist with Paganini's skills this maximum notes can be fairly high. It also depends on the sensation we want to transmit to the listener. For a quieter and peaceful sensation the minimum and maximum permitted notes has to be lower than for a fuller and denser sensation.

Minimum/Maximum added notes : The pop-up menus (10) and (11) allows the users to choose the minimum and maximum notes we want to add to the notes that are generated from attached sub-blocks or voice objects. These menus are only relevant if we have attached blocks or a voice object to the inlets of the block.

6.1.2 Time constraint panel :

This panel presents rhythmic constraints. We present the different pop-up menus, check-boxes and sliders from top to bottom.

Minimum/Maximum note length : Pop-up menus (12) and (13) constrain

the minimum and maximum length of the notes. For fast-paced melodies, the maximum note length can be reasonably short. Contrarily to slow-paced melodies where the minimum length notes can be fairly long. Once again, these fields depend on the emotion you want to convey.

Quantification : Pop-up menu (14) select the smallest beat fraction allowed in the block. This allows the user to play with varied rhythms and include duplets and triplets to your piece.

Rhythm repetition : The pop-up menu (15) allows the user to choose the length of the repeated rhythmic pattern throughout the block. This is a common practice in many musical genres where the rhythmic pattern is usually one or two long.

Pause quantity : By checking the box (16) the composer can decide the percentage of silences that the block has. If the slider is completely to the right the block will contain no notes. While, if it is completely to the left the block won't contain any silences.

Pause distribution : By checking the box (17) the composer can decide through the slider the minimal length between pauses. With the slider completely to the left the pauses occurs very frequently. Contrarily, with the slider completely to the right the pauses occurs rarely.

6.1.3 Pitch constraint panel

This panel provides melodic and harmonic related constraints. Again, we present the different pop-up menus, check-boxes and sliders from top to bottom.

Key and Mode selection : With the pop-up menus (18) and (19) the composer chooses the key and mode of the block. In other words, at chosen key, if they can decide to follow a major, minor or a pentatonic scale amongst other. Figure 6.6 all the available modes and its associated in semitones between two consecutive notes.

Chord key and quality : Pop-up menus (20) and (21) permits to choose the type of chord the composer wants to represent. Figure 6.7 all the available modes.

Minimum/Maximum pitch : The check-boxes and sliders (22) and (23) allow the users to define the minimum and maximum pitch of the notes. This can be practical for representing instruments ranges or voices tessitura for example.

Note repetition : By checking the box (24), the user chooses whether they

modes	intervals
Ionian (major)	2 2 1 2 2 2 1
Dorian	2 1 2 2 2 1 2
Phrygian	1 2 2 2 1 2 2
Lydian	2 2 2 1 2 2 1
Mixolydian	2 2 1 2 2 1 2
Aeolian (natural minor)	2 1 2 2 1 2 2
Locrian	1 2 2 1 2 2 2
Harmonic minor	2 1 2 2 1 3 1
Pentatonic	2 2 3 2 3
Chromatic	1 1 1 1 1 1 1 1 1 1 1 1

Figure 6.6: scale modes and the associated intervals between notes

modes	intervals
Major	4 3 5
Minor	3 4 5
Augmented	4 4 4
Diminished	3 3 6
Major 7	4 3 4 1
Minor 7	3 4 3 2
Dominant 7	4 3 3 2
Minor 7 flat 5	3 3 4 2
Diminished 7	3 3 3 3
Minor-major 7	3 4 4 1

Figure 6.7: chord modes and the associated intervals between notes

would rather like to have many repeated notes or not. A slider tuned completely to the left imposes all the notes to be different. While a slider tuned completely to the right generates a single-noted melody.

Repetition type : The pop-up menu (25) allows to select which type of repetition is going to be set with the value from Note repetition. It gives a choice between Random, Soft an Hard repetition. Random will randomly link two time slots and impose the same notes triggered at these moments. Soft will decrease the size of available notes to make them more repetitive. Hard will force a note to repeat itself as much as the Note repetition value forces it.

Pitch direction : This pop-up menu (26) allows to constrain the melody's direction. The composer can choose between an increasing, strictly increasing, decreasing or strictly decreasing pitch melody.

Golomb ruler size : The pop-up menu (27) allows you to choose how many notes constitute your Golomb ruler. A Golomb ruler is a list of integer positions

along a ruler such that no two pairs of marks are the same distance apart [26]. The distance between two integer position is here represented as the interval between two pitches. This constraint generates very creative solution since it benefits from a very uncommon tool in the musical field.

6.2 Search object

When we create a Search object, the box in figure 6.8 appears. We can observe that it has three inlets and four outlets. Again, the inlets represent the setter of the object while the outlets are the getters and the first inlet/outlet, also called 'self', represents the object itself. The second inlet is used to receive a Block object as shown in figure 6.9.

The Search object functionalities are :

- Select the tempo expressed in beats per minute in which the piece will be played.
- Start the search.
- Ask for the next solution.
- Stop the search if it is taking too much time.
- Select the Branching. Currently there are three types of branching : Top Down, Full and Top Down random. Top Down and Full are the most efficient while Top Down random can provide the most unexpected original solutions. Full is the most efficient when dealing with big musical pieces.
- Set the percentage of diversity from one solution to another. If you like the solution provided, set a low percentage. If not, set a high percentage in the hope of finding better solutions.

Figure 6.8: Search object's box

6.3 Connecting blocks to form a structured piece

Now that we have explained the Block and Search object's interface editor as well as what can be connected to its inlets and outlets, let's recapitulate how a whole piece can be structured by connecting Voice, Block and Search objects.

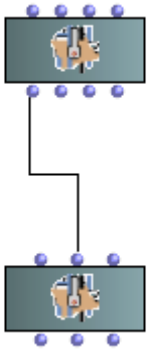


Figure 6.9: Block object connected to a Search object

First of all, a Voice object, a Block or a list of Blocks along with its starting positions can be passed to a Block object, that can be also passed to one or more Blocks. Then, a Block or a list of Blocks along with its starting positions can be passed to a Search object.

Let's represent this using a simple example where we connect the first outlet of Block object representing a chord to the second inlet of a Search object as illustrated in figure 6.9. We select the constraints available in the editor's interface to represent a C Major chord as shown in figure 6.10. And finally, we evaluate the Search object to obtain the solution 6.11. This example is explained in details in section 7.1.

Figure 6.10: Block object editor's interface to represent a C Major chord

Taking everything into consideration, we can deduce that there are several possible manners that a piece can be structured using this Block tree structure. In chapter 7 we provide many examples to clarify how this box interconnection can be applied to real musical scenarios.

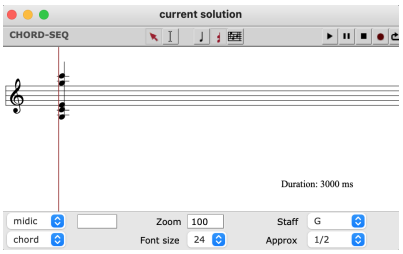


Figure 6.11: C Major chord solution

Chapter 7

Making music with Melodizer

This section is dedicated to the composition of music with Melodizer. It is separated in different scenarios, each showing the potential of an aspect of Melodizer. As the scenarios are organised in increasing level of complexity, we advise you to follow their order.

Melodizer uses two types of objects : Block and Search. The Block object is used to describe the piece of music you want to create using mathematical constraints. The Search object receives the described "mathemusical" problem from the Block structure and tries to find a solution to it.

7.1 Scenario 1 : Playing with a chord

7.1.1 Description

This very first scenario is a very easy introduction to the use of Melodizer. After this scenario, you will be able to use the chord constraints of Melodizer and combine them with other constraints to create interesting results.

What we want to create here is a melody around a chord with some interesting rhythmic and melodic patterns. All the musical solutions found in this scenario are available in this SoundCloud playlist.

7.1.2 Patch set up

This scenario uses the most basic block set up : a Block and a Search. The *self* outlet from the Block is linked to the *block-csp* inlet of the Search. After having created our block structure, we must not forget to evaluate the Search, that is clicking on it and pressing *v*. This also evaluates the Block due to their link. Figure 7.1 shows what the OM patch looks like.

7.1.3 Modus operandi

First, we can open the Block window by double clicking on the Block object. We now have to set some Block constraints to describe the melody that we have in mind. We can start by setting Bar length to 1 since we only want to create a short 1 chord melody. We can set Voices to 5. Minimum pushed notes can be set to 5 to force a full chord style of play. In the time constraints window, we can set Minimum note length to the maximum value to force a long chord. Finally, in the pitch constraints window, we can select C in Chord key and Major in Chord quality. We can also raise the Minimum pitch value and lower the Maximum pitch value. Our Block window now looks like Figure 7.2.

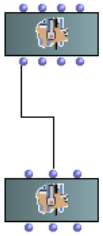


Figure 7.1: Scenario 1 : Patch set up

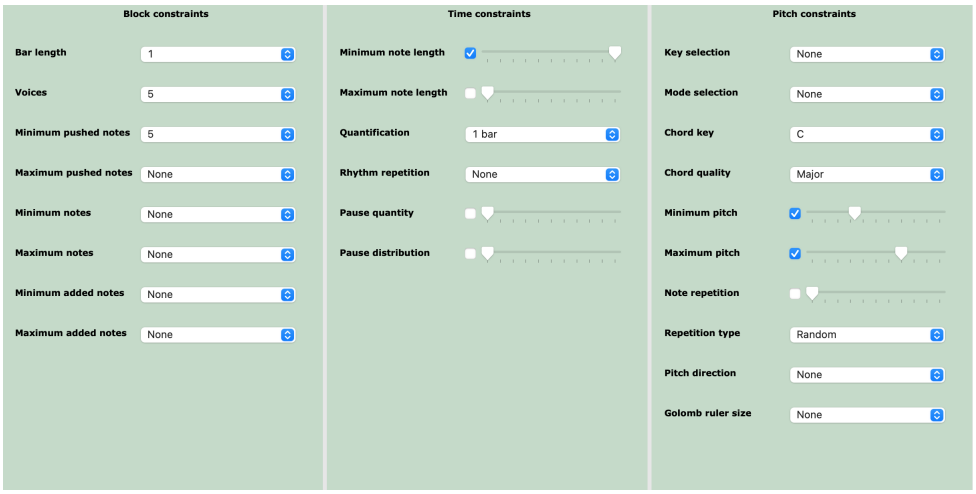


Figure 7.2: Scenario 1 : Block window

We can now start the search of a solution to our CSP. To do so, we have to open the Search window, set the BPM to 80 for example, and press Start. We can see the message "new-melodizer CSP constructed" printed on the OM listener. Pressing Next starts the search and shows us the solution. Figure 7.3 shows the CHORD-SEQ object containing the solution.

Now lets try to create some arpeggios around this chord. To do so, we can set Minimum pushed notes to None and Maximum pushed notes to 1. We also have

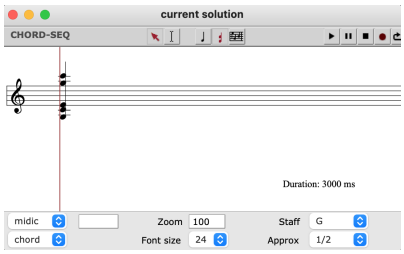


Figure 7.3: Scenario 1 : Solution 1

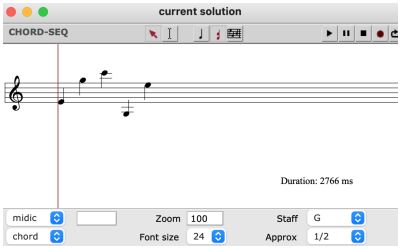


Figure 7.4: Scenario 1 : Solution 2

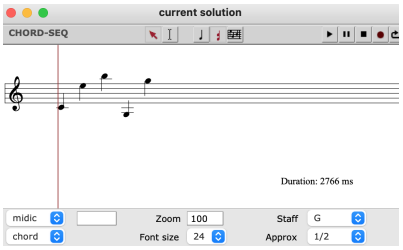


Figure 7.5: Scenario 1 : Solution 3

to reduce Minimum note length to less than half of the slider as notes of 1 bar length can only fit in a 1 bar length melody if they start at the very beginning, which we do not want. Then we can set Quantification to 1/2 bar. The solution to these settings can be found on Figure 7.4.

Finally, let's try to use a different, more complex type of chord. We are going to set Chord quality to Major 7. Figure 7.5 shows the solution to this set of constraints.

7.2 Scenario 2 : Playing with two chords

7.2.1 Description

In this scenario, we are going to learn how to use links between Blocks to create more complex melodies. The objective of this scenario is to create a 2 bars melody containing two different chords. All the musical solutions found in this scenario are available in this SoundCloud playlist.

7.2.2 Patch set up

As explained in the description, Melodizer uses OM links to create dependencies between Block objects. Since we want our melody to contain two chords, we are going to use three different Blocks. The two first Blocks each describe constraints for one of the chords and the last Block describe the constraints for the whole melody. We can start by naming our Blocks by pressing *cmd + i*, entering a name, then pressing *n* to display it on the Block. Our three Blocks are

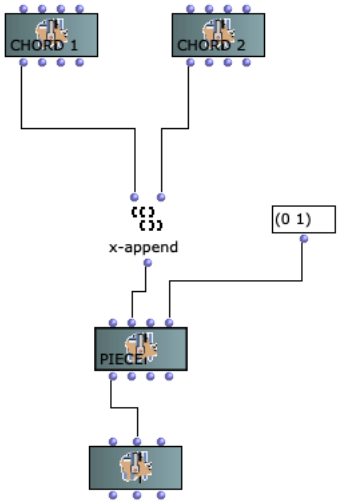


Figure 7.6: Scenario 2 : Patch set up

named Chord 1, Chord 2 and Piece. The patch we want to create is in Figure 7.6. We are using a *x-append* OM operator to create a list containing Chord 1 and Chord 2. This list is linked to the *block-list* inlet from Piece. They should be considered as sub blocks of Piece, meaning that every constraint from Piece also constrain them but not the opposite. The last element from the patch is the *(0 1)* list. Those numbers are the respective starting positions (in bars) of Chord 1 and Chord 2 in Piece.

7.2.3 Modus operandi

Figures 7.7, 7.8 and 7.9 show the settings for the three different blocks. A good idea when creating music is to start with very simple melodies and add to them step by step. This is especially true with Melodizer where it is really difficult to predict how different constraints might interact with each other. Starting with too many constraints often results in creating a problem with no solution. We thus start this scenario with two chords, a C major followed by a D minor, playing for 1 bar each. Figure 7.10 shows the results of these settings.

Now let's try to create some arpeggios around these chords two have a bit more rhythmic complexity. The first thing to do is to set a Maximum pushed notes constraint to 1 and lower the Quantification to 1/2 beat on Piece. Then we can lower the Minimum note length to around half the slider. Figure 7.11 shows the solution to these settings.

Figure 7.7: Scenario 2 : Piece window

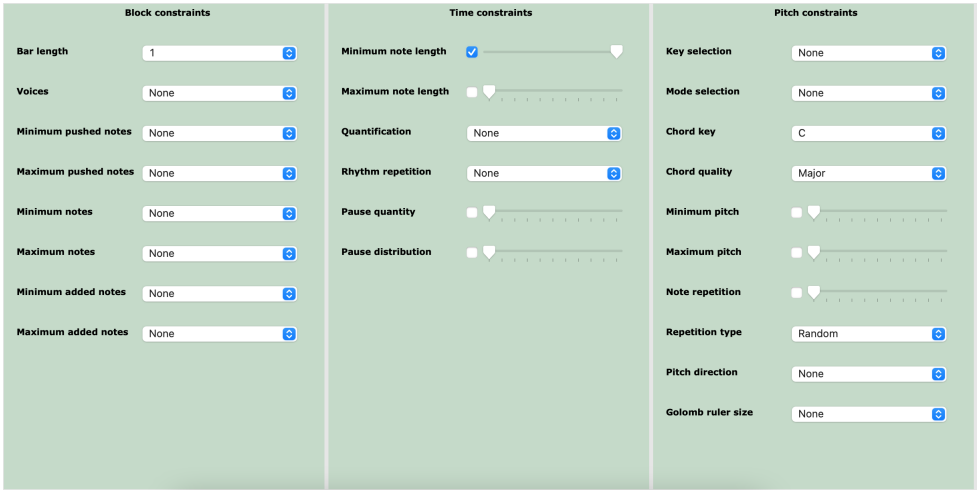


Figure 7.8: Scenario 2 : Chord 1 window

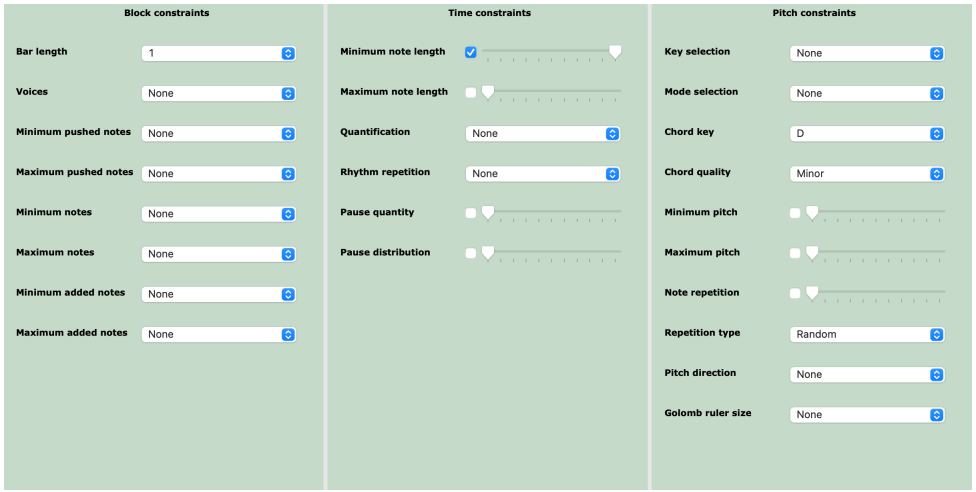


Figure 7.9: Scenario 2 : Chord 2 window

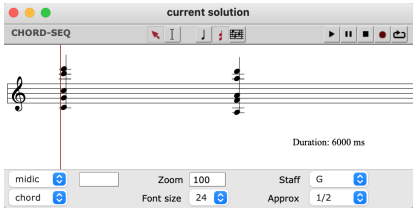


Figure 7.10: Scenario 2 : Solution 1

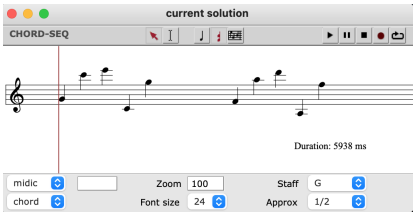


Figure 7.11: Scenario 2 : Solution 2

7.3 Scenario 3 : Melody on top of chords

7.3.1 Description

In this scenario, we are going to create a 4 bars melody composed of 4 different chords, following a simple I VI II V progression, and a melody on top. Both the chords and the melody are in the C major scale. All the musical solutions found in this scenario are available in this SoundCloud playlist.

7.3.2 Patch set up

The set up for this scenario is more complex than for the previous ones. Figure 7.12 shows what the patch looks like. The idea is to recursively divide components of the melody into smaller pieces, each represented by Blocks, until you can describe these pieces with the constraints of a Block. We thus start with the Piece, which represents the whole melody. We can divide the Piece into two parts : the Chord progression and the Melody. The Melody can be described precisely enough with a Block so there is no need to separate it. The chord progression, on the other hand, needs to be told what chords to play. We are thus going to create 4 more Blocks, one for each chord.

7.3.3 Modus operandi

For this scenario, we are going to change settings of blocks starting from the most general to the most precise. The Piece needs it's Bar length constraint set 4 and Voices set to 5 (to allow the Piece to feel full but not too much). As we do not want to add more notes than the chords and the melody, we have to set Maximum added notes to 0. We can set the Quantification to 1/4 beat and the Rhythm repetition to 1 bar to keep the same rhythmic between chords. We can set the Key selection to C and the Mode selection to ionian (major).

The Melody Block also has a Bar length of 4. We are going to set the Voices to 1 as we want the melody to be monophonic. We can raise the Minimum note length a bit. We can set the Minimum pitch and Maximum pitch in order to have a melody higher than the chords.

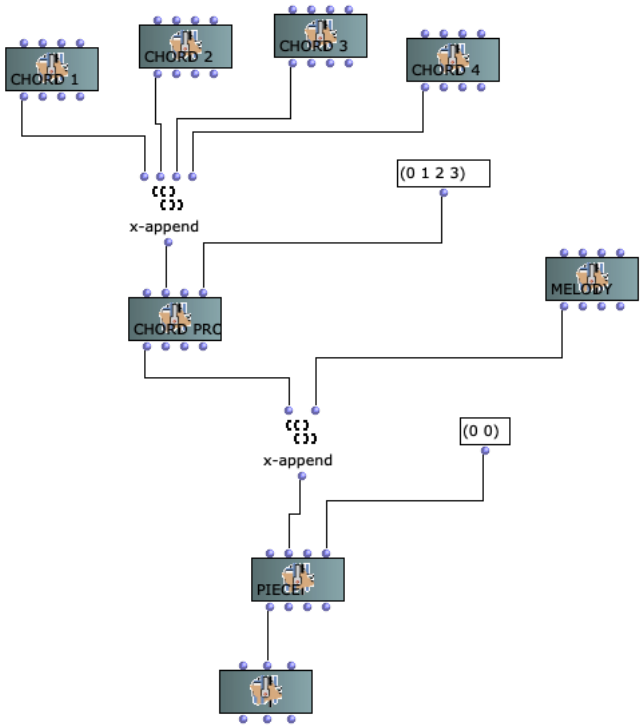


Figure 7.12: Scenario 3 : Patch set up

The Chord progression Block must contain the constraints that affect all the chords. Let's start by setting Bar length to 4 and Voices to 4. We can set the Quantification to 1 bar. Finally, we can set the Minimum pitch and Maximum pitch around the middle of the slider and close enough to each other to limit large pitch differences between chords notes.

Finally, we can set Bar length to 1 for all blocks and give them respectively C major, A minor, D minor and G major as Chord key and quality. We can also set the minimum length to the maximum value to force full long chords. The result for this setting is on Figure 7.14.

If you listen to the resulting melody, you will probably find it completely overloaded and not beautiful at all. To change that, we can tweak a few buttons in the Melody Block. The first thing we can do is to add silences to the melody. To do so, we can increase the pause quantity. We can also increase the Pause distribution to spread the silences more evenly in the melody. Figures 7.13 and 7.15 show the Melody pause settings and the resulting melody.

Now, we can hear that the pitch range of the melody feels slightly too large and that the notes seems to go all over the place. We can increase the Minimum

pitch to narrow the pitch range down. We can also increase the note repetition and set the Repetition type to Soft. Finally, we can lower the tempo of the melody in the Search down to 100 as it felt a bit too fast. Figure 7.16 shows the result of these changes in settings. Figure 7.17 shows what happens if you select Hard for Repetition type (it's obviously more repetitive). Random repetition does not work well with a Rhythm repetition constraint so we don't use it in this scenario.

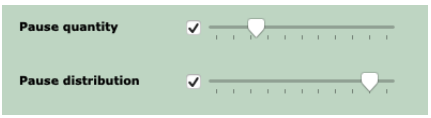


Figure 7.13: Scenario 3 : Melody pause settings

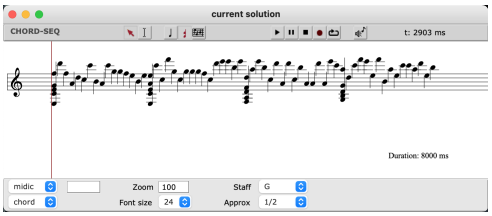


Figure 7.14: Scenario 3 : Solution 1

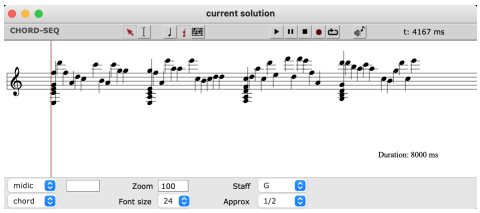


Figure 7.15: Scenario 3 : Solution 2

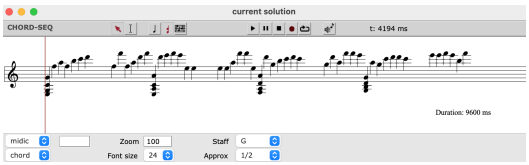


Figure 7.16: Scenario 3 : Solution 3

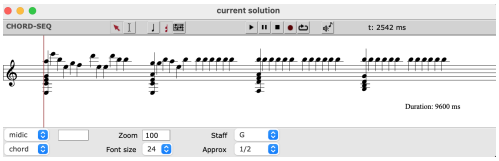


Figure 7.17: Scenario 3 : Solution 4

7.4 Scenario 4 : Blues in C Major

7.4.1 Description

In this scenario, we are going to explore the technique for repeating Blocks multiple times in the same song. This is particularly useful when the piece of music we want to create uses the same chord multiple times in the progression. For example, let's have a look at a simple C major blues chord progression on Figure 7.18 [1]. We can see that both C, F and G are repeated at least twice. We could simply create a Block for each bar but that would be very exhausting. Instead, we are simply going to use three chord Blocks, one for each chord. All the musical solutions found in this scenario are available in this SoundCloud playlist.

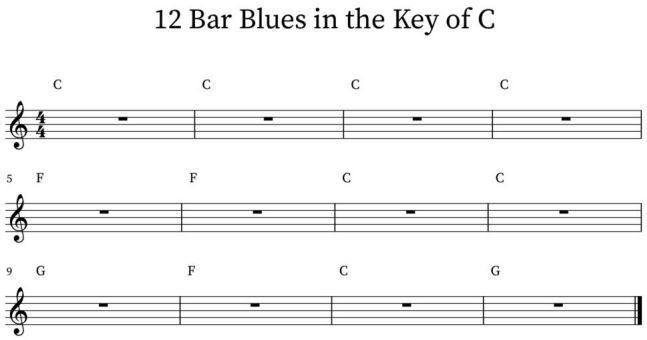


Figure 7.18: C major Blues

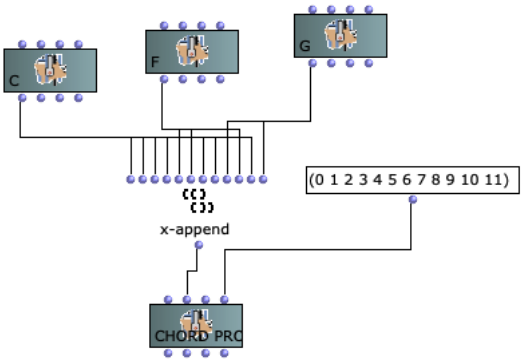


Figure 7.19: Scenario 4 : Wrong patch set up

7.4.2 Patch set up

As explained earlier, we want to minimise the quantity of Blocks for clarity and not spending too much time changing Block settings. At first, we could be tempted to use the set up shown on Figure 7.19 to describe the Blues chord progression but this does not work as expected. When being evaluated, the Chord progression Block creates as many chord Blocks as there are links to the x-append function. As a result, there is 7 C Blocks, 3 F blocks and 2 G blocks created. These Blocks are different and completely independent from each other thus changing settings on a Block window only affect one of them.

Figure 7.20 shows the patch organisation for this scenario. To solve the multiple Block problem, we are going to use the OM function create-list to create a list of pointers to the same Block. With this structure, changing settings on a chord Block (C, F or G) has an impact on all occurrences of this Block. If we take a look at the input list of positions of the Chord progression Block, we can see that the list is not ordered. This is because the first seven values of this list describe all C chord starting positions, then the following three are the F chord starting positions and finally the two last define the G chord starting positions.

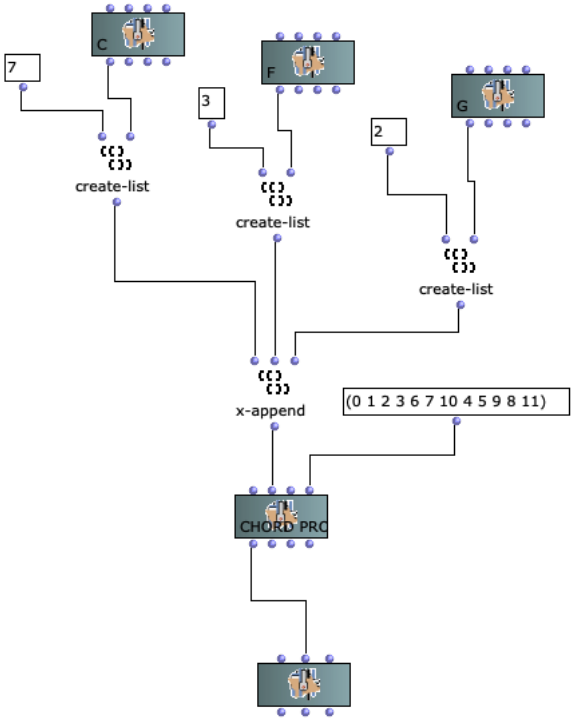


Figure 7.20: Scenario 4 : Patch set up

7.4.3 Modus operandi

Settings for this scenario are really basic and do not require much explanations. The Chord progression Block has its length set to 12. We can set Voices to 5 for a good balance between rich and simple chords. We also set Maximum added notes to 0 and Quantification to 1 bar. Finally, we can apply a small interval of pitches by setting Minimum pitch and Maximum pitch close to each other.

Settings for chords are also really basic. We simply set Bar Length to 1, Minimum note length to the maximum, Chord Key to C, F or G, and Chord quality to dominant 7 to get that characteristic bluesy warmth in the chords.

7.5 Scenario 5 : The strumming effect

7.5.1 Description

In the previous scenarios, we were more focusing on creating the melody rather than making it expressive or giving it personality. What really makes a melody interesting is the way it is played on the instrument. Some instruments have very recognizable expressive signatures that give strength to the melody. Some of

theses signatures can actually be described with constraints. In this scenario, we are going to describe the way guitarists strum chords on a guitar with constraints. All the musical solution found in this scenario are available in this SoundCloud playlist.

7.5.2 Patch set up

The patch set up for this scenario is really simple and does not require much description. Figure 7.21 shows the patch.

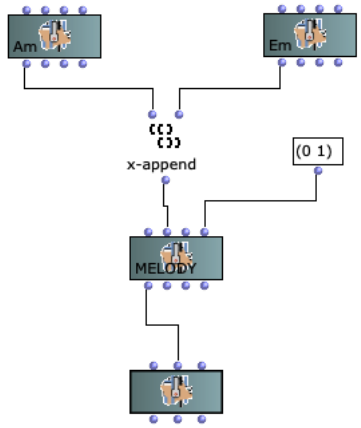


Figure 7.21: Scenario 5 : Patch set up

7.5.3 Modus operandi

In order to describe strumming into constraints, we first have to fully understand what it means. Strumming is the action of playing strings on a guitar in a single movement. This results in chords being activated in increasing order of pitch highness at different but really close times. In this scenario, we have chosen to describe a 4 string guitar strumming. Figure 7.22 shows the settings for this scenario. There are three important constraints here. The first one is the Maximum pushed notes set to 1 to make sure notes are activated at different times. The second is the quantification which is set to 1/12. The quantification makes the strumming speed vary. The last important constraint is the increasing constraint in Pitch direction. This truly makes that strumming feels real. Settings for the Em Block are similar except for the Chord key and Chord quality.

Figure 7.23 and 7.24 show solutions for a Quantification set to respectively 1/12 beat and 1/4 beat. We can see that, as we increase Quantification, the solutions tends to feel more like arpeggios rather than strumming.

Figure 7.22: Scenario 5 : Am window

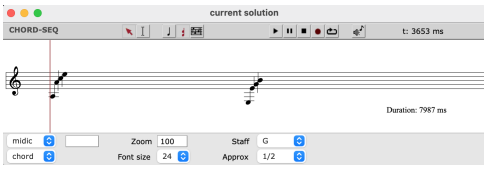


Figure 7.23: Scenario 5 : Solution 1

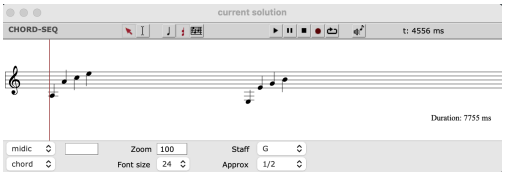


Figure 7.24: Scenario 5 : Solution 2

7.6 Scenario 6 : Unexpected results

7.6.1 Description

In this section, we are going to try generating the most unexpected results from Melodizer. There are many Block constraints which may have unexpected results. However, result unexpectedness can also reside in the Search object settings. In this scenario, we are going to focus on the Search object and its capability to get original melodies from a simple 1-Block CSP structure. All the musical solution found in this scenario are available in this SoundCloud playlist.

7.6.2 Patch set up

Patch set up for this scenario is exactly the same as for Scenario 1. Figure 7.1 shows what it looks like.

7.6.3 Modus operandi

As for the patch set up, we want really simple settings on the Block object to remove as few unexpected results from the set of solutions as possible. On the other hand, we will also use less efficient searching options for which we cannot afford too large search spaces. Figure 7.25 shows the Block window for this scenario. As we can see, we have set Voices to 1 for a very simple monophonic

melody. We have set Quantification to 1/4 beat to allow more complex rhythmic patterns. We have decided to set the scale to F major.

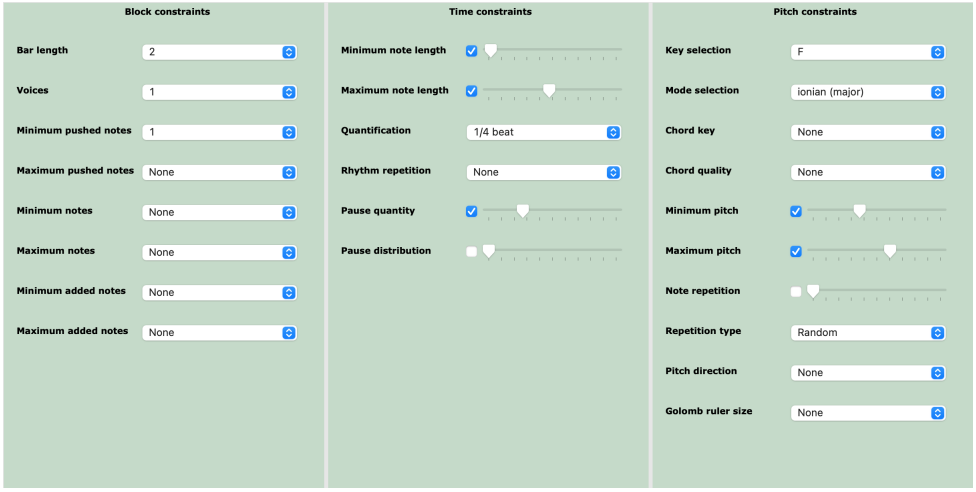


Figure 7.25: Scenario 6 : Block window

In order to fully understand what this scenario is about, we recommend having a listen at all the results on the playlist. The first 5 solutions were found with Top down Branching setting. Difference percentage was set around 70 percent. The first solution has all its pauses at the end of the melody. As we search for next solutions and impose a 70 percent difference, we can see that the pauses start diffusing across the melody and give more variety in rhythm. The rhythmic difference between Solution 1 and Solution 5 speaks for itself.

The last 3 solutions were found with a Top down random Branching setting. This immediately gives a more unexpected nature to the melody. Length of notes vary and pauses are spread all across the melody. A 70 percent difference creates almost completely different melodies.

Chapter 8

Conclusion

8.1 Melodizer 2.0 major achievements

As a whole, if we have to explain to a musician why they should upgrade to the new version of Melodizer, there would be four principal reasons :

- The new faculty of generating polyphonic pieces. Contrarily to the previous version that would only generate monophonic melodies.
- Its ability to combine both rhythm and pitch constraints. The anterior tool would only present pitch related constraints.
- The definition and implementation of Block and Search objects that allows you to structure your piece. The composer can now combine Blocks, each one with its specific constraints, to be played simultaneously or consecutively. This would open the door to many new musical scenarios compared to the previous version where the same constraints were applied to the whole piece. Blocks, similarly to musical phrases can be repeated. For example, in the basic ternary form ABA, parts A and B can be represented by two different Blocks and we juxtapose Block A, Block B and then repeat Block A.
- Melodizer 2.0 guarantees that the solver provides diverse solutions. In fact, the users can now decide what is the percentage that they would like to change from one solution to another. This is a big improvement compared to the previous version of Melodizer that provided solutions where only one note would change. In order to achieve this, the Branch-and-Bound search engine had to be introduced to Melodizer 2.0 .

8.1.1 Necessary steps to develop Melodizer 2.0

In order to achieve these goals there were many steps to complete :

- We conceived an entire new model in Gecode that would allow us to state a Constraint Satisfaction Problem to generate musical polyphonic solutions with the composer's pitch and rhythm constraints. Within this framework,

we had to translate musical constraints into mathematical constraints. Furthermore, we had to implement the base structure that would allow block connection to combine musical phrases to be played sequentially or simultaneously. We analyzed the proposed branching heuristics and exploration strategies to generate more diverse solutions efficiently.

- We introduced our model with all the musical constraints, and, Branch-and-Bound to GiL, the interface between Gecode and Lisp.
- Create an interactive user-friendly interface in OpenMusic that would easily allow composers to choose amongst the implemented musical constraints, the ones they would like to incorporate into their theme. We developed the Block objects that can eventually be connected to recreate a structured piece with different constrained parts played together or successively.
- We played with our tool by recreating musical scenarios to see what aspects could improve the user experience and what interesting constraints could be added.

8.2 Further improvements and using Melodizer 2.0 as a cornerstone

With a tool that offers as many possibilities as Melodizer 2.0 there is still a lot of room for improvements and for additional features. The following subsections contains both the features we would have liked to add and the ideas we had when testing our final product. The following element are all the relevant things that should be incorporated specifically to Melodizer 2.0 and not to GiL. Since we consider GiL as a tool that should be upgraded along the way.

8.2.1 Some general ideas

- **Rhythm control** : As of now the control over rhythm given to the composer is on 3 main parts, choosing the quantification, changing the quantity and distribution of rests and tweaking the number of notes in the melody. Those elements together with some complex blocks tree structure can give a good control over rhythm but this method is tough, not intuitive and still not perfect as rhythm is often more complex. Accomplishing this can be done through multiple means, like making the solver more creative in its solution, adding new constraints that can give better control over rhythm, or even completely separating melody and rhythm in two different elements that can work together but we talk more about that in the next subsection 8.2.2.
- **Improving the Searching experience** : Without much surprise the research algorithm and branching are the most determining elements to find solutions and there is always place for upgrade in this domain. For

the moment, we have settled for what felt like a good compromise between time of research and interest of the solution but offering more option and letting the user choose between them could be a great improvement. In the specific case of music creation, as there is no wrong or bad solution, just new musical ideas that are always interesting to someone, there are good reasons to add some randomness to the solver, always with the possibility to reproduce the results in order to not loose anything by inadvertence. Randomness is a great way to find unexpected solutions that doesn't follow any pattern, it can be a great motor for creativity.

- **Better user experience** : Using Melodizer is probably not an easy task for the uninitiated but of course this can be improved by various additions. One of the most important is probably to give constructive feedback on error. When dealing with a lot of constraints it's easy to end up with no solution, sometimes simply because two constraints are in direct contradiction and the user didn't realize it because they are not supposed to deal with constraint but with music. In this case, in its current state Melodizer would return a simple "no more solutions" which is not helping much.
  Improving the experience can also be done by adding a database to the tool, in order to save some solutions for later usage, and allowing to modify and concatenate these solution to create a bigger piece. The sets of constraint that was used to create a solution can also be saved in order to retry them later and find similar solutions with some new branching for example.

8.2.2 Extending the block structure

The block structure is the main feature of Melodizer 2.0, it allows to create a lot of complexity in the melody found and give a different vision of music, in a tree like structure, similarly to what is usually done by most music composition software. But the blocks implementation is far from perfect and many improvements can be brought.

First of all, new blocks. Melodizer 2.0 only comes with two Open Music classes, the block and search objects, blocks alone are used to describe all the aspects of a melody. This versatility of the blocks in representing many different things was an objective for us in order to keep things simple, but it turned out that it might not be that simple. Trying to do everything from the same class make some things harder, like as stated previously rhythm control. This can be improved by dividing the constraints among different classes, we can imagine a pitch block which, as its name suggest, take the constraint to find the pitch of the notes and a rhythm block, that find some rhythm, obviously they have to work together as the number of pitch to find is linked to the rhythm of the melody, yet this is completely possible with the tree like structure already introduced. Figure 8.1 presents an example of how the interaction between these new blocks could work. Here, a pitch block take the combination of two other pitch blocks to

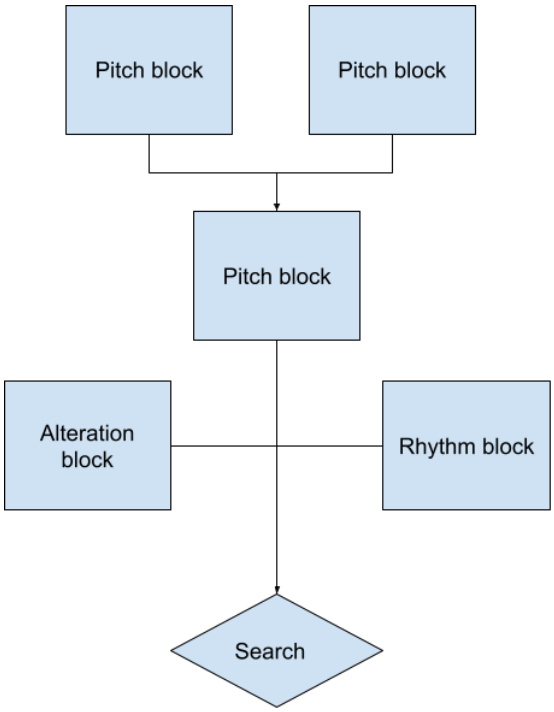


Figure 8.1: An idea for the future of the block structure

create its output, this result is mixed with the output of a rhythm block and an alteration block, in this case an alteration block is a class that add constraint to create irregularities in the result of the pitch block, resulting in some dissonance for example. There is a lot of these kind of blocks that can be added either from completely new ideas or by separating the current functionalities of the block object. It is nevertheless important to not get carried away, a block should not replace what can be a single constraint, for example an increasing pitch block which only create melody with increasing pitch is probably not a good idea as it multiplies objects for what could be a simple constraint in the pitch block.

Next is the interaction between blocks. At the moment blocks only interaction is through some kind of constraints sharing while there is a lot more that can be done. Allowing to fix the melody generated by a specific block for example, instead of sharing its constraint, this block does it's own search, fix its melody output and shares it with the parent block, then when the composer search for new melody there is always a part of the solution that doesn't change which is useful when we found a nice musical voice for the lead, but the chords are not that good.

Furthermore, we could include genre Blocks such as a Jazz, Blues, Rock, Metal or Pop Blocks by extending the already existing Block. This would allow the composer to select musical constraints, rhythms and chord progressions that are more genre specific. For example, it would not make sense in Pop music to

include a constraint allowing the devil's tritone [1] intervals since Pop music follows quiet rigorously the musical rules of consonance. This is why it tends to sound catchy at first but can quickly become monotonous. However in Metal, it would perfectly make sense to include dissonance by allowing devil's tritone intervals.

Finally, along the same line, we could include instrument specific Blocks also by extending the already existing Block. This would allow to generate an orchestral pieces where each block would represent an instrument. These blocks would already contain its specific physical constraints such as the tessitura or range of the instrument in order for the solver to provide realistic scores. For instance, it wouldn't generate a fast sequence of high pitched notes for a tuba. Whereas for a piccolo it could be perfectly possible.

8.2.3 A final word about musical constraints

Adding new possibility of musical constraints is the most obvious and easiest way to improve Melodizer 2.0. Giving a list of constraint ideas would be useless because while working on this project we quickly realized that any constraint can in a fact be a musical constraint, it's a matter of applying it on the right musical aspect. When creating new constraint opportunities one should not be stopped by thinking that a constraint will not serve any purpose as composers always enjoy more possibilities and find way to create some melodies out of the most unexpected tools.

It is needless to say that there can be an immense amount of possible musical constraints with all the existing genres and compositional styles. This is why, in the context of our thesis we focused more in developing the main building blocks where more features could be added on top. Rather than attempting to represent all the possible musical constraints. Nevertheless, we questioned ourselves how we could eventually integrate as many musical constraints as possible. We came up with the following ideas :

- Ask advise to several composer of different genres in order to cover the most widely used constraints.
- Create an Open Source project. This would allow for computer-savvy composers to add the musical constraints they need but that aren't available yet.
- Allow the composer to write some kind of code to create themselves the constraints that fits them best. This was a question asked by a spectator of the audience at IRCAM when Damien Sprockeels presented Melodizer. He requested if it was possible for the user to encode himself his own musical constraints. However, the major drawback is that not all the composers know nor want to code!

[1] also known as augmented fourth or diminished fifth

Bibliography

[1] Aidan. *An Introduction To The 12 Bar Blues*. Happy Bluesman. Aug. 25, 2019. URL: https://happybluesman.com/introduction-12-bar-blues/ (visited on 06/03/2022).

[2] Stephen Boyd and Jacob Mattingley. "Branch and bound methods". In: *Notes for EE364b, Stanford University* (2007), pp. 2006–07.

[3] C. W. Choi et al. "Finite Domain Bounds Consistency Revisited". In: *AI 2006: Advances in Artificial Intelligence*. Ed. by Abdul Sattar and Byeong-ho Kang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 49–58. ISBN: 978-3-540-49788-2.

[4] Adolphe Danhauser. *Théorie de la musique*. Lemoine, 1994.

[5] Romuald Debruyne and Christian Bessiere. "Some practicable filtering techniques for the constraint satisfaction problem". In: *In Proceedings of IJCAI'97*. Citeseer. 1997.

[6] Jean-Louis Foucart. *Vingt leçons d'harmonie pour comprendre et composer la musique*. 2004.

[7] Robert M Haralick and Gordon L Elliott. "Increasing tree search efficiency for constraint satisfaction problems". In: *Artificial intelligence* 14.3 (1980), pp. 263–313.

[8] William D Harvey and Matthew L Ginsberg. "Limited discrepancy search". In: *IJCAI (1)*. 1995, pp. 607–615.

[9] Scott Jarrett and Holly Day. *Music composition for dummies*. John Wiley & Sons, 2008.

[10] Baptiste Lapière. "Computer-aided musical composition Constraint programming and music". MA thesis. UCLouvain, 2019–2020.

[11] Pascal Van Hentenryck Laurent Michel Pierre Schaus. *Part 1: Overview of CP, Filtering, Search, Consistency, Fixpoint*. 2021. URL: https://minicp.readthedocs.io/en/latest/learning_minicp/part_1.html (visited on 05/04/2022).

[12] Pascal Van Hentenryck Laurent Michel Pierre Schaus. *Part 5: Circuit Constraint, TSP, Optimization, LNS, and VRP*. 2021. URL: https://minicp.readthedocs.io/en/latest/learning_minicp/part_5.html (visited on 05/04/2022).

[13] Deborah Lupton. *Digital sociology*. Routledge, 2014.

[14] Laurent Michel, Pierre Schaus, and Pascal Van Hentenryck. "MiniCP: a lightweight solver for constraint programming". In: *Mathematical Programming Computation* 13.1 (2021), pp. 133–184.

[15] Global Guitar Network. *Chord Progressions in a Major Key*. URL: `https://globalguitarnetwork.com/chord-progressions-major-key/` (visited on 05/25/2022).

[16] Michael Pilhofer and Holly Day. *Music theory for dummies*. John Wiley & Sons, 2019.

[17] David Pisinger and Stefan Ropke. "Large neighborhood search". In: *Handbook of metaheuristics*. Springer, 2010, pp. 399–419.

[18] Ircam - Centre Pompidou. *OpenMusic Documentation*. URL: `https://support.ircam.fr/docs/om/om6-manual/co/OM-Documentation.html` (visited on 05/17/2022).

[19] Ircam - Centre Pompidou. *OpenMusic Documentation : Score objects*. URL: `https://support.ircam.fr/docs/om/om6-manual/co/ScoreObjects.html` (visited on 05/17/2022).

[20] Christopher G Reeson et al. "An interactive constraint-based approach to Sudoku". In: *AAAI*. 2007, pp. 1976–1977.

[21] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. "Modeling and programming with gecode". In: vol. 1. 2010. Chap. 4, pp. 55–290.

[22] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. "Modeling and programming with gecode". In: vol. 1. 2010. Chap. 8, pp. 121–148.

[23] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. "Modeling and programming with gecode". In: vol. 1. 2010. Chap. 2, pp. 13–34.

[24] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. "Modeling and programming with gecode". In: vol. 1. 2010. Chap. 5, pp. 83–92.

[25] Christian Schulte, Guido Tack, and Mikael Z Lagerkvist. "Modeling and programming with gecode". In: vol. 1. 2010. Chap. 9, pp. 149–184.

[26] Barabara M Smith, Kostas Stergiou, and Toby Walsh. "Modelling the Golomb ruler problem". In: *Research report series-university of Leeds school of computer studies LU SCS RR* (1999).

[27] Piano Time [Computer Software]. URL: `https://apps.microsoft.com/store/detail/piano-time/9WZDNCRFJC9S?hl=en-us&gl=US`.

[28] Damien Sprockeels. "Melodizer : A Constraint Programming Tool For Computer-aided Musical Composition". MA thesis. UCLouvain, 2020-2021.

[29] ThePiano.SG. *How To Tell The Quality Of A Musical Chord*. URL: `https://www.thepiano.sg/piano/read/how-tell-quality-musical-chord` (visited on 05/25/2022).

Appendix A

How to install Melodizer 2.0

This appendix explains how to download and install Melodizer 2.0. GiL does not work on Windows because Lisp's license used by OpenMusic is a 32bit version, and the Gecode Windows version is 64 bits by default. Therefore Melodizer 2.0 work only in the MacOS and Linux operating systems.

A.1 Download and install

In order to use Melodizer 2.0 inside OpenMusic, we frstly have to :

- Download and install Gecode : https://www.gecode.org/download.html
- Download and install OpenMusic : https://openmusic-project.github.io/openmusic/
- Download GiL : https://github.com/sprockeelsd/GiLv2.0
- Download Melodizer 2.0 : https://github.com/clemsky/TFE-Composition-Musicale

A.2 Loading the libraries to OpenMusic

To load the libraries and start using it you first have to launch Open Music. On the first window you can either create a new workspace or open the previous one as seen in figure A.1. When you workspace is open, in the toolbar in the upper part of the interface click the "windows" button highlighted in figure A.2 and then "Library" in the dropdown menu. A new window will open, select "File" in the toolbar and the "Add remote library", from there you will be able to navigate your file system to find the path to your Melodizer and Gil library previously downloaded. Finally the two libraries should appear in the "Library" window under the "libraries" folder as seen in figure A.3, right click on Melodizer and select "load", if no error appears everything should be up and ready to go.
Nevertheless if an error does appear, there is great chance that it is due to a linking problem with the Gecode library, to solve it on MacOS you can use the script found in the c++ folder of the gil library, before using it you should edit

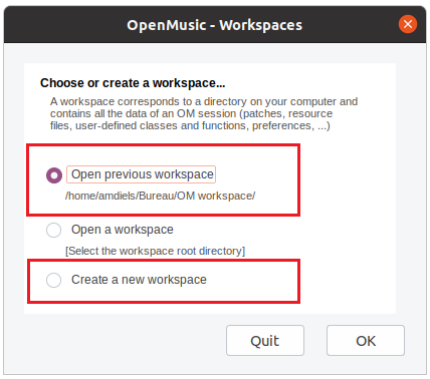


Figure A.1: First window when launching Open Music

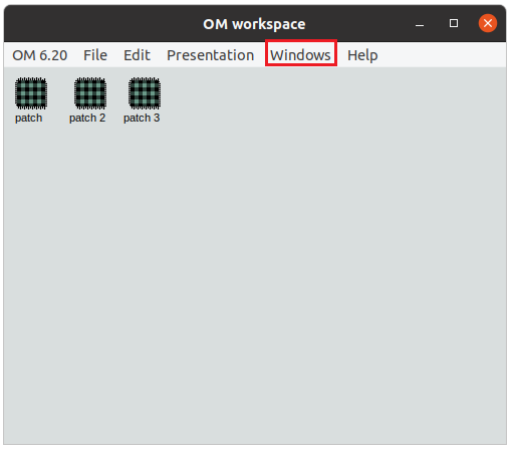


Figure A.2: The workspace of Open Music

the path to Gecode inside it to be the one used by your system. If you are using Linux you should add the Gecode library to the LD_LIBRARY_PATH variable, to do that head to the folder */etc/ld.so.conf.d* of your system and add a new *.conf* file if one is not already present. Paste the full path to your Gecode library in this file and save it, then run *sudo ldconfig* to update your system with the new library and you should be ready to properly use Melodizer.

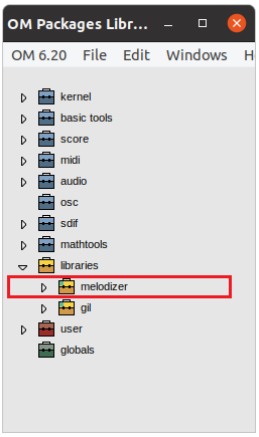


Figure A.3: The library window of Open Music

Appendix B

Gecode source code

B.1 Sudoku propagation example

```
#include <gecode/driver.hh>
#include <gecode/int.hh>
#include <gecode/minimodel.hh>
#include <gecode/int/arithmetic.hh>
#include <gecode/set.hh>
#include <vector>

using namespace Gecode;

class Melody : public Space {
protected:
  IntVarArray x;
public:
  Melody(void) : x(*this, 9, 1, 9) {

      int v[] = { 7, 9 };
      IntSet c(v, 2);

      rel(*this, x[1], IRT_EQ, 2);
      rel(*this, x[3], IRT_EQ, 1);
      rel(*this, x[6], IRT_EQ, 3);

      dom(*this, x[0], 1, 6);
      dom(*this, x[2], 1, 5);
      dom(*this, x[4], 1, 5);
      dom(*this, x[5], 7, 9);
      dom(*this, x[7], c);
      dom(*this, x[8], c);

      distinct(*this, x, IPL_DOM); //for domain propagation
      //distinct(*this, x, IPL_VAL); //for value propagation
      //distinct(*this, x, IPL_BND); //for bound propagation

      branch(*this, x, INT_VAR_NONE(), INT_VAL_MIN());
  }
  Melody(Melody& s) : Space(s) {
    x.update(*this, s.x);
  }
  virtual Space* copy(void) {
    return new Melody(*this);
  }
  void print(std::ostream& os) const {
    os << x << std::endl;
  }
};

int main(int argc, char* argv[]) {
  Melody* m = new Melody;
  Gist::Print<Melody> p("Print solution");
  Gist::Options o;
  o.inspect.click(&p);
  Gist::dfs(m, o);
  delete m;
  return 0;
}
```

Appendix C

Gil source code

This appendix contains the code of Gil, the interface between Lisp and Gecode which was first implemented by Baptiste Lapière [10], then improved by Damien Sprockeels [28] and finally we further developed it during our thesis. The code is divided in two main parts, the C wrapper in section C.1 and the lisp wrapper in section C.2. You can find more about Gil and how we upgraded it in chapter 4. The complete code can also be found on github at https://github.com/sprockeelsd/GiLv2.0

C.1 C Wrapper

The C wrapper is used to wrap Gecode function with C functions, in order to be able to call them from Lisp using CFFI. It is made of 4 files :

- **space_wrapper.hpp** : header file that contains the declaration of the c++ function and classes wrapping gecode elements, but also the declaration of the space and the arrays containing all the variables.
- **space_wrapper.cpp** : implements the function and classes to wrap Gecode.
- **gecode_wrapper.hpp** : declares the C function that will wrap the C++ functions of space_wrapper.cpp.
- **gecode_wrapper.hpp** : implements the function of gecode_wrapper.hpp.

C.1.1 space_wrapper.hpp

```
1  #ifndef space_wrapper_hpp
2  #define space_wrapper_hpp
3
4  #include <vector>
5  #include <iostream>
6  #include <stdlib.h>
7  #include <exception>
8  #include "gecode/kernel.hh"
9  #include "gecode/int.hh"
10 #include "gecode/search.hh"
11 #include "gecode/minimodel.hh"
12 #include "gecode/set.hh"
```

```
13
14 using namespace Gecode;
15 using namespace Gecode::Int;
16 using namespace Gecode::Set;
17 using namespace std;
18 using namespace Gecode::Search;
19
20 class WSpace: public IntMinimizeSpace {
21 protected:
22     vector<IntVar> int_vars;
23     vector<BoolVar> bool_vars;
24     vector<SetVar> set_vars;
25     int i_size;
26     int b_size;
27     int s_size;
28     int cost_id;
29
30     //int* solution_variable_indexes;// to know what variables will hold the solution, useful
       for bab
31     int* solution_variable_indexes;
32     int var_sol_size;
33     int percent_diff;
34
35     //========================
36     //= Variables from idx =
37     //========================
38
39     /**
40      Return the IntVar contained in int_vars at index vid.
41      */
42     IntVar get_int_var(int vid);
43
44     /**
45      Return the BoolVar contained in bool_vars at index vid.
46      */
47     BoolVar get_bool_var(int vid);
48
49     /**
50     Return the SetVar contained in set_vars at index vid.
51     */
52     SetVar get_set_var(int vid);
53
54
55     //======================
56     //= Args for methods =
57     //======================
58
59     /**
60      Return an IntVarArgs of size n, containing the n IntVars contained in
61      int_vars at indices vids.
62      */
63     IntVarArgs int_var_args(int n, int* vids);
64
65     /**
66      Return a BoolVarArgs of size n, containing the n BoolVars contained in
67      bool_vars at indices vids.
68      */
69     BoolVarArgs bool_var_args(int n, int* vids);
70
71         /**
72      Return a SetVarArgs of size n, containing the n SetVars contained in
73      set_vars at indices vids.
74      */
75     SetVarArgs set_var_args(int n, int* vids);
76
77     /**
78      Return an IntArgs of size n, containing the n values in vals
79     */
80     IntArgs int_args(int n, int* vals);
81
82     /**
83      Return the expression int_rel(vid, val)
84     */
85     BoolVar bool_expr_val(int vid, int int_rel, int val);
86
87     /**
88      Return the expression int_rel(vid1, vid2)
89     */
90     BoolVar bool_expr_var(int vid1, int int_rel, int vid2);
91
92 public:
93     /**
94      Default constructor
95      */
96     WSpace();
97
98     //============================
99     //= Variables and domains =
100    //============================
101
102    /**
103     Add an IntVar to the WSpace ranging from min to max.
104     In practice, push a new IntVar at the end of the vector int_vars.
```

```
105      Return the index of the IntVar in int_vars
106      */
107     int add_intVar(int min, int max);
108
109     /**
110      Add an IntVar to the WSpace with domain dom of size s.
111      In practice, push a new IntVar at the end of the vector int_vars.
112      Return the index of the IntVar in int_vars
113      */
114     int add_intVarWithDom(int s, int* dom);
115
116     /**
117      Add n IntVars to the WSpace ranging from min to max.
118      In practice, push n new IntVars at the end of the vector int_vars.
119      Return the indices of the IntVars in int_vars.
120      */
121     int* add_intVarArray(int n, int min, int max);
122
123     /**
124      Add n IntVars to the WSpace with domain dom of size s.
125      In practice, push n new IntVars at the end of the vector int_vars.
126      Return the indices of the IntVars in int_vars.
127      */
128     int* add_intVarArrayWithDom(int n, int s, int* dom);
129
130     /**
131      Define which variables are to be the solution so they can be accessed to add a
       constraint with bab
132     */
133     void set_as_solution_variables(int n, int* vids);
134
135     /**
136     Define the percentage of the solution that should change when searching for the next
       solution with BAB
137     */
138     void set_percent_diff(int diff);
139
140     /**
141      Return the number of IntVars in the space.
142      */
143     int nvars();
144
145     enum {
146         //Relations for BoolExpr
147         B_EQ,
148         B_NQ,
149         B_LE,
150         B_LQ,
151         B_GQ,
152         B_GR
153     };
154
155     /**
156      Add a BoolVar to the WSpace ranging from min to max.
157      In practice, push a new BoolVar at the end of the vector bool_vars.
158      Return the index of the BoolVar in bool_vars
159      */
160     int add_boolVar(int min, int max);
161
162     /**
163     Add n BoolVars to the WSpace ranging from min to max.
164     In practice, push n new BoolVars at the end of the vector bool_vars.
165     Return the indices of the BoolVars in bool_vars.
166     */
167     int* add_boolVarArray(int n, int min, int max);
168
169     /**
170      Add a BoolVar to the WSpace corresponding to the evaluation of int_rel(vid, val).
171      In practice, push a new BoolVar at the end of the vector bool_vars.
172      Return the index of the BoolVar in bool_vars
173      */
174     int add_boolVar_expr_val(int vid, int int_rel, int val);
175
176     /**
177      Add a BoolVar to the WSpace corresponding to the evaluation of int_rel(vid1, vid2).
178      In practice, push a new BoolVar at the end of the vector bool_vars.
179      Return the index of the BoolVar in bool_vars
180      */
181     int add_boolVar_expr_var(int vid1, int int_rel, int vid2);
182
183     /**
184     Add a SetVar to the WSpace initialized with n integer from array r.
185     In practice, push a new SetVar at the end of the vector set_vars.
186     Return the index of the SetVar in set_vars.
187     */
188     int add_setVar(int lub_min, int lub_max, int card_min, int card_max);
189
190     /**
191     Add n SetVars to the WSpace ranging with cardinality card_min to card_max.
192     In practice, push n new SetVars at the end of the vector set_vars.
193     Return the indices of the SetVars in set_vars.
194     */
195     int* add_setVarArray(int n, int lub_min, int lub_max, int card_min, int card_max);
```

```
196
197     //=======================
198     //= Posting constraints =
199     //=======================
200
201     //=== INTVARS ===
202
203     /**
204      Post a relation constraint between the IntVar denoted by vid and the val.
205      */
206     void cst_val_rel(int vid, int rel_type, int val);
207
208     /**
209      Post a relation constraint between the IntVars denoted by vid1 and vid2.
210     */
211     void cst_var_rel(int vid1, int rel_type, int vid2);
212
213     /**
214     Post a relation constraint between the IntVars denoted by vid1 and vid2 with reification.
215     */
216     void cst_var_rel_reify(int vid1, int rel_type, int vid2, int vid3, int mode);
217
218     /**
219     Post a relation constraint between the IntVars denoted by vid1 and val with reification.
220     */
221     void cst_val_rel_reify(int vid1, int rel_type, int val, int vid2, int mode);
222
223     /**
224      Post a relation constraint between the n IntVars denoted by vids and the val.
225     */
226     void cst_arr_val_rel(int n, int* vids, int rel_type, int val);
227
228     /**
229      Post a relation constraint between the n IntVars denoted by vids and the the IntVar vid.
230     */
231     void cst_arr_var_rel(int n, int* vids, int rel_type, int vid);
232
233     /**
234      Post a relation constraint between the n IntVars denoted by vids.
235     */
236     void cst_arr_rel(int n, int* vids, int rel_type);
237
238     /**
239      Post a lexicographic relation constraint between the n1 IntVars denoted by vids1 and
240      the n2 IntVars denoted by vids2.
241     */
242     void cst_arr_arr_rel(int n1, int* vids1, int rel_type, int n2, int* vids2);
243
244     /**
245      Post the constraint that the n IntVars denoted by vids are distinct
246      */
247     void cst_distinct(int n, int* vids);
248
249     /**
250      Post the linear constraint [c]*[vids] rel val.
251      */
252     void cst_val_linear(int n, int* c, int* vids, int rel_type, int val);
253
254     /**
255      Post the linear constraint [c]*[vids] rel_type vid.
256     */
257     void cst_var_linear(int n, int* c, int* vids, int rel_type, int vid);
258
259     /**
260      Post the constraint that |vid1| = vid2.
261      */
262     void cst_abs(int vid1, int vid2);
263
264     /**
265      Post the constraint that dom(vid) = d, where d is a set of size n.
266      */
267     void cst_dom(int vid, int n, int* d);
268
269     /**
270      Post the constraint that vid is included in {vids[0], ..., vids[n-1]}
271      */
272     void cst_member(int n, int* vids, int vid);
273
274     /**
275      Post the constraint that vid1 / vid2 = vid3.
276      */
277     void cst_div(int vid1, int vid2, int vid3);
278
279     /**
280      Post the constraint that vid1 % vid2 = vid3.
281      */
282     void cst_mod(int vid1, int vid2, int vid3);
283
284     /**
285      Post the constraint that vid1 / vid2 = vid3
286      and vid1 % vid2 = div4
287      */
288     void cst_divmod(int vid1, int vid2, int vid3, int vid4);
```

```
289
290     /**
291      Post the constraint that min(vid1, vid2) = vid3.
292      */
293     void cst_min(int vid1, int vid2, int vid3);
294
295     /**
296      Post the constraint that vid = min(vids).
297      */
298     void cst_arr_min(int n, int* vids, int vid);
299
300     /**
301      Post the constraint that vid = argmin(vids).
302      */
303     void cst_argmin(int n, int* vids, int vid);
304
305     /**
306      Post the constraint that max(vid1, vid2) = vid3.
307      */
308     void cst_max(int vid1, int vid2, int vid3);
309
310     /**
311      Post the constraint that vid = max(vids).
312      */
313     void cst_arr_max(int n, int* vids, int vid);
314
315     /**
316      Post the constraint that vid = argmax(vids).
317      */
318     void cst_argmax(int n, int* vids, int vid);
319
320     /**
321      Post the constraint that vid1 * vid2 = vid3.
322      */
323     void cst_mult(int vid1, int vid2, int vid3);
324
325     /**
326      Post the constraint that sqr(vid1) = vid2.
327      */
328     void cst_sqr(int vid1, int vid2);
329
330     /**
331      Post the constraint that sqrt(vid1) = vid2.
332      */
333     void cst_sqrt(int vid1, int vid2);
334
335     /**
336      Post the constraint that pow(vid1, n) = vid2.
337      */
338     void cst_pow(int vid1, int n, int vid2);
339
340     /**
341      Post the constraint that nroot(vid1, n) = vid2.
342      */
343     void cst_nroot(int vid1, int n, int vid2);
344
345     /**
346      Post the constraint that vid = sum(vids).
347      */
348     void cst_sum(int vid, int n, int* vids);
349
350     /**
351      Post the constraint that the number of variables in vids equal to val1 has relation
     rel_type
352      with val2.
353      */
354     void cst_count_val_val(int n, int* vids, int val1, int rel_type, int val2);
355
356     /**
357      Post the constraint that the number of variables in vids equal to val has relation
     rel_type
358      with vid.
359      */
360     void cst_count_val_var(int n, int* vids, int val, int rel_type, int vid);
361
362     /**
363      Post the constraint that the number of variables in vids equal to vid has relation
     rel_type
364      with val.
365      */
366     void cst_count_var_val(int n, int* vids, int vid, int rel_type, int val);
367
368     /**
369      Post the constraint that the number of variables in vids equal to vid1 has relation
     rel_type
370      with vid2.
371      */
372     void cst_count_var_var(int n, int* vids, int vid1, int rel_type, int vid2);
373
374     /**
375      Post the constraint that the number of variables in vids in the set set has relation
     rel_type with vid2
376      */
```

```
377     void cst_count_var_set_val(int n, int*vids, int s, int* set, int rel_type, int val);
378
379     /**
380      Post the constraint that the number of variables in vids where vars[i] = c[i] and c is
        an array of integers has rel_type to val
381      */
382     void cst_count_array_val(int n, int*vids, int* c, int rel_type, int val);
383
384     /**
385      Post the constraint that the number of occurences of s-set in every subsequence of
        length
386      val1 in vids must be higher than val2 and lower than val3
387      */
388     void cst_sequence_var(int n, int*vids, int s, int* set, int val1, int val2, int val3);
389
390     /**
391      Post the constraint the number of distinct values in the n variables denoted by vids
392      has the given rel_type relation with the variable vid.
393      */
394     void cst_nvalues(int n, int* vids, int rel_type, int vid);
395
396     /**
397      Post the constraint that values of vids1 are the edges of an hamiltonian circuit in
398      the graph formed by the n variables in vids1, vids2 are the costs of these edges
        described
399      by c, and vid is the total cost of the circuit, i.e. sum(vids2).
400      */
401     void cst_circuit(int n, int* c, int* vids1, int* vids2, int vid);
402
403     /**
404      Post the constraint that if there exists j (0[U+FFFD]j < |x|) such thatx[j] = t,
405      then there must exist i with i < j such that x[i] = s
406     */
407     void cst_precede(int n, int* vids, int t, int u);
408
409     //=== BOOLVARS ===
410
411     /**
412      Post the constraint that vid1 bool_op vid2 = val.
413      */
414     void cst_boolop_val(int vid1, int bool_op, int vid2, int val);
415
416     /**
417     Post the constraint that elements of vids bool_op val.
418     */
419     void cst_boolop_arr_val(int bool_op, int s, int* vids, int val);
420
421     /**
422     Post the constraint that y is the result of bool_op between all element of vids.
423     */
424     void cst_boolop_arr_var(int bool_op, int s, int* vids, int vid1);
425
426     /**
427      Post the constraint that vid1 bool_op vid2 = vid3.
428      */
429     void cst_boolop_var(int vid1, int bool_op, int vid2, int vid3);
430
431     /**
432      Post a relation constraint between vid and val.
433      */
434     void cst_boolrel_val(int vid, int rel_type, int val);
435
436     /**
437      Post a relation constraint between vid1 and vid2.
438      */
439     void cst_boolrel_var(int vid1, int rel_type, int vid2);
440
441     //=== SETVARS ===
442
443     /**
444      Post the constraint that vid1 set_op vid2 = vid3.
445      */
446     void cst_setop_var(int vid1, int set_op, int vid2, int set_rel, int vid3);
447
448     /**
449     Post the constraint that y set_op x.
450     */
451     void cst_setop_arr(int set_op, int s, int* vid1, int vid2);
452
453     /**
454      Post a relation constraint between vid1 and vid2.
455      */
456     void cst_setrel_var(int vid1, int rel_type, int vid2);
457
458     /**
459     Post a relation constraint between vid1 and vid2.
460     */
461     void cst_setrel_val(int vid1, int rel_type, int* dom, int s);
462
463     /**
464     Post a relation constraint between vid1 and domain dom with reify.
465     */
466     void cst_setrel_val_reify(int vid1, int rel_type, int* dom, int s, int r, int mode);
```

```
467
468     /**
469     Post a relation constraint between vid1 and vid2 with reify.
470     */
471     void cst_setrel_var_reify(int vid1, int rel_type, int vid2, int r, int mode);
472
473     /**
474     Post a dom constraint between vid1 and dom {i,..., j}.
475     */
476     void cst_setdom_ints(int vid1, int rel_type, int i, int j);
477
478     /**
479     Post a dom constraint between vid1 and vid2.
480     */
481     void cst_setdom_set(int vid1, int vid2);
482
483     /**
484     Post a constraint that SetVar vid1 has to be empty
485     */
486     void cst_set_empty(int vid1);
487
488     /**
489      Post a cardinality constraint on vid1.
490      */
491     void cst_card_val(int n, int* vids, int min_card, int max_card);
492
493     /**
494     Post a cardinality constraint on vid1 with intvar vid2
495     */
496     void cst_card_var(int vid1, int vid2);
497
498     /**
499     Post a channeling constraint between vid1 and vid2
500     */
501     void cst_channel(int n1, int* vids1, int n2, int* vids2);
502
503     /**
504     Post a channeling constraint between boolVarArray vid1 and SetVar vid2
505     */
506     void cst_channel_sb(int n1, int* vids1, int vid2);
507
508     /**
509     Return an intvar constrained to the minimum of setvar vid1
510     */
511     int cst_setmin(int vid1);
512
513     /**
514     Return an intvar constrained to the maximum of the setvar vid1
515     */
516     int cst_setmax(int vid1);
517
518     /**
519     Return an intvar constrained to the minimum of the setvar vid1 with reification
520     */
521     void cst_setmin_reify(int vid1, int vid2, int r, int mode);
522
523     /**
524     Return an intvar constrained to the maximum of the setvar vid1 with reification
525     */
526     void cst_setmax_reify(int vid1, int vid2, int r, int mode);
527
528     /**
529     Post a relation constraint beween setvar vid1 and the union of the set in vids
530     */
531     void cst_setunion(int vid1, int n, int* vids);
532
533     /**
534     Post a relation constraint beween setvar vid1 and the union of the set in vids
535     */
536     void cst_element(int set_op, int n, int* vids, int vid1, int vid2);
537
538
539     //=======================================
540     //Branch and bound constraint function =
541     //=======================================
542
543     /*
544      Constrain method for BAB search
545      This is called everytime the solver finds a solution
546     */
547     virtual void constrain(const Space& _b);
548
549     //============================
550     //= Exploration strategies =
551     //============================
552
553     /**
554      Post a branching strategy on the n IntVars in vids, with strategies denoted by
      var_strategy and
555      val_strategy.
556      */
557     void branch(int n, int* vids, int var_strategy, int val_strategy);
558
```

```
559     /**
560      Post a branching strategy on the n BoolVars in vids, with strategies denoted by
         var_strategy and
561      val_strategy.
562      */
563     void branch_b(int n, int* vids, int var_strategy, int val_strategy);
564
565     /**
566      Post a branching strategy on the n SetVars in vids, with strategies denoted by
         var_strategy and
567      val_strategy.
568      */
569     void branch_set(int n, int* vids, int var_strategy, int val_strategy);
570
571     //==================
572     //= Search support =
573     //==================
574
575     void cost(int vid);
576
577     virtual IntVar cost(void) const;
578
579     WSpace(WSpace& s);
580
581     virtual Space* copy(void);
582
583     //=====================
584     //= Getting solutions =
585     //=====================
586
587     /**
588      Return the current values of the variable denoted by vid.
589      */
590     int value(int vid);
591
592     /**
593      Return the current values of the variable denoted by vid.
594      */
595     int value_bool(int vid);
596
597      /**
598      Return the current values of the SetVar denoted by vid.
599      */
600     int* value_set(int vid, int n);
601
602     /**
603      Return the current size of the SetVar denoted by vid.
604      */
605     int value_size(int vid);
606
607
608     /**
609      Return the current values of the n variables denoted by vids.
610      */
611     int* values(int n, int* vids);
612
613     //=====================
614     //= Printing solutions =
615     //=====================
616
617     /**
618      Print the n variables denoted by vids.
619      */
620     void print(int n, int* vids);
621 };
622
623 //==================
624 //= Search options =
625 //==================
626
627 class WTimeStop {
628     protected:
629     Gecode::Search::TimeStop stop;
630     Gecode::Search::TimeStop* stop_ptr;
631
632     public:
633     WTimeStop(int maxTime);
634     ~WTimeStop();
635
636     void reset();
637     TimeStop getStop();
638     TimeStop* getStopPtr();
639 };
640
641 class WSearchOptions {
642     protected:
643         Gecode::Search::Options opts;
644
645     public:
646     WSearchOptions();
647     ~WSearchOptions();
648
649     /**
```

```
650      getter for the opts field
651     */
652     Options getOpts();
653
654     /**
655      Different functions to add options
656      */
657
658     /**
659      set the number of threads to use for parallel search
660      */
661     int setNbThreads(int nThreads);
662
663     /**
664     Set the time stopping mechanism that is to be used during the search to a certain
         duration in ms
665     */
666     void setTimeStop(WTimeStop* timestop);
667 };
668
669
670
671 //=================
672 //= Search engine =
673 //=================
674
675 class WbabEngine { // new version
676 protected:
677     BAB<WSpace>* bab;
678 public:
679     WbabEngine(WSpace* sp, Options opts);
680     ~WbabEngine();
681
682     /**
683      Search the next solution for this search engine.
684      */
685     WSpace* next();
686     int stopped();
687 };
688
689 class WdfsEngine {
690 protected:
691     DFS<WSpace>* dfs;
692 public:
693     WdfsEngine(WSpace* sp, Options opts);
694     ~WdfsEngine();
695
696     /**
697      Search the next solution for this search engine.
698      */
699     WSpace* next();
700     int stopped();
701 };
702
703 #endif
```

C.1.2 space_wrapper.cpp

```
1  #include "headers/space_wrapper.hpp"
2  #include <iostream>
3  #include <fstream>
4
5  using namespace Gecode;
6  using namespace Gecode::Int;
7  using namespace Gecode::Set;
8  using namespace std;
9
10 /*
11 To Print value to a file :
12     ofstream myfile;
13     myfile.open ("/home/amdiels/Bureau/example.txt", ios::app);
14     myfile << value << endl;
15     myfile.close();
16 */
17
18 /**
19  Default constructor
20  */
21 WSpace::WSpace() {
22     i_size = 0;
23     b_size = 0;
24     s_size = 0;
25 }
26
27 //=====================
28 //= Variables from idx =
29 //=====================
30
31 /**
32  Return the IntVar contained in int_vars at index vid
```

```
 */
IntVar WSpace::get_int_var(int vid) {
    return int_vars.at(vid);
}

/**
 Return the BoolVar contained in bool_vars at index vid
 */
BoolVar WSpace::get_bool_var(int vid) {
    return bool_vars.at(vid);
}

/**
Return the SetVar contained in set_vars at index vid.
*/
SetVar WSpace::get_set_var(int vid){
    return set_vars.at(vid);
}

//======================
//= Args for methods =
//======================

/**
 Return an IntVarArgs of size n, containing the n IntVars contained in
 int_vars at indices vids.
 */
IntVarArgs WSpace::int_var_args(int n, int* vids) {
    IntVarArgs x(n);
    for(int i = 0; i < n; i++)
        x[i] = get_int_var(vids[i]);
    return x;
}

/**
 Return an BoolVarArgs of size n, containing the n BoolVars contained in
 bool_vars at indices vids.
 */
BoolVarArgs WSpace::bool_var_args(int n, int* vids) {
    BoolVarArgs x(n);
    for(int i = 0; i < n; i++)
        x[i] = get_bool_var(vids[i]);
    return x;
}

/**
 Return a SetVarArgs of size n, containing the n SetVars contained in
 set_vars at indices vids.
 */
SetVarArgs WSpace::set_var_args(int n, int* vids) {
    SetVarArgs x(n);
    for(int i = 0; i < n; i++)
        x[i] = get_set_var(vids[i]);
    return x;
}

/**
 Return an IntArgs of size n, containing the n values in vals
*/
IntArgs WSpace::int_args(int n, int* vals) {
    IntArgs c(n);
    for(int i = 0; i < n; i++)
        c[i] = vals[i];
    return c;
}

/**
 Return the expression int_rel(vid, val)
*/
BoolVar WSpace::bool_expr_val(int vid, int int_rel, int val) {
    switch(int_rel) {
        case B_EQ: return expr(*this, get_int_var(vid) == val);
        case B_NQ: return expr(*this, get_int_var(vid) != val);
        case B_LE: return expr(*this, get_int_var(vid) < val);
        case B_LQ: return expr(*this, get_int_var(vid) <= val);
        case B_GQ: return expr(*this, get_int_var(vid) >= val);
        case B_GR: return expr(*this, get_int_var(vid) > val);
        default:
            cout << "Wrong expression type in BoolVar creation." << endl;
            return BoolVar();
    }
}

/**
 Return the expression int_rel(vid1, vid2)
*/
BoolVar WSpace::bool_expr_var(int vid1, int int_rel, int vid2) {
    switch(int_rel) {
        case B_EQ: return expr(*this, get_int_var(vid1) == get_int_var(vid2));
        case B_NQ: return expr(*this, get_int_var(vid1) != get_int_var(vid2));
        case B_LE: return expr(*this, get_int_var(vid1) < get_int_var(vid2));
        case B_LQ: return expr(*this, get_int_var(vid1) <= get_int_var(vid2));
        case B_GQ: return expr(*this, get_int_var(vid1) >= get_int_var(vid2));
```

```
126         case B_GR: return expr(*this, get_int_var(vid1) > get_int_var(vid2));
127         default:
128             cout << "Wrong expression type in BoolVar creation." << endl;
129             return BoolVar();
130     }
131 }
132 
133 
134 //============================
135 //= Variables and domains =
136 //============================
137 
138 /**
139  Add an IntVar to the WSpace ranging from min to max.
140  In practice, push a new IntVar at the end of the vector int_vars.
141  Return the index of the IntVar in int_vars
142  */
143 int WSpace::add_intVar(int min, int max) {
144     int_vars.push_back(IntVar(*this, min, max));
145     return i_size++;
146 }
147 
148 /**
149  Add an IntVar to the WSpace with domain dom of size s.
150  In practice, push a new IntVar at the end of the vector int_vars.
151  Return the index of the IntVar in int_vars
152  */
153 int WSpace::add_intVarWithDom(int s, int* dom) {
154     int_vars.push_back(IntVar(*this, IntSet(dom, s)));
155     return i_size++;
156 }
157 
158 /**
159  Add n IntVars to the WSpace ranging from min to max.
160  In practice, push n new IntVars at the end of the vector int_vars.
161  Return the indices of the IntVars in int_vars.
162  */
163 int* WSpace::add_intVarArray(int n, int min, int max) {
164     int* vids = new int[n];
165     for(int i = 0; i < n; i++)
166         vids[i] = this->add_intVar(min, max);
167     return vids;
168 }
169 
170 /**
171  Add n IntVars to the WSpace with domain dom of size s.
172  In practice, push n new IntVars at the end of the vector int_vars.
173  Return the indices of the IntVars in int_vars.
174  */
175 int* WSpace::add_intVarArrayWithDom(int n, int s, int* dom) {
176     int* vids = new int[n];
177     for(int i = 0; i < n; i++)
178         vids[i] = this->add_intVarWithDom(s, dom);
179     return vids;
180 }
181 
182 /**
183  Define which variables are to be the solution so they can be accessed to add a constraint
     with bab
184  */
185 void WSpace::set_as_solution_variables(int n, int* vids){
186     solution_variable_indexes = new int[n];
187     for (int i=0; i<n; i++) {
188         solution_variable_indexes[i]=vids[i];
189     }
190     var_sol_size = n;
191 }
192 
193 /**
194  Define the percentage of the solution that should change when searching for the next
     solution with BAB
195  */
196 void WSpace::set_percent_diff(int diff){
197     percent_diff = diff;
198 }
199 
200 /**
201  Return the number of IntVars in the space.
202  */
203 int WSpace::nvars() {
204     return i_size;
205 }
206 
207 /**
208  Add a BoolVar to the WSpace ranging from min to max.
209  In practice, push a new BoolVar at the end of the vector bool_vars.
210  Return the index of the BoolVar in bool_vars
211  */
212 int WSpace::add_boolVar(int min, int max) {
213     bool_vars.push_back(BoolVar(*this, min, max));
214     return b_size++;
215 }
216 
```

```
217 /**
218  Add n BoolVars to the WSpace ranging from min to max.
219  In practice, push n new BoolVars at the end of the vector bool_vars.
220  Return the indices of the BoolVars in bool_vars.
221  */
222 int* WSpace::add_boolVarArray(int n, int min, int max) {
223     int* vids = new int[n];
224     for(int i = 0; i < n; i++)
225         vids[i] = this->add_boolVar(min, max);
226     return vids;
227 }
228
229 /**
230  Add a BoolVar to the WSpace corresponding to the evaluation of int_rel(vid, val).
231  In practice, push a new BoolVar at the end of the vector bool_vars.
232  Return the index of the BoolVar in bool_vars
233  */
234 int WSpace::add_boolVar_expr_val(int vid, int int_rel, int val) {
235     bool_vars.push_back(bool_expr_val(vid, int_rel, val));
236     return b_size++;
237 }
238
239 /**
240  Add a BoolVar to the WSpace corresponding to the evaluation of int_rel(vid1, vid2).
241  In practice, push a new BoolVar at the end of the vector bool_vars.
242  Return the index of the BoolVar in bool_vars
243  */
244 int WSpace::add_boolVar_expr_var(int vid1, int int_rel, int vid2) {
245     bool_vars.push_back(bool_expr_var(vid1, int_rel, vid2));
246     return b_size++;
247 }
248
249 /**
250 Add a SetVar to the WSpace initialized with n integer from array r.
251 In practice, push a new SetVar at the end of the vector set_vars.
252 Return the index of the SetVar in set_vars.
253 */
254 int WSpace::add_setVar(int lub_min, int lub_max, int card_min, int card_max) {
255     set_vars.push_back(SetVar(*this,IntSet::empty, lub_min, lub_max, card_min, card_max));
256     return s_size++;
257 }
258
259 /**
260  Add n SetVars to the WSpace ranging with cardinality card_min to card_max.
261  In practice, push n new SetVars at the end of the vector set_vars.
262  Return the indices of the SetVars in set_vars.
263  */
264 int* WSpace::add_setVarArray(int n, int lub_min, int lub_max, int card_min, int card_max) {
265     int* vids = new int[n];
266     for(int i = 0; i < n; i++)
267         vids[i] = this->add_setVar(lub_min, lub_max, card_min, card_max);
268     return vids;
269 }
270
271
272 //=========================
273 //= Posting constraints =
274 //=========================
275
276 //=== INTVAR ===
277
278 /**
279  Post a relation constraint between the IntVar denoted by vid and the val.
280  */
281 void WSpace::cst_val_rel(int vid, int rel_type, int val) {
282     rel(*this, get_int_var(vid), (IntRelType) rel_type, val);
283 }
284
285 /**
286  Post a relation constraint between the IntVars denoted by vid1 and vid2.
287 */
288 void WSpace::cst_var_rel(int vid1, int rel_type, int vid2) {
289     rel(*this, get_int_var(vid1), (IntRelType) rel_type, get_int_var(vid2));
290 }
291
292 /**
293  Post a relation constraint between the IntVars denoted by vid1 and vid2 with reification.
294 */
295 void WSpace::cst_var_rel_reify(int vid1, int rel_type, int vid2, int vid3, int mode) {
296     rel(*this, get_int_var(vid1), (IntRelType) rel_type, get_int_var(vid2), Reify(
      get_bool_var(vid3), (ReifyMode) mode));
297 }
298
299 /**
300  Post a relation constraint between the IntVars denoted by vid1 and vid2 with reification.
301 */
302 void WSpace::cst_val_rel_reify(int vid1, int rel_type, int val, int vid2, int mode) {
303     rel(*this, get_int_var(vid1), (IntRelType) rel_type, val, Reify(get_bool_var(vid2), (
      ReifyMode) mode));
304 }
305
306 /**
307  Post a relation constraint between the n IntVars denoted by vids and the val.
```

```
*/
void WSpace::cst_arr_val_rel(int n, int* vids, int rel_type, int val) {
    rel(*this, int_var_args(n, vids), (IntRelType) rel_type, val);
}

/**
 Post a relation constraint between the n IntVars denoted by vids and the the IntVar vid.
*/
void WSpace::cst_arr_var_rel(int n, int* vids, int rel_type, int vid) {
    rel(*this, int_var_args(n, vids), (IntRelType) rel_type, get_int_var(vid));
}

/**
 Post a relation constraint between the n IntVars denoted by vids.
*/
void WSpace::cst_arr_rel(int n, int* vids, int rel_type) {
    rel(*this, int_var_args(n, vids), (IntRelType) rel_type);
}

/**
 Post a lexicographic relation constraint between the n1 IntVars denoted by vids1 and
 the n2 IntVars denoted by vids2.
*/
void WSpace::cst_arr_arr_rel(int n1, int* vids1, int rel_type, int n2, int* vids2) {
    rel(*this, int_var_args(n1, vids1), (IntRelType) rel_type, int_var_args(n2, vids2));
}

/**
 Post the constraint that all IntVars denoted by vids are distinct
 */
void WSpace::cst_distinct(int n, int* vids) {
    distinct(*this, int_var_args(n, vids));
}

/**
 Post the linear constraint [c]*[vids] rel_type val.
 */
void WSpace::cst_val_linear(int n, int* c, int* vids, int rel_type, int val) {
    linear(*this, int_args(n, c), int_var_args(n, vids), (IntRelType) rel_type, val);
}

/**
 Post the linear constraint [c]*[vids] rel_type vid.
*/
void WSpace::cst_var_linear(int n, int* c, int* vids, int rel_type, int vid) {
    linear(*this, int_args(n, c), int_var_args(n, vids), (IntRelType) rel_type, get_int_var(
     vid));
}

/**
 Post the constraint that |vid1| = vid2.
 */
void WSpace::cst_abs(int vid1, int vid2) {
    abs(*this, get_int_var(vid1), get_int_var(vid2));
}

/**
 Post the constaraint that dom(vid) = d.
 */
void WSpace::cst_dom(int vid, int n, int* d) {
    dom(*this, get_int_var(vid), IntSet(d, n));
}

/**
 Post the constraint that vid is included in {vids[0], ..., vids[n-1]}
*/
void WSpace::cst_member(int n, int* vids, int vid) {
    member(*this, int_var_args(n, vids), get_int_var(vid));
}

/**
 Post the constraint that vid1 / vid2 = vid3.
 */
void WSpace::cst_div(int vid1, int vid2, int vid3) {
    div(*this, get_int_var(vid1), get_int_var(vid2), get_int_var(vid3));
}

/**
 Post the constraint that vid1 % vid2 = vid3.
 */
void WSpace::cst_mod(int vid1, int vid2, int vid3) {
    mod(*this, get_int_var(vid1), get_int_var(vid2), get_int_var(vid3));
}

/**
 Post the constraint that vid1 / vid2 = vid3
 and vid1 % vid2 = div4
 */
void WSpace::cst_divmod(int vid1, int vid2, int vid3, int vid4) {
    divmod(*this, get_int_var(vid1), get_int_var(vid2), get_int_var(vid3), get_int_var(vid4))
     ;
}

```

```
399 /**
400  Post the constraint that min(vid1, vid2) = vid3.
401  */
402 void WSpace::cst_min(int vid1, int vid2, int vid3) {
403     Gecode::min(*this, get_int_var(vid1), get_int_var(vid2), get_int_var(vid3));
404 }
405
406 /**
407  Post the constraint that vid = min(vids).
408  */
409 void WSpace::cst_arr_min(int n, int* vids, int vid) {
410     Gecode::min(*this, int_var_args(n, vids), get_int_var(vid));
411 }
412
413 /**
414  Post the constraint that vid = argmin(vids).
415  */
416 void WSpace::cst_argmin(int n, int* vids, int vid) {
417     Gecode::argmin(*this, int_var_args(n, vids), get_int_var(vid));
418 }
419
420 /**
421  Post the constraint that max(vid1, vid2) = vid3.
422  */
423 void WSpace::cst_max(int vid1, int vid2, int vid3) {
424     Gecode::max(*this, get_int_var(vid1), get_int_var(vid2), get_int_var(vid3));
425 }
426
427 /**
428  Post the constraint that vid = max(vids).
429  */
430 void WSpace::cst_arr_max(int n, int* vids, int vid) {// corrig[U+FFFD]
431     Gecode::max(*this, int_var_args(n, vids), get_int_var(vid));
432 }
433
434 /**
435  Post the constraint that vid = argmax(vids).
436  */
437 void WSpace::cst_argmax(int n, int* vids, int vid) {
438     Gecode::argmax(*this, int_var_args(n, vids), get_int_var(vid));
439 }
440
441 /**
442  Post the constraint that vid1 * vid2 = vid3.
443  */
444 void WSpace::cst_mult(int vid1, int vid2, int vid3) {
445     mult(*this, get_int_var(vid1), get_int_var(vid2), get_int_var(vid3));
446 }
447
448 /**
449  Post the constraint that sqr(vid1) = vid2.
450  */
451 void WSpace::cst_sqr(int vid1, int vid2) {
452     sqr(*this, get_int_var(vid1), get_int_var(vid2));
453 }
454
455 /**
456  Post the constraint that sqrt(vid1) = vid2.
457  */
458 void WSpace::cst_sqrt(int vid1, int vid2) {
459     Gecode::sqrt(*this, get_int_var(vid1), get_int_var(vid2));
460 }
461
462 /**
463  Post the constraint that pow(vid1, n) = vid2.
464  */
465 void WSpace::cst_pow(int vid1, int n, int vid2) {
466     Gecode::pow(*this, get_int_var(vid1), n, get_int_var(vid2));
467 }
468
469 /**
470  Post the constraint that nroot(vid1, n) = vid2.
471  */
472 void WSpace::cst_nroot(int vid1, int n, int vid2) {
473     nroot(*this, get_int_var(vid1), n, get_int_var(vid2));
474 }
475
476 /**
477  Post the constraint that vid = sum(vids).
478  */
479 void WSpace::cst_sum(int vid, int n, int* vids) {
480     rel(*this, get_int_var(vid), IRT_EQ, expr(*this, sum(int_var_args(n, vids))));
481 }
482
483 /**
484  Post the constraint that the number of variables in vids equal to val1 has relation rel_type
485  with val2.
486  */
487 void WSpace::cst_count_val_val(int n, int* vids, int val1, int rel_type, int val2) {
488     count(*this, int_var_args(n, vids), val1, (IntRelType) rel_type, val2);
489 }
490
491 /**
```

```
492  Post the constraint that the number of variables in vids equal to val has relation rel_type
493  with vid.
494  */
495 void WSpace::cst_count_val_var(int n, int* vids, int val, int rel_type, int vid) {
496     count(*this, int_var_args(n, vids), val, (IntRelType) rel_type, get_int_var(vid));
497 }
498
499 /**
500  Post the constraint that the number of variables in vids equal to vid has relation rel_type
501  with val.
502  */
503 void WSpace::cst_count_var_val(int n, int* vids, int vid, int rel_type, int val) {// corrig[U+FFFD]
504     count(*this, int_var_args(n, vids), get_int_var(vid), (IntRelType) rel_type, val);
505 }
506
507 /**
508  Post the constraint that the number of variables in vids equal to vid1 has relation rel_type
509  with vid2.
510  */
511 void WSpace::cst_count_var_var(int n, int* vids, int vid1, int rel_type, int vid2) {
512     count(*this, int_var_args(n, vids), vid1, (IntRelType) rel_type, get_int_var(vid2));
513 }
514
515 /**
516  Post the constraint that the number of variables in vids in the set set has relation
       rel_type with val
517  */
518 void WSpace::cst_count_var_set_val(int n, int*vids, int s, int* set, int rel_type, int val){
       // ajout[U+FFFD]
519     count(*this, int_var_args(n, vids), IntSet(set, s), (IntRelType) rel_type, val);
520 }
521
522 /**
523  Post the constraint that the number of variables in vids where vars[i] = c[i] and c is an
       array of integers has rel_type to val
524  */
525 void WSpace::cst_count_array_val(int n, int*vids, int* c, int rel_type, int val){
526     count(*this, int_var_args(n, vids), int_args(n, c), (IntRelType) rel_type, val);
527 }
528
529 /**
530   Post the constraint that the number of occurences of s-set in every subsequence of length
531   val1 in vids must be higher than val2 and lower than val3
532  */
533 void WSpace::cst_sequence_var(int n, int*vids, int s, int* set, int val1, int val2, int val3)
       {// ajout[U+FFFD]
534     sequence(*this, int_var_args(n, vids), IntSet(set, s), val1, val2, val3);
535 }
536
537
538 /**
539  Post the constraint that the number of distinct values in the n variables denoted by vids
540  has the given rel_type relation with the variable vid.
541  */
542 void WSpace::cst_nvalues(int n, int* vids, int rel_type, int vid) {
543     nvalues(*this, int_var_args(n, vids), (IntRelType) rel_type, get_int_var(vid));
544 }
545
546 /**
547  Post the constraint that values of vids1 are the edges of an hamiltonian circuit in
548  the graph formed by the n variables in vids1, vids2 are the costs of these edges described
549  by c, and vid is the total cost of the circuit, i.e. sum(vids2).
550  */
551 void WSpace::cst_circuit(int n, int* c, int* vids1, int* vids2, int vid) {
552     circuit(*this, int_args(n*n, c), int_var_args(n, vids1), int_var_args(n, vids2),
       get_int_var(vid));
553 }
554
555 /**
556  Post the constraint that if there exists j (0[U+FFFD]j < |x|) such that x[j] = u,
557  then there must exist i with i < j such that x[i] = s
558 */
559 void WSpace::cst_precede(int n, int* vids, int s, int u){
560     precede(*this, int_var_args(n, vids), s, u);
561 }
562
563
564 //=== BOOLVAR ===
565
566 /**
567  Post the constraint that vid1 bool_op vid2 = val.
568  */
569 void WSpace::cst_boolop_val(int vid1, int bool_op, int vid2, int val) {
570     rel(*this, get_bool_var(vid1), (BoolOpType) bool_op, get_bool_var(vid2), val);
571 }
572
573 /**
574  Post the constraint that elements of vids bool_op val.
575  */
576 void WSpace::cst_boolop_arr_val(int bool_op, int s, int* vids, int val) {
577     rel(*this, (BoolOpType) bool_op, bool_var_args(s, vids), val);
578 }
579
```

```
580 /**
581  Post the constraint that y is the result of bool_op between all element of vids.
582  */
583 void WSpace::cst_boolop_arr_var(int bool_op, int s, int* vids, int vid1) {
584     rel(*this, (BoolOpType) bool_op, bool_var_args(s, vids), get_bool_var(vid1));
585 }
586
587 /**
588  Post the constraint that vid1 bool_op vid2 = vid3.
589  */
590 void WSpace::cst_boolop_var(int vid1, int bool_op, int vid2, int vid3) {
591     rel(*this, get_bool_var(vid1), (BoolOpType) bool_op, get_bool_var(vid2), get_bool_var(
      vid3));
592 }
593
594 /**
595  Post a relation constraint between vid and val.
596  */
597 void WSpace::cst_boolrel_val(int vid, int rel_type, int val) {
598     rel(*this, get_bool_var(vid), (IntRelType) rel_type, val);
599 }
600
601 /**
602  Post a relation constraint between vid1 and vid2.
603  */
604 void WSpace::cst_boolrel_var(int vid1, int rel_type, int vid2) {
605     rel(*this, get_bool_var(vid1), (IntRelType) rel_type, get_bool_var(vid2));
606 }
607
608 //=== SETVAR ===
609
610 /**
611  Post the constraint that vid1 set_op vid2 = vid3.
612  */
613 void WSpace::cst_setop_var(int vid1, int set_op, int vid2, int set_rel, int vid3) {
614     rel(*this, get_set_var(vid1), (SetOpType) set_op, get_set_var(vid2), (SetRelType) set_rel
      , get_set_var(vid3));
615 }
616
617 /**
618  Post the constraint that y set_op x.
619  */
620 void WSpace::cst_setop_arr(int set_op, int s, int* vid1, int vid2) {
621     rel(*this, (SetOpType) set_op, set_var_args(s, vid1), get_set_var(vid2));
622 }
623
624 /**
625  Post a relation constraint between vid1 and vid2.
626  */
627 void WSpace::cst_setrel_var(int vid1, int rel_type, int vid2) {
628     rel(*this, get_set_var(vid1), (SetRelType) rel_type, get_set_var(vid2));
629 }
630
631 /**
632  Post a relation constraint between vid1 and domain dom.
633  */
634 void WSpace::cst_setrel_val(int vid1, int rel_type, int* domain, int s) {
635     dom(*this, get_set_var(vid1), (SetRelType) rel_type, IntSet(domain, s));
636 }
637
638 /**
639  Post a relation constraint between vid1 and domain dom with a reify variable
640  */
641 void WSpace::cst_setrel_val_reify(int vid1, int rel_type, int* domain, int s, int r, int mode
      ) {
642     dom(*this, get_set_var(vid1), (SetRelType) rel_type, IntSet(domain, s), Reify(
      get_bool_var(r), (ReifyMode) mode));
643 }
644
645 /**
646  Post a relation constraint between vid1 and vid2 with a reify variable
647  */
648 void WSpace::cst_setrel_var_reify(int vid1, int rel_type, int vid2, int r, int mode) {
649     rel(*this, get_set_var(vid1), (SetRelType) rel_type, get_set_var(vid2), Reify(
      get_bool_var(r), (ReifyMode) mode));
650 }
651
652 /**
653  Post a constraint that SetVar vid1 has to be empty
654  */
655 void WSpace::cst_set_empty(int vid1) {
656     dom(*this, get_set_var(vid1), (SetRelType) 0, IntSet::empty);
657 }
658
659 /**
660  Post a dom constraint between vid1 and dom {i,..., j}.
661  */
662 void WSpace::cst_setdom_ints(int vid1, int rel_type, int i, int j) {
663     dom(*this, get_set_var(vid1), (SetRelType) rel_type, i, j);
664 }
665
666 /**
667  Post a dom constraint between vid1 and dom vid2.
```

```
 */
void WSpace::cst_setdom_set(int vid1, int vid2) {
    dom(*this, get_set_var(vid1), get_set_var(vid2));
}

/**
 Post a cardinality constraint on vids with 2 bounds min_card max_card
 */
void WSpace::cst_card_val(int n, int* vids, int min_card, int max_card) {
    cardinality(*this, set_var_args(n, vids), min_card, max_card);
}

/**
 Post a cardinality constraint on vid1 with intvar vid2
 */
void WSpace::cst_card_var(int vid1, int vid2) {
    cardinality(*this, get_set_var(vid1), get_int_var(vid2));
}

/**
Post a channeling constraint between vid1 and vid2
*/
void WSpace::cst_channel(int n1, int* vids1, int n2, int* vids2){
    channel(*this, set_var_args(n1, vids1), set_var_args(n2, vids2));
}

/**
Post a channeling constraint between boolVarArray vid1 and SetVar vid2
*/
void WSpace::cst_channel_sb(int n1, int* vids1, int vid2){
    channel(*this, bool_var_args(n1, vids1), get_set_var(vid2));
}

/**
Return an intvar constrained to the minimum of the setvar vid1
*/
int WSpace::cst_setmin(int vid1){
    int_vars.push_back(expr(*this, min(get_set_var(vid1))));
    return i_size++ ;
}

/**
Return an intvar constrained to the maximum of the setvar vid1
*/
int WSpace::cst_setmax(int vid1){
    int_vars.push_back(expr(*this, max(get_set_var(vid1))));
    return i_size++ ;
}

/**
Post a constraint between vid2 and the minimum of the setvar vid1 with reification
*/
void WSpace::cst_setmin_reify(int vid1, int vid2, int r, int mode){
    min(*this, get_set_var(vid1), get_int_var(vid2), Reify(get_bool_var(r), (ReifyMode) mode)
     );
}

/**
Post a constraint between vid2 and the maximum of the setvar vid1 with reification
*/
void WSpace::cst_setmax_reify(int vid1, int vid2, int r, int mode){
    max(*this, get_set_var(vid1), get_int_var(vid2), Reify(get_bool_var(r), (ReifyMode) mode)
     );
}

/**
Post a relation constraint beween setvar vid1 and the union of the set in vids
*/
void WSpace::cst_setunion(int vid1, int n, int* vids){
    rel(*this, SOT_UNION, set_var_args(n, vids), get_set_var(vid1));
}

/**
Post an element constraints
*/
void WSpace::cst_element(int set_op, int n, int* vids, int vid1, int vid2){
    element(*this, (SetOpType) set_op, set_var_args(n, vids), get_set_var(vid1), get_set_var(
     vid2));
}

//=========================================
//Branch and bound constraint function =
//=========================================

/*
 Constrain method for BAB search
 This is called everytime the solver finds a solution
 This is a virtual method as declared in space_wrapper.h
*/
void WSpace::constrain(const Space& _b) {
    const WSpace& b = static_cast<const WSpace&>(_b);

    SetVarArgs bvars(b.var_sol_size);
```

```
758     for(int i = 0; i < b.var_sol_size; i++)
759         bvars[i] = (b.set_vars).at((b.solution_variable_indexes)[i]);
760 
761     SetVarArgs vars(b.var_sol_size);
762     for(int i = 0; i < b.var_sol_size; i++)
763         vars[i] = (set_vars).at((solution_variable_indexes)[i]);
764 
765     for(int i=0; i<b.var_sol_size; i++){
766       if((rand()%100)< b.percent_diff){
767       SetVar tmp(bvars[i]);
768       rel(*this,(vars[i] != tmp) );
769       }
770     }
771   }
772 
773 //============================
774 //= Exploration strategies =
775 //============================
776 
777 /**
778  Post a branching strategy on the variables in vids, with strategies denoted by var_strategy
        and
779  val_strategy.
780  var_strategy:
781     - 0 : INT_VAR_SIZE_MIN()
782     - 1 : INT_VAR_RND(r)
783     - 2 : INT_VAR_DEGREE_MAX()
784     - 3 : INT_VAR_NONE()
785  val_strategy:
786     - 0 : INT_VAL_MIN()
787     - 1 : INT_VAL_RND(r)
788     - 2 : INT_VAL_SPLIT_MIN()
789     - 3 : INT_VAL_SPLIT_MAX()
790     - 4 : INT_VAL_MED()
791  */
792 void WSpace::branch(int n, int* vids, int var_strategy, int val_strategy) {
793     IntVarBranch var_strat;
794     IntValBranch val_strat;
795 
796     Rnd r1(1U);
797     Rnd r2(3U);
798 
799     //determine the variable strategy
800     if(var_strategy == 0){//INT_VAR_SIZE_MIN()
801         var_strat = INT_VAR_SIZE_MIN();
802     }
803     else if(var_strategy == 1){//INT_VAR_RND(r1)
804         var_strat = INT_VAR_RND(r1);
805     }
806     else if(var_strategy == 2){//INT_VAR_DEGREE_MAX()
807         var_strat = INT_VAR_DEGREE_MAX();
808     }
809     else if(var_strategy == 3){//INT_VAR_NONE()
810         var_strat = INT_VAR_NONE();
811     }
812 
813     //determine the value strategy
814     if(val_strategy == 0){//INT_VAL_MIN()
815         val_strat = INT_VAL_MIN();
816     }
817     else if(val_strategy == 1){//INT_VAL_RND(r2)
818         val_strat = INT_VAL_RND(r2);
819     }
820     else if(val_strategy == 2){//INT_VAL_SPLIT_MIN()
821         val_strat = INT_VAL_SPLIT_MIN();
822     }
823     else if(val_strategy == 3){//INT_VAL_SPLIT_MAX()
824         val_strat = INT_VAL_SPLIT_MAX();
825     }
826     else if(val_strategy == 4){//INT_VAL_MED()
827         val_strat = INT_VAL_MED();
828     }
829 
830     Gecode::branch(*this, int_var_args(n, vids), var_strat, val_strat);
831 }
832 
833 
834 /**
835  Post a branching strategy on the n BoolVars in vids, with strategies denoted by var_strategy
        and
836  val_strategy.
837  */
838 void WSpace::branch_b(int n, int* vids, int var_strategy, int val_strategy) {
839     Gecode::branch(*this, bool_var_args(n, vids), BOOL_VAR_NONE(), BOOL_VAL_MIN()); //default
        for now
840 }
841 
842 /**
843  Post a branching strategy on the n SetVars in vids.
844  */
845 void WSpace::branch_set(int n, int* vids, int var_strategy, int val_strategy) {
846     SetVarBranch var_strat;
847     SetValBranch val_strat;
```

```
848
849     Rnd r1(1U);
850     Rnd r2(3U);
851
852     //determine the variable strategy
853     if(var_strategy == 0){//SET_VAR_SIZE_MIN()
854         var_strat = SET_VAR_SIZE_MIN();
855     }
856     else if(var_strategy == 1){//SET_VAR_RND(r1)
857         var_strat = SET_VAR_RND(r1);
858     }
859     else if(var_strategy == 2){//SET_VAR_DEGREE_MAX()
860         var_strat = SET_VAR_DEGREE_MAX();
861     }
862     else if(var_strategy == 3){//SET_VAR_NONE()
863         var_strat = SET_VAR_NONE();
864     }
865
866     //determine the value strategy
867     if(val_strategy == 0){//SET_VAL_MIN()
868         val_strat = SET_VAL_MIN_INC();
869     }
870     else if(val_strategy == 1){//SET_VAL_RND(r2)
871         val_strat = SET_VAL_RND_INC(r2);
872     }
873     else if(val_strategy == 2){//SET_VAL_SPLIT_MIN()
874         val_strat = SET_VAL_MIN_EXC();
875     }
876     else if(val_strategy == 3){//SET_VAL_SPLIT_MAX()
877         val_strat = SET_VAL_RND_EXC(r2);
878     }
879     else if(val_strategy == 4){//SET_VAL_MED()
880         val_strat = SET_VAL_MED_INC();
881     }
882
883     Gecode::branch(*this, set_var_args(n, vids), var_strat, val_strat);
884 }
885
886 //====================
887 //= Search support =
888 //====================
889
890 /**
891  Define which variable, denoted by vid, will be considered as the cost.
892  */
893 void WSpace::cost(int vid) {
894     cost_id = vid;
895 }
896
897 IntVar WSpace::cost(void) const {
898     return int_vars.at(cost_id);
899 }
900
901 WSpace::WSpace(WSpace& s): IntMinimizeSpace(s), int_vars(s.i_size), bool_vars(s.b_size),
      set_vars(s.s_size), i_size(s.i_size), b_size(s.b_size), s_size(s.s_size), cost_id(s.
      cost_id),
902                            var_sol_size(s.var_sol_size), solution_variable_indexes(s.
      solution_variable_indexes), percent_diff(s.percent_diff) {
903     //IntVars update
904     vector<IntVar>::iterator itd, its;
905     for(itd = int_vars.begin(), its = s.int_vars.begin(); itd != int_vars.end(); ++itd, ++its
      )
906         itd->update(*this, *its);
907
908     //BoolVars update
909     vector<BoolVar>::iterator btd, bts;
910     for(btd = bool_vars.begin(), bts = s.bool_vars.begin(); btd != bool_vars.end(); ++btd, ++
      bts)
911         btd->update(*this, *bts);
912
913     //SetVars update
914     vector<SetVar>::iterator std, sts;
915     for(std = set_vars.begin(), sts = s.set_vars.begin(); std != set_vars.end(); ++std, ++sts
      )
916         std->update(*this, *sts);
917
918     //Solutions for BAB
919     for(int i=0; i<var_sol_size; i++)
920         s.solution_variable_indexes[i]=solution_variable_indexes[i];
921
922 }
923
924 Space* WSpace::copy(void) {
925     return new WSpace(*this);
926 }
927
928 //=====================
929 //= Getting solutions =
930 //=====================
931
932 /**
933  Return the current values of the variable denoted by vid.
934  */
```

```
935 int WSpace::value(int vid) {
936     return get_int_var(vid).val();
937 }
938
939 /**
940  Return the current values of the variable denoted by vid.
941  */
942 int WSpace::value_bool(int vid) {
943     return get_bool_var(vid).val();
944 }
945
946
947 /**
948  Return the current values of the variable denoted by vid.
949  */
950 int* WSpace::value_set(int vid, int n) {
951     SetVar sv = get_set_var(vid);
952     int* vals = new int[n] ;
953     int i = 0 ;
954     for (SetVarGlbValues d(sv);d();++d){
955         vals[i] = d.val() ;
956         i++ ;
957     }
958     return vals;
959 }
960
961 int WSpace::value_size(int vid) {
962     return get_set_var(vid).glbSize() ;
963 }
964
965 /**
966  Return the current values of the n variables denoted by vids.
967  */
968 int* WSpace::values(int n, int* vids) {
969     int* vals = new int[n];
970     for(int i = 0; i < n; i++)
971         vals[i] = get_int_var(vids[i]).val();
972     return vals;
973 }
974
975 //=======================
976 //= Printing solutions =
977 //=======================
978
979 void WSpace::print(int n, int* vids) {
980     std::cout << "{";
981     for(int i = 0; i < n; i++) {
982         std::cout << get_int_var(vids[i]);
983         if(i < n - 1) std::cout << ", ";
984     }
985     std::cout << "}" << std::endl;
986 }
987
988 //=============================
989 //= Search options managment =
990 //=============================
991
992 // ==== TIME STOP OBJECT ====
993
994 /**
995  Default constructor
996  */
997 WTimeStop::WTimeStop(int maxTime) : stop(Gecode::Search::TimeStop(maxTime)) {
998     stop_ptr = &stop;
999 }
1000
1001 WTimeStop::~WTimeStop(){
1002
1003 }
1004
1005 TimeStop WTimeStop::getStop(){
1006     return stop;
1007 }
1008
1009 TimeStop* WTimeStop::getStopPtr(){
1010     return stop_ptr;
1011 }
1012
1013 /**
1014  Reset the time value of the time stop object
1015  */
1016 void WTimeStop::reset(){
1017     stop.reset();
1018 }
1019
1020 // ==== OPTIONS OBJECT ====
1021
1022 /**
1023  Default constructor
1024  */
1025 WSearchOptions::WSearchOptions(){
1026
1027 }
```

```
1028
1029 WSearchOptions::~WSearchOptions(){
1030
1031 }
1032
1033 /**
1034  getter for the opts field
1035  */
1036 Options WSearchOptions::getOpts(){
1037     return opts;
1038 }
1039 /**
1040  set the number of threads to use for parallel search
1041  */
1042 int WSearchOptions::setNbThreads(int nThreads){
1043     opts.threads = nThreads;
1044     return opts.threads;
1045 }
1046
1047 /**
1048  Set the time stopping mechanism that is to be used during the search to a certain duration
         in ms
1049  Takes a WTimeStop object as argument, and sets the WSearchOptions object's opts.stop field
         to the TimeStop pointer of the WTimeStop object
1050  */
1051 void WSearchOptions::setTimeStop(WTimeStop* timestop){
1052     opts.stop = timestop->getStopPtr();
1053 }
1054
1055 //====================
1056 //= Search engine =
1057 //====================
1058
1059 /*
1060  Branch and bound
1061  */
1062 WbabEngine::WbabEngine(WSpace* sp, Options opts) {
1063     bab = new BAB<WSpace>(sp, opts);
1064 }
1065
1066 WbabEngine::~WbabEngine() {
1067     delete bab;
1068 }
1069
1070 /**
1071  Search the next solution for this search engine.
1072  */
1073 WSpace* WbabEngine::next() {
1074     return bab->next();
1075 }
1076
1077 /**
1078  Returns true if the search has been stopped by a search object
1079  */
1080 int WbabEngine::stopped(){
1081     return bab->stopped();
1082 }
1083
1084 /*
1085  Depth-first search
1086 */
1087 WdfsEngine::WdfsEngine(WSpace* sp, Options opts) {
1088     dfs = new DFS<WSpace>(sp, opts);
1089 }
1090
1091 WdfsEngine::~WdfsEngine() {
1092     delete dfs;
1093 }
1094
1095 /**
1096  Search the next solution for this search engine.
1097  */
1098 WSpace* WdfsEngine::next() {
1099     return dfs->next();
1100 }
1101
1102 /**
1103  Returns true if the search has been stopped by a search object
1104  */
1105 int WdfsEngine::stopped(){
1106     return dfs->stopped();
1107 }
```

C.1.3 gecode_wrapper.hpp

```
1 #ifndef gecode_wrapper_hpp
2 #define gecode_wrapper_hpp
3
4 #include <stdlib.h>
5
```

```
#ifdef __cplusplus
extern "C" {
#endif

enum {
    IRT_EQ,
    IRT_NQ,
    IRT_LQ,
    IRT_LE,
    IRT_GQ,
    IRT_GR
};

enum {
    BOT_AND,
    BOT_OR,
    BOT_IMP,
    BOT_EQV,
    BOT_XOR
};

/**
 Wraps the WSpace constructor.
 */
void* computation_space();

/**
 Wraps the WSpace add_intVar method.
 */
int add_intVar(void* sp, int min, int max);

/**
 Wraps the WSpace add_intVarWithDom method.
 */
int add_intVarWithDom(void* sp, int s, int* dom);

/**
 Wraps the WSpace add_intVarArray method.
 */
int* add_intVarArray(void* sp, int n, int min, int max);

/**
 Wraps the WSpace add_intVarArrayWithDom method.
 */
int* add_intVarArrayWithDom(void* sp, int n, int s, int* dom);

/**
 Wraps the WSpace set_as_solution_variables method.
 */
void set_solution_vars(void* sp, int n, int* vids);

/**
 Wraps the WSpace set_percent_diff method.
 */
void set_percent_diff(void* sp, int diff);

/**
 Wraps the WSpace nvars method.
 */
int nvars(void* sp);

/**
 Wraps the WSpace add_boolVar method.
 */
int add_boolVar(void* sp, int min, int max);

/**
 Wraps the WSpace add_boolVarArray method.
 */
int* add_boolVarArray(void* sp, int n, int min, int max);

/**
 Wraps the WSpace add_boolVar_expr_val method.
 */
int add_boolVar_expr_val(void* sp, int vid, int rel_type, int val);

/**
 Wraps the WSpace add_boolVar_expr_var method.
 */
int add_boolVar_expr_var(void* sp, int vid1, int rel_type, int vid2);

/**
Wraps the WSpace add_setVar method
*/
int add_setVar(void* sp, int lub_min, int lub_max, int card_min, int card_max);

/**
Wraps the WSpace add_setVarArray method
*/
int* add_setVarArray(void* sp, int n, int lub_min, int lub_max, int card_min, int card_max);

/**
 Wraps the WSpace cst_var_relr method.
```

```
 */
void var_rel(void* sp, int vid1, int rel_type, int vid2);

/**
 Wraps the WSpace cst_var_rel_reify method.
 */
void var_rel_reify(void* sp, int vid1, int rel_type, int vid2, int vid3, int mode);

/**
 Wraps the WSpace cst_val_rel_reify method.
 */
void val_rel_reify(void* sp, int vid1, int rel_type, int val, int vid2, int mode);

/**
 Wraps the WSpace cst_val_rel method.
 */
void val_rel(void* sp, int vid, int rel_type, int val);

/**
 Wraps the WSpace cst_arr_val_rel method.
 */
void arr_val_rel(void* sp, int n, int* vids, int rel_type, int val);

/**
 Wraps the WSpace cst_arr_var_rel method.
 */
void arr_var_rel(void* sp, int n, int* vids, int rel_type, int vid);

/**
 Wraps the WSpace cst_arr_rel method.
 */
void arr_rel(void* sp, int n, int* vids, int rel_type);

/**
 Wraps the WSpace cst_arr_arr_rel method.
 */
void arr_arr_rel(void* sp, int n1, int* vids1, int rel_type, int n2, int* vids2);

/**
 Wraps the WSpace cst_distinct method.
 */
void distinct(void* sp, int n, int* vids);

/**
 Wraps the WSpace cst_val_linear method.
 */
void val_linear(void* sp, int n, int* c, int* vids, int rel_type, int value);

/**
 Wraps the WSpace cst_var_linear method.
 */
void var_linear(void* sp, int n, int* c, int* vids, int rel_type, int vid);

/**
 Wraps the WSpace cst_abs method.
 */
void arithmetics_abs(void* sp, int vid1, int vid2);

/**
 Wraps the WSpace acst_div method.
 */
void arithmetics_div(void* sp, int vid1, int vid2, int vid3);

/**
 Wraps the WSpace cst_var_mod method.
 */
void arithmetics_mod(void* sp, int vid1, int vid2, int vid3);

/**
 Wraps the WSpace cst_divmod method.
 */
void arithmetics_divmod(void* sp, int vid1, int vid2, int vid3, int vid4);

/**
 Wraps the WSpace cst_min method.
 */
void arithmetics_min(void* sp, int vid1, int vid2, int vid3);

/**
 Wraps the WSpace cst_arr_min method.
 */
void arithmetics_arr_min(void* sp, int n, int* vids, int vid);

/**
 Wraps the WSpace cst_argmin method.
 */
void arithmetics_argmin(void* sp, int n, int* vids, int vid);

/**
 Wraps the WSpace cst_max method.
 */
void arithmetics_max(void* sp, int vid1, int vid2, int vid3);

```

```
192 /**
193  Wraps the WSpace cst_arr_max method.
194  */
195 void arithmetics_arr_max(void* sp, int n, int* vids, int vid);
196
197 /**
198  Wraps the WSpace cst_argmax method.
199  */
200 void arithmetics_argmax(void* sp, int n, int* vids, int vid);
201
202 /**
203  Wraps the WSpace cst_mult method.
204  */
205 void arithmetics_mult(void* sp, int vid1, int vid2, int vid3);
206
207 /**
208  Wraps the WSpace cst_sqr method.
209  */
210 void arithmetics_sqr(void* sp, int vid1, int vid2);
211
212 /**
213  Wraps the WSpace cst_sqrt method.
214  */
215 void arithmetics_sqrt(void* sp, int vid1, int vid2);
216
217 /**
218  Wraps the WSpace cst_pow method.
219  */
220 void arithmetics_pow(void* sp, int vid1, int n, int vid2);
221
222 /**
223  Wraps the WSpace cst_nroot method.
224  */
225 void arithmetics_nroot(void* sp, int vid1, int n, int vid2);
226
227 /**
228  Wraps the WSpace cst_dom method.
229  */
230 void set_dom(void* sp, int vid, int n, int* d);
231
232 /**
233  Wraps the WSpace cst_member method.
234  */
235 void set_member(void* sp, int n, int* vids, int vid);
236
237 /**
238  Wraps the WSpace cst_sum method.
239  */
240 void rel_sum(void* sp, int vid, int n, int* vids);
241
242 /**
243  Wraps the WSpace cst_count_val_val method.
244  */
245 void count_val_val(void* sp, int n, int* vids, int val1, int rel_type, int val2);
246
247 /**
248  Wraps the WSpace cst_count_val_var method.
249  */
250 void count_val_var(void* sp, int n, int* vids, int val, int rel_type, int vid);
251
252 /**
253  Wraps the WSpace cst_count_var_val method.
254  */
255 void count_var_val(void* sp, int n, int* vids, int vid, int rel_type, int val);
256
257 /**
258  Wraps the WSpace cst_count_var_var method.
259  */
260 void count_var_var(void* sp, int n, int* vids, int vid1, int rel_type, int vid2);
261
262 /**
263  Wraps the WSpace cst_count_var_set_val method.
264  */
265 void count_var_set_val(void*sp, int n, int* vids, int s, int* set, int rel_type, int val);
266
267 /**
268  Wraps the WSpace cst_count_array_val method.
269  */
270 void count_array_val(void* sp, int n, int* vids, int* c, int rel_type, int val);
271
272 /**
273  Wraps the WSpace cst_sequence_var method.
274  */
275 void sequence_var(void*sp, int n, int* vids, int s, int* set, int val1, int val2, int val3);
276
277 /**
278  Wraps the WSpace cst_nvalues method.
279  */
280 void nvalues(void* sp, int n, int* vids, int rel_type, int vid);
281
282 /**
283  Wraps the WSpace cst_circuit method.
284  */
```

```
285 void circuit(void* sp, int n, int* c, int* vids1, int* vids2, int vid);
286
287 /**
288  Wraps the WSpace cst_precede method
289 */
290 void precede(void* sp, int n, int* vids, int s, int u);
291
292 /**
293  Wraps the WSpace cst_boolop_val method.
294  */
295 void val_boolop(void* sp, int vid1, int bool_op, int vid2, int val);
296
297 /**
298  Wraps the WSpace cst_boolop_arr_val method.
299  */
300 void val_arr_boolop(void* sp, int bool_op, int s, int* vids, int val);
301
302 /**
303  Wraps the WSpace cst_boolop_arr_var method.
304  */
305 void var_arr_boolop(void* sp, int bool_op, int s, int* vids, int vid1);
306
307 /**
308  Wraps the WSpace cst_boolop_var method.
309  */
310 void var_boolop(void* sp, int vid1, int bool_op, int vid2, int vid3);
311
312 /**
313  Wraps the WSpace cst_boolrel_val method.
314  */
315 void val_boolrel(void* sp, int vid, int rel_type, int val);
316
317 /**
318  Wraps the WSpace cst_boolrel_var method.
319  */
320 void var_boolrel(void* sp, int vid1, int rel_type, int vid2);
321
322 /**
323  Wraps the WSpace cst_setop_var method.
324  */
325 void var_setop(void* sp, int vid1, int set_op, int vid2, int set_rel, int vid3);
326
327 /**
328  Wraps the WSpace cst_setop_arr method.
329  */
330 void arr_setop(void* sp, int set_op, int s, int* vid1, int vid2);
331
332 /**
333  Wraps the WSpace cst_setrel_var method.
334  */
335 void var_setrel(void* sp, int vid1, int rel_type, int vid2);
336
337 /**
338  Wraps the WSpace cst_setrel_val method.
339  */
340 void val_setrel(void* sp, int vid1, int rel_type, int* dom, int s);
341
342 /**
343  Wraps the WSpace cst_setrel_val_reify method.
344  */
345 void val_setrel_reify(void* sp, int vid1, int rel_type, int* dom, int s, int r, int mode);
346
347 /**
348  Wraps the WSpace cst_setrel_var_reify method.
349  */
350 void var_setrel_reify(void* sp, int vid1, int rel_type, int vid2, int r, int mode);
351
352 /**
353  Wraps the WSpace cst_setdom_ints method.
354  */
355 void ints_setdom(void* sp, int vid1, int rel_type, int i, int j);
356
357 /**
358  Wraps the WSpace cst_setdom_set method.
359  */
360 void set_setdom(void* sp, int vid1, int vid2);
361
362 /**
363  Wraps the WSpace cst_set_empty method.
364  */
365 void empty_set(void* sp, int vid1);
366
367 /**
368  Wraps the WSpace cst_card_val method.
369  */
370 void val_card(void* sp, int n, int* vids, int min_card, int max_card);
371
372 /**
373  Wraps the WSpace cst_setrel_var method.
374  */
375 void var_card(void* sp, int vid1, int vid2);
376
377 /**
```

```
 Wraps the WSpace cst_channel method.
 */
void channel_set(void* sp, int n1, int* vids1, int n2, int* vids2);

/**
Wraps the WSpace cst_channel_sb method.
*/
void channel_set_bool(void* sp, int n1, int* vids1, int vid2);

/**
Wraps the WSpace cst_setmin method.
*/
int set_min(void* sp, int vid1);

/**
Wraps the WSpace cst_setmax method.
*/
int set_max(void* sp, int vid1);

/**
Wraps the WSpace cst_setmin_reify method.
*/
void set_min_reify(void* sp, int vid1, int vid2, int r, int mode);

/**
Wraps the WSpace cst_setmax_reify method.
*/
void set_max_reify(void* sp, int vid1, int  vid2, int r, int mode);

/**
Wraps the WSpace cst_setunion method.
*/
void set_union(void* sp, int vid1, int n, int* vids);

/**
Wraps the WSpace cst_element method.
*/
void element(void* sp, int set_op, int n, int* vids, int vid1, int vid2);


/**
 Wraps the WSpace branch method.
 */
void branch(void* sp, int n, int* vids, int var_strategy, int val_strategy);

/**
 Wraps the WSpace branch_b method.
 */
void branch_b(void* sp, int n, int* vids, int var_strategy, int val_strategy);

/**
 Wraps the WSpace branch_set method.
 */
void branch_set(void* sp, int n, int* vids, int var_strategy, int val_strategy);

/**
 Wraps the WSpace cost method.
 */
void cost(void* sp, int vid);

/**
 Wraps the WTimeStop constructor
 */
void* new_time_stop(int maxTime);

/**
 Wraps the WTimeStop reset method
 */
void reset_time_stop(void* tStop);

/**
 Wraps the WSearchOptions constructor
 */
void* new_search_options();

/**
 Wraps the WSearchOptions setNbThreads method.
 */
int set_nb_threads(void* sOpts, int nThreads);

/**
 Wraps the WSearchOptions setTimeStop method.
 */
void* set_time_stop(void* sOpts, void* tStop);

//new version
/**
 Wraps the WbabEngine constructor.
 */
void* new_bab_engine(void* sp, void* opts);

/**
 Wraps the WbabEngine next method.
```

```
 */
void* bab_next(void* se);

/**
 Wraps the WbabEngine stopped method.
 */
int bab_stopped(void* se);

/**
 Wraps the WdfsEngine constructor.
 */
void* new_dfs_engine(void* sp, void* opts);

/**
 Wraps the WdfsEngine next method.
 */
void* dfs_next(void* se);

/**
 Wraps the WdfsEngine stopped method.
 */
int dfs_stopped(void* se);

/**
 Wraps the WSpace destructor.
 */
void release(void* sp);

/**
 Wraps the WSpace value method.
 */
int get_value(void* sp, int vid);

/**
 Wraps the WSpace value method.
 */
int get_value_bool(void* sp, int vid);

/**
 Wraps the WSpace value method.
 */
int* get_value_set(void* sp, int vid, int n);

/**
 Wraps the WSpace value method.
 */
int get_value_size(void* sp, int vid);

/**
 Wraps the WSpace values method.
 */
int* get_values(void* sp, int n, int* vids);

/**
 Wraps the WSpace print method.
 */
void print_vars(void* sp, int n, int* vids);

#ifdef __cplusplus
};
#endif
#endif
```

C.1.4 gecode_wrapper.cpp

```
#include "headers/gecode_wrapper.hpp"
#include "headers/space_wrapper.hpp"

/**
 Wraps the WSpace constructor.
 */
void* computation_space() {
    return (void*) new WSpace();
}

/**
 Wraps the WSpace add_intVar method.
 */
int add_intVar(void* sp, int min, int max) {
    return static_cast<WSpace*>(sp)->add_intVar(min, max);
}

/**
 Wraps the WSpace add_intVarWithDom method.
 */
int add_intVarWithDom(void* sp, int s, int* dom) {
    return static_cast<WSpace*>(sp)->add_intVarWithDom(s, dom);
}

/**
```

```
26  Wraps the WSpace add_intVarArray method.
27  */
28 int* add_intVarArray(void* sp, int n, int min, int max) {
29     return static_cast<WSpace*>(sp)->add_intVarArray(n, min, max);
30 }
31
32 /**
33  Wraps the WSpace add_intVarArrayWithDom method.
34  */
35 int* add_intVarArrayWithDom(void* sp, int n, int s, int* dom) {
36     return static_cast<WSpace*>(sp)->add_intVarArrayWithDom(n, s, dom);
37 }
38
39 /**
40  Wraps the WSpace set_as_solution_variables method.
41  */
42 void set_solution_vars(void* sp, int n, int* vids){
43     return static_cast<WSpace*>(sp)->set_as_solution_variables(n, vids);
44 }
45
46 /**
47  Wraps the WSpace set_percent_diff method.
48  */
49 void set_percent_diff(void* sp, int diff){
50     return static_cast<WSpace*>(sp)->set_percent_diff(diff);
51 }
52
53 /**
54  Wraps the WSpace nvars method.
55  */
56 int nvars(void* sp) {
57     return static_cast<WSpace*>(sp)->nvars();
58 }
59
60 /**
61  Wraps the WSpace add_boolVar method.
62  */
63 int add_boolVar(void* sp, int min, int max) {
64     return static_cast<WSpace*>(sp)->add_boolVar(min, max);
65 }
66
67 /**
68  Wraps the WSpace add_boolVarArray method.
69  */
70 int* add_boolVarArray(void* sp, int n, int min, int max) {
71     return static_cast<WSpace*>(sp)->add_boolVarArray(n, min, max);
72 }
73
74 /**
75  Wraps the WSpace add_boolVar_expr_val method.
76  */
77 int add_boolVar_expr_val(void* sp, int vid, int rel_type, int val) {
78     return static_cast<WSpace*>(sp)->add_boolVar_expr_val(vid, rel_type, val);
79 }
80
81 /**
82  Wraps the WSpace add_boolVar_expr_var method.
83  */
84 int add_boolVar_expr_var(void* sp, int vid1, int rel_type, int vid2) {
85     return static_cast<WSpace*>(sp)->add_boolVar_expr_var(vid1, rel_type, vid2);
86 }
87
88 int add_setVar(void* sp, int lub_min, int lub_max, int card_min, int card_max) {
89     return static_cast<WSpace*>(sp)->add_setVar(lub_min, lub_max, card_min, card_max);
90 }
91
92 /**
93  Wraps the WSpace add_setVarArray method.
94  */
95 int* add_setVarArray(void* sp, int n, int lub_min, int lub_max, int card_min, int card_max) {
96     return static_cast<WSpace*>(sp)->add_setVarArray(n, lub_min, lub_max, card_min, card_max)
       ;
97 }
98
99 /**
100  Wraps the WSpace cst_val_rel method.
101  */
102 void val_rel(void* sp, int vid, int rel_type, int val) {
103     return static_cast<WSpace*>(sp)->cst_val_rel(vid, rel_type, val);
104 }
105
106 /**
107  Wraps the WSpace cst_var_rel method.
108  */
109 void var_rel(void* sp, int vid1, int rel_type, int vid2) {
110     return static_cast<WSpace*>(sp)->cst_var_rel(vid1, rel_type, vid2);
111 }
112
113 /**
114  Wraps the WSpace cst_var_rel_reify method.
115  */
116 void var_rel_reify(void* sp, int vid1, int rel_type, int vid2, int vid3, int mode) {
117     return static_cast<WSpace*>(sp)->cst_var_rel_reify(vid1, rel_type, vid2, vid3, mode);
```

```
}

/**
 Wraps the WSpace cst_val_rel_reify method.
 */
void val_rel_reify(void* sp, int vid1, int rel_type, int val, int vid2, int mode) {
    return static_cast<WSpace*>(sp)->cst_val_rel_reify(vid1, rel_type, val, vid2, mode);
}

/**
 Wraps the WSpace cst_arr_val_rel method.
 */
void arr_val_rel(void* sp, int n, int* vids, int rel_type, int val) {
    return static_cast<WSpace*>(sp)->cst_arr_val_rel(n, vids, rel_type, val);
}

/**
 Wraps the WSpace cst_arr_var_rel method.
 */
void arr_var_rel(void* sp, int n, int* vids, int rel_type, int vid) {
    return static_cast<WSpace*>(sp)->cst_arr_var_rel(n, vids, rel_type, vid);
}

/**
 Wraps the WSpace cst_arr_rel method.
 */
void arr_rel(void* sp, int n, int* vids, int rel_type) {
    return static_cast<WSpace*>(sp)->cst_arr_rel(n, vids, rel_type);
}

/**
 Wraps the WSpace cst_arr_arr_rel method.
 */
void arr_arr_rel(void* sp, int n1, int* vids1, int rel_type, int n2, int* vids2) {
    return static_cast<WSpace*>(sp)->cst_arr_arr_rel(n1, vids1, rel_type, n2, vids2);
}

/**
 Wraps the WSpace cst_distinct method.
 */
void distinct(void* sp, int n, int* vids) {
    return static_cast<WSpace*>(sp)->cst_distinct(n, vids);
}

/**
 Wraps the WSpace cst_val_linear method.
 */
void val_linear(void* sp, int n, int* c, int* vids, int rel_type, int value) {
    return static_cast<WSpace*>(sp)->cst_val_linear(n, c, vids, rel_type, value);
}

/**
 Wraps the WSpace cst_var_linear method.
 */
void var_linear(void* sp, int n, int* c, int* vids, int rel_type, int vid) {
    return static_cast<WSpace*>(sp)->cst_var_linear(n, c, vids, rel_type, vid);
}

/**
 Wraps the WSpace cst_abs method.
 */
void arithmetics_abs(void* sp, int vid1, int vid2) {
    return static_cast<WSpace*>(sp)->cst_abs(vid1, vid2);
}

/**
 Wraps the WSpace acst_div method.
 */
void arithmetics_div(void* sp, int vid1, int vid2, int vid3) {
    return static_cast<WSpace*>(sp)->cst_div(vid1, vid2, vid3);
}

/**
 Wraps the WSpace cst_mod method.
 */
void arithmetics_mod(void* sp, int vid1, int vid2, int vid3) {
    return static_cast<WSpace*>(sp)->cst_mod(vid1, vid2, vid3);
}

/**
 Wraps the WSpace cst_divmod method.
 */
void arithmetics_divmod(void* sp, int vid1, int vid2, int vid3, int vid4) {
    return static_cast<WSpace*>(sp)->cst_divmod(vid1, vid2, vid3, vid4);
}

/**
 Wraps the WSpace cst_min method.
 */
void arithmetics_min(void* sp, int vid1, int vid2, int vid3) {
    return static_cast<WSpace*>(sp)->cst_min(vid1, vid2, vid3);
}

```

```
/**
 Wraps the WSpace cst_arr_min method.
 */
void arithmetics_arr_min(void* sp, int n, int* vids, int vid) {
    return static_cast<WSpace*>(sp)->cst_arr_min(n, vids, vid);
}

/**
 Wraps the WSpace cst_argmin method.
 */
void arithmetics_argmin(void* sp, int n, int* vids, int vid) {
    return static_cast<WSpace*>(sp)->cst_argmin(n, vids, vid);
}

/**
 Wraps the WSpace cst_max method.
 */
void arithmetics_max(void* sp, int vid1, int vid2, int vid3) {
    return static_cast<WSpace*>(sp)->cst_max(vid1, vid2, vid3);
}

/**
 Wraps the WSpace cst_arr_max method.
 */
void arithmetics_arr_max(void* sp, int n, int* vids, int vid) {
    return static_cast<WSpace*>(sp)->cst_arr_max(n, vids, vid);
}

/**
 Wraps the WSpace cst_argmax method.
 */
void arithmetics_argmax(void* sp, int n, int* vids, int vid) {
    return static_cast<WSpace*>(sp)->cst_argmax(n, vids, vid);
}

/**
 Wraps the WSpace cst_mult method.
 */
void arithmetics_mult(void* sp, int vid1, int vid2, int vid3) {
    return static_cast<WSpace*>(sp)->cst_mult(vid1, vid2, vid3);
}

/**
 Wraps the WSpace cst_sqr method.
 */
void arithmetics_sqr(void* sp, int vid1, int vid2) {
    return static_cast<WSpace*>(sp)->cst_sqr(vid1, vid2);
}

/**
 Wraps the WSpace cst_sqrt method.
 */
void arithmetics_sqrt(void* sp, int vid1, int vid2) {
    return static_cast<WSpace*>(sp)->cst_sqrt(vid1, vid2);
}

/**
 Wraps the WSpace cst_pow method.
 */
void arithmetics_pow(void* sp, int vid1, int n, int vid2) {
    return static_cast<WSpace*>(sp)->cst_pow(vid1, n, vid2);
}

/**
 Wraps the WSpace cst_nroot method.
 */
void arithmetics_nroot(void* sp, int vid1, int n, int vid2) {
    return static_cast<WSpace*>(sp)->cst_nroot(vid1, n, vid2);
}

/**
 Wraps the WSpace cst_dom method.
 */
void set_dom(void* sp, int vid, int n, int* d) {
    return static_cast<WSpace*>(sp)->cst_dom(vid, n, d);
}

/**
 Wraps the WSpace cst_member method.
 */
void set_member(void* sp, int n, int* vids, int vid) {
    return static_cast<WSpace*>(sp)->cst_member(n, vids, vid);
}

/**
 Wraps the WSpace cst_sum method.
 */
void rel_sum(void* sp, int vid, int n, int* vids) {
    return static_cast<WSpace*>(sp)->cst_sum(vid, n, vids);
}

/**
 Wraps the WSpace cst_count_val_val method.
```

```
 */
void count_val_val(void* sp, int n, int* vids, int val1, int rel_type, int val2) {
    return static_cast<WSpace*>(sp)->cst_count_val_val(n, vids, val1, rel_type, val2);
}

/**
 Wraps the WSpace cst_count_val_var method.
 */
void count_val_var(void* sp, int n, int* vids, int val, int rel_type, int vid) {
    return static_cast<WSpace*>(sp)->cst_count_val_var(n, vids, val, rel_type, vid);
}

/**
 Wraps the WSpace cst_count_var_val method.
 */
void count_var_val(void* sp, int n, int* vids, int vid, int rel_type, int val) {
    return static_cast<WSpace*>(sp)->cst_count_var_val(n, vids, vid, rel_type, val);
}

/**
 Wraps the WSpace cst_count_var_var method.
 */
void count_var_var(void* sp, int n, int* vids, int vid1, int rel_type, int vid2) {
    return static_cast<WSpace*>(sp)->cst_count_var_var(n, vids, vid1, rel_type, vid2);
}

/**
 Wraps the WSpace cst_count_var_set_val method.
 */
void count_var_set_val(void*sp, int n, int* vids, int s, int* set, int rel_type, int val){
    return static_cast<WSpace*>(sp)->cst_count_var_set_val(n, vids, s, set, rel_type, val);
}

/**
 Wraps the WSpace cst_count_array_val method.
 */
void count_array_val(void* sp, int n, int* vids, int* c, int rel_type, int val){
    return static_cast<WSpace*>(sp)->cst_count_array_val(n, vids, c, rel_type, val);
}

/**
 Wraps the WSpace cst_sequence_var method.
 */
void sequence_var(void*sp, int n, int* vids, int s, int* set, int val1, int val2, int val3){
    return static_cast<WSpace*>(sp)->cst_sequence_var(n, vids, s, set, val1, val2, val3);
}

/**
 Wraps the WSpace cst_nvalues method.
 */
void nvalues(void* sp, int n, int* vids, int rel_type, int vid) {
    return static_cast<WSpace*>(sp)->cst_nvalues(n, vids, rel_type, vid);
}

/**
 Wraps the WSpace cst_circuit method.
 */
void circuit(void* sp, int n, int* c, int* vids1, int* vids2, int vid) {
    return static_cast<WSpace*>(sp)->cst_circuit(n, c, vids1, vids2, vid);
}

/**
 Wraps the WSpace cst_precede method
*/
void precede(void* sp, int n, int* vids, int s, int u){
    return static_cast<WSpace*>(sp)->cst_precede(n, vids, s, u);
}

/**
 Wraps the WSpace cst_boolop_val method.
 */
void val_boolop(void* sp, int vid1, int bool_op, int vid2, int val) {
    return static_cast<WSpace*>(sp)->cst_boolop_val(vid1, bool_op, vid2, val);
}

/**
 Wraps the WSpace cst_boolop_arr_val method.
 */
void val_arr_boolop(void* sp, int bool_op, int s, int* vids, int val) {
    return static_cast<WSpace*>(sp)->cst_boolop_arr_val(bool_op, s, vids, val);
}

/**
 Wraps the WSpace cst_boolop_arr_var method.
 */
void var_arr_boolop(void* sp, int bool_op, int s, int* vids, int vid1) {
    return static_cast<WSpace*>(sp)->cst_boolop_arr_var(bool_op, s, vids, vid1);
}

/**
 Wraps the WSpace cst_boolop_var method.
 */
void var_boolop(void* sp, int vid1, int bool_op, int vid2, int vid3) {
```

```
397     return static_cast<WSpace*>(sp)->cst_boolop_var(vid1, bool_op, vid2, vid3);
398 }
399
400 /**
401  Wraps the WSpace cst_boolrel_val method.
402  */
403 void val_boolrel(void* sp, int vid, int rel_type, int val) {
404     return static_cast<WSpace*>(sp)->cst_boolrel_val(vid, rel_type, val);
405 }
406
407 /**
408  Wraps the WSpace cst_boolrel_var method.
409  */
410 void var_boolrel(void* sp, int vid1, int rel_type, int vid2) {
411     return static_cast<WSpace*>(sp)->cst_boolrel_var(vid1, rel_type, vid2);
412 }
413
414 /**
415  Wraps the WSpace cst_setop_var method.
416  */
417 void var_setop(void* sp, int vid1, int set_op, int vid2, int set_rel, int vid3) {
418     return static_cast<WSpace*>(sp)->cst_setop_var(vid1, set_op, vid2, set_rel, vid3);
419 }
420
421 /**
422  Wraps the WSpace cst_setop_arr method.
423  */
424 void arr_setop(void* sp, int set_op, int s, int* vid1, int vid2) {
425     return static_cast<WSpace*>(sp)->cst_setop_arr(set_op, s, vid1, vid2);
426 }
427
428 /**
429  Wraps the WSpace cst_setrel_var method.
430  */
431 void var_setrel(void* sp, int vid1, int rel_type, int vid2) {
432     return static_cast<WSpace*>(sp)->cst_setrel_var(vid1, rel_type, vid2);
433 }
434
435 /**
436  Wraps the WSpace cst_setrel_val method.
437  */
438 void val_setrel(void* sp, int vid1, int rel_type, int* dom, int s) {
439     return static_cast<WSpace*>(sp)->cst_setrel_val(vid1, rel_type, dom, s);
440 }
441
442 /**
443  Wraps the WSpace cst_setrel_val_reify method.
444  */
445 void val_setrel_reify(void* sp, int vid1, int rel_type, int* dom, int s, int r, int mode) {
446     return static_cast<WSpace*>(sp)->cst_setrel_val_reify(vid1, rel_type, dom, s, r, mode);
447 }
448
449 /**
450  Wraps the WSpace cst_setrel_var_reify method.
451  */
452 void var_setrel_reify(void* sp, int vid1, int rel_type, int vid2, int r, int mode) {
453     return static_cast<WSpace*>(sp)->cst_setrel_var_reify(vid1, rel_type, vid2, r, mode);
454 }
455
456 /**
457  Wraps the WSpace cst_setdom_ints method.
458  */
459 void ints_setdom(void* sp, int vid1, int rel_type, int i, int j) {
460     return static_cast<WSpace*>(sp)->cst_setdom_ints(vid1, rel_type, i, j);
461 }
462
463 /**
464  Wraps the WSpace cst_setdom_set method.
465  */
466 void set_setdom(void* sp, int vid1, int vid2) {
467     return static_cast<WSpace*>(sp)->cst_setdom_set(vid1, vid2);
468 }
469
470 /**
471  Wraps the WSpace cst_set_empty method.
472  */
473 void empty_set(void* sp, int vid1) {
474     return static_cast<WSpace*>(sp)->cst_set_empty(vid1);
475 }
476
477 /**
478  Wraps the WSpace cst_setrel_val method.
479  */
480 void val_card(void* sp, int n, int* vids, int min_card, int max_card) {
481     return static_cast<WSpace*>(sp)->cst_card_val(n, vids, min_card, max_card);
482 }
483
484 /**
485  Wraps the WSpace cst_setrel_var method.
486  */
487 void var_card(void* sp, int vid1, int vid2) {
488     return static_cast<WSpace*>(sp)->cst_card_var(vid1, vid2);
489 }
```

```
490
491 /**
492 Wraps the WSpace cst_channel method.
493 */
494 void channel_set(void* sp, int n1, int* vids1, int n2, int* vids2) {
495     return static_cast<WSpace*>(sp)->cst_channel(n1, vids1, n2, vids2);
496 }
497
498 /**
499 Wraps the WSpace cst_channel_sb method.
500 */
501 void channel_set_bool(void* sp, int n1, int* vids1, int vid2) {
502     return static_cast<WSpace*>(sp)->cst_channel_sb(n1, vids1, vid2);
503 }
504
505 /**
506 Wraps the WSpace cst_setmin method.
507 */
508 int set_min(void* sp, int vid1){
509     return static_cast<WSpace*>(sp)->cst_setmin(vid1);
510 }
511
512 /**
513 Wraps the WSpace cst_setmax method.
514 */
515 int set_max(void* sp, int vid1){
516     return static_cast<WSpace*>(sp)->cst_setmax(vid1);
517 }
518
519 /**
520 Wraps the WSpace cst_setmin_reify method.
521 */
522 void set_min_reify(void* sp, int vid1, int vid2, int r, int mode){
523     return static_cast<WSpace*>(sp)->cst_setmin_reify(vid1, vid2, r, mode);
524 }
525
526 /**
527 Wraps the WSpace cst_setmax_reify method.
528 */
529 void set_max_reify(void* sp, int vid1, int vid2, int r, int mode){
530     return static_cast<WSpace*>(sp)->cst_setmax_reify(vid1, vid2, r, mode);
531 }
532
533 /**
534 Wraps the WSpace cst_setunion method.
535 */
536 void set_union(void* sp, int vid1, int n, int* vids){
537     return static_cast<WSpace*>(sp)->cst_setunion(vid1, n, vids);
538 }
539
540 /**
541 Wraps the WSpace cst_element method.
542 */
543 void element(void* sp, int set_op, int n, int* vids, int vid1, int vid2){
544     return static_cast<WSpace*>(sp)->cst_element(set_op, n, vids, vid1, vid2);
545 }
546
547 /**
548  Wraps the WSpace branch method.
549  */
550 void branch(void* sp, int n, int* vids, int var_strategy, int val_strategy) {
551     return static_cast<WSpace*>(sp)->branch(n, vids, var_strategy, val_strategy);
552 }
553
554 /**
555  Wraps the WSpace branch_b method.
556  */
557 void branch_b(void* sp, int n, int* vids, int var_strategy, int val_strategy) {
558     return static_cast<WSpace*>(sp)->branch_b(n, vids, var_strategy, val_strategy);
559 }
560
561 /**
562  Wraps the WSpace branch_set method.
563  */
564 void branch_set(void* sp, int n, int* vids, int var_strategy, int val_strategy) {
565     return static_cast<WSpace*>(sp)->branch_set(n, vids, var_strategy, val_strategy);
566 }
567
568 /**
569  Wraps the WSpace cost method.
570  */
571 void cost(void* sp, int vid) {
572     return static_cast<WSpace*>(sp)->cost(vid);
573 }
574
575 /**
576  Wraps the WTimeStop constructor
577  */
578 void* new_time_stop(int maxTime){
579     return (void*) new WTimeStop(maxTime);
580 }
581
582 /**
```

```
 Wraps the WTimeStop reset method
 */
void reset_time_stop(void* tStop){
    WTimeStop* _tStop = static_cast<WTimeStop*>(tStop);
    _tStop->reset();
}

/**
 Wraps the WSearchOptions constructor.
 */
void* new_search_options(){
    return (void*) new WSearchOptions();
}

/**
 Wraps the WSearchOptions setNbThreads method.
 */
int set_nb_threads(void* sOpts, int nThreads){
    return static_cast<WSearchOptions*>(sOpts)->setNbThreads(nThreads);
}

/**
 Wraps the WSearchOptions setTimeStop method.
 Returns the options object passed as an argument as a void pointer
 */
void* set_time_stop(void* sOpts, void* tStop){
    WTimeStop* _tStop = static_cast<WTimeStop*>(tStop);
    WSearchOptions* _sOpts = static_cast<WSearchOptions*>(sOpts);
    _sOpts->setTimeStop(_tStop);
    return (void*) _sOpts;
}

//new version
/**
 Wraps the WbabEngine constructor.
 */
void* new_bab_engine(void* sp, void* opts) {
    WSpace* _sp = static_cast<WSpace*>(sp);
    WSearchOptions* _opts = static_cast<WSearchOptions*>(opts);
    return (void*) new WbabEngine(_sp, _opts->getOpts());
}

/**
 Wraps the WbabEngine next method.
 */
void* bab_next(void* se) {
    return (void*) static_cast<WbabEngine*>(se)->next();
}

/**
 Wraps the WbabEngine stopped method.
 */
int bab_stopped(void* se){
    return static_cast<WbabEngine*>(se)->stopped();
}

/**
 Wraps the WdfsEngine constructor.
 */
void* new_dfs_engine(void* sp, void* opts) {
    WSpace* _sp = static_cast<WSpace*>(sp);
    WSearchOptions* _opts = static_cast<WSearchOptions*>(opts);
    return (void*) new WdfsEngine(_sp, _opts->getOpts());
}

/**
 Wraps the WdfsEngine next method.
 */
void* dfs_next(void* se) {
    return (void*) static_cast<WdfsEngine*>(se)->next();
}

/**
 Wraps the WdfsEngine stopped method.
 */
int dfs_stopped(void* se){
    return static_cast<WdfsEngine*>(se)->stopped();
}

/**
 Wraps the WSpace destructor.
 */
void release(void* sp) {
    delete static_cast<WSpace*>(sp);
}

/**
 Wraps the WSpace value method.
 */
int get_value(void* sp, int vid) {
    return static_cast<WSpace*>(sp)->value(vid);
}

```

```
676 /**
677  Wraps the WSpace value method.
678  */
679 int get_value_bool(void* sp, int vid) {
680     return static_cast<WSpace*>(sp)->value_bool(vid);
681 }
682
683 /**
684  Wraps the WSpace value method.
685  */
686 int* get_value_set(void* sp, int vid, int n) {
687     return static_cast<WSpace*>(sp)->value_set(vid, n);
688 }
689
690 /**
691  Wraps the WSpace value method.
692  */
693 int get_value_size(void* sp, int vid) {
694     return static_cast<WSpace*>(sp)->value_size(vid);
695 }
696
697 /**
698  Wraps the WSpace values method.
699  */
700 int* get_values(void* sp, int n, int* vids) {
701     return static_cast<WSpace*>(sp)->values(n, vids);
702 }
703
704 /**
705  Wraps the WSpace print method.
706  */
707 void print_vars(void* sp, int n, int* vids) {
708     return static_cast<WSpace*>(sp)->print(n, vids);
709 }
```

C.2 Lisp Wrapper

The lisp wrapper is used to call the C function previously defined from Lisp code, using the C Foreign Function Interface, it is composed of two files :

- **gecode-wrapper.lisp** : implements the calls to the function from the C library of Gil.
- **gecode-wrapper-ui.lisp** : wraps the function of gecode-wrapper.lisp to make them more user-friendly.

C.2.1 gecode-wrapper.lisp

```
1  (cl:defpackage "gil"
2    (:nicknames "GIL")
3     (:use common-lisp :cl-user :cl :cffi))
4
5  (in-package :gil)
6
7  (cffi::defcfun ("computation_space" new-space) :pointer
8      "Create a new computation space."
9  )
10
11 (cffi::defcfun ("add_intVar" add-int-var-low) :int
12     "Add an IntVar ranging from min to max to the specified space. Return the reference of
       this variable for this space."
13     (sp :pointer)
14     (min :int)
15     (max :int)
16 )
17
18 (cffi::defcfun ("add_intVarWithDom" add-int-var-dom-aux) :int
19     "Add an IntVar with domain dom of size s to the specified space. Return the reference of
       this variable for this space."
20     (sp :pointer)
21     (s :int)
22     (dom :pointer)
23 )
24
25 (defun add-int-var-dom-low (sp dom)
26     "Add an IntVar with domain dom to the specified space. Return the reference of this
       variable for this space."
27     (let ((x (cffi::foreign-alloc :int :initial-contents dom)))
28         (add-int-var-dom-aux sp (length dom) x))
```

```
)

(cffi::defcfun ("add_intVarArray" add-int-var-array-aux) :pointer
    "Add n IntVar ranging from min to max to the specified space."
    (sp :pointer)
    (n :int)
    (min :int)
    (max :int)
)

(defun add-int-var-array-low (sp n min max)
    "Add n IntVar ranging from min to max to the specified space. Return the references of
     those variables for this space"
    (let ((p (add-int-var-array-aux sp n min max)))
        (loop for i from 0 below n
            collect (cffi::mem-aref p :int i)))
)

(cffi::defcfun ("add_boolVarArray" add-bool-var-array-aux) :pointer
    "Add n boolVar ranging from min to max to the specified space."
    (sp :pointer)
    (n :int)
    (min :int)
    (max :int)
)

(defun add-bool-var-array-low (sp n min max)
    "Add n BoolVar ranging from min to max to the specified space. Return the references of
     those variables for this space"
    (let ((p (add-bool-var-array-aux sp n min max)))
        (loop for i from 0 below n
            collect (cffi::mem-aref p :int i)))
)

(cffi::defcfun ("add_intVarArrayWithDom" add-int-var-array-dom-aux) :pointer
    "Add n IntVar with domain dom of size s to the specified space."
    (sp :pointer)
    (n :int)
    (s :int)
    (dom :pointer)
)

(defun add-int-var-array-dom-low (sp n dom)
    "Add n IntVar with domain dom to the specified space. Return the references of those
     variables for this space"
    (let ((x (cffi::foreign-alloc :int :initial-contents dom))
          p)
        (setq p (add-int-var-array-dom-aux sp n (length dom) x))
        (loop for i from 0 below n
            collect (cffi::mem-aref p :int i)))
)

(cffi::defcfun ("set_solution_vars" set-solution-vars-aux) :void
    (sp :pointer)
    (n :int)
    (vids :pointer)
)

(defun set-solution-vars (sp vids)
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (set-solution-vars-aux sp (length vids) x)
    )
)

(cffi::defcfun ("set_percent_diff" set-percent-diff) :void
    (sp :pointer)
    (diff :int)
)

(cffi::defcfun ("nvars" nvars) :int
    "Return the number of variables in the space."
    (sp :pointer)
)

;IntVar relation flags
(defparameter gil::IRT_EQ 0)    ; equality relation
(defparameter gil::IRT_NQ 1)    ; inequality
(defparameter gil::IRT_LQ 2)    ; Less or equal
(defparameter gil::IRT_LE 3)    ; Strictly lower
(defparameter gil::IRT_GQ 4)    ; Greater or equal
(defparameter gil::IRT_GR 5)    ; Strictly greater

(cffi::defcfun ("add_boolVar" add-bool-var-range) :int
    "Add a BoolVar ranging from l to h. Return the index to this BoolVar."
    (sp :pointer)
    (l :int)
    (h :int)
)

(cffi::defcfun ("add_boolVar_expr_val" add-bool-var-expr-val) :int
    "Add a BoolVar corresponding to the evalueation of rel-type(vid, val)."
    (sp :pointer)
    (vid :int)
```

```
    (rel-type :int)
    (val :int)
)

(cffi::defcfun ("add_boolVar_expr_var" add-bool-var-expr-var) :int
    "Add a BoolVar corresponding to the evalueation of rel-type(vid1, vid2)."
    (sp :pointer)
    (vid1 :int)
    (rel-type :int)
    (vid2 :int)
)

(cffi::defcfun ("add_setVar" add-set-var-card) :int
    "Add a SetVar ranging from l to h. Return the index to this BoolVar."
    (sp :pointer)
    (lub-min :int)
    (lub-max :int)
    (card-min :int)
    (card-max :int)
)

(cffi::defcfun ("add_setVarArray" add-set-var-array-aux) :pointer
    "Add n setVar with cardinality card-min to card-max to the specified space."
    (sp :pointer)
    (n :int)
    (lub-min :int)
    (lub-max :int)
    (card-min :int)
    (card-max :int)
)

(defun add-set-var-array-card (sp n lub-min lub-max card-min card-max)
    "Add n SetVar ranging cardinality from card-min to card-max to the specified space.
     Return the references of those variables for this space"
    (let ((p (add-set-var-array-aux sp n lub-min lub-max card-min card-max)))
        (loop for i from 0 below n
            collect (cffi::mem-aref p :int i)))
)

(cffi::defcfun ("val_rel" val-rel) :void
    "Post a variable/value rel constraint."
    (sp :pointer)
    (vid :int)
    (rel-type :int)
    (val :int)
)

(cffi::defcfun ("var_rel" var-rel) :void
    "Post a variable/variable rel constraint."
    (sp :pointer)
    (vid1 :int)
    (rel-type :int)
    (vid2 :int)
)

(cffi::defcfun ("var_rel_reify" var-rel-reify) :void
    "Post a variable/variable rel constraint with reification."
    (sp :pointer)
    (vid1 :int)
    (rel-type :int)
    (vid2 :int)
    (vid3 :int)
    (mode :int)
)

(cffi::defcfun ("val_rel_reify" val-rel-reify) :void
    "Post a variable/value rel constraint with reification."
    (sp :pointer)
    (vid1 :int)
    (rel-type :int)
    (val :int)
    (vid2 :int)
    (mode :int)
)

(cffi::defcfun ("arr_val_rel" arr-val-rel-aux) :void
    "Post a variable-array/value rel constraint."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (rel-type :int)
    (val :int)
)

(defun arr-val-rel (sp vids rel-type val)
    "Post a variable-array/value rel constraint."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (arr-val-rel-aux sp (length vids) x rel-type val))
)

(cffi::defcfun ("arr_var_rel" arr-var-rel-aux) :void
    "Post a variable-array/variable rel constraint."
    (sp :pointer)
```

```
    (n :int)
    (vids :pointer)
    (rel-type :int)
    (vid :int)
)

(defun arr-var-rel (sp vids rel-type vid)
    "Post a variable-array/variable rel constraint."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (arr-var-rel-aux sp (length vids) x rel-type vid))
)

(cffi::defcfun ("arr_rel" arr-rel-aux) :void
    "Post a variable-array rel constraint."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (rel-type :int)
)

(defun arr-rel (sp vids rel-type)
    "Post a variable-array rel constraint."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (arr-rel-aux sp (length vids) x rel-type))
)

(cffi::defcfun ("arr_arr_rel" arr-arr-rel-aux) :void
    "Post a variable-array/variable-array rel constraint."
    (sp :pointer)
    (n1 :int)
    (vids1 :pointer)
    (rel-type :int)
    (n2 :int)
    (vids2 :pointer)
)

(defun arr-arr-rel (sp vids1 rel-type vids2)
    "Post a variable-array/variable-array rel constraint."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids1))
          (y (cffi::foreign-alloc :int :initial-contents vids2)))
        (arr-arr-rel-aux sp (length vids1) x rel-type (length vids2) y))
)

(cffi::defcfun ("distinct" distinct-aux) :void
    "Post a distinct constraint on the n variables denoted in vids."
    (sp :pointer)
    (n :int)
    (vids :pointer)
)

(defun distinct (sp vids)
    "Post a distinct constraint on the variables denoted in vids."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (distinct-aux sp (length vids) x))
)

(cffi::defcfun ("val_linear" val-linear-aux) :void
    "Post a linear equation constraint."
    (sp :pointer)
    (n :int)
    (c :pointer)
    (vids :pointer)
    (rel-type :int)
    (val :int)
)

(defun val-linear (sp coeffs vars rel-type value)
    "Post a linear equation constraint. coeffs and vars must have the same number of elements
    ."
    (let ((c (cffi::foreign-alloc :int :initial-contents coeffs))
          (x (cffi::foreign-alloc :int :initial-contents vars)))
        (val-linear-aux sp (length coeffs) c x rel-type value))
)

(cffi::defcfun ("var_linear" var-linear-aux) :void
    "Post a linear equation constraint."
    (sp :pointer)
    (n :int)
    (c :pointer)
    (vids :pointer)
    (rel-type :int)
    (vid :int)
)

(defun var-linear (sp coeffs vars rel-type vid)
    "Post a linear equation constraint. coeffs and vars must have the same number of elements
    ."
    (let ((c (cffi::foreign-alloc :int :initial-contents coeffs))
          (x (cffi::foreign-alloc :int :initial-contents vars)))
        (var-linear-aux sp (length coeffs) c x rel-type vid))
)

(cffi::defcfun ("arithmetics_abs" ge-abs) :void
```

```
    "Post the constraint that |vid1| = vid2."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
)

(cffi::defcfun ("arithmetics_div" ge-div) :void
    "Post the constraint that vid3 = vid1/vid2."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
    (vid3 :int)
)

(cffi::defcfun ("arithmetics_mod" var-mod) :void
    "Post the constraint that vid1 % vid2 = vid3."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
    (vid3 :int)
)

(cffi::defcfun ("arithmetics_divmod" ge-divmod) :void
    "Post the constraint that vid3 = vid1/vid2 and vid4 = vid1 % vid2."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
    (vid3 :int)
    (vid4 :int)
)

(cffi:defcfun ("arithmetics_min" ge-min) :void
    "Post the constraint that vid3 = min(vid1, vid2)."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
    (vid3 :int)
)

(cffi:defcfun ("arithmetics_arr_min" ge-arr-min-aux) :void
    "Post the constraint that vid = min(vids)."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (vid :int)
)

(defun ge-arr-min (sp vid vids)
    "Post the constraint that vid = min(vids)."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (ge-arr-min-aux sp (length vids) x vid))
)

(cffi:defcfun ("arithmetics_argmin" ge-argmin-aux) :void
    "Post the constraint that vid = argmin(vids)."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (vid :int)
)

(defun ge-argmin (sp vids vid)
    "Post the constraint that vid = argmin(vids)."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (ge-argmin-aux sp (length vids) x vid))
)

(cffi:defcfun ("arithmetics_max" ge-max) :void
    "Post the constraint that vid3 = max(vid1, vid2)."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
    (vid3 :int)
)

(cffi:defcfun ("arithmetics_arr_max" ge-arr-max-aux) :void
    "Post the constraint that vid = max(vids)."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (vid :int)
)

(defun ge-arr-max (sp vid vids)
    "Post the constraint that vid = max(vids)."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (ge-arr-max-aux sp (length vids) x vid))
)

(cffi:defcfun ("arithmetics_argmax" ge-argmax-aux) :void
    "Post the constraint that vid = argmax(vids)."
    (sp :pointer)
    (n :int)
```

```
    (vids :pointer)
    (vid :int)
)

(defun ge-argmax (sp vids vid)
    "Post the constraint that vid = argmax(vids)."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (ge-argmax-aux sp (length vids) x vid))
)

(cffi:defcfun ("arithmetics_mult" ge-mult) :void
    "Post the constraint that vid3 = vid1 * vid2."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
    (vid3 :int)
)

(cffi:defcfun ("arithmetics_sqr" ge-sqr) :void
    "Post the constraint that vid2 = vid1^2."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
)

(cffi:defcfun ("arithmetics_sqrt" ge-sqrt) :void
    "Post the constraint that vid2 = vid1^(1/2)."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
)

(cffi:defcfun ("arithmetics_pow" ge-pow) :void
    "Post the constraint that vid2 = vid1^n."
    (sp :pointer)
    (vid1 :int)
    (n :int)
    (vid2 :int)
)

(cffi:defcfun ("arithmetics_nroot" ge-nroot) :void
    "Post the constraint that vid2 = vid1^(1/n)."
    (sp :pointer)
    (vid1 :int)
    (n :int)
    (vid2 :int)
)

(cffi::defcfun ("set_dom" set-dom-aux) :void
    "Post the constraint that dom(vid) = domain of size n."
    (sp :pointer)
    (vid :int)
    (n :int)
    (domain :pointer)
)

(defun set-dom (sp vid domain)
    "Post the constraint that dom(vid) = domain."
    (let ((x (cffi::foreign-alloc :int :initial-contents domain)))
        (set-dom-aux sp vid (length domain) x))
)

(cffi::defcfun ("set_member" set-member-aux) :void
    "Post the constraint that vid is a member vids."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (vid :int)
)

(defun set-member (sp vids vid)
    "Post the constraint that vid is a member vids."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (set-member-aux sp (length vids) x vid))
)

(cffi::defcfun ("rel_sum" rel-sum-aux) :void
    "Post the constraint that vid = sum(vids). n is the number of vars in vids."
    (sp :pointer)
    (vid :int)
    (n :int)
    (vids :pointer)
)

(defun rel-sum (sp vid vids)
    "Post the constraint that vid = sum(vids)."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (rel-sum-aux sp vid (length vids) x))
)

(cffi::defcfun ("count_val_val" count-val-val-aux) :void
    "Post the constraint that the number of variables in vids equal to val1 has relation
    rel-type with val2."
```

```
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (val1 :int)
    (rel-type :int)
    (val2 :int)
)

(defun count-val-val (sp vids val1 rel-type val2)
    "Post the constraint that the number of variables in vids equal to val1 has relation
    rel-type with val2."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (count-val-val-aux sp (length vids) x val1 rel-type val2))
)

(cffi::defcfun ("count_val_var" count-val-var-aux) :void
    "Post the constraint that the number of variables in vids equal to val1 has relation
    rel-type with vid."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (val :int)
    (rel-type :int)
    (vid :int)
)

(defun count-val-var (sp vids val rel-type vid)
    "Post the constraint that the number of variables in vids equal to val has relation
    rel-type with vid."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (count-val-var-aux sp (length vids) x val rel-type vid))
)

(cffi::defcfun ("count_var_val" count-var-val-aux) :void
    "Post the constraint that the number of variables in vids equal to vid has relation
    rel-type with val."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (vid :int)
    (rel-type :int)
    (val :int)
)

(defun count-var-val (sp vids vid rel-type val)
    "Post the constraint that the number of variables in vids equal to vid has relation
    rel-type with val."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (count-var-val-aux sp (length vids) x vid rel-type val))
)

(cffi::defcfun ("count_var_var" count-var-var-aux) :void
    "Post the constraint that the number of variables in vids equal to vid1 has relation
    rel-type with vid2."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (vid1 :int)
    (rel-type :int)
    (vid2 :int)
)

(defun count-var-var (sp vids vid1 rel-type vid2)
    "Post the constraint that the number of variables in vids equal to vid1 has relation
    rel-type with vid2."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (count-var-var-aux sp (length vids) x vid1 rel-type vid2))
)

(cffi::defcfun ("count_var_set_val" count-var-set-val-aux) :void
    "Post the constraint that the number of variables in vids belonging to the set set has
    relation rel-type with val."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (s :int)
    (s-set :pointer)
    (rel-type :int)
    (val :int)
)

(defun count-var-set-val (sp vids s-set rel-type val)
"Post the constraint that the number of variables in vids belonging to the set set has
    relation rel-type with val."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids))
            (y (cffi::foreign-alloc :int :initial-contents s-set)))
        (count-var-set-val-aux sp (length vids) x (length s-set) y rel-type val))
)

(cffi::defcfun ("count_array_val" count-array-val-aux) :void
    (sp :pointer)
    (n :int)
    (vids :pointer)
```

```
579     (c :pointer)
580     (rel-type :int)
581     (val :int)
582 )
583
584 (defun count-array-val (sp vids c rel-type val)
585 "Post the constraint that the number of times that vars[i] = c[i] is equal to val"
586         (let ((x (cffi::foreign-alloc :int :initial-contents vids))
587                 (y (cffi::foreign-alloc :int :initial-contents c)))
588             (count-array-val-aux sp (length vids) x y rel-type val))
589 )
590
591 (cffi::defcfun ("sequence_var" sequence-var-aux) :void
592     "Post the constraint that the number of occurences of s-set in every subsequence of
     length
593     val1 in vids must be higher than val2 and lower than val3"
594     (sp :pointer)
595     (n :int)
596     (vids :pointer)
597     (s :int)
598     (s-set :pointer)
599     (val1 :int)
600     (val2 :int)
601     (val3 :int)
602 )
603
604 (defun sequence-var (sp vids s-set val1 val2 val3)
605     (let ((x (cffi::foreign-alloc :int :initial-contents vids))
606             (y (cffi::foreign-alloc :int :initial-contents s-set)))
607         (sequence-var-aux sp (length vids) x (length s-set) y val1 val2 val3))
608 )
609
610 (cffi::defcfun ("nvalues" nvalues-aux) :void
611     "Post the constraint the number of distinct values in the n variables denoted by vids
612      has the given rel-type relation with the variable vid."
613     (sp :pointer)
614     (n :int)
615     (vids :pointer)
616     (rel-type :int)
617     (vid :int)
618 )
619
620 (defun nvalues (sp vids rel-type vid)
621     "Post the constraint the number of distinct values in the n variables denoted by vids
622     has the given rel-type relation with the variable vid."
623     (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
624         (nvalues-aux sp (length vids) x rel-type vid))
625 )
626
627 (cffi::defcfun ("circuit" hcircuit-aux) :void
628     "Post the constraint that values of vids1 are the edges of an hamiltonian circuit in
629     the graph formed by the n variables in vids1, vids2 are the costs of these edges
     described
630     by c, and vid is the total cost of the circuit, i.e. sum(vids2)."
631     (sp :pointer)
632     (n :int)
633     (c :pointer)
634     (vids1 :pointer)
635     (vids2 :pointer)
636     (vid :int)
637 )
638
639 (defun hcircuit (sp c vids1 vids2 vid)
640     "Post the constraint that values of vids1 are the edges of an hamiltonian circuit in
641     the graph formed by the variables in vids1, vids2 are the costs of these edges described
642     by c, and vid is the total cost of the circuit, i.e. sum(vids2)."
643     (let ((costs (cffi::foreign-alloc :int :initial-contents c))
644           (x (cffi::foreign-alloc :int :initial-contents vids1))
645           (y (cffi::foreign-alloc :int :initial-contents vids2)))
646         (hcircuit-aux sp (length vids1) costs x y vid))
647 )
648
649 (cffi::defcfun ("precede" precede-aux) :void
650     "Post the constraint that if there exists j (0[U+FFFD]j < |x|) such that x[j] = u,
651     then there must exist i with i < j such that x[i] = s"
652     (sp :pointer)
653     (n :int)
654     (vids :pointer)
655     (s :int)
656     (u :int)
657 )
658
659 (defun precede (sp vids s u)
660     "Post the constraint that if there exists j (0[U+FFFD]j < |x|) such that x[j] = u,
661     then there must exist i with i < j such that x[i] = s"
662     (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
663         (precede-aux sp (length vids) x s u)
664     )
665 )
666
667 ;Reification mode
668 (defparameter gil::RM_EQV 0)     ; Equivalent
669 (defparameter gil::RM_IMP 1)     ; Implication
```

```
(defparameter gil::RM_PMI 2)     ; Inverse implication

;BoolVar operation flags
(defparameter gil::BOT_AND 0)     ; logical and
(defparameter gil::BOT_OR 1)      ; logical or
(defparameter gil::BOT_IMP 2)     ; logical implication
(defparameter gil::BOT_EQV 3)     ; logical equivalence
(defparameter gil::BOT_XOR 4)     ; logical exclusive or

(cffi::defcfun ("val_boolop" val-bool-op) :void
    "Post the constraint that val = bool-op(vid1, vid2)."
    (sp :pointer)
    (vid1 :int)
    (bool-op :int)
    (vid2 :int)
    (val :int)
)

(cffi::defcfun ("val_arr_boolop" val-arr-bool-op-aux) :void
    "Post the constraint that elements of vids bool_op val"
    (sp :pointer)
    (bool_op :int)
    (s :int)
    (vids :pointer)
    (val :int)
)

(defun val-arr-bool-op (sp bool-op vids val)
    "Post the constraint that elements of vids bool-op val"
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (val-arr-bool-op-aux sp bool-op (length vids) x val)
    )
)

(cffi::defcfun ("var_arr_boolop" var-arr-bool-op-aux) :void
    "Post the constraint that vid equals bool_op between all element of vids"
    (sp :pointer)
    (bool_op :int)
    (s :int)
    (vids :pointer)
    (vid :int)
)

(defun var-arr-bool-op (sp bool-op vids vid)
    "Post the constraint that vid equals bool_op between all element of vids"
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (var-arr-bool-op-aux sp bool-op (length vids) x vid)
    )
)

(cffi::defcfun ("var_boolop" var-bool-op) :void
    "Post the constraint that vid3 = bool-op(vid1, vid2)."
    (sp :pointer)
    (vid1 :int)
    (bool-op :int)
    (vid2 :int)
    (vid3 :int)
)

(cffi::defcfun ("val_boolrel" val-bool-rel) :void
    "Post boolean rel constraint."
    (sp :pointer)
    (vid :int)
    (rel-type :int)
    (val :int)
)

(cffi::defcfun ("var_boolrel" var-bool-rel) :void
    "Post boolean rel constraint."
    (sp :pointer)
    (vid1 :int)
    (rel-type :int)
    (vid2 :int)
)

;SetVar relation flags
(defparameter gil::SRT_EQ 0)     ; equality relation
(defparameter gil::SRT_NQ 1)     ; inequality
(defparameter gil::SRT_SUB 2)    ; Subset
(defparameter gil::SRT_SUP 3)    ; Superset
(defparameter gil::SRT_DISJ 4)   ; Disjoint
(defparameter gil::SRT_CMPL 5)   ; Complement
(defparameter gil::SRT_LQ 6)     ; Less or equal
(defparameter gil::SRT_LE 7)     ; Strictly lower
(defparameter gil::SRT_GQ 8)     ; Greater or equal
(defparameter gil::SRT_GR 9)     ; Strictly greater

;SetVar operation flags
(defparameter gil::SOT_UNION 0)    ; union
(defparameter gil::SOT_DUNION 1)   ; disjoint union
(defparameter gil::SOT_INTER 2)    ; intersection
(defparameter gil::SOT_MINUS 3)    ; difference

```

```
(cffi::defcfun ("var_setop" var-set-op) :void
    "Post the constraint that vid3 set_rel( set-op(vid1, vid2))."
    (sp :pointer)
    (vid1 :int)
    (set-op :int)
    (vid2 :int)
    (set-rel :int)
    (vid3 :int)
)

(cffi::defcfun ("arr_setop" arr-set-op-aux) :void
    "Post the constraint that vid2 set_op vid1."
    (sp :pointer)
    (set_op :int)
    (s :int)
    (vid1 :pointer)
    (vid2 :int)
)

(defun arr-set-op (sp set_op vid1 vid2)
 "Post the constraint that vid2 set_op vid1."
    (let ((x (cffi::foreign-alloc :int :initial-contents vid1)))
        (arr-set-op-aux sp set_op (length vid1) x vid2))
)

(cffi::defcfun ("var_setrel" var-set-rel) :void
    "Post setVar rel constraint."
    (sp :pointer)
    (vid1 :int)
    (rel-type :int)
    (vid2 :int)
)

(cffi::defcfun ("empty_set" empty-set) :void
    "post that vid1 has to be empty"
    (sp :pointer)
    (vid1 :int)
)

(cffi::defcfun ("val_setrel" val-set-rel-aux) :void
    "Post setVar rel constraint."
    (sp :pointer)
    (vid :int)
    (rel-type :int)
    (dom :pointer)
    (s :int)
)

(defun val-set-rel (sp vid1 rel-type dom)
    "Post the constraint that vid = min(vids)."
    (let ((x (cffi::foreign-alloc :int :initial-contents dom)))
        (val-set-rel-aux sp vid1 rel-type x (length dom)))
)

(cffi::defcfun ("val_setrel_reify" val-set-rel-reify-aux) :void
    "Post setVar rel constraint with reify."
    (sp :pointer)
    (vid :int)
    (rel-type :int)
    (dom :pointer)
    (s :int)
    (r :int)
    (mode :int)
)

(defun val-set-rel-reify (sp vid1 rel-type dom r mode)
    "Post the constraint that vid = min(vids)."
    (let ((x (cffi::foreign-alloc :int :initial-contents dom)))
        (val-set-rel-reify-aux sp vid1 rel-type x (length dom) r mode))
)

(cffi::defcfun ("var_setrel_reify" var-set-rel-reify) :void
    "Post setVar rel constraint with reify."
    (sp :pointer)
    (vid1 :int)
    (rel-type :int)
    (vid2 :int)
    (r :int)
    (mode :int)
)

(cffi::defcfun ("ints_setdom" ints-set-dom) :void
    "Post setVar dom constraint."
    (sp :pointer)
    (vid1 :int)
    (rel-type :int)
    (i :int)
    (j :int)
)

(cffi::defcfun ("set_setdom" set-setdom) :void
    "Post setVar dom constraint."
    (sp :pointer)
```

```
    (vid1 :int)
    (vid2 :int)
)

(cffi::defcfun ("val_card" val-card-aux) :void
    "Post setVar cardinality constraint."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (min-card :int)
    (max-card :int)
)

(defun val-card (sp vids min-card max-card)
    "Post cardinality constraint on the SetVars denoted by vids."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (val-card-aux sp (length vids) x min-card max-card))
)

(cffi::defcfun ("var_card" var-card) :void
    "Post setVar cardinality constraint."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
)

(cffi::defcfun ("var_setrel" var-set-rel) :void
    "Post setVar rel constraint."
    (sp :pointer)
    (vid1 :int)
    (rel-type :int)
    (vid2 :int)
)

(cffi::defcfun ("channel_set" channel-set-aux) :void
    "Post setVar channel constraint."
    (sp :pointer)
    (n1 :int)
    (vids1 :pointer)
    (n2 :int)
    (vids2 :pointer)
)

(defun channel-set (sp vids1 vids2)
    "Post channel constraint on the SetVars denoted by vids."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids1))
          (y (cffi::foreign-alloc :int :initial-contents vids2)))
        (channel-set-aux sp (length vids1) x (length vids2) y))
)

(cffi::defcfun ("channel_set_bool" channel-set-bool-aux) :void
    "Post setVar channel constraint."
    (sp :pointer)
    (n1 :int)
    (vids1 :pointer)
    (vid2 :int)
)

(defun channel-set-bool (sp vids1 vid2)
    "Post channel constraint on the SetVar vid2 and boolVarArray vids1."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids1)))
        (channel-set-bool-aux sp (length vids1) x vid2))
)

(cffi::defcfun ("set_min" set-min) :int
    "Post minimum of SetVar constraint."
    (sp :pointer)
    (vid1 :int)
)

(cffi::defcfun ("set_max" set-max) :int
    "Post maximum of SetVar constraint."
    (sp :pointer)
    (vid1 :int)
)

(cffi::defcfun ("set_min_reify" set-min-reify) :void
    "Post minimum of SetVar constraint with reification."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
    (r :int)
    (mode :int)
)

(cffi::defcfun ("set_max_reify" set-max-reify) :void
    "Post maximum of SetVar constraint with reification"
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
    (r :int)
    (mode :int)
)
```

```
949
950 (cffi::defcfun ("set_union" set-union-aux) :void
951     "Post setVar cardinality constraint."
952     (sp :pointer)
953     (vid1 :int)
954     (n :int)
955     (vids :pointer)
956 )
957
958 (defun set-union (sp vid1 vids)
959     "Post cardinality constraint on the SetVars denoted by vids."
960     (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
961         (set-union-aux sp vid1 (length vids) x))
962 )
963
964 (cffi::defcfun ("element" element-aux) :void
965     "Post setVar element constraint."
966     (sp :pointer)
967     (set-op :int)
968     (n :int)
969     (vids :pointer)
970     (vid1 :int)
971     (vid2 :int)
972 )
973
974 (defun element (sp set-op vids vid1 vid2)
975     "Post cardinality constraint on the SetVars denoted by vids."
976     (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
977         (element-aux sp set-op (length vids) x vid1 vid2))
978 )
979
980 (cffi::defcfun ("branch" branch-aux) :void
981     "Post branching on the n IntVars denoted by vids."
982     (sp :pointer)
983     (n :int)
984     (vids :pointer)
985     (var-strat :int)
986     (val-strat :int)
987 )
988
989 (defun branch (sp vids var-strat val-strat)
990     "Post branching on the IntVars denoted by vids."
991     (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
992         (branch-aux sp (length vids) x var-strat val-strat))
993 )
994
995 (cffi::defcfun ("branch_b" branch-b-aux) :void
996     "Post branching on the n BoolVars denoted by vids."
997     (sp :pointer)
998     (n :int)
999     (vids :pointer)
1000    (var-strat :int)
1001    (val-strat :int)
1002 )
1003
1004 (defun branch-b (sp vids var-strat val-strat)
1005     "Post branching on the BoolVars denoted by vids."
1006     (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
1007         (branch-b-aux sp (length vids) x var-strat val-strat))
1008 )
1009
1010 (cffi::defcfun ("branch_set" branch-set-aux) :void
1011     "Post branching on the n SetVars denoted by vids."
1012     (sp :pointer)
1013     (n :int)
1014     (vids :pointer)
1015     (var_strat :int)
1016     (val_strat :int)
1017 )
1018
1019 (defun branch-set (sp vids var_strat val_strat)
1020     "Post branching on the SetVars denoted by vids."
1021     (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
1022         (branch-set-aux sp (length vids) x var_strat val_strat))
1023 )
1024
1025 (cffi::defcfun ("cost" set-cost) :void
1026     "Define which variable is to be the cost."
1027     (sp :pointer)
1028     (vid :int)
1029 )
1030
1031 (cffi::defcfun ("new_time_stop" new-time-stop) :pointer
1032     "Create a new TimeStop object to specify the time after which the search should stop"
1033     (max-time :int)
1034 )
1035
1036 (cffi::defcfun ("reset_time_stop" reset-time-stop) :void
1037     "Reset the timer of the timeStop object"
1038     (t-stop :pointer)
1039 )
1040
1041 (cffi::defcfun ("new_search_options" new-search-options) :pointer
```

```
1042     "Create a new options object to specify the search options"
1043 )
1044
1045 (cffi::defcfun ("set_nb_threads" set-nb-threads) :int
1046     "Sets the number of threads to use during the search"
1047     (s-opts :pointer)
1048     (n-threads :int)
1049 )
1050
1051 (cffi::defcfun ("set_time_stop" set-t-stop) :pointer
1052     "Sets the stop field of the Options object to the timeStop object"
1053     (s-opts :pointer)
1054     (t-stop :pointer)
1055 )
1056
1057 (cffi::defcfun ("new_bab_engine" bab-engine-low) :pointer
1058     "Create a new branch and bound search-engine."
1059     (sp :pointer)
1060     (opts :pointer)
1061 )
1062
1063 (cffi::defcfun ("bab_next" bab-next) :pointer
1064     "Find the next solution for the search-engine se."
1065     (se :pointer)
1066 )
1067
1068 (cffi::defcfun ("bab_stopped" bab-stopped) :int
1069     "returns t if the search engine has been stopped, nil otherwise"
1070     (se :pointer)
1071 )
1072
1073 (cffi::defcfun ("new_dfs_engine" dfs-engine-low) :pointer
1074     "Create a new depth-first search search-engine."
1075     (sp :pointer)
1076     (opts :pointer)
1077 )
1078
1079 (cffi::defcfun ("dfs_next" dfs-next) :pointer
1080     "Find the next solution for the search-engine se."
1081     (se :pointer)
1082 )
1083
1084 (cffi::defcfun ("dfs_stopped" dfs-stopped) :int
1085     "returns t if the search engine has been stopped, nil otherwise"
1086     (se :pointer)
1087 )
1088
1089 (cffi::defcfun ("get_value" get-value) :int
1090     "Get the value of the variable denoted by vid."
1091     (sp :pointer)
1092     (vid :int)
1093 )
1094
1095 (cffi::defcfun ("get_value_bool" get-value-bool) :int
1096     "Get the value of the variable denoted by vid."
1097     (sp :pointer)
1098     (vid :int)
1099 )
1100
1101 (cffi::defcfun ("get_value_set" get-value-set-aux) :pointer
1102     "Get the value of the variable denoted by vid."
1103     (sp :pointer)
1104     (vid :int)
1105     (n :int)
1106 )
1107
1108 (defun get-value-set (sp vid n)
1109     "get all the values of a SetVar"
1110     (let ((p (get-value-set-aux sp vid n)))
1111         (loop for i from 0 below n
1112             collect (cffi::mem-aref p :int i)))
1113 )
1114
1115 (cffi::defcfun ("get_value_size" get-value-size) :int
1116     "Get the size of the solution of SetVar denoted by vid."
1117     (sp :pointer)
1118     (vid :int)
1119 )
1120
1121 (cffi::defcfun ("get_values" get-values-aux) :pointer
1122     "Get the values of the n variables denoted by vids."
1123     (sp :pointer)
1124     (n :int)
1125     (vids :pointer)
1126 )
1127
1128 (defun get-values (sp vids)
1129     "Print the values of the variables denoted by vids."
1130     (let ((x (cffi::foreign-alloc :int :initial-contents vids))
1131           p)
1132         (setq p (get-values-aux sp (length vids) x))
1133         (loop for i from 0 below (length vids)
1134             collect (cffi::mem-aref p :int i)))
```

```
)

(cffi::defcfun ("print_vars" print-vars-aux) :void
    "Print the values of the n variables denoted by vids."
    (sp :pointer)
    (n :int)
    (vids :pointer)
)

(defun print-vars (sp vids)
    "Print the values of the variables denoted by vids."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (print-vars-aux sp (length vids) x))
)
```

C.2.2 gecode-wrapper-ui.lisp

```
(cl:defpackage "gil"
  (:nicknames "GIL")
   (:use common-lisp :cl-user :cl :cffi))

(in-package :gil)
;;;;;;;;;;;;;;;;;;;;;;;;;
; Creating int variables ;
;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass int-var ()
    ((id :initarg :id :accessor id))
)

(defmethod add-int-var (sp l h)
    "Adds a integer variable with domain [l,h] to sp"
    (make-instance 'int-var :id (add-int-var-low sp l h)))

(defmethod add-int-var-dom (sp dom)
    "Adds a integer variable with domain dom to sp"
    (make-instance 'int-var :id (add-int-var-dom-low sp dom)))

(defmethod add-int-var-array (sp n l h)
    "Adds an array of n integer variables with domain [l,h] to sp"
    (loop for v in (add-int-var-array-low sp n l h) collect
        (make-instance 'int-var :id v)))

(defmethod add-int-var-array-dom (sp n dom)
    "Adds an array of n integer variables with domain dom to sp"
    (loop for v in (add-int-var-array-dom-low sp n dom) collect
        (make-instance 'int-var :id v)))

(defmethod g-specify-sol-variables (sp vids)
    "Specifies the variables that will contain the solution"
    (set-solution-vars sp (vid vids)))

(defmethod g-specify-percent-diff (sp diff)
    "Specifies the percent of modification when searching the next solution"
    (set-percent-diff sp diff))

;id getter
(defmethod vid ((self int-var))
    "Gets the vid of the variable self"
    (id self))

(defmethod vid ((self list))
    "Gets the vids of the variables in self"
    (loop for v in self collect (vid v)))

;;;;;;;;;;;;;;;;;;;;;;;;;
; Creating bool variables ;
;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass bool-var ()
    ((id :initarg :id :accessor id))
)

(defmethod add-bool-var (sp l h)
    "Adds a boolean variable with domain [l,h] to sp"
    (make-instance 'bool-var :id (add-bool-var-range sp l h)))

(defmethod add-bool-var-array (sp n l h)
    "Adds an array of n boolean variables with domain [l,h] to sp"
    (loop for v in (add-bool-var-array-low sp n l h) collect
        (make-instance 'bool-var :id v)))

(defmethod add-bool-var-expr (sp (v1 int-var) rel-type (v2 fixnum))
    "Adds a boolean variable representing the expression
    v1 rel-type v2 to sp"
    (make-instance 'bool-var
        :id (add-bool-var-expr-val sp (vid v1) rel-type v2)))

(defmethod add-bool-var-expr (sp (v1 int-var) rel-type (v2 int-var))
    (make-instance 'bool-var
```

```
        :id (add-bool-var-expr-var sp (vid v1) rel-type (vid v2))))

;id getter
(defmethod vid ((self bool-var)) (id self))

;;;;;;;;;;;;;;;;;;;;;;;;;
; Creating set variables ;
;;;;;;;;;;;;;;;;;;;;;;;;;

(defclass set-var ()
    ((id :initarg :id :accessor id))
)

(defmethod add-set-var (sp lub-min lub-max card-min card-max)
    "Adds a set variable with minimum cardinality card-min and max card-max"
    (make-instance 'set-var :id (add-set-var-card sp lub-min lub-max card-min card-max)))

(defmethod add-set-var-array (sp n lub-min lub-max card-min card-max)
    "Adds an array of n set variables with cardinality card-min to card-max to sp"
    (loop for v in (add-set-var-array-card sp n lub-min lub-max card-min card-max) collect
        (make-instance 'set-var :id v)))

;id getter
(defmethod vid ((self set-var)) (id self))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Methods for int constraints ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;REL
(defmethod g-rel (sp (v1 int-var) rel-type (v2 fixnum))
    "Post the constraint that v1 rel-type v2."
    (val-rel sp (vid v1) rel-type v2))

(defmethod g-rel (sp (v1 int-var) rel-type (v2 int-var))
    (var-rel sp (vid v1) rel-type (vid v2)))

(defmethod g-rel (sp (v1 list) rel-type (v2 null))
    (arr-rel sp (vid v1) rel-type))

(defmethod g-rel (sp (v1 list) rel-type (v2 fixnum))
    (arr-val-rel sp (vid v1) rel-type v2))

(defmethod g-rel (sp (v1 list) rel-type (v2 int-var))
    (arr-var-rel sp (vid v1) rel-type (vid v2)))

(defmethod g-rel (sp (v1 list) rel-type (v2 list))
    (arr-arr-rel sp (vid v1) rel-type (vid v2)))

(defmethod g-rel-reify (sp (v1 int-var) rel-type (v2 int-var) (v3 bool-var) &optional mode)
    (if (not mode)
        (setf mode gil::RM_EQV))
    (var-rel-reify sp (vid v1) rel-type (vid v2) (vid v3) mode))

(defmethod g-rel-reify (sp (v1 int-var) rel-type (v2 fixnum) (v3 bool-var) &optional mode)
    (if (not mode)
        (setf mode gil::RM_EQV))
    (val-rel-reify sp (vid v1) rel-type v2 (vid v3) mode))

;DISTINCT
(defmethod g-distinct (sp vars)
    "Post the constraint that the given vars are distinct."
    (distinct sp (vid vars)))

;LINEAR
(defmethod g-linear (sp coeffs vars rel-type (v fixnum))
    "Post the linear relation coeffs*vars rel-type v."
    (val-linear sp coeffs (vid vars) rel-type v))

(defmethod g-linear (sp coeffs vars rel-type (v int-var))
    (var-linear sp coeffs (vid vars) rel-type (vid v)))

;ARITHMETICS
(defmethod g-abs (sp (v1 int-var) (v2 int-var))
    "Post the constraints that v2 = |v1|."
    (ge-abs sp (vid v1) (vid v2)))

(defmethod g-div (sp (v1 int-var) (v2 int-var) (v3 int-var))
    "Post the constraints that v3 = v1/v2."
    (ge-div sp (vid v1) (vid v2) (vid v3)))

(defmethod g-mod (sp (v1 int-var) (v2 int-var) (v3 int-var))
    "Post the constraints that v3 = v1%v2."
    (var-mod sp (vid v1) (vid v2) (vid v3)))

(defmethod g-divmod (sp (v1 int-var) (v2 int-var) (v3 int-var) (v4 int-var))
    "Post the constraints that v3 = v1/v2 and v4 = v1%v2."
    (ge-divmod sp (vid v1) (vid v2) (vid v3) (vid v4)))

(defmethod g-min (sp (v1 int-var) (v2 int-var) (v3 int-var) &rest vars)
    "Post the constraints that v1 = min(v2, v3, ...)."
    (cond
        ((null vars)
```

```
            (ge-min sp (vid v2) (vid v3) (vid v1)))
        (t (ge-arr-min sp (vid v1)
            (append (list (vid v2) (vid v3)) (vid vars))))))

(defmethod g-lmin (sp (v int-var) vars)
    "Post the constraints that v = min(vars)."
    (ge-arr-min sp  (vid v) (vid vars)))

(defmethod g-argmin (sp vars (v int-var))
    "Post the constraints that v = argmin(vars)."
    (ge-argmin sp (vid vars) (vid v2)))

(defmethod g-max (sp (v1 int-var) (v2 int-var) (v3 int-var) &rest vars)
    "Post the constraints that v1 = max(v2, v3, ...)."
    (cond ((null vars) (ge-max sp (vid v2) (vid v3) (vid v1)))
          (t (ge-arr-max sp (vid v1) (append (list (vid v2) (vid v3)) (vid vars))))))

(defmethod g-lmax (sp (v int-var) vars)
    "Post the constraints that v = max(vars)."
    (ge-arr-max sp (vid v) (vid vars)))

(defmethod g-argmax (sp vars (v int-var))
    "Post the constraints that v2 = argmax(vars)."
    (ge-argmax sp (vid vars) (vid v)))

(defmethod g-mult (sp (v1 int-var) (v2 int-var) (v3 int-var))
    "Post the constraints that v3 = v1*v2."
    (ge-mult sp (vid v1) (vid v2) (vid v3)))

(defmethod g-sqr (sp (v1 int-var) (v2 int-var))
    "Post the constraints that v2 is the square of v1."
    (ge-sqr sp (vid v1) (vid v2)))

(defmethod g-sqrt (sp (v1 int-var) (v2 int-var))
    "Post the constraints that v2 square root of v1."
    (ge-sqrt sp (vid v1) (vid v2)))

(defmethod g-pow (sp (v1 int-var) n (v2 int-var))
    "Post the constraints that v2 nth power of v1."
    (ge-pow sp (vid v1) n (vid v2)))

(defmethod g-nroot (sp (v1 int-var) n (v2 int-var))
    "Post the constraints that v2 is the nth root of v1."
    (ge-nroot sp (vid v1) n (vid v2)))

(defmethod g-sum (sp (v int-var) vars)
    "Post the constraints that v = sum(vars)."
    (rel-sum sp (vid v) (vid vars)))

;DOM
(defmethod g-dom (sp (v int-var) dom)
    "Post the constraints that dom(v) = dom."
    (set-dom sp (vid v) dom))

(defmethod g-member (sp vars (v int-var))
    "Post the constraints that v is in vars."
    (set-member sp (vid vars) (vid v)))


;COUNT
(defmethod g-count (sp vars (v1 fixnum) rel-type (v2 fixnum))
    "Post the constraints that v2 is the number of times v1 occurs in vars."
    (count-val-val sp (vid vars) v1 rel-type v2))

(defmethod g-count (sp vars (v1 fixnum) rel-type (v2 int-var))
    (count-val-var sp (vid vars) v1 rel-type (vid v2)))

(defmethod g-count (sp vars (v1 int-var) rel-type (v2 fixnum))
    (count-var-val sp (vid vars) (vid v1) rel-type v2))

(defmethod g-count (sp vars (v1 int-var) rel-type (v2 int-var))
    (count-var-var sp (vid vars) (vid v1) rel-type (vid v2)))

(defmethod g-count (sp vars (s-set list) rel-type (val fixnum)); ajout[U+FFFD]
    (count-var-set-val sp (vid vars) s-set rel-type val)
)

(defmethod g-count-array (sp vars (c list) rel-type (val fixnum)); ajout[U+FFFD]
    (count-array-val sp (vid vars) c rel-type val)
)

;SEQUENCE
(defmethod g-sequence (sp vars (s-set list) (v1 fixnum) (v2 fixnum) (v3 fixnum))
    (sequence-var sp (vid vars) s-set v1 v2 v3)
)


;NUMBER OF VALUES
(defmethod g-nvalues (sp vars rel-type (v int-var))
    "Post the constraints that v is the number of distinct values in vars."
    (nvalues sp (vid vars) rel-type (vid v)))

;HAMILTONIAN PATH/CIRCUIT
```

```
(defmethod g-circuit (sp costs vars1 vars2 v)
    "Post the constraint that values of vars1 are the edges of an hamiltonian circuit in
    the graph formed by the n variables in vars1, vars2 are the costs of these edges
     described
    by costs, and v is the total cost of the circuit, i.e. sum(vars2)."
    (hcircuit sp costs (vid vars1) (vid vars2) (vid v)))

;VALUE PRECEDENCE
(defmethod g-precede (sp vars s u)
    "Post the constraint that if there exists j (0[U+FFFD]j < |x|) such that x[j] = u,
    then there must exist i with i < j such that x[i] = s"
    (precede sp (vid vars) s u)
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Methods for bool constraints ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;OP
(defmethod g-op (sp (v1 bool-var) bool-op (v2 bool-var) (v3 fixnum))
    "Post the constraints that v1 bool-op v2 = v3."
    (val-bool-op sp (vid v1) bool-op (vid v2) v3))

(defmethod g-op (sp (v1 bool-var) bool-op (v2 bool-var) (v3 bool-var))
    (var-bool-op sp (vid v1) bool-op (vid v2) (vid v3)))

;REL
(defmethod g-rel (sp (v1 bool-var) rel-type (v2 fixnum))
    "Post the constraints that v1 rel-type v2."
    (val-bool-rel sp (vid v1) rel-type v2))

(defmethod g-rel (sp (v1 bool-var) rel-type (v2 bool-var))
    (var-bool-rel sp (vid v1) rel-type (vid v2)))

(defmethod g-rel (sp bool-op (v1 list) (v2 fixnum))
    (val-arr-bool-op sp bool-op (vid v1)  v2))

(defmethod g-rel (sp bool-op (v1 list) (v2 bool-var))
    (var-arr-bool-op sp bool-op (vid v1)  (vid v2)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Methods for setVar constraints ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;OP
(defmethod g-op (sp (v1 set-var) set-op (v2 set-var) (v3 set-var))
    (var-set-op sp (vid v1) set-op (vid v2) gil::SRT_EQ (vid v3)))

(defmethod g-set-op (sp (v1 set-var) set-op (v2 set-var) set_rel (v3 set-var))
    (var-set-op sp (vid v1) set-op (vid v2) set_rel (vid v3)))

(defmethod g-arr-op (sp set-op (v1 list) (v2 set-var))
    (arr-set-op sp set-op (vid v1) (vid v2)))

;REL
(defmethod g-rel (sp (v1 set-var) rel-type (v2 set-var))
    "Post the constraints that v1 rel-type v2."
    (var-set-rel sp (vid v1) rel-type (vid v2)))

(defmethod g-rel (sp (v1 set-var) rel-type dom)
    "Post the constraints that v1 rel-type domain dom."
    (val-set-rel sp (vid v1) rel-type dom))

(defmethod g-rel-reify (sp (v1 set-var) rel-type (dom list) r &optional mode)
    "Post the constraints that v1 rel-type domain dom."
    (if (not mode)
        (setf mode gil::RM_EQV))
    (val-set-rel-reify sp (vid v1) rel-type dom (vid r) mode))

(defmethod g-rel-reify (sp (v1 set-var) rel-type (v2 set-var) r &optional mode)
    "Post the constraints that v1 rel-type domain dom."
    (if (not mode)
        (setf mode gil::RM_EQV))
    (var-set-rel-reify sp (vid v1) rel-type (vid v2) (vid r) mode))

;DOM
(defmethod g-dom (sp (v1 set-var) (v2 set-var))
    "Post the constraints that dom(v) = dom."
    (set-setdom sp (vid v1) (vid v2)))

(defmethod g-dom-ints (sp (v1 set-var) rel-type i j)
    "Post the constraints that v1 rel-type domain {i, ..., j}."
    (ints-set-dom sp (vid v1) rel-type i j))

(defmethod g-empty (sp (v1 set-var))
    "Post the constraints that v1 is empty."
    (empty-set sp (vid v1)))

;CARDINALITY
(defmethod g-card (sp (v1 set-var) min-card max-card)
    (val-card sp (list (vid v1)) min-card max-card))

(defmethod g-card (sp (v list) min-card max-card)
```

```
352     (val-card sp (vid v) min-card max-card))
353
354 (defmethod g-card-var (sp (v1 set-var) (v2 int-var))
355     (var-card sp (vid v1) (vid v2)))
356
357 ;CHANNEL
358 (defmethod g-channel (sp (v1 list) (v2 list))
359     (channel-set sp (vid v1) (vid v2)))
360
361 (defmethod g-channel (sp (v1 list) (v2 set-var))
362     (channel-set-bool sp (vid v1) (vid v2)))
363
364 ;MINIMUM
365 (defmethod g-setmin (sp (v1 set-var))
366     (make-instance 'int-var :id (set-min sp (vid v1))))
367
368 (defmethod g-setmin-reify (sp (v1 set-var) (v2 int-var) (r bool-var) &optional mode)
369     (if (not mode)
370         (setf mode gil::RM_EQV))
371     (set-min-reify sp (vid v1) (vid v2) (vid r) mode))
372
373 ;MAXIMUM
374 (defmethod g-setmax (sp (v1 set-var))
375     (make-instance 'int-var :id (set-max sp (vid v1))))
376
377 (defmethod g-setmax-reify (sp (v1 set-var) (v2 int-var) (r bool-var) &optional mode)
378     (if (not mode)
379         (setf mode gil::RM_EQV))
380     (set-max-reify sp (vid v1) (vid v2) (vid r) mode))
381
382 ;SETUNION
383 (defmethod g-setunion (sp (v1 set-var) (v2 list))
384     (set-union sp (vid v1) (vid v2)))
385
386 ;ELEMENT
387 (defmethod g-element (sp set-op (v1 list) (v2 set-var) (v3 set-var))
388     (element sp set-op (vid v1) (vid v2) (vid v3))
389 )
390
391 ;;;;;;;;;;;;;;;;;;;;;;;;;;
392 ; Methods for exploration ;
393 ;;;;;;;;;;;;;;;;;;;;;;;;;;
394
395 ;;;INTVARS
396
397 ;Variable branching strategies
398 (defparameter gil::INT_VAR_SIZE_MIN 0)    ; select first the variable with the smallest
        domain
399 (defparameter gil::INT_VAR_RND 1) ; select first a random variable
400 (defparameter gil::INT_VAR_DEGREE_MAX 2); select the variable with the highest degree
401 (defparameter gil::INT_VAR_NONE 3) ;select first unassigned
402
403 ;Value branching strategies
404 (defparameter gil::INT_VAL_MIN 0)    ; select first the smallest value of the domain
405 (defparameter gil::INT_VAL_RND 1) ; select first a random value
406 (defparameter gil::INT_VAL_SPLIT_MIN 2) ; select the values not greater than the (min+max)/2
407 (defparameter gil::INT_VAL_SPLIT_MAX 3) ; select the values greater than (min+max)/2
408 (defparameter gil::INT_VAL_MED 4) ; selects the greatest value not bigger than the median
409
410 ;;;SETVARS
411
412 ;Variable branching strategies
413 (defparameter gil::SET_VAR_SIZE_MIN 0)    ; select first the variable with the smallest
        unknown domain
414 (defparameter gil::SET_VAR_RND 1) ; select first a random variable
415 (defparameter gil::SET_VAR_DEGREE_MAX 2); select the variable with the highest degree
416 (defparameter gil::SET_VAR_NONE 3) ;select first unassigned
417
418 ;Value branching strategies
419 (defparameter gil::SET_VAL_MIN_INC 0)    ; select first the smallest value of the domain
420 (defparameter gil::SET_VAL_RND_INC 1) ; select first a random value
421 (defparameter gil::SET_VAL_MIN_EXC 2) ; select the values not greater than the (min+max)/2
422 (defparameter gil::SET_VAL_RND_EXC 3) ; select the values greater than (min+max)/2
423 (defparameter gil::SET_VAL_MED_INC 4) ; selects the greatest value not bigger than the median
424
425 (defmethod g-branch (sp (v int-var) var-strat val-strat)
426     "Post a branching on v with strategies var-strat and val-strat."
427     (branch sp (list (vid v)) var-strat val-strat))
428
429 (defmethod g-branch (sp (v bool-var) var-strat val-strat)
430     (branch-b sp (list (vid v)) var-strat val-strat))
431
432 (defmethod g-branch (sp (v set-var) var-strat val-strat)
433     (branch-set sp (list (vid v)) var-strat val-strat))
434
435 (defmethod g-branch (sp (v list) var-strat val-strat)
436     (if (typep (car v) 'int-var)
437         (branch sp (vid v) var-strat val-strat)
438         (if (typep (car v) 'bool-var)
439             (branch-b sp (vid v) var-strat val-strat)
440             (branch-set sp (vid v) var-strat val-strat))))
441
442 ;cost
```

```
(defmethod g-cost (sp (v int-var))
    "Defines that v is the cost of sp."
    (set-cost sp (vid v)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Methods for search engines ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Methods for search engine options

(defclass time-stop ()
    ((ts :initform nil :initarg ts :accessor ts)); ts is a void pointer to a WTimeStop object
      in Gecode
)

(defmethod t-stop ()
    (make-instance 'time-stop)
)

(defmethod time-stop-init (tstop max-time)
    (setf (ts tstop) (new-time-stop max-time))
)

(defmethod time-stop-reset (tstop)
    (reset-time-stop (ts tstop))
)

(defclass search-options ()
    ((opts :initform nil :initarg opts :accessor opts)); opts is a void pointer to a
     WSearchOptions object in Gecode
)

(defmethod search-opts ()
    (make-instance 'search-options)
)

(defmethod init-search-opts (sopts)
    (setf (opts sopts) (new-search-options))
)

(defmethod set-n-threads (s-opts nthreads)
    (set-nb-threads (opts s-opts) nthreads)
)

(defmethod set-time-stop (s-opts t-stop)
    (set-t-stop (opts s-opts) (ts t-stop))
)

;Search-engine types
(defparameter gil::DFS "dfs")
(defparameter gil::BAB "bab")

(defclass BAB-engine ()
    ((bab :initform nil :initarg :bab :accessor bab))
)

(defclass DFS-engine ()
    ((dfs :initform nil :initarg :dfs :accessor dfs))
)

(defmethod search-engine (sp opts se-type)
    "Creates a new search engine (dfs or bab)."
    (cond
        ((string-equal se-type gil::DFS) (make-instance 'DFS-engine :dfs (dfs-engine-low sp
     opts)))
        ((string-equal se-type gil::BAB) (make-instance 'BAB-engine :bab (bab-engine-low sp
     opts)))
    )
)

;solution exist?
(defun sol? (sol)
    "Existence predicate for a solution"
    (and (not (cffi::null-pointer-p sol)) sol))

;next solution
(defmethod search-next ((se BAB-engine))
    "Search the next solution of se."
    (sol? (bab-next (bab se))))

(defmethod search-next ((se DFS-engine))
    (sol? (dfs-next (dfs se))))

(defmethod search-next ((se null))
    nil)

;stopped
; returns 1 if stopped, 0 if not
(defmethod stopped ((se BAB-engine))
    (bab-stopped (bab se))
)

(defmethod stopped ((se DFS-engine))
```

```
    (dfs-stopped (dfs se))
)

;;;;;;;;;;;;;;;;;;;;;;;;
; Methods for solutions ;
;;;;;;;;;;;;;;;;;;;;;;;;

;values
(defmethod g-values (sp (v int-var))
    "Get the values assigned to v."
    (get-value sp (vid v)))

(defmethod g-values (sp (v bool-var))
    "Get the values assigned to v."
    (get-value-bool sp (vid v)))

(defmethod g-values (sp (v set-var))
    "Get the values assigned to v."
    (get-value-set sp (vid v) (g-value-size sp v)))

(defmethod g-value-size (sp (v set-var))
    "Get the size of a SetVar"
    (get-value-size sp (vid v)))

(defmethod g-values (sp (v list))
    (get-values sp (vid v)))

(defmethod g-values ((sp null) v)
    nil)

;print
(defmethod g-print (sp (v int-var))
    "Print v."
    (print-vars sp (list (vid v))))

(defmethod g-print (sp (v list))
    (print-vars sp (vid v)))

(defmethod g-print ((sp null) v)
    nil)
```

Appendix D

Melodizer source code

In this appendix is the code for Melodizer 2.0, the first iteration of Melodizer was implemented by Damien Sprockeels in 2021 [28], we started from his work to create our version. Nearly everything was modified and written from scratch as a lot of things have been modified between the two versions but we were strongly inspired by Damien's work. This code is composed of four files :

- **block.lisp** : contains the code of the two Open Music objects "block" and "search" and their interfaces.
- **melodizer-csp.lisp** : contains the creation of the musical CSP, the creation of the search engine and the call to find the next solutions.
- **melodizer-csts.lisp** : contains some of the musical constraints that would obfuscate melodizer-csp.lisp.
- **melodizer-utils.lisp** : contains some utility functions used throughout the other files

You can find the complete code for melodizer and more on github at https://github.com/clemsky/TFE-Composition-Musicale

D.1 block.lisp

```
(in-package :mldz)

;;;=====================
;;;= BLOCK OBJECT =
;;;=====================

(om::defclass! block ()
  ;attributes
  ((block-list :accessor block-list :initarg :block-list :initform nil :documentation "")
   (melody-source :accessor melody-source :initarg :melody-source :initform nil :
     documentation "")
   (position-list :accessor position-list :initarg :position-list :initform nil :
     documentation "")
   (bar-length :accessor bar-length :initform 0 :type integer)
   (beat-length :accessor beat-length :initform 0 :type integer)
   (voices :accessor voices :initform nil :type integer)
   (min-pushed-notes :accessor min-pushed-notes :initform nil :type integer)
   (max-pushed-notes :accessor max-pushed-notes :initform nil :type integer)
   (min-notes :accessor min-notes :initform nil :type integer)
   (max-notes :accessor max-notes :initform nil :type integer)
   (min-added-notes :accessor min-added-notes :initform nil :type integer)
```

```
20    (max-added-notes :accessor max-added-notes :initform nil :type integer)
21    (min-note-length-flag :accessor min-note-length-flag :initform nil :type integer)
22    (min-note-length :accessor min-note-length :initform 0 :type integer)
23    (max-note-length-flag :accessor max-note-length-flag :initform nil :type integer)
24    (max-note-length :accessor max-note-length :initform 192 :type integer)
25    (quantification :accessor quantification :initform nil :type string)
26    (note-repartition-flag :accessor note-repartition-flag :initform nil :type integer)
27    (note-repartition :accessor note-repartition :initform nil :type integer)
28    (rhythm-repetition :accessor rhythm-repetition :initform nil :type string)
29    (pause-quantity-flag :accessor pause-quantity-flag :initform nil :type integer)
30    (pause-quantity :accessor pause-quantity :initform 0 :type integer)
31    (pause-repartition-flag :accessor pause-repartition-flag :initform nil :type integer)
32    (pause-repartition :accessor pause-repartition :initform 0 :type integer)
33    (key-selection :accessor key-selection :initform nil :type string)
34    (mode-selection :accessor mode-selection :initform nil :type string)
35    (chord-key :accessor chord-key :initform nil :type string)
36    (chord-quality :accessor chord-quality :initform nil :type string)
37    (all-chord-notes :accessor all-chord-notes :initform nil :type integer)
38    (min-pitch :accessor min-pitch :initform 1 :type integer)
39    (min-pitch-flag :accessor min-pitch-flag :initform nil :type integer)
40    (max-pitch :accessor max-pitch :initform 127 :type integer)
41    (max-pitch-flag :accessor max-pitch-flag :initform nil :type integer)
42    (pitch-direction :accessor pitch-direction :initform nil :type string)
43    (golomb-ruler-size :accessor golomb-ruler-size :initform 0 :type integer)
44    (note-repetition-flag :accessor note-repetition-flag :initform nil :type integer)
45    (note-repetition-type :accessor note-repetition-type :initform "Random" :type string)
46    (note-repetition :accessor note-repetition :initform 0 :type integer)
47  )
48  (:icon 225)
49  (:documentation "This class implements Melodizer.
50        Melodizer is a constraints based application aiming to improve composer's expression
    and exploration abilities
51        by generating interesting and innovative melodies based on a set of constraints
    expressing musical rules.
52        More information and a tutorial can be found at https://github.com/sprockeelsd/
    Melodizer")
53 )
54
55 (om::defclass! search ()
56   ;attributes
57   (
58     (block-csp :accessor block-csp :initarg :block-csp :initform nil)
59     (solution :accessor solution :initarg :solution :initform nil :documentation "The current
       solution of the CSP in the form of a voice object.")
60     (result :accessor result
61       :result :initform (list) :documentation
62       "A temporary list holder to store the result of the call to the CSPs, shouldn't be
    touched.")
63     (stop-search :accessor stop-search :stop-search :initform nil :documentation
64       "A boolean variable to tell if the user wishes to stop the search or not.")
65     (input-rhythm :accessor input-rhythm :input-rhythm :initform (make-instance 'voice) :
    documentation
66       "The rhythm of the melody or a melody in the form of a voice object. ")
67     (tempo :accessor tempo :initform 120 :type integer :documentation
68       "The tempo (BPM) of the project")
69     (branching :accessor branching :initform "Top down" :type string :documentation
70       "The tempo (BPM) of the project")
71     (percent-diff :accessor percent-diff :initform 0 :type integer)
72   )
73   (:icon 225)
74   (:documentation "This class implements Melodizer.
75        Melodizer is a constraints based application aiming to improve composer's expression
    and exploration abilities
76        by generating interesting and innovative melodies based on a set of constraints
    expressing musical rules.
77        More information and a tutorial can be found at https://github.com/sprockeelsd/
    Melodizer")
78 )
79
80 ; the editor for the object
81 (defclass block-editor (om::editorview) ())
82
83 (defmethod om::class-has-editor-p ((self block)) t)
84 (defmethod om::get-editor-class ((self block)) 'block-editor)
85
86 (defmethod om::om-draw-contents ((view block-editor))
87   (let* ((object (om::object view)))
88     (om::om-with-focused-view
89       view
90       ;;; DRAW SOMETHING ?
91     )
92   )
93 )
94
95 ; To access the melodizer object, (om::object self)
96
97 (defmethod initialize-instance ((self block-editor) &rest args)
98   ;;; do what needs to be done by default
99   (call-next-method) ; start the search by default?
100  (make-my-interface self)
101 )
102
103 ; function to create the tool's interface
```

```
(defmethod make-my-interface ((self block-editor))

  ; create the main view of the object
  (make-main-view self)

  ;;;;;;;;;;;;;;;;
  ;;; sub views ;;;
  ;;;;;;;;;;;;;;;;

  (let*
    (
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ;;; setting the different regions of the tool ;;;
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

      (block-constraints-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 400 605)
        :position (om::om-make-point 5 5)
        :bg-color om::*azulito*)
      )


      ; part of the display for everything that has to do with adding new constraints to the
    problem
      (time-constraints-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 400 605)
        :position (om::om-make-point 410 5)
        :bg-color om::*azulito*)
      )
      ; part of the display to put different solutions together
      (pitch-constraints-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 400 605)
        :position (om::om-make-point 815 5)
        :bg-color om::*azulito*)
      )
    )

    (setf elements-block-constraints-panel (make-block-constraints-panel self
     block-constraints-panel))

    (setf elements-time-constraints-panel (make-time-constraints-panel self
     time-constraints-panel))
    ; create the pitch constrains panel
    (setf elements-pitch-constraints-panel (make-pitch-constraints-panel self
     pitch-constraints-panel))
    ; add the subviews for the different parts into the main view
    (om::om-add-subviews
      self
      block-constraints-panel
      time-constraints-panel
      pitch-constraints-panel
    )
  )
  ; return the editor
  self
)



    ;;;;;;;;;;;;;;;;;
    ;;; main view ;;;
    ;;;;;;;;;;;;;;;;;

; this function creates the elements for the main panel
(defun make-main-view (editor)
  ; background colour
  (om::om-set-bg-color editor om::*om-light-gray-color*) ;pour changer le bg color. om peut
     fabriquer sa propre couleur: (om-make-color r g b)
)

    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ;;; creating the constraints panel ;;;
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun make-block-constraints-panel (editor block-constraints-panel)
  (om::om-add-subviews
    block-constraints-panel
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 150 2)
      (om::om-make-point 120 20)
      "Block constraints"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 15 50)
      (om::om-make-point 200 20)
      "Bar length"
      :font om::*om-default-font1b*
    )

```

```
192     (om::om-make-dialog-item
193       'om::pop-up-menu
194       (om::om-make-point 170 50)
195       (om::om-make-point 200 20)
196       "Bar length"
197       :range (loop :for n :from 0 :upto 32 collect n)
198       :di-action #'(lambda (m)
199         (setf (bar-length (om::object editor)) (nth (om::om-get-selected-item-index m) (om::
    om-get-item-list m)))
200       )
201     )
202
203     ; (om::om-make-dialog-item
204     ;   'om::om-static-text
205     ;   (om::om-make-point 15 100)
206     ;   (om::om-make-point 200 20)
207     ;   "Beat length"
208     ;   :font om::*om-default-font1b*
209     ; )
210     ;
211     ; (om::om-make-dialog-item
212     ;   'om::pop-up-menu
213     ;   (om::om-make-point 170 100)
214     ;   (om::om-make-point 200 20)
215     ;   "Beat length"
216     ;   :range '(0 1 2 3)
217     ;   :di-action #'(lambda (m)
218     ;     (setf (beat-length (om::object editor)) (nth (om::om-get-selected-item-index m) (om
    ::om-get-item-list m)))
219     ;   )
220     ; )
221
222     (om::om-make-dialog-item
223       'om::om-static-text
224       (om::om-make-point 15 100)
225       (om::om-make-point 200 20)
226       "Voices"
227       :font om::*om-default-font1b*
228     )
229
230     (om::om-make-dialog-item
231       'om::pop-up-menu
232       (om::om-make-point 170 100)
233       (om::om-make-point 200 20)
234       "Voices"
235       :range (append '("None") (loop :for n :from 0 :upto 15 collect n))
236       :di-action #'(lambda (m)
237         (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
238         (if (typep check 'string)
239           (setf (voices (om::object editor)) nil)
240           (setf (voices (om::object editor)) check))
241       )
242     )
243
244     (om::om-make-dialog-item
245       'om::om-static-text
246       (om::om-make-point 15 150)
247       (om::om-make-point 200 20)
248       "Minimum pushed notes"
249       :font om::*om-default-font1b*
250     )
251
252     (om::om-make-dialog-item
253       'om::pop-up-menu
254       (om::om-make-point 170 150)
255       (om::om-make-point 200 20)
256       "Minimum pushed notes"
257       :range (append '("None") (loop :for n :from 0 :upto 10 collect n))
258       :di-action #'(lambda (m)
259         (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
260         (if (typep check 'string)
261           (setf (min-pushed-notes (om::object editor)) nil)
262           (setf (min-pushed-notes (om::object editor)) check))
263       )
264     )
265
266     (om::om-make-dialog-item
267       'om::om-static-text
268       (om::om-make-point 15 200)
269       (om::om-make-point 200 20)
270       "Maximum pushed notes"
271       :font om::*om-default-font1b*
272     )
273
274     (om::om-make-dialog-item
275       'om::pop-up-menu
276       (om::om-make-point 170 200)
277       (om::om-make-point 200 20)
278       "Maximum pushed notes"
279       :range (append '("None") (loop :for n :from 0 :upto 10 collect n))
280       :di-action #'(lambda (m)
281         (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
282         (if (typep check 'string)
```

```
283             (setf (max-pushed-notes (om::object editor)) nil)
284             (setf (max-pushed-notes (om::object editor)) check))
285       )
286     )
287
288     (om::om-make-dialog-item
289       'om::om-static-text
290       (om::om-make-point 15 250)
291       (om::om-make-point 200 20)
292       "Minimum notes"
293       :font om::*om-default-font1b*
294     )
295
296     (om::om-make-dialog-item
297       'om::pop-up-menu
298       (om::om-make-point 170 250)
299       (om::om-make-point 200 20)
300       "Minimum notes"
301       :range (append '("None") (loop :for n :from 0 :upto 100 collect n))
302       :di-action #'(lambda (m)
303         (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
304         (if (typep check 'string)
305           (setf (min-notes (om::object editor)) nil)
306           (setf (min-notes (om::object editor)) check))
307       )
308     )
309
310     (om::om-make-dialog-item
311       'om::om-static-text
312       (om::om-make-point 15 300)
313       (om::om-make-point 200 20)
314       "Maximum notes"
315       :font om::*om-default-font1b*
316     )
317
318     (om::om-make-dialog-item
319       'om::pop-up-menu
320       (om::om-make-point 170 300)
321       (om::om-make-point 200 20)
322       "Maximum notes"
323       :range (append '("None") (loop :for n :from 0 :upto 100 collect n))
324       :di-action #'(lambda (m)
325         (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
326         (if (typep check 'string)
327           (setf (max-notes (om::object editor)) nil)
328           (setf (max-notes (om::object editor)) check))
329       )
330     )
331
332     (om::om-make-dialog-item
333       'om::om-static-text
334       (om::om-make-point 15 350)
335       (om::om-make-point 200 20)
336       "Minimum added notes"
337       :font om::*om-default-font1b*
338     )
339
340     (om::om-make-dialog-item
341       'om::pop-up-menu
342       (om::om-make-point 170 350)
343       (om::om-make-point 200 20)
344       "Minimum added notes"
345       :range (append '("None") (loop :for n :from 0 :upto 100 collect n))
346       :di-action #'(lambda (m)
347         (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
348         (if (typep check 'string)
349           (setf (min-added-notes (om::object editor)) nil)
350           (setf (min-added-notes (om::object editor)) check))
351       )
352     )
353
354     (om::om-make-dialog-item
355       'om::om-static-text
356       (om::om-make-point 15 400)
357       (om::om-make-point 200 20)
358       "Maximum added notes"
359       :font om::*om-default-font1b*
360     )
361
362     (om::om-make-dialog-item
363       'om::pop-up-menu
364       (om::om-make-point 170 400)
365       (om::om-make-point 200 20)
366       "Maximum added notes"
367       :range (append '("None") (loop :for n :from 0 :upto 100 collect n))
368       :di-action #'(lambda (m)
369         (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
370         (if (typep check 'string)
371           (setf (max-added-notes (om::object editor)) nil)
372           (setf (max-added-notes (om::object editor)) check))
373       )
374     )
375   )
```

```
376
377
378 )
379
380 ; this function creates the elements of the main additional constraints panel
381 ; coordinates here are local to constraint-panel
382 (defun make-time-constraints-panel (editor time-constraints-panel)
383   (om::om-add-subviews
384     time-constraints-panel
385
386     ; title
387     (om::om-make-dialog-item
388       'om::om-static-text
389       (om::om-make-point 150 2)
390       (om::om-make-point 120 20)
391       "Time constraints"
392       :font om::*om-default-font1b*
393     )
394
395     (om::om-make-dialog-item
396       'om::om-static-text
397       (om::om-make-point 15 50)
398       (om::om-make-point 200 20)
399       "Minimum note length"
400       :font om::*om-default-font1b*
401     )
402
403     (om::om-make-dialog-item
404       'om::om-check-box
405       (om::om-make-point 170 50)
406       (om::om-make-point 20 20)
407       ""
408       :di-action #'(lambda (c)
409                      (if (om::om-checked-p c)
410                        (setf (min-note-length-flag (om::object editor)) 1)
411                        (setf (min-note-length-flag (om::object editor)) nil)
412                      )
413       )
414     )
415
416     ; slider to express how different the solutions should be (100 = completely different, 1
      = almost no difference)
417     (om::om-make-dialog-item
418       'om::om-slider
419       (om::om-make-point 190 50)
420       (om::om-make-point 180 20); size
421       "Minimum note length"
422       :range '(0 192)
423       :increment 1
424       :di-action #'(lambda (s)
425         (setf (min-note-length (om::object editor)) (om::om-slider-value s))
426       )
427     )
428
429     (om::om-make-dialog-item
430       'om::om-static-text
431       (om::om-make-point 15 100)
432       (om::om-make-point 200 20)
433       "Maximum note length"
434       :font om::*om-default-font1b*
435     )
436
437     (om::om-make-dialog-item
438       'om::om-check-box
439       (om::om-make-point 170 100)
440       (om::om-make-point 200 20)
441       ""
442       :di-action #'(lambda (c)
443                      (if (om::om-checked-p c)
444                        (setf (max-note-length-flag (om::object editor)) 1)
445                        (setf (max-note-length-flag (om::object editor)) nil)
446                      )
447       )
448     )
449
450     (om::om-make-dialog-item
451       'om::om-slider
452       (om::om-make-point 190 100)
453       (om::om-make-point 180 20); size
454       "Maximum note length"
455       :range '(0 192)
456       :increment 1
457       :di-action #'(lambda (s)
458         (setf (max-note-length (om::object editor)) (om::om-slider-value s))
459       )
460     )
461
462     (om::om-make-dialog-item
463       'om::om-static-text
464       (om::om-make-point 15 150)
465       (om::om-make-point 200 20)
466       "Quantification"
467       :font om::*om-default-font1b*
```

```
468     )
469
470     (om::om-make-dialog-item
471       'om::pop-up-menu
472       (om::om-make-point 170 150)
473       (om::om-make-point 200 20)
474       "Quantification"
475       :range '("None" "1 bar" "1/2 bar" "1 beat" "1/2 beat" "1/4 beat" "1/8 beat" "1/3 bar" "
      1/6 bar" "1/3 beat" "1/6 beat" "1/12 beat")
476       :di-action #'(lambda (m)
477         (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
478         (if (string= check "None")
479           (setf (quantification (om::object editor)) nil)
480           (setf (quantification (om::object editor)) check))
481       )
482     )
483
484     ; (om::om-make-dialog-item
485     ;   'om::om-static-text
486     ;   (om::om-make-point 15 200)
487     ;   (om::om-make-point 200 20)
488     ;   "Note repartition"
489     ;   :font om::*om-default-font1b*
490     ; )
491     ;
492     ; (om::om-make-dialog-item
493     ;   'om::om-check-box
494     ;   (om::om-make-point 170 200)
495     ;   (om::om-make-point 200 20)
496     ;   ""
497     ;   :di-action #'(lambda (c)
498     ;                  (if (om::om-checked-p c)
499     ;                    (setf (note-repartition-flag (om::object editor)) 1)
500     ;                    (setf (note-repartition-flag (om::object editor)) nil)
501     ;                  )
502     ;   )
503     ; )
504     ;
505     ; ; slider to express how different the solutions should be (100 = completely different,
      1 = almost no difference)
506     ; (om::om-make-dialog-item
507     ;   'om::om-slider
508     ;   (om::om-make-point 190 200)
509     ;   (om::om-make-point 180 20); size
510     ;   "Note repartition"
511     ;   :range '(1 100)
512     ;   :increment 1
513     ;   :di-action #'(lambda (s)
514     ;     (setf (note-repartition (om::object editor)) (om::om-slider-value s))
515     ;   )
516     ; )
517
518     (om::om-make-dialog-item
519       'om::om-static-text
520       (om::om-make-point 15 200)
521       (om::om-make-point 200 20)
522       "Rhythm repetition"
523       :font om::*om-default-font1b*
524     )
525
526     (om::om-make-dialog-item
527       'om::pop-up-menu
528       (om::om-make-point 170 200)
529       (om::om-make-point 200 20)
530       "Rhythm repetition"
531       :range '("None" "1 bar" "1/2 bar" "1 beat" "1/2 beat" "1/4 beat" "1/8 beat" "1/3 bar" "
      1/6 bar" "1/3 beat" "1/6 beat" "1/12 beat")
532       :di-action #'(lambda (m)
533         (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
534         (if (string= check "None")
535           (setf (rhythm-repetition (om::object editor)) nil)
536           (setf (rhythm-repetition (om::object editor)) check))
537       )
538     )
539
540     (om::om-make-dialog-item
541       'om::om-static-text
542       (om::om-make-point 15 250)
543       (om::om-make-point 200 20)
544       "Pause quantity"
545       :font om::*om-default-font1b*
546     )
547
548     (om::om-make-dialog-item
549       'om::om-check-box
550       (om::om-make-point 170 250)
551       (om::om-make-point 20 20)
552       ""
553       :di-action #'(lambda (c)
554                      (if (om::om-checked-p c)
555                        (setf (pause-quantity-flag (om::object editor)) 1)
556                        (setf (pause-quantity-flag (om::object editor)) nil)
557                      )
```

```
558       )
559     )
560
561     ; slider to express how different the solutions should be (100 = completely different, 1
        = almost no difference)
562     (om::om-make-dialog-item
563       'om::om-slider
564       (om::om-make-point 190 250)
565       (om::om-make-point 180 20); size
566       "Pause quantity"
567       :range '(1 192)
568       :increment 1
569       :di-action #'(lambda (s)
570         (setf (pause-quantity (om::object editor)) (om::om-slider-value s))
571       )
572     )
573
574     (om::om-make-dialog-item
575       'om::om-static-text
576       (om::om-make-point 15 300)
577       (om::om-make-point 200 20)
578       "Pause repartition"
579       :font om::*om-default-font1b*
580     )
581
582     (om::om-make-dialog-item
583       'om::om-check-box
584       (om::om-make-point 170 300)
585       (om::om-make-point 20 20)
586       ""
587       :di-action #'(lambda (c)
588                      (if (om::om-checked-p c)
589                        (setf (pause-repartition-flag (om::object editor)) 1)
590                        (setf (pause-repartition-flag (om::object editor)) nil)
591                      )
592       )
593     )
594
595     ; slider to express how different the solutions should be (100 = completely different, 1
        = almost no difference)
596     (om::om-make-dialog-item
597       'om::om-slider
598       (om::om-make-point 190 300)
599       (om::om-make-point 180 20); size
600       "Pause repartition"
601       :range '(0 191)
602       :increment 1
603       :di-action #'(lambda (s)
604         (setf (pause-repartition (om::object editor)) (om::om-slider-value s))
605       )
606     )
607   )
608 )
609
610 (defun make-pitch-constraints-panel (editor pitch-constraints-panel)
611   (om::om-add-subviews
612     pitch-constraints-panel
613
614     ; title
615     (om::om-make-dialog-item
616       'om::om-static-text
617       (om::om-make-point 150 2)
618       (om::om-make-point 200 20)
619       "Pitch constraints"
620       :font om::*om-default-font1b*
621     )
622
623     ; Key
624
625     (om::om-make-dialog-item
626       'om::om-static-text
627       (om::om-make-point 15 50)
628       (om::om-make-point 200 20)
629       "Key selection"
630       :font om::*om-default-font1b*
631     )
632
633     (om::om-make-dialog-item
634       'om::pop-up-menu
635       (om::om-make-point 170 50)
636       (om::om-make-point 200 20)
637       "Key selection"
638       :range '("None" "C" "C#" "D" "Eb" "E" "F" "F#" "G" "Ab" "A" "Bb" "B")
639       :di-action #'(lambda (m)
640         (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
641         (if (string= check "None")
642           (setf (key-selection (om::object editor)) nil)
643           (setf (key-selection (om::object editor)) check))
644       )
645     )
646
647     ; Mode
648     (om::om-make-dialog-item
```

```
      'om::om-static-text
      (om::om-make-point 15 100)
      (om::om-make-point 200 20)
      "Mode selection"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 170 100)
      (om::om-make-point 200 20)
      "Mode selection"
      :range '("None" "ionian (major)" "dorian" "phrygian" "lydian" "mixolydian" "aeolian (natural minor)" "locrian" "pentatonic" "harmonic minor" "chromatic")
      :di-action #'(lambda (m)
        (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
        (if (string= check "None")
          (setf (mode-selection (om::object editor)) nil)
          (setf (mode-selection (om::object editor)) check))
      )
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 15 150)
      (om::om-make-point 200 20)
      "Chord key"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 170 150)
      (om::om-make-point 200 20)
      "Chord key"
      :range '("None" "C" "C#" "D" "Eb" "E" "F" "F#" "G" "Ab" "A" "Bb" "B")
      :di-action #'(lambda (m)
        (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
        (if (string= check "None")
          (setf (chord-key (om::object editor)) nil)
          (setf (chord-key (om::object editor)) check))
      )
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 15 200)
      (om::om-make-point 200 20)
      "Chord quality"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 170 200)
      (om::om-make-point 200 20)
      "Chord quality"
      :range '("None" "Major" "Minor" "Augmented" "Diminished" "Major 7" "Minor 7" "Dominant 7" "Minor 7 flat 5" "Diminished 7" "Minor-major 7"
        "Major 9" "Minor 9" "9 Augmented 5" "9 flatted 5" "7 flat 9" "Augmented 9" "Minor 11" "Major 11" "Dominant 11" "Dominant # 11" "Major # 11")
      :di-action #'(lambda (m)
        (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
        (if (string= check "None")
          (setf (chord-quality (om::object editor)) nil)
          (setf (chord-quality (om::object editor)) check))
      )
    )

    ; ;checkbox for all-different constraint
    ; (om::om-make-dialog-item
    ;   'om::om-check-box
    ;   (om::om-make-point 170 250)
    ;   (om::om-make-point 200 20)
    ;   "All chord notes"
    ;   ;:checked-p (find "all-different-notes" (optional-constraints (om::object editor)) :test #'equal)
    ;   :di-action #'(lambda (c)
    ;                  (if (om::om-checked-p c)
    ;                    (setf (all-chord-notes (om::object editor)) 1)
    ;                    (setf (all-chord-notes (om::object editor)) nil)
    ;                  )
    ;   )
    ;   :font om::*om-default-font1*
    ; )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 15 250)
      (om::om-make-point 200 20)
      "Minimum pitch"
      :font om::*om-default-font1b*
    )
```

```
738
739     (om::om-make-dialog-item
740       'om::om-check-box
741       (om::om-make-point 170 250)
742       (om::om-make-point 20 20)
743       ""
744       :di-action #'(lambda (c)
745                      (if (om::om-checked-p c)
746                        (setf (min-pitch-flag (om::object editor)) 1)
747                        (setf (min-pitch-flag (om::object editor)) nil)
748                      )
749       )
750     )
751
752     (om::om-make-dialog-item
753       'om::slider
754       (om::om-make-point 190 250)
755       (om::om-make-point 180 20)
756       "Minimum pitch"
757       :range '(1 127)
758       :increment 1
759       :di-action #'(lambda (s)
760         (setf (min-pitch (om::object editor)) (om::om-slider-value s))
761       )
762     )
763
764     (om::om-make-dialog-item
765       'om::om-static-text
766       (om::om-make-point 15 300)
767       (om::om-make-point 200 20)
768       "Maximum pitch"
769       :font om::*om-default-font1b*
770     )
771
772     (om::om-make-dialog-item
773       'om::om-check-box
774       (om::om-make-point 170 300)
775       (om::om-make-point 20 20)
776       ""
777       :di-action #'(lambda (c)
778                      (if (om::om-checked-p c)
779                        (setf (max-pitch-flag (om::object editor)) 1)
780                        (setf (max-pitch-flag (om::object editor)) nil)
781                      )
782       )
783     )
784
785     (om::om-make-dialog-item
786       'om::slider
787       (om::om-make-point 190 300)
788       (om::om-make-point 180 20)
789       "Maximum pitch"
790       :range '(1 127)
791       :increment 1
792       :di-action #'(lambda (s)
793         (setf (max-pitch (om::object editor)) (om::om-slider-value s))
794       )
795     )
796
797     (om::om-make-dialog-item
798       'om::om-static-text
799       (om::om-make-point 15 350)
800       (om::om-make-point 200 20)
801       "Note repetition"
802       :font om::*om-default-font1b*
803     )
804
805     (om::om-make-dialog-item
806       'om::om-check-box
807       (om::om-make-point 170 350)
808       (om::om-make-point 20 20)
809       ""
810       :di-action #'(lambda (c)
811                      (if (om::om-checked-p c)
812                        (setf (note-repetition-flag (om::object editor)) 1)
813                        (setf (note-repetition-flag (om::object editor)) nil)
814                      )
815       )
816     )
817
818     (om::om-make-dialog-item
819       'om::slider
820       (om::om-make-point 190 350)
821       (om::om-make-point 180 20)
822       "Note repetition"
823       :range '(0 100)
824       :increment 1
825       :di-action #'(lambda (s)
826         (setf (note-repetition (om::object editor)) (om::om-slider-value s))
827       )
828     )
829
830     (om::om-make-dialog-item
```

```
831         'om::om-static-text
832         (om::om-make-point 15 400)
833         (om::om-make-point 200 20)
834         "Repetition type"
835         :font om::*om-default-font1b*
836       )
837
838       (om::om-make-dialog-item
839           'om::pop-up-menu
840           (om::om-make-point 170 400)
841           (om::om-make-point 200 20)
842           "Repetition type"
843           :range '("Random" "Soft" "Hard")
844           :di-action #'(lambda (m)
845              (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
846              (setf (note-repetition-type (om::object editor)) check)
847           )
848       )
849
850       (om::om-make-dialog-item
851         'om::om-static-text
852         (om::om-make-point 15 450)
853         (om::om-make-point 200 20)
854         "Pitch direction"
855         :font om::*om-default-font1b*
856       )
857
858       (om::om-make-dialog-item
859           'om::pop-up-menu
860           (om::om-make-point 170 450)
861           (om::om-make-point 200 20)
862           "Pitch direction"
863           :range '("None" "Increasing" "Strictly increasing" "Decreasing" "Strictly decreasing")
864           :di-action #'(lambda (m)
865             (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
866             (if (string= check "None")
867               (setf (pitch-direction (om::object editor)) nil)
868               (setf (pitch-direction (om::object editor)) check))
869           )
870       )
871
872      (om::om-make-dialog-item
873           'om::om-static-text
874         (om::om-make-point 15 500)
875         (om::om-make-point 200 20)
876         "Golomb ruler size"
877         :font om::*om-default-font1b*
878       )
879
880       (om::om-make-dialog-item
881           'om::pop-up-menu
882           (om::om-make-point 170 500)
883           (om::om-make-point 200 20)
884           "Golomb ruler size"
885           :range '("None" "1" "2" "3" "4" "5" "6" "7" "8" "9")
886           :di-action #'(lambda (m)
887             (setf (golomb-ruler-size (om::object editor)) (om::om-get-selected-item-index m))
888           )
889       )
890
891     )
892   )
893
894   ; the editor for the object
895   (defclass search-editor (om::editorview) ())
896
897   (defmethod om::class-has-editor-p ((self search)) t)
898   (defmethod om::get-editor-class ((self search)) 'search-editor)
899
900   (defmethod om::om-draw-contents ((view search-editor))
901     (let* ((object (om::object view)))
902       (om::om-with-focused-view
903         view
904         ;;; DRAW SOMETHING ?
905       )
906     )
907   )
908
909   (defmethod initialize-instance ((self search-editor) &rest args)
910     ;;; do what needs to be done by default
911     (call-next-method) ; start the search by default?
912     (make-my-interface self)
913   )
914
915   ; function to create the tool's interface
916   (defmethod make-my-interface ((self search-editor))
917
918     ; create the main view of the object
919     (make-main-view self)
920
921     (let*
922       (
923         (search-panel (om::om-make-view 'om::om-view
```

```
924           :size (om::om-make-point 400 605)
925           :position (om::om-make-point 5 5)
926           :bg-color om::*azulito*)
927         )
928       )
929
930       (setf elements-search-panel (make-search-panel self search-panel))
931
932       (om::om-add-subviews
933         self
934         search-panel
935       )
936     )
937     self
938   )
939
940   (defun make-search-panel (editor search-panel)
941     (om::om-add-subviews
942       search-panel
943       (om::om-make-dialog-item
944         'om::om-static-text
945         (om::om-make-point 145 2)
946         (om::om-make-point 120 20)
947         "Search Parameters"
948         :font om::*om-default-font1b*
949       )
950
951       (om::om-make-dialog-item
952         'om::om-button
953         (om::om-make-point 5 50) ; position (horizontal, vertical)
954         (om::om-make-point 130 20) ; size (horizontal, vertical)
955         "Start"
956         :di-action #'(lambda (b)
957           (let init
958             (setq init (new-melodizer (block-csp (om::object editor)) (percent-diff (om::object
      editor)) (branching (om::object editor))))
959             (setf (result (om::object editor)) init)
960             ; TO TEST THE GOLOMB RULER PROGRAM
961             ;(setq init (golomb-ruler 5))
962             ;(setf (result (om::object editor)) init)
963           )
964         )
965       )
966
967       (om::om-make-dialog-item
968         'om::om-button
969         (om::om-make-point 135 50) ; position
970         (om::om-make-point 130 20) ; size
971         "Next"
972         :di-action #'(lambda (b)
973           (if (typep (result (om::object editor)) 'null); if the problem is not initialized
974             (error "The problem has not been initialized. Please set the input and press Start.
      ")
975           )
976           (print "Searching for the next solution")
977           ;reset the boolean because we want to continue the search
978           (setf (stop-search (om::object editor)) nil)
979           ;get the next solution
980           (mp:process-run-function ; start a new thread for the execution of the next method
981             "next thread" ; name of the thread, not necessary but useful for debugging
982             nil ; process initialization keywords, not needed here
983             (lambda () ; function to call
984               (setf (solution (om::object editor)) (new-search-next (result (om::object editor)
      ) (om::object editor)))
985               ;TO TEST THE GOLOMB-RULER PROGRAM
986               ;(setf (solution (om::object editor)) (search-next-golomb-ruler (result (om::
      object editor))))
987               (om::openeditorframe ; open a voice window displaying the solution
988                 (om::omNG-make-new-instance (solution (om::object editor)) "current solution")
989               )
990             )
991           )
992         )
993       )
994
995       (om::om-make-dialog-item
996           'om::om-button
997           (om::om-make-point 265 50) ; position (horizontal, vertical)
998           (om::om-make-point 130 20) ; size (horizontal, vertical)
999           "Stop"
1000          :di-action #'(lambda (b)
1001            (setf (stop-search (om::object editor)) t)
1002          )
1003      )
1004
1005      (om::om-make-dialog-item
1006        'om::om-static-text
1007        (om::om-make-point 15 100)
1008        (om::om-make-point 200 20)
1009        "Tempo (BPM)"
1010        :font om::*om-default-font1b*
1011      )
1012
```

```
    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 170 100)
      (om::om-make-point 200 20)
      "Tempo"
      :range (loop :for n :from 30 :upto 200 collect n)
      :di-action #'(lambda (m)
        (setf (tempo (om::object editor)) (nth (om::om-get-selected-item-index m) (om::
om-get-item-list m)))
      )
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 15 150)
      (om::om-make-point 200 20)
      "Branching"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 170 150)
      (om::om-make-point 200 20)
      "Branching"
      :range '("Top down" "Full" "Top down random")
      :di-action #'(lambda (m)
        (setf (branching (om::object editor)) (nth (om::om-get-selected-item-index m) (om::
om-get-item-list m)))
      )
    )


    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 15 200)
      (om::om-make-point 200 20)
      "Difference Percentage"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::slider
      (om::om-make-point 170 200)
      (om::om-make-point 200 20)
      "Difference Percentage"
      :range '(0 100)
      :increment 1
      :di-action #'(lambda (s)
        (setf (percent-diff (om::object editor)) (om::om-slider-value s))
      )
    )
  )
)
```

D.2 melodizer-csp.lisp

```
(in-package :mldz)

;;;;;;;;;;;;;;;;;
; NEW-MELODIZER ;
;;;;;;;;;;;;;;;;;

; <block-csp> list of the child block objects
; <percent-diff> percentage of difference wanted for the solutions
; This function creates the CSP by creating the space and the variables, posting the
    constraints and the branching, specifying
; the search options and creating the search engine.
(defmethod new-melodizer (block-csp percent-diff branching)
    (let ((sp (gil::new-space)); create the space;
        push pull playing pushMap pullMap dfs tstop sopts scaleset pitch temp push-card
    q-push
        pos

        (max-pitch 127)
        (bars (bar-length block-csp))
        (quant 192)
        (min-length 1) ;minimum length of a note with associated constraint
        (chord-rhythm 2) ;a chord is played every [chord-rhythm] quant
        (chord-min-length 2)) ; minimum length of a chord with associated constraint

        (print block-csp)

        (setq push-list (list))
        (setq pull-list (list))
        (setq playing-list (list))
        (setq debug (list))
        (setq debug2 (list))

```

```
31          ;Setting constraint for this block and child blocks
32          (setq temp (get-sub-block-values sp block-csp))
33          (setq push (nth 0 temp))
34          (setq pull (nth 1 temp))
35          (setq playing (nth 2 temp))
36          (setq notes (nth 3 temp))
37          (setq added-notes (nth 4 temp))
38          (setq push-card (nth 5 temp))
39          (setq q-push (nth 6 temp))
40
41          (gil::g-specify-sol-variables sp q-push)
42          (gil::g-specify-percent-diff sp percent-diff)
43
44          (cond
45              ((string-equal branching "Top down")
46                  (loop :for i :from (- (length push-list) 1) :downto 0 :do
47                      (gil::g-branch sp (append (nth i push-list) (nth i pull-list)) gil::
    SET_VAR_SIZE_MIN gil::SET_VAL_RND_INC)
48                  )
49              )
50              ((string-equal branching "Full")
51                  (progn
52                      (setq branch-push (list))
53                      (setq branch-pull (list))
54                      (loop :for l :in push-list :do
55                          (setq branch-push (append branch-push l))
56                      )
57                      (loop :for l :in pull-list :do
58                          (setq branch-pull (append branch-pull l))
59                      )
60                      (gil::g-branch sp (append branch-push branch-pull) gil::SET_VAR_SIZE_MIN
    gil::SET_VAL_RND_INC)
61                  )
62              )
63              ((string-equal branching "Top down random")
64                  (loop :for i :from (- (length push-list) 1) :downto 0 :do
65                      (gil::g-branch sp (append (nth i push-list) (nth i pull-list)) gil::
    SET_VAR_RND gil::SET_VAL_RND_INC)
66                  )
67              )
68          )
69
70          ;time stop
71          (setq tstop (gil::t-stop)); create the time stop object
72          (gil::time-stop-init tstop 500); initialize it (time is expressed in ms)
73
74          ;search options
75          (setq sopts (gil::search-opts)); create the search options object
76          (gil::init-search-opts sopts); initialize it
77          (gil::set-n-threads sopts 1); set the number of threads to be used during the search
    (default is 1, 0 means as many as available)
78          (gil::set-time-stop sopts tstop); set the timestop object to stop the search if it
    takes too long
79
80          ; search engine
81          (setq se (gil::search-engine sp (gil::opts sopts) gil::BAB))
82
83          (print "new-melodizer CSP constructed")
84          ; return
85          (list se push pull tstop sopts bars quant push-list pull-list playing-list debug
    debug2)
86      )
87  )
88
89  ;recursive function to set the constraint on all the blocks in the tree structure
90  (defun get-sub-block-values (sp block-csp)
91      ; for block child of block-csp
92      ; (pull supersets de get-sub-block-values(block) )
93      ; constraints
94      ; return pull push playing
95      (let (pull push notes playing pushMap pushMap-card pullMap block-list positions max-notes
        sub-push sub-pull
96              push-card added-push added-notes added-push-card q-push q-push-card
97          (bars (bar-length block-csp))
98          (quant 192)
99          (prevNotes (list))
100         (major-natural (list 2 2 1 2 2 2 1))
101         (max-pitch 127))
102
103         (setq max-notes (* 127 (+ (* bars quant) 1)))
104
105         ;initialize the variables
106
107         (setq push (gil::add-set-var-array sp (+ (* bars quant) 1) 0 max-pitch 0 max-pitch))
108         (setq pull (gil::add-set-var-array sp (+ (* bars quant) 1) 0 max-pitch 0 max-pitch))
109         (setq playing (gil::add-set-var-array sp (+ (* bars quant) 1) 0 max-pitch 0 max-pitch
    ))
110
111         (setq push-list (nconc push-list (list push)))
112         (setq pull-list (nconc pull-list (list pull)))
113         (setq playing-list (nconc playing-list (list playing)))
114
115         ;channeling array with time as index to array with pitch as index
```

```
116         (setq pushMap (gil::add-set-var-array sp (+ max-pitch 1) 0 (+ (* bars quant) 1) 0 (+ (* bars quant) 1)))
117         (setq pullMap (gil::add-set-var-array sp (+ max-pitch 1) 0 (+ (* bars quant) 1) 0 (+ (* bars quant) 1)))
118         (gil::g-channel sp push pushMap)
119         (gil::g-channel sp pull pullMap)
120 
121         (setq pushMap-card (gil::add-int-var-array sp 128 0 (+ (* bars quant) 1)))
122         (loop :for i :from 0 :below (length pushMap) :by 1 :do
123             (gil::g-card-var sp (nth i pushMap) (nth i pushMap-card))
124         )
125 
126         (setq block-list (block-list block-csp))
127         (if (not (typep block-list 'list))
128             (setq block-list (list block-list))
129         )
130         (setq positions (position-list block-csp))
131 
132         ;initial constraint on pull, push, playing and durations
133         (gil::g-empty sp (first pull)) ; pull[0] == empty
134         (gil::g-empty sp (car (last push)))  ; push[bars*quant] == empty
135         (gil::g-empty sp (car (last playing)))  ; playing[bars*quant] == empty
136         (gil::g-rel sp (first push) gil::SRT_EQ (first playing)) ; push[0] == playing [0]
137 
138         ;compute notes
139         (setq notes (gil::add-int-var sp 0 max-notes))
140         (setq push-card (gil::add-int-var-array sp (+ (* bars quant) 1) 0 127))
141 
142         (loop :for i :from 0 :below (+ (* bars quant) 1) :by 1 :do
143             (gil::g-card-var sp (nth i push) (nth i push-card))
144         )
145         (gil::g-sum sp notes push-card)
146 
147 
148         ;compute added notes
149         (setq added-push (gil::add-set-var-array sp (+ (* bars quant) 1) 0 max-pitch 0 max-pitch))
150         (setq sub-push (gil::add-set-var-array sp (+ (* bars quant) 1) 0 max-pitch 0 max-pitch))
151         (setq sub-pull (gil::add-set-var-array sp (+ (* bars quant) 1) 0 max-pitch 0 max-pitch))
152         (setq added-notes (gil::add-int-var sp 0 127))
153         (setq added-push-card (gil::add-int-var-array sp (+ (* bars quant) 1) 0 127))
154         (loop :for i :from 0 :below (+ (* bars quant) 1) :by 1 :do
155             (gil::g-card-var sp (nth i added-push) (nth i added-push-card))
156         )
157         (gil::g-sum sp added-notes added-push-card)
158 
159         ;compute q-push
160         (setq q-push (gil::add-set-var-array sp (* bars (get-quant (quantification block-csp))) 0 max-pitch 0 max-pitch))
161         (loop :for i :from 0 :below (length q-push) :by 1 :do
162             (gil::g-rel sp (nth i q-push) gil::SRT_EQ (nth (* i (get-length (quantification block-csp))) push))
163         )
164         (setq q-push-card (gil::add-int-var-array sp (length q-push) 0 127))
165         (loop :for i :from 0 :below (length q-push) :by 1 :do
166             (gil::g-card-var sp (nth i q-push) (nth i q-push-card))
167         )
168 
169 
170         ;connect push, pull and playing
171         (loop :for j :from 1 :below (+ (* bars quant) 1) :do ;for each interval
172             (let (temp z c)
173                 (setq temp (gil::add-set-var sp 0 max-pitch 0 max-pitch)); temporary variables
174                 (gil::g-op sp (nth (- j 1) playing) gil::SOT_MINUS (nth j pull) temp); temp[0] = playing[j-1] - pull[j]
175                 (gil::g-op sp temp gil::SOT_UNION (nth j push) (nth j playing)); playing[j] == playing[j-1] - pull[j] + push[j] Playing note
176                 (gil::g-rel sp (nth j pull) gil::SRT_SUB (nth (- j 1) playing)) ; pull[j] <= playing[j-1] cannot pull a note not playing
177                 (gil::g-set-op sp (nth (- j 1) playing) gil::SOT_MINUS (nth j pull) gil::SRT_DISJ (nth j push)); push[j] || playing[j-1] - pull[j] Cannot push a note still playing
178 
179             )
180         )
181 
182         (if (melody-source block-csp)
183             (let (melody-temp melody-push melody-pull melody-playing)
184                 (setq melody-temp (create-push-pull (melody-source block-csp) quant))
185                 (setq melody-push (gil::add-set-var-array sp (length (first melody-temp)) 0 max-pitch 0 max-pitch))
186                 (setq melody-pull (gil::add-set-var-array sp (length (second melody-temp)) 0 max-pitch 0 max-pitch))
187                 (setq melody-playing (gil::add-set-var-array sp (length (third melody-temp)) 0 max-pitch 0 max-pitch))
188                 (loop :for i :from 0 :below (length (first melody-temp)) :by 1 :do
189                     (if (or (typep (nth i (first melody-temp)) 'list) (/= (nth i (first melody-temp)) -1))
190                         (gil::g-rel sp (nth i melody-push) gil::SRT_EQ (nth i (first melody-temp)))
```

```
                        (gil::g-empty sp (nth i push))
                    )
                )
                (loop :for i :from 0 :below (length (second melody-temp)) :by 1 :do
                    (if (or (typep (nth i (second melody-temp)) 'list) (/= (nth i (second
melody-temp)) -1))
                        (gil::g-rel sp (nth i melody-pull) gil::SRT_EQ (nth i (second
melody-temp)))
                        (gil::g-empty sp (nth i pull))
                    )
                )
                (loop :for i :from 0 :below (length (third melody-temp)) :by 1 :do
                    (if (or (typep (nth i (third melody-temp)) 'list) (/= (nth i (third
melody-temp)) -1))
                        (gil::g-rel sp (nth i melody-playing) gil::SRT_EQ (nth i (third
melody-temp)))
                        (gil::g-empty sp (nth i melody-playing))
                    )
                )
                (loop :for j :from 0 :below (length melody-push) :by 1 :do
                        (gil::g-rel sp (nth j melody-push) gil::SRT_SUB (nth j push))
                        (gil::g-rel sp (nth j melody-pull) gil::SRT_SUB (nth j pull))
                )
            )
        )


        (if (not (endp block-list))
            ; make the push and pull array supersets of the corresponding array of the child
blocks
            (let ((sub-push-list (list)) (sub-pull-list (list)))

                (loop :for i :from 0 :below (+ (* bars quant) 1) :by 1 :do
                    (setq temp1 (gil::add-set-var-array sp (length block-list) 0 max-pitch 0
max-pitch))
                    (setq temp2 (gil::add-set-var-array sp (length block-list) 0 max-pitch 0
max-pitch))
                    (gil::g-setunion sp (nth i sub-push) temp1)
                    (setq sub-push-list (nconc sub-push-list (list temp1)))
                    (gil::g-setunion sp (nth i sub-pull) temp2)
                    (setq sub-pull-list (nconc sub-pull-list (list temp2)))
                    (gil::g-op sp (nth i push) gil::SOT_MINUS (nth i sub-push) (nth i
added-push))
                )
                (loop :for i :from 0 :below (length block-list) :by 1 :do
                      (let (tempPush tempPull tempPlaying tempList (start (* (nth i positions
) quant)))
                            (setq tempList (get-sub-block-values sp (nth i block-list)))
                            (setq tempPush (first tempList))
                            (setq tempPull (second tempList))
                            (setq tempPlaying (third tempList))
                            (setq prevNotes (nth 7 tempList))

                            (loop :for j :from start :below (+ start (length tempPlaying)) :by
 1 :do
                                (gil::g-rel sp (nth (- j start) tempPush) gil::SRT_SUB (nth j
 push))
                                (gil::g-rel sp (nth (- j start) tempPull) gil::SRT_SUB (nth j
 pull))
                                (gil::g-rel sp (nth (- j start) tempPlaying) gil::SRT_SUB (
nth j playing))
                            )

                            (loop :for j :from 0 :below (length push) :by 1 :do
                                (if (and (>= j start) (< j (+ start (length tempPlaying))))
                                    (gil::g-rel sp (nth (- j start) tempPush) gil::SRT_EQ (
nth i (nth j sub-push-list)))
                                    (gil::g-empty sp (nth i (nth j sub-push-list)))
                                )
                            )

                            (loop :for j :from 0 :below (length pull) :by 1 :do
                                (if (and (>= j start) (< j (+ start (length tempPlaying))))
                                    (gil::g-rel sp (nth (- j start) tempPull) gil::SRT_EQ (
nth i (nth j sub-pull-list)))
                                    (gil::g-empty sp (nth i (nth j sub-pull-list)))
                                )
                            )
                      )
                )
            )
            ; if no block-list
            (progn
                (gil::g-rel sp added-notes gil::SRT_EQ notes)
                (loop :for p :in sub-push :do (gil::g-empty sp p))
                (loop :for p :in sub-pull :do (gil::g-empty sp p))
            )


        )

        ;constraints
        (post-optional-constraints sp block-csp push pull playing pushMap pushMap-card notes
```

```
    added-notes push-card sub-push sub-pull q-push q-push-card)
269         (pitch-range sp push (min-pitch block-csp) (max-pitch block-csp))
270         (list push pull playing notes added-notes push-card q-push)
271     )
272 )
273
274 ;posts the optional constraints specified in the list
275 ; TODO CHANGE LATER SO THE FUNCTION CAN BE CALLED FROM THE STRING IN THE LIST AND NOT WITH A SERIES OF IF STATEMENTS
276 (defun post-optional-constraints (sp block push pull playing pushMap pushMap-card notes
    added-notes push-card sub-push sub-pull q-push q-push-card)
277
278     ; Block constraints
279     (if (voices block)
280         (gil::g-card sp playing 0 (voices block))
281     )
282
283     (if (min-pushed-notes block)
284         (loop :for i :from 0 :below (length push-card) :by 1 :do
285             (setq b1 (gil::add-bool-var sp 0 1))
286             (gil::g-rel-reify sp (nth i push-card) gil::IRT_EQ 0 b1)
287             (setq b2 (gil::add-bool-var sp 0 1))
288             (gil::g-rel-reify sp (nth i push-card) gil::IRT_GQ (min-pushed-notes block) b2)
289             (gil::g-rel sp b1 gil::BOT_OR b2)
290         )
291     )
292
293     (if (max-pushed-notes block)
294         (gil::g-card sp push 0 (max-pushed-notes block))
295     )
296
297     (if (min-notes block)
298         (progn
299             (gil::g-rel sp notes gil::IRT_GQ (min-notes block))
300         )
301     )
302
303     (if (max-notes block)
304         (gil::g-rel sp notes gil::IRT_LQ (max-notes block))
305     )
306
307     (if (min-added-notes block)
308         (gil::g-rel sp added-notes gil::IRT_GQ (min-added-notes block))
309     )
310
311     (if (max-added-notes block)
312         (if (= 0 (max-added-notes block))
313             (progn
314                 (loop :for i :from 0 :below (length push) :by 1 :do
315                     (gil::g-rel sp (nth i push) gil::SRT_EQ (nth i sub-push))
316                 )
317             )
318             (gil::g-rel sp added-notes gil::IRT_LQ (max-added-notes block))
319         )
320
321     )
322
323     ; Time constraints
324     (if (min-note-length-flag block)
325         (note-min-length sp push pull (min-note-length block))
326     )
327
328     (if (max-note-length-flag block)
329         (note-max-length sp push pull (max-note-length block))
330     )
331
332     (if (quantification block)
333         (set-quantification sp push pull (quantification block))
334     )
335
336     (if (rhythm-repetition block)
337         (set-rhythm-repetition sp push-card (get-length (rhythm-repetition block)))
338     )
339
340     (if (pause-quantity-flag block)
341         (set-pause-quantity sp q-push-card (pause-quantity block) (bar-length block) (
    get-quant (quantification block)))
342     )
343
344     (if (pause-repartition-flag block)
345         (set-pause-repartition sp q-push-card (pause-repartition block))
346     )
347
348     ; Pitch constraints
349     ; following a scale
350     (if (key-selection block)
351         (if (mode-selection block)
352             (let (scaleset
353                   (bool (gil::add-bool-var sp 0 1)) ; c[U+FFFD]- le booleen pour la reify
354                   (scale (get-scale (mode-selection block)))  ;if - mode selectionn[U+FFFD]
355                   (offset (- (name-to-note-value (key-selection block)) 60)))
356                 (setq scaleset (build-scaleset scale offset))
357                 (gil::g-rel sp bool gil::SRT_EQ 1) ;forcer le reify a true dans ce cas
```

```
                (scale-follow-reify sp push scaleset bool))
            (let (scaleset
                  (bool (gil::add-bool-var sp 0 1)) ; α[U+FFFD]r le booleen pour la reify
                  (scale (get-scale "ionian (major)"))  ;else - pas de mode selectionn[U+FFFD] => major natural
                  (offset (- (name-to-note-value (key-selection block)) 60)))
                 (gil::g-rel sp bool gil::SRT_EQ 1) ;forcer le reify a true dans ce cas
                 (setq scaleset (build-scaleset scale offset))
                 (scale-follow-reify sp push scaleset bool))
        )
        (if (mode-selection block)
            (let ((bool-array (gil::add-bool-var-array sp 12 0 1))) ; α[U+FFFD]r le booleen pour la reify
                (loop :for key :from 0 :below 12 :by 1 :do
                    (setq scale (get-scale (mode-selection block)))
                    (setq scaleset (build-scaleset scale key))
                    (scale-follow-reify sp push scaleset (nth key bool-array))
                )
                (gil::g-rel sp gil::BOT_OR bool-array 1)
            )
        )
    )

    (if (chord-key block)
        (if (chord-quality block)
            (if (all-chord-notes block)
                (let ((bool (gil::add-bool-var sp 0 1)) ; α[U+FFFD]r le booleen pour la reify
                      (bool2 (gil::add-bool-var sp 0 1))
                      (chord (get-chord (chord-quality block)))  ;if - mode selectionn[U+FFFD]
                      (offset (- (name-to-note-value (chord-key block)) 60))
                      (all-notes (gil::add-set-var sp 0 127 0 127))
                      chordset notesets bool-array)
                    (setq chordset (build-scaleset chord offset))
                    (scale-follow-reify sp push chordset bool)
                    (setq notesets (build-notesets chord offset))
                    (setq bool-array (gil::add-bool-var-array sp (length notesets) 0 1))
                    (loop :for i :from 0 :below (length notesets) :do
                         (let ((push-bool-array (gil::add-bool-var-array sp (length push) 0 1)))
                             (loop :for j :from 0 :below (length push) :do
                                 (gil::g-rel-reify sp (nth j push) gil::SRT_DISJ (nth i notesets) (nth j push-bool-array))
                             )
                             (gil::g-rel sp gil::BOT_AND push-bool-array (nth i bool-array))
                         )
                    )
                    (setq debug (nconc debug (list bool-array)))
                    (setq debug2 (nconc debug2 (list bool2)))

                    (gil::g-rel sp gil::BOT_OR bool-array bool2)
                    (gil::g-rel sp bool gil::SRT_EQ 1)
                    )

                (let ((bool (gil::add-bool-var sp 0 1)) ; α[U+FFFD]r le booleen pour la reify
                      (chord (get-chord (chord-quality block)))  ;if - mode selectionn[U+FFFD]
                      (offset (- (name-to-note-value (chord-key block)) 60))
                      (all-notes (gil::add-set-var sp 0 127 0 127))
                      chordset)
                    (gil::g-setunion sp all-notes push)
                    (setq chordset (build-scaleset chord offset))
                    (gil::g-rel sp bool gil::SRT_EQ 1) ;forcer le reify a true dans ce cas
                    (scale-follow-reify sp push chordset bool))
            )
        )
        (if (chord-quality block)
            (if (all-chord-notes block)
                (let (chord chordset notesets
                      (bool-array (gil::add-bool-var-array sp 12 0 1)); α[U+FFFD]r le booleen pour la reify
                      (all-notes (gil::add-set-var sp 0 127 01 127)))
                    (gil::g-setunion sp all-notes push)
                    (loop :for key :from 0 :below 12 :by 1 :do
                        (let ((bool1 (gil::add-bool-var sp 0 1))
                              (bool2 (gil::add-bool-var sp 0 1))
                              (bool-array-note (gil::add-bool-var-array sp (length notesets) 0 1))
                              chordset notesets)
                            (setq chord (get-chord (chord-quality block)))
                            (setq chordset (build-scaleset chord key))
                            (setq notesets (build-notesets chord key))

                            (loop :for i :from 0 :below (length notesets) :do
                                (gil::g-rel-reify sp all-notes gil::SRT_DISJ (nth i notesets) (nth i bool-array-note))
                            )
                            (gil::g-rel sp gil::BOT_AND bool-array-note bool1)
                            (scale-follow-reify sp push chordset bool2)
                            (gil::g-op sp (nth key bool-array) gil::BOT_AND bool 0))
                    )
                    (gil::g-rel sp gil::BOT_OR bool-array 1)
                )
                (let (chord chordset
                      (bool-array (gil::add-bool-var-array sp 12 0 1)))
```

```
                    (loop :for key :from 0 :below 12 :by 1 :do
                        (setq chord (get-chord (chord-quality block)))
                        (setq chordset (build-scaleset chord key))
                        (scale-follow-reify sp push chordset (nth key bool-array))
                    )
                    (gil::g-rel sp gil::BOT_OR bool-array 1)
                )
            )

        )
    )


    (if (pitch-direction block)
        (let ((allPlayed (gil::add-set-var sp 0 (+ (length push) 1) 0 (+ (length push) 1)))
              (isPlayed (gil::add-bool-var-array sp (+ (length push) 1) 0 1)))
            (gil::g-arr-op sp gil::SOT_UNION pushMap allPlayed)
            (gil::g-channel sp isPlayed allPlayed)

            (cond
                ((string= (pitch-direction block) "Increasing")           (increasing-pitch sp push isPlayed))
                ((string= (pitch-direction block) "Strictly increasing")  (strictly-increasing-pitch sp push isPlayed))
                ((string= (pitch-direction block) "Decreasing")           (decreasing-pitch sp push isPlayed))
                ((string= (pitch-direction block) "Strictly decreasing")  (strictly-decreasing-pitch sp push isPlayed))
            )
        )
    )

    (if (/= (golomb-ruler-size block) 0)
         (golomb-rule sp (golomb-ruler-size block) push (/ 192 (get-quant (quantification block))))
    )



    (if (note-repetition-flag block)
        (cond
          ((string-equal (note-repetition-type block) "Random")
            (random-repeat-note sp push (note-repetition block) (get-length (quantification block))))
          ((string-equal (note-repetition-type block) "Soft")
            (soft-repeat-note sp (note-repetition block) pushMap-card))
          ((string-equal (note-repetition-type block) "Hard")
            (hard-repeat-note sp (note-repetition block) pushMap-card (length q-push)))
        )
    )
)

;;;;;;;;;;;;;;;
; SEARCH-NEXT ;
;;;;;;;;;;;;;;;

; <l> is a list containing the search engine for the problem and the variables
; <melodizer-object> is a melodizer object
; this function finds the next solution of the CSP using the search engine given as an argument
(defmethod new-search-next (l melodizer-object)
    (let ((se (first l))
          (push (second l))
          (pull (third l))
          (tstop (fourth l))
          (sopts (fifth l))
          (bars (sixth l))
          (quant (seventh l))
          (push-list (eighth l))
          (pull-list (ninth l))
          (playing-list (nth 9 l))
          (debug (nth 10 l))
          (debug2 (nth 11 l))
          (check t); for the while loop
          sol score)

          (print "in search")

        (om::while check :do
            (gil::time-stop-reset tstop);reset the tstop timer before launching the search
            (setq sol (gil::search-next se)); search the next solution
            (if (null sol)
                (stopped-or-ended (gil::stopped se) (stop-search melodizer-object) tstop); check if there are solutions left and if the user wishes to continue searching
                (setf check nil); we have found a solution so break the loop
            )
        )

        ;SOME CODE PIECES FOR DEBUGGING

        ;(print "PUSH")
        ;(loop :for p :in push-list :do
        ;    (let (l (list))
```

```
529         ;         (print (gil::vid p))
530         ;         (setq l (nconc l (mapcar (lambda (n) (to-midicent (gil::g-values sol n))) p)
    ))
531         ;         (print l)
532         ;     )
533         ;)
534
535         ;(print "PULL")
536         ;(loop :for p :in pull-list :do
537         ;     (let (l (list))
538         ;         (setq l (nconc l (mapcar (lambda (n) (to-midicent (gil::g-values sol n))) p)
    ))
539         ;         (print l)
540         ;     )
541         ;)
542
543         ;(print "PLAYING")
544         ;(loop :for p :in playing-list :do
545         ;     (let (l (list))
546         ;         (setq l (nconc l (mapcar (lambda (n) (to-midicent (gil::g-values sol n))) p)
    ))
547         ;         (print l)
548         ;     )
549         ;)
550
551         ;(print "DEBUG")
552         ;(print debug)
553         ;(loop :for p :in debug :do
554         ;  (let (l (list))
555         ;      (setq l (nconc l (mapcar (lambda (n) (gil::g-values sol n)) p)))
556         ;      (print l)
557         ;  )
558         ;)
559
560         ;(print "DEBUG")
561         ;(loop :for p :in debug2 :do
562         ;  (print (gil::g-values sol p))
563         ;)
564
565
566          ;α[U+FFFD] score qui retourne la liste de pitch et la rhythm tree
567         (setq score-chord-seq (build-chord-seq sol push pull bars quant (tempo
    melodizer-object)))
568
569         (make-instance 'chord-seq
570             :LMidic (first score-chord-seq)
571             :LOnset (second score-chord-seq)
572             :Ldur (third score-chord-seq)
573         )
574     )
575 )
576
577 ; determines if the search has been stopped by the solver because there are no more solutions
      or if the user has stopped the search
578 (defun stopped-or-ended (stopped-se stop-user tstop)
579     (if (= stopped-se 0); if the search has not been stopped by the TimeStop object, there is
      no more solutions
580         (error "There are no more solutions.")
581     )
582     ;otherwise, check if the user wants to keep searching or not
583     (if stop-user
584         (error "The search has been stopped. Press next to continue the search.")
585     )
586 )
```

D.3 melodizer-csts.lisp

```
1  (in-package :mldz)
2
3  ;;;;;;;;;;;;;;;;;;;;;
4  ; FOLLOWING A SCALE ;
5  ;;;;;;;;;;;;;;;;;;;;;
6
7  (defun scale-follow (sp push scaleset)
8      (loop :for j :from 0 :below (length push) :do
9              (gil::g-rel sp (nth j push) gil::SRT_SUB scaleset)
10     )
11 )
12
13 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
14 ; FOLLOWING A SCALE WITH REIFICATION ;
15 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
16
17 (defun scale-follow-reify (sp push scaleset reify)
18     (setq r (gil::add-bool-var-array sp (length push) 0 1))
19     (loop :for j :from 0 :below (length push) :do
20         (gil::g-rel-reify sp (nth j push) gil::SRT_SUB scaleset (nth j r))
21     )
22     (gil::g-rel sp gil::BOT_AND r reify)
```

```
23
24 )
25
26 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
27 ; FOLLOWING A CHORD PROGRESSION ;
28 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
29
30 (defun chordprog-follow (sp push chordset progsize)
31     (loop :for j :from 0 :below (length chordset) :by 1 :do
32         (loop :for k :from 0 :below (/ (length push) progsize) :by 1 :do
33             (gil::g-rel sp (nth (+ k (/ (* (length push) j) progsize)) push) gil::SRT_SUB (
    nth j chordset))
34         )
35     )
36 )
37
38 ;;;;;;;;;;;;;;;;;;;;;;;;
39 ; LIMITING PITCH RANGE ;
40 ;;;;;;;;;;;;;;;;;;;;;;;;
41
42 (defun pitch-range (sp push min-pitch max-pitch)
43     (loop :for j :below (length push) :by 1 :do
44         (gil::g-dom-ints sp (nth j push) gil::SRT_SUB min-pitch max-pitch)
45     )
46 )
47
48 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
49 ; LIMITING MINIMUM NOTE LENGTH ;
50 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
51
52 (defun note-min-length (sp push pull min-length)
53     (setq l (floor (* (- (length push) 1) min-length) 192))
54     (loop :for j :from 0 :below (length push) :by 1 :do
55         (loop :for k :from 1 :below l  :while (< (+ j k) (length pull)) :do
56             (gil::g-rel sp (nth (+ j k) pull) gil::SRT_DISJ (nth j push))
57         )
58     )
59 )
60
61 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
62 ; LIMITING MAXIMUM NOTE LENGTH ;
63 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
64
65 (defun note-max-length (sp push pull max-length)
66     (setq l (floor (* (- (length push) 1) max-length) 192))
67     (loop :for j :from 0 :below (+ (- (length push) l) 1) :by 1 :do
68         (let ((l-pull (gil::add-set-var-array sp l 0 127 0 127))
69               (l-pull-union (gil::add-set-var sp 0 127 0 127)))
70             (loop :for k :from 0 :below l :by 1 :do
71                 (gil::g-rel sp (nth k l-pull) gil::SRT_EQ (nth (+ 1 (+ j k)) pull))
72             )
73             (gil::g-setunion sp l-pull-union l-pull)
74             (gil::g-rel sp (nth j push) gil::SRT_SUB l-pull-union)
75         )
76     )
77 )
78
79 ;;;;;;;;;;;;;;;;;;;;;;;;
80 ; DEFINING CHORD RHYTHM ;
81 ;;;;;;;;;;;;;;;;;;;;;;;;
82
83 (defun chords-rhythm (sp push chord-rhythm chord-size-min chord-size-max)
84     (loop :for j :from 0 :below (length push) :by 1 :do
85         (if (= (mod j chord-rhythm) 0)
86             (gil::g-card sp (nth j push) chord-size-min chord-size-max)
87             (gil::g-card sp (nth j push) 0 1)
88         )
89     )
90 )
91
92 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
93 ; LIMITING MINIMUM CHORD LENGTH ;
94 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
95
96 (defun chords-length (sp push pull chord-rhythm chord-min-length)
97     (loop :for j :from 0 :below (length push) :by 1 :do
98         (if (= (mod j chord-rhythm) 0)
99             (loop :for k :from 1 :below chord-min-length :while (< (+ j k) (length pull)) :
    do
100                 (gil::g-rel sp (nth (+ j k) pull) gil::SRT_DISJ (nth j push))
101             )
102         )
103     )
104 )
105
106 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
107 ; LIMIT NUMBER OF ADDED NOTE ;
108 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
109
110 (defun num-added-note (sp playing min-card max-card)
111     (gil::g-card sp playing min-card max-card)
112 )
113
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; SETS QUANTIFICATION FOR PUSH ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun set-quantification (sp push pull quantification)
    (setq q (/ 192 (get-quant quantification)))
    (loop :for j :from 0 :below (length push) :by 1 :do
         (if (/= (mod j q) 0)
            (gil::g-empty sp (nth j push))
         )
    )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; SETS REPETITION FOR PUSH CARDINALITIES ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun set-rhythm-repetition (sp push-card len)
    (loop :for i :from 0 :below len :while (< i (length push-card)) :do
        (loop :for j :from 1 :below (length push-card) :while (< (+ i (* j len)) (- (length
    push-card) 1)) :do
            (gil::g-rel sp (nth i push-card) gil::IRT_EQ (nth (+ i (* j len)) push-card))
        )
    )
)

;;;;;;;;;;;;;;;;;;;;;;;;
; SETS PAUSE QUANTITY ;
;;;;;;;;;;;;;;;;;;;;;;;;

(defun set-pause-quantity (sp q-push-card quantity bars quant)
    (setq c (floor (* (length q-push-card) quantity) 192))
    (gil::g-count sp q-push-card 0 gil::IRT_GQ c)
)

;;;;;;;;;;;;;;;;;;;;;;;;;;
; SETS PAUSE REPARTITION ;
;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun set-pause-repartition (sp q-push-card repartition)
    (setq l (ceiling (* (length q-push-card) (- 192 repartition)) 192))
    (gil::g-sequence sp q-push-card (list 0) l 1 1)
)

;;;;;;;;;;;;;;;;;;;
; PITCH DIRECTION ;
;;;;;;;;;;;;;;;;;;;

(defun increasing-pitch (sp playing isPlayed)
    (loop :for i :from 0 :below (- (length playing) 1) :by 1 :do
          (let ((tempVar1 (gil::add-int-var sp 0 127)))
             (gil::g-setmin-reify sp (nth i playing) tempVar1 (nth i isPlayed) gil::RM_IMP)
             (loop :for j :from (+ i 1) :below (length playing) :by 1 :do
                   (let ((tempBool (gil::add-bool-var sp 0 1)) tempVar2)
                        (setq tempVar2 (gil::add-int-var sp 0 127))
                        (gil::g-op sp (nth i isPlayed) gil::BOT_AND (nth j isPlayed) tempBool
    )
                        (gil::g-setmin-reify sp (nth j playing) tempVar2 tempBool gil::RM_IMP
    )
                        (gil::g-rel-reify sp tempVar1 gil::IRT_LQ tempVar2 tempBool gil::
    RM_IMP)
                   )
             )
          )
    )
)

(defun decreasing-pitch (sp playing isPlayed)
    (loop :for i :from 0 :below (- (length playing) 1) :by 1 :do
          (let ((tempVar1 (gil::add-int-var sp 0 127)))
             (gil::g-setmax-reify sp (nth i playing) tempVar1 (nth i isPlayed) gil::RM_IMP)
             (loop :for j :from (+ i 1) :below (length playing) :by 1 :do
                   (let ((tempBool (gil::add-bool-var sp 0 1)) tempVar2)
                        (setq tempVar2 (gil::add-int-var sp 0 127))
                        (gil::g-op sp (nth i isPlayed) gil::BOT_AND (nth j isPlayed) tempBool
    )
                        (gil::g-setmax-reify sp (nth j playing) tempVar2 tempBool gil::RM_IMP
    )
                        (gil::g-rel-reify sp tempVar1 gil::IRT_GQ tempVar2 tempBool gil::
    RM_IMP)
                   )
             )
          )
    )
)

(defun strictly-increasing-pitch (sp playing isPlayed)
    (loop :for i :from 0 :below (- (length playing) 1) :by 1 :do
          (let ((tempVar1 (gil::add-int-var sp 0 127)))
             (gil::g-setmax-reify sp (nth i playing) tempVar1 (nth i isPlayed) gil::RM_IMP)
             (loop :for j :from (+ i 1) :below (length playing) :by 1 :do
                   (let ((tempBool (gil::add-bool-var sp 0 1)) tempVar2)
                        (setq tempVar2 (gil::add-int-var sp 0 127))
```

```
                        (gil::g-op sp (nth i isPlayed) gil::BOT_AND (nth j isPlayed) tempBool
    )
                        (gil::g-setmin-reify sp (nth j playing) tempVar2 tempBool gil::RM_IMP
    )
                        (gil::g-rel-reify sp tempVar1 gil::IRT_LE tempVar2 tempBool gil::
    RM_IMP)
                    )
                )
            )
    )
)

(defun strictly-decreasing-pitch (sp playing isPlayed)
    (loop :for i :from 0 :below (- (length playing) 1) :by 1 :do
          (let ((tempVar1 (gil::add-int-var sp 0 127)))
             (gil::g-setmin-reify sp (nth i playing) tempVar1 (nth i isPlayed) gil::RM_IMP)
             (loop :for j :from (+ i 1) :below (length playing) :by 1 :do
                   (let ((tempBool (gil::add-bool-var sp 0 1)) tempVar2)
                        (setq tempVar2 (gil::add-int-var sp 0 127))
                        (gil::g-op sp (nth i isPlayed) gil::BOT_AND (nth j isPlayed) tempBool
    )
                        (gil::g-setmax-reify sp (nth j playing) tempVar2 tempBool gil::RM_IMP
    )
                        (gil::g-rel-reify sp tempVar1 gil::IRT_GR tempVar2 tempBool gil::
    RM_IMP)
                    )
                )
          )
    )
)


;;;;;;;;;;;;;;;;
; GOLOMB RULER ;
;;;;;;;;;;;;;;;;

(defun golomb-rule (sp size push quant)

    (setf size-d (/ (- (* size size) size) 2))

    ; array of differences
    (setq d (gil::add-int-var-array sp size-d 0 127))

    (setf k 0)
    (loop :for i :from 0 :below (* (- size 1) quant) :by quant :do
       (loop :for j :from (+ i quant) :below (* size quant) :by quant :do
            (progn
                (gil::g-linear sp '(1 -1) (list (gil::g-setmax sp (nth j push)) (gil::g-setmax
     sp (nth i push))) gil::IRT_EQ (nth k d))
                (setf k (+ k 1))
            )
       )
    )

    (gil::g-distinct sp d)
)

;;;;;;;;;;;;;;;;;;
; NOTE REPETITION ;
;;;;;;;;;;;;;;;;;;

(defun random-repeat-note (sp push percent quant)
    (let ((index (list-shuffler (range (length push) :min 0 :step quant))))
        (loop :for i :from 0 :below (- (length index) 1) :by 1 :do
            (if (< (random 100) percent)
                (gil::g-rel sp (nth (nth i index) push) gil::SRT_EQ (nth (+ (nth i index) 1)
     push))
                (gil::g-rel sp (nth (nth i index) push) gil::SRT_DISJ (nth (+ (nth i index)
     1) push))
            )
        )
    )
)

(defun soft-repeat-note (sp percent pushMap-card)
    (let ((c (round (* percent (- (length pushMap-card) 1)) 100)))
        (gil::g-count sp pushMap-card 0 gil::IRT_GQ c)
    )
)

(defun hard-repeat-note (sp percent pushMap-card max-repetition)
    (let ((repetition (round (* percent max-repetition) 100)))
        (gil::g-count sp pushMap-card repetition gil::IRT_GQ 1)
    )
)
```

D.4 melodizer-utils.lisp

```
(in-package :mldz)
```

```
2
3  ; converts a list of MIDI values to MIDIcent
4  (defun to-midicent (l)
5      (if (null l)
6          nil
7          (cons (* 100 (first l)) (to-midicent (rest l)))
8      )
9  )
10
11 ; convert from MIDIcent to MIDI
12 (defun to-midi (l)
13     (if (null l)
14         nil
15         (cons (/ (first l) 100) (to-midi (rest l)))
16     )
17 )
18
19 ;converts the value of a note to its name
20 (defmethod note-value-to-name (note)
21     (cond
22         ((eq note 60) "C")
23         ((eq note 61) "C#")
24         ((eq note 62) "D")
25         ((eq note 63) "Eb")
26         ((eq note 64) "E")
27         ((eq note 65) "F")
28         ((eq note 66) "F#")
29         ((eq note 67) "G")
30         ((eq note 68) "Ab")
31         ((eq note 69) "A")
32         ((eq note 70) "Bb")
33         ((eq note 71) "B")
34     )
35 )
36
37 ;converts the name of a note to its value
38 (defmethod name-to-note-value (name)
39     (cond
40         ((string-equal name "C") 60)
41         ((string-equal name "C#") 61)
42         ((string-equal name "D") 62)
43         ((string-equal name "Eb") 63)
44         ((string-equal name "E") 64)
45         ((string-equal name "F") 65)
46         ((string-equal name "F#") 66)
47         ((string-equal name "G") 67)
48         ((string-equal name "Ab") 68)
49         ((string-equal name "A") 69)
50         ((string-equal name "Bb") 70)
51         ((string-equal name "B") 71)
52     )
53 )
54
55 ; finds the smallest element of a list
56 (defun min-list (L)
57     (cond
58         ((null (car L)) nil); the list is empty -> return nil
59         ((null (cdr L)) (car L)); the list has 1 element -> return it
60         (T
61             (let ((head (car L)); default behavior
62                   (tailMin (min-list (cdr L))))
63                 (if (< head tailMin) head tailMin)
64             )
65         )
66     )
67 )
68
69 ; finds the biggest element of a list
70 (defun max-list (L)
71     (cond
72         ((null (car L)) nil); the list is empty -> return nil
73         ((null (cdr L)) (car L)); the list has 1 element -> return it
74         (T
75             (let ((head (car L)); default behavior
76                   (tailMax (max-list (cdr L))))
77                 (if (> head tailMax) head tailMax)
78             )
79         )
80     )
81 )
82
83
84 ; finds the biggest element in a list of lists
85 (defun max-list-list (L)
86     (cond
87         ((null (car L)) nil); the list is empty -> return nil
88         ((null (cdr L)) (max-list (car L))); the list has 1 element -> return it
89         (T
90             (let ((head (max-list (car L))); default behavior
91                   (tailMax (max-list-list (cdr L))))
92                 (if (> head tailMax) head tailMax)
93             )
94         )
```

```
    )
)

; create a list from min to max by step
(defun range (max &key (min 0) (step 1))
   (loop for n from min below max by step
      collect n))

; function to update the list of solutions in a pop-up menu without having to close and
    re-open the window
; TODO find a more efficient way to do this
(defun update-pop-up (self my-panel data position size output)
  (om::om-add-subviews my-panel
    (om::om-make-dialog-item
      'om::om-pop-up-dialog-item
      position ;(om::om-make-point 5 130)
      size ;(om::om-make-point 320 20)
      "list of solutions"
      :range (loop for item in (make-data-sol data) collect (car item))
      :di-action #'(lambda (m)
                     (cond
                          ((string-equal output "output-solution")
                              (setf (output-solution (om::object self)) (nth (om::
    om-get-selected-item-index m) data)); set the output solution to the currently selected
    solution
                              (let ((indx (om::om-get-selected-item-index m)))
                                  (om::openeditorframe ; open the editor of the selected
    solution
                                      (om::omNG-make-new-instance
                                          (nth indx data)
                                          (format nil "melody ~D" (1+ indx)); name of the
    window
                                      )
                                  )
                              )
                          )
                          ((string-equal output "output-motif")
                              (setf (output-motif (om::object self)) (nth (om::
    om-get-selected-item-index m) data))
                              (let ((indx (om::om-get-selected-item-index m)))
                                  (om::openeditorframe
                                      (om::omNG-make-new-instance
                                          (output-motif (om::object self))
                                          (format nil "motif ~D" (1+ indx)); name of the window
                                      )
                                  )
                              )
                          )
                          ((string-equal output "output-phrase")
                              (setf (output-phrase (om::object self)) (nth (om::
    om-get-selected-item-index m) data))
                              (let ((indx (om::om-get-selected-item-index m)))
                                  (om::openeditorframe
                                      (om::omNG-make-new-instance
                                          (output-phrase (om::object self))
                                          (format nil "phrase ~D" (1+ indx)); name of the
    window
                                      )
                                  )
                              )
                          )
                          ((string-equal output "output-period")
                              (setf (output-period (om::object self)) (nth (om::
    om-get-selected-item-index m) data))
                              (let ((indx (om::om-get-selected-item-index m)))
                                  (om::openeditorframe
                                      (om::omNG-make-new-instance
                                          (output-period (om::object self))
                                          (format nil "period ~D" (1+ indx))
                                      )
                                  )
                              )
                          )
                     )
      )
    )
  )
)

;function to get the starting times (in ms) of the notes
; from karim haddad (OM)
(defmethod voice-onsets ((self voice))
  "on passe de voice a chord-seq juste pour avoir les onsets"
    (let ((obj (om::objfromobjs self (make-instance 'om::chord-seq))))
        (butlast (om::lonset obj))
    )
)

;function to get the duration (in ms) of the notes
(defmethod voice-durs ((self voice))
  "on passe de voice a chord-seq juste pour avoir les onsets"
    (let ((obj (om::objfromobjs self (make-instance 'om::chord-seq))))
        (om::ldur obj)
```

```
    )
)

; returns the list of intervals defining a given mode
(defun get-scale (mode)
    (cond
        ((string-equal mode "ionian (major)")
            (list 2 2 1 2 2 2 1)
        )
        ((string-equal mode "dorian")
            (list 2 1 2 2 2 1 2)
        )
        ((string-equal mode "phrygian")
            (list 1 2 2 2 1 2 2)
        )
        ((string-equal mode "lydian")
            (list 2 2 2 1 2 2 1)
        )
        ((string-equal mode "mixolydian")
            (list 2 2 1 2 2 1 2)
        )
        ((string-equal mode "aeolian (natural minor)")
            (list 2 1 2 2 1 2 2)
        )
        ((string-equal mode "locrian")
            (list 1 2 2 1 2 2 2)
        )
        ((string-equal mode "harmonic minor")
            (list 2 1 2 2 1 3 1)
        )
        ((string-equal mode "pentatonic")
            (list 2 2 3 2 3)
        )
        ((string-equal mode "chromatic")
            (list 1 1 1 1 1 1 1 1 1 1 1 1)
        )
    )
)

(defun get-chord (quality)
    (cond
        ((string-equal quality "Major")
            (list 4 3 5)
        )
        ((string-equal quality "Minor")
            (list 3 4 5)
        )
        ((string-equal quality "Augmented")
            (list 4 4 4)
        )
        ((string-equal quality "Diminished")
            (list 3 3 6)
        )
        ((string-equal quality "Major 7")
            (list 4 3 4 1)
        )
        ((string-equal quality "Minor 7")
            (list 3 4 3 2)
        )
        ((string-equal quality "Dominant 7" )
            (list 4 3 3 2)
        )
        ((string-equal quality "Minor 7 flat 5")
            (list 3 3 4 2)
        )
        ((string-equal quality "Diminished 7")
            (list 3 3 3 3)
        )
        ((string-equal quality "Minor-major 7")
            (list 3 4 4 1)
        )

        ; TODO gfU+FFFDr les accords 9 ou +
        ((string-equal quality "Major 9")
            (list 3 4 5)
        )
        ((string-equal quality "Minor 9")
            (list 4 3 5)
        )
        ((string-equal quality "9 Augmented 5")
            (list 3 4 5)
        )
        ((string-equal quality "9 flatted 5")
            (list 3 4 5)
        )
        ((string-equal quality "7 flat 9")
            (list 4 3 5)
        )
        ((string-equal quality "Augmented 9")
            (list 3 4 5)
        )
        ((string-equal quality "Minor 11")
            (list 3 4 5)
```

```
272         )
273         ((string-equal quality "Major 11")
274             (list 4 3 5)
275         )
276         ((string-equal quality "Dominant 11")
277             (list 3 4 5)
278         )
279         ((string-equal quality "Dominant # 11")
280             (list 4 3 5)
281         )
282         ((string-equal quality "Major # 11")
283             (list 3 4 5)
284         )
285     )
286 )
287
288 ; function to get all of a given note (e.g. C)
289 (defun get-all-notes (note)
290     (let ((acc '()) (backup note))
291         (om::while (<= note 127) :do
292             (setq acc (cons note acc)); add it to the list
293             (incf note 12)
294         )
295         (setf note (- backup 12))
296         (om::while (>= note 0) :do
297             (setq acc (cons note acc)); add it to the list
298             (decf note 12)
299         )
300         acc
301     )
302 )
303
304 ; function to get all notes playable on top of a given chord CHECK WHAT NOTES CAN BE PLAYED
      FOR OTHER CASES THAN M/m
305 (defun get-admissible-notes (chords mode inversion)
306     (let ((return-list '()))
307         (cond
308             ((string-equal mode "major"); on top of a major chord, you can play either of the
       notes from the chord though the preferred order is 1-5-3
309                 (setf return-list (reduce #'cons
310                     (get-all-notes (first chords))
311                     :initial-value return-list
312                     :from-end t
313                 ))
314                 (setf return-list (reduce #'cons
315                     (get-all-notes (second chords))
316                     :initial-value return-list
317                     :from-end t
318                 ))
319                 (setf return-list (reduce #'cons
320                     (get-all-notes (third chords))
321                     :initial-value return-list
322                     :from-end t
323                 ))
324             )
325             ((string-equal mode "minor"); on top of a minor chord, you can play either of the
       notes from the chord though the preferred order is 1-5-3
326                 (setf return-list (reduce #'cons
327                     (get-all-notes (first chords))
328                     :initial-value return-list
329                     :from-end t
330                 ))
331                 (setf return-list (reduce #'cons
332                     (get-all-notes (second chords))
333                     :initial-value return-list
334                     :from-end t
335                 ))
336                 (setf return-list (reduce #'cons
337                     (get-all-notes (third chords))
338                     :initial-value return-list
339                     :from-end t
340                 ))
341             )
342             ((string-equal mode "diminished"); only the third can be played on top of
      diminished chords
343                 (cond
344                     ((= inversion 0)
345                         (setf return-list (reduce #'cons
346                             (get-all-notes (second chords))
347                             :initial-value return-list
348                             :from-end t
349                         ))
350                     )
351                     ((= inversion 1)
352                         (setf return-list (reduce #'cons
353                             (get-all-notes (first chords))
354                             :initial-value return-list
355                             :from-end t
356                         ))
357                     )
358                     ((= inversion 2)
359                         (setf return-list (reduce #'cons
360                             (get-all-notes (third chords))
```

```
                                :initial-value return-list
                                :from-end t
                            ))
                        )
                    )
                )
            )
        )
    )
)

; function to get the mode of the chord (major, minor, diminished,...) and the inversion (0 = classical form, 1 = first inversion, 2 = second inversion)
(defun get-mode-and-inversion (intervals)
    (let ((major-intervals (list (list 4 3) (list 3 5) (list 5 4))); possible intervals in midi for major chords
        (minor-intervals (list (list 3 4) (list 4 5) (list 5 3))) ; possible intervals in midi for minor chords
        (diminished-intervals (list (list 3 3) (list 3 6) (list 6 3)))); possible intervals in midi for diminished chords
        (cond
            ((position intervals major-intervals :test #'equal); if the chord is major
                (list "major" (position intervals major-intervals :test #'equal))
            )
            ((position intervals minor-intervals :test #'equal); if the chord is minor
                (list "minor" (position intervals minor-intervals :test #'equal))
            )
            ((position intervals diminished-intervals :test #'equal); if the chord is diminished
                (list "diminished" (position intervals diminished-intervals :test #'equal))
            )
        )
    )
)

;makes a list (name voice-instance) from a list of voices:
;(from Karim Haddad)
(defun make-data-sol (liste)
  (loop for l in liste
        for i from 1 to (length liste)
        collect (list (format nil "solution ~D: ~A"  i l) l)))


; taken from rhythm box
; https://github.com/blapiere/Rhythm-Box
(defun rel-to-gil (rel)
"Convert a relation operator symbol to a GiL relation value."
    (cond
        ((eq rel '=) gil::IRT_EQ)
        ((eq rel '=/=) gil::IRT_NQ)
        ((eq rel '<) gil::IRT_LE)
        ((eq rel '=<) gil::IRT_LQ)
        ((eq rel '>) gil::IRT_GR)
        ((eq rel '>=) gil::IRT_GQ)
    )
)

; Create push and pull list from a voice object
(defun create-push-pull (input-chords quant)
    (let (temp
         (next 0)
         (push (list))
         (pull (list '-1))
         (playing (list))
         (tree (om::tree input-chords))
         (pitch (to-pitch-list (om::chords input-chords))))
         (setq tree (second tree))
         (loop :for i :from 0 :below (length tree) :by 1 :do
            (setq temp (read-tree (make-list quant :initial-element -1) (make-list quant :initial-element -1) (make-list quant :initial-element -1) (second (first (second (nth i tree)))) pitch 0 quant next))
            (setq push (append push (first temp)))
            (setq pull (append pull (second temp)))
            (setq playing (append playing (third temp)))
            (setf next (fourth temp))
        )
        (list push pull playing))
)

; <tree> is the rhythm tree to read
; <pitch> is the ordered list of pitch
; <pos> is the next position in push to add values
; <length> is the current duration of a note to add
; <next> is the index in pitch of the next notes we will add
;recursive function to read a rhythm tree and create push and pull
(defun read-tree (push pull playing tree pitch pos length next)
    (progn
        (setf length (/ length (length tree)))
        (loop :for i :from 0 :below (length tree) :by 1 :do
            (if (typep (nth i tree) 'list)
                (let (temp)
                    (setq temp (read-tree push pull playing (second (nth i tree)) pitch pos length next))
                    (setq push (first temp))
```

```
                    (setq pull (second temp))
                    (setq playing (third temp))
                    (setf next (fourth temp))
                    (setf pos (fifth temp))
                )
                (progn
                    (setf (nth pos push) (nth next pitch))
                    (loop :for j :from pos :below (+ pos (* length (nth i tree))) :by 1 :do
                        (setf (nth j playing) (nth next pitch))
                    )
                    (setf pos (+ pos (* length (nth i tree))))
                    (setf (nth (- pos 1) pull) (nth next pitch))
                    (setf next (+ next 1))
                )
            )
        )
        (list push pull playing next pos)
    )
)

; <input-chords> is the voice objects for the chords
; <quantOrig> quantification used by melodizer
; Return a list in which each element i represent a note starting at a time i*quant
; -1 means no note starting at that time, a chord object means multiple note starting
(defun create-push (input-chords quantOrig)
    (let ((note-starting-times (voice-onsets input-chords))
          (quant (/ (second (first (om::tempo input-chords))) (/ quantOrig 16)))
          (tree (om::tree input-chords))
          (push-list (list))
          (chords (to-pitch-list (om::chords input-chords))) ; get chords list
         )
         (setf note-starting-times (mapcar (lambda (n) (/ n quant)) note-starting-times)) ;
    dividing note-starting-times by quant
         (loop :for j :from 0 :below (+ (max-list note-starting-times) 1) :by 1 :do
            (if (= j (car note-starting-times)); if j == note-starting-times[0]
                (progn
                    (setq push-list (nconc push-list (list (car chords))))
                    (setf chords (cdr chords))
                    (setf note-starting-times (cdr note-starting-times))) ;add chords[0] to
    push and prune qt[0] and pchords[0]
                (setq push-list (nconc push-list (list -1)))) ; else add -1 to push
        )
    )
)


; <input-chords> is the voice objects for the chords
; <quant> NOT USED YET (FORCED TO 500) smallest possible note length
; Return a list in which each element i represent a note stopping at a time i*quant
; -1 means no note stop at that time, a chord object means multiple note starting
(defun create-pull (input-chords)
    (let ((note-starting-times (voice-onsets input-chords)) ; note-starting-times = start
    time of each chord
          (note-dur-times (voice-durs input-chords)) ; note-dur-times = duration of each note
          (note-stopping-times (list))
          (quant 500)
          (pull-list (list))
          (pitch (to-pitch-list (om::chords input-chords))) ; get chords list
         )
         (setf note-starting-times (mapcar (lambda (n) (/ n quant)) note-starting-times)) ;
    dividing note-starting-times by quant
         (setf note-dur-times (mapcar (lambda (n) (mapcar (lambda (m) (/ m quant)) n))
    note-dur-times)) ; dividing note-dur-times by quant
         (loop :for j :from 0 :below (length note-starting-times) :by 1 :do
             (setq note-stopping-times (nconc note-stopping-times (list (mapcar (lambda (n)
    (+ n (nth j note-starting-times))) (nth j note-dur-times))))) ; Adding
    note-starting-times to note-dur-times to get note-stopping-times
        )
        (loop :for j :from 0 :below (+ (max-list-list note-stopping-times) 1) :by 1 :do
              (setq pull-list (nconc pull-list (list -1))))
        (loop for l in note-stopping-times
              for k in pitch do
            (loop for i in l
                  for j in k do
                  (if (typep (nth i pull-list) 'list)
                      (setf (nth i pull-list) (nconc (nth i pull-list) (list j)))
                      (setf (nth i pull-list) (list j)))
             )
        )
    )
)

; reformat a scale to be a canvas of pitch and not intervals
(defun adapt-scale (scale)
    (let ((major-modified (list (first scale))))
         (loop :for i :from 1 :below (length scale) :by 1 :do
            (setq major-modified (nconc major-modified (list (+ (nth i scale) (nth (- i 1)
    major-modified)))))
         )
    (return-from adapt-scale major-modified)
    )
)

```

```
; build the list of acceptable pitch based on the scale and a key offset
(defun build-scaleset (scale offset)
    (let ((major-modified (adapt-scale scale))
          (scaleset (list)))
        (loop :for octave :from -1 :below 11 :by 1 append
              (setq scaleset (nconc scaleset (mapcar (lambda (n) (+ (+ n (* octave 12))
    offset)) major-modified)))
        )
        (setq scaleset (remove-if 'minusp scaleset))
    )
)

; build the list of acceptable pitch based on the scale and a key offset
(defun build-notesets (chord offset)
    (let ((chord-modified (adapt-scale chord))
          (notesets (list)))
        (loop :for i :from 0 :below (length chord-modified) :by 1 :do
            (setq noteset (list))
            (loop :for octave :from -1 :below 11 :by 1 append
                  (setq noteset (nconc noteset (list (+ (+ (nth i chord-modified) (* octave
    12)) offset))))
            )
            (setq noteset (remove-if 'minusp noteset))
            (setq notesets (nconc notesets (list noteset)))
        )
        notesets
    )
)



; <chords> a list of chord object
; Return the list of pitch contained in chords in midi format
(defun to-pitch-list (chords)
     (loop :for n :from 0 :below (length chords) :by 1 collect (to-midi (om::lmidic (nth n
    chords))))
)


; Getting a list of chords and a rhythm tree from the playing list of intvar
(defun build-voice (sol push pull bars quant tempo)
    (let ((p-push (list))
          (p-pull (list))
          (chords (list))
          (tree (list))
          (ties (list))
          (prev 0)
          )

    (setq p-pull (nconc p-pull (mapcar (lambda (n) (to-midicent (gil::g-values sol n))) pull)
    ))
    (setq p-push (nconc p-push (mapcar (lambda (n) (to-midicent (gil::g-values sol n))) push)
    ))

    (setq count 1)
    (loop :for b :from 0 :below bars :by 1 :do
        (if (not (nth (* b quant) p-push))
            (setq rest 1)
            (setq rest 0)
        )
        (setq rhythm (list))
        (loop :for q :from 0 :below quant :by 1 :do
            (setq i (+ (* b quant) q))
            (cond
                ((nth i p-push)
                     ; if rhythm impulse
                     (progn
                        (setq durations (list))
                        (loop :for m :in (nth i p-push) :do
                            (setq j (+ i 1))
                            (loop
                                (if (nth j p-pull)
                                    (if (find m (nth j p-pull))
                                        (progn
                                            (setq dur (* (floor 60000 (* tempo quant)) (- j i
    )))
                                            (setq durations (nconc durations (list dur)))

                                            (return)
                                        )
                                    )
                                )
                                (incf j)
                            )
                        )
                        (setq chord (make-instance 'chord :LMidic (nth i p-push) :Ldur
    durations))
                        (setq chords (nconc chords (list chord)))
                        (cond
                            ((= rest 1)
                                (progn
                                    (setq rhythm (nconc rhythm (list (* -1 count))))
                                    (setq rest 0)))
```

```
                            ((/= q 0)
                                (setq rhythm (nconc rhythm (list count))))
                        )
                        (setq count 1))
                )
                ; else
                (t (setq count (+ count 1)))
            )
        )
        (if (= rest 1)
            (setq rhythm (nconc rhythm '(list (* -1 count))))
            (setq rhythm (nconc rhythm (list count)))
        )
        (setq count 0)
        (setq rhythm (list '(4 4) rhythm))

        (setq tree (nconc tree (list rhythm)))
    )
    (setq tree (list '? tree))

    (list chords tree)
    )
)

(defun build-chord-seq (sol push pull bars quant tempo)
    (let ((p-push (list))
          (p-pull (list))
          (chords (list))
          (durations (list))
          (onsets (list)))

        (setq p-pull (nconc p-pull (mapcar (lambda (n) (to-midicent (gil::g-values sol n))) pull)))
        (setq p-push (nconc p-push (mapcar (lambda (n) (to-midicent (gil::g-values sol n))) push)))

        (loop :for i :from 0 :below (+ (* bars quant) 1) :do
            (if (nth i p-push)
                (progn
                    (setq onset (* (/ 60000 (* tempo (/ quant 4))) i))
                    (setq duration (list))
                    (loop :for m :in (nth i p-push) :do
                        (setq j (+ i 1))
                        (loop
                            (if (nth j p-pull)
                                (if (find m (nth j p-pull))
                                    (progn
                                        (setq dur (* (/ 60000 (* tempo (/ quant 4))) (- j i)))
                                        (setq duration (nconc duration (list dur)))

                                        (return)
                                    )
                                )
                            )
                            (incf j)
                        )
                    )
                    (setq chords (nconc chords (list (nth i p-push))))
                    (setq durations (nconc durations (list duration)))
                    (setq onsets (nconc onsets (list onset)))
                )
            )
        )

        (list chords onsets durations)
    )
)

;return T if the two list have the same elements (order doesn't matter)
(defun compare (l1 l2)
  (and (subsetp l1 l2) (subsetp l2 l1)))

; return the quant value based on the index selected
(defun get-quant (str)
  (cond ((string= str "1 bar") 1)
    ((string= str "1/2 bar") 2)
    ((string= str "1 beat") 4)
    ((string= str "1/2 beat") 8)
    ((string= str "1/4 beat") 16)
    ((string= str "1/8 beat") 32)
    ((string= str "1/3 bar") 3)
    ((string= str "1/6 bar") 6)
    ((string= str "1/3 beat") 12)
    ((string= str "1/6 beat") 24)
    ((string= str "1/12 beat") 48)
    ((not str) 192))
)

; return the quant value based on the index selected
(defun get-length (str)
  (cond ((string= str "1 bar") 192)
    ((string= str "1/2 bar") 96)
```

```
    ((string= str "1 beat") 48)
    ((string= str "1/2 beat") 24)
    ((string= str "1/4 beat") 12)
    ((string= str "1/8 beat") 6)
    ((string= str "1/3 bar") 64)
    ((string= str "1/6 bar") 32)
    ((string= str "1/3 beat") 16)
    ((string= str "1/6 beat") 8)
    ((string= str "1/12 beat") 4)
    ((not str) 1))
)

; shuffles a list
; from https://gist.github.com/shortsightedsid/62d0ee21bfca53d9b69e
(defun list-shuffler (input-list &optional accumulator)
  "Shuffle a list using tail call recursion."
  (if (eq input-list nil)
      accumulator
      (progn
  (rotatef (car input-list)
     (nth (random (length input-list)) input-list))
  (list-shuffler (cdr input-list)
         (append accumulator (list (car input-list)))))))
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**