

*UNIVERSITE PARIS 7 - DENIS DIDEROT*

*UFR INFORMATIQUE*

**DOCTORAT**

Informatique

**CHARLOTTE TRUCHET**

CONTRAINTES, RECHERCHE LOCALE  
ET COMPOSITION ASSISTÉE PAR ORDINATEUR

**Thèse dirigée par Philippe CODOGNET**

**JURY**

|                                                         |              |
|---------------------------------------------------------|--------------|
| M. Patrick COUSOT, Ecole Normale Supérieure             | Président    |
| M. Edward TSANG, University of Essex                    | Rapporteur   |
| Mme Myriam DESAINTE-CATHERINE, Université de Bordeaux 1 | Rapporteur   |
| M. Philippe CODOGNET, Université de Paris 6             | Directeur    |
| M. Gérard ASSAYAG, IRCAM                                | Co-directeur |
| M. Guy COUSINEAU, Université de Paris 7                 | Examineur    |

Membres invités :

|                                            |           |
|--------------------------------------------|-----------|
| M. Frédéric BENHAMOU, Université de Nantes | Examineur |
| M. Marc CHEMILLIER, Université de Caen     | Examineur |



# Remerciements

Les remerciements de thèse s'étendent parfois sur plusieurs pages, au risque d'apparaître convenus, voire peu sincères. Il est évident que cette thèse doit beaucoup à beaucoup de gens. Ils le savent bien je crois, et je leur présente mes excuses pour ne pas en dresser la liste.

Mais trop de remerciements tuent les remerciements.

Je remercie le dieu des thésards <sup>1</sup> d'avoir travaillé sous la direction de Gérard Assayag et Philippe Codognet. D'abord, la liste de ce que je leur dois est longue. Mais surtout, bien que très différents, ils sont tous les deux de ces rares personnes que connaître est, déjà, une chance. C'est ici l'occasion de leur dire ma gratitude et mon admiration. Je ne les remercierai jamais assez.

A titre personnel, je remercie Cécile Le Bourdon, Florent Tournois, Didier, Catherine, Marie, Camille et Thomas Truchet.

Et Olivier, toujours plus.

---

<sup>1</sup>aidé par Patrick Cousot



# Table des matières

|                                                                        |              |
|------------------------------------------------------------------------|--------------|
| <b>Résumé</b>                                                          | <b>1</b>     |
| 0.1 Introduction (français) . . . . .                                  | 5            |
| 0.2 Introduction (english) . . . . .                                   | 6            |
| <br><b>I Informatique musicale, programmation par contraintes</b>      | <br><b>9</b> |
| <b>1 Summary of Part I : Computer Music and Constraint Programming</b> | <b>11</b>    |
| <b>2 Informatique musicale</b>                                         | <b>13</b>    |
| 2.1 Ordinateur et musique . . . . .                                    | 13           |
| 2.2 Composition Assistée par Ordinateur . . . . .                      | 14           |
| <b>3 OpenMusic</b>                                                     | <b>17</b>    |
| 3.1 OpenMusic, introduction générale . . . . .                         | 17           |
| 3.2 Le Langage visuel d'OpenMusic . . . . .                            | 18           |
| 3.3 Implémentation . . . . .                                           | 22           |
| 3.4 L'environnement OpenMusic . . . . .                                | 25           |
| 3.5 Un exemple musical . . . . .                                       | 29           |
| 3.6 Modèle de distribution et méthodologie . . . . .                   | 32           |
| 3.7 Les utilisateurs, musiciens . . . . .                              | 33           |
| 3.8 Conclusion sur OpenMusic . . . . .                                 | 35           |
| <b>4 Programmation par contraintes</b>                                 | <b>37</b>    |
| <b>5 Problème de Satisfaction de Contraintes (CSP)</b>                 | <b>39</b>    |
| 5.1 Définition . . . . .                                               | 39           |
| 5.2 Modélisation et problème des solutions approchées . . . . .        | 40           |
| 5.3 Modélisation en contraintes binaires . . . . .                     | 42           |
| <b>6 Quelques familles de CSPs</b>                                     | <b>43</b>    |
| 6.1 SAT . . . . .                                                      | 43           |
| 6.2 Tournées de véhicules . . . . .                                    | 44           |
| 6.3 Allocation de fréquences . . . . .                                 | 44           |
| 6.4 Problèmes de sac à dos (knapsack problem) . . . . .                | 44           |

|           |                                                                    |           |
|-----------|--------------------------------------------------------------------|-----------|
| <b>7</b>  | <b>Méthodes complètes</b>                                          | <b>45</b> |
| 7.1       | Principe général, arbre de recherche . . . . .                     | 45        |
| 7.2       | Backtracking (retour arrière chronologique) . . . . .              | 46        |
| 7.3       | Arc consistance . . . . .                                          | 48        |
| 7.4       | Forward checking . . . . .                                         | 49        |
| <b>8</b>  | <b>Les soft constraints</b>                                        | <b>53</b> |
| 8.1       | Langage de contraintes . . . . .                                   | 54        |
| <b>9</b>  | <b>Les métaheuristiques</b>                                        | <b>57</b> |
| 9.1       | SAT . . . . .                                                      | 57        |
| 9.2       | La recherche locale . . . . .                                      | 58        |
| 9.3       | Autres . . . . .                                                   | 60        |
| 9.4       | Les méthodes hybrides . . . . .                                    | 61        |
| 9.5       | Langages pour la recherche locale . . . . .                        | 62        |
| <b>10</b> | <b>Contraintes en musique</b>                                      | <b>65</b> |
| 10.1      | Harmonisation automatique . . . . .                                | 65        |
| 10.2      | En contraintes et musique, divers . . . . .                        | 66        |
| 10.3      | En Composition Assistée par Ordinateur . . . . .                   | 67        |
| 10.4      | Conclusion, la place de la programmation par contraintes . . . . . | 68        |
| <b>II</b> | <b>Modélisation des CSPs musicaux</b>                              | <b>71</b> |
| <b>11</b> | <b>Summary of Part II : Musical CSPs modelization</b>              | <b>73</b> |
| <b>12</b> | <b>Introduction, le problème du classement</b>                     | <b>77</b> |
| <b>13</b> | <b>Problèmes portant sur l'harmonie</b>                            | <b>79</b> |
| 13.1      | Problème 1 : Suite d'accords avec note commune . . . . .           | 79        |
| 13.2      | Problème 2 : Harmonisation à structure rythmique donnée . . . . .  | 81        |
| 13.3      | Problème 3 : Classement d'accords . . . . .                        | 86        |
| 13.4      | Conclusion . . . . .                                               | 86        |
| <b>14</b> | <b>Problèmes portant sur le rythme</b>                             | <b>87</b> |
| 14.1      | Problème 4 : Rythmes asynchrones . . . . .                         | 87        |
| 14.2      | Problème 5 : Durées dans Fibonacci . . . . .                       | 90        |
| 14.3      | Problème 6 : Tempi et accelerando . . . . .                        | 91        |
| 14.4      | Problème 7 : Suivi de tempo . . . . .                              | 92        |
| 14.5      | Conclusion . . . . .                                               | 93        |
| <b>15</b> | <b>Problèmes portant sur une mélodie</b>                           | <b>95</b> |
| 15.1      | Problème 8 : Suite de gestes . . . . .                             | 95        |
| 15.2      | Problème 9 : Séries tous intervalles . . . . .                     | 96        |
| 15.3      | Conclusion . . . . .                                               | 97        |

|                                                                         |            |
|-------------------------------------------------------------------------|------------|
| <b>16 Problèmes d'analyse musicale</b>                                  | <b>99</b>  |
| 16.1 Problème 10 : Textures de Ligeti . . . . .                         | 99         |
| 16.2 Problème 11 : Imparité rythmique . . . . .                         | 101        |
| 16.3 Problème 12 : Canons Nzakara . . . . .                             | 103        |
| 16.4 Conclusion . . . . .                                               | 104        |
| <b>17 Autres problèmes</b>                                              | <b>105</b> |
| 17.1 Problème 13 : Echelles en quarts de ton . . . . .                  | 105        |
| 17.2 Problème 14 : Doigtés pour la guitare . . . . .                    | 106        |
| <b>18 Résumé</b>                                                        | <b>113</b> |
| <b>19 Conclusions</b>                                                   | <b>115</b> |
| 19.1 Variété multiple . . . . .                                         | 115        |
| 19.2 Variables . . . . .                                                | 115        |
| 19.3 Contraintes "oubliées" . . . . .                                   | 116        |
| 19.4 On ne veut pas de solution . . . . .                               | 116        |
| 19.5 Solutions approchées . . . . .                                     | 116        |
| <b>III Recherche adaptative et applications musicales</b>               | <b>117</b> |
| <b>20 Summary of Part III : Adaptive Search and Musical Application</b> | <b>119</b> |
| <b>21 Recherche Adaptative</b>                                          | <b>121</b> |
| 21.1 Introduction . . . . .                                             | 121        |
| 21.2 La Recherche Adaptative, présentation générale . . . . .           | 122        |
| 21.3 Algorithme . . . . .                                               | 123        |
| 21.4 Exemple pour les carrés magiques . . . . .                         | 124        |
| 21.5 Fonctions de coût . . . . .                                        | 125        |
| 21.6 Paramétrage . . . . .                                              | 128        |
| <b>22 Résolution des CSPs musicaux</b>                                  | <b>131</b> |
| 22.1 Problème 1 : Suite d'accords avec note commune . . . . .           | 131        |
| 22.2 Problème 2 : Harmonisation à structure rythmique donnée . . . . .  | 131        |
| 22.3 Problème 3 : Classement d'accords . . . . .                        | 132        |
| 22.4 Problème 4 : Rythmes asynchrones . . . . .                         | 132        |
| 22.5 Problème 5 : Durées dans Fibonacci . . . . .                       | 136        |
| 22.6 Problème 6 : Tempi et accelerando . . . . .                        | 136        |
| 22.7 Problème 7 : Suivi de tempo . . . . .                              | 136        |
| 22.8 Problème 8 : Suite de gestes . . . . .                             | 137        |
| 22.9 Problème 9 : Séries tous intervalles . . . . .                     | 139        |
| 22.10 Problème 10 : Textures de Ligeti . . . . .                        | 140        |
| 22.11 Problème 11 : Imparité rythmique . . . . .                        | 142        |
| 22.12 Problème 12 : Canons Nzakara . . . . .                            | 143        |
| 22.13 Problème 14 : Doigtés pour la guitare . . . . .                   | 144        |
| 22.14 Conclusion . . . . .                                              | 146        |

---

|                                                             |            |
|-------------------------------------------------------------|------------|
| 23 Recherche Adaptative et probabilités                     | 151        |
| <b>IV OMClouds</b>                                          | <b>153</b> |
| 24 Summary of Part IV : OMClouds                            | 155        |
| 25 Choix techniques                                         | 157        |
| 25.1 Modélisation . . . . .                                 | 157        |
| 25.2 Interface . . . . .                                    | 157        |
| 25.3 Objets . . . . .                                       | 158        |
| 26 Méthode complète, intégration de Screamer dans OpenMusic | 159        |
| 27 OMClouds                                                 | 163        |
| 27.1 Modélisation du problème . . . . .                     | 163        |
| 27.2 Définition des contraintes . . . . .                   | 165        |
| 27.3 Génération des coûts . . . . .                         | 166        |
| 27.4 Résolution . . . . .                                   | 167        |
| 27.5 Détail d'un exemple . . . . .                          | 171        |
| 27.6 Performances . . . . .                                 | 173        |
| 27.7 Conclusions . . . . .                                  | 175        |
| 28 Conclusions                                              | 177        |
| 28.1 Conclusion (français) . . . . .                        | 177        |
| 28.2 Conclusion (english) . . . . .                         | 177        |
| Bibliographie                                               | 178        |
| Annexe                                                      | 184        |

# Résumé

La Programmation par Contraintes (CP) permet de modéliser et de résoudre des problèmes fortement combinatoires par la spécification d'une certaine information partielle sur des variables, inconnues du problème. Dans cette thèse, nous avons étudié des problèmes de contraintes musicaux, posés par des compositeurs contemporains, d'analyse musicale et d'instrumentation, en collaboration avec l'IRCAM. Quatorze problèmes ont été modélisés et résolus au cas par cas, ce qui a permis d'en dresser une typologie détaillée. Ceci a servi à la conception et l'implémentation d'un algorithme de recherche locale dans l'environnement de Composition Assistée par Ordinateur OpenMusic, sous la forme d'une librairie dont l'architecture permet notamment de définir visuellement un problème de contraintes, de le résoudre, et le cas échéant d'éditer des résultats partiels ou approchés pendant le processus de résolution.

---

# Abstract

Constraint Programming (CP) allows to modelize and solve combinatory problems by specifying some partial information on variables, unknowns of the problem. In this thesis, we have studied musical problems, either stated by contemporary composers, or of musical analysis, or of instrumentation, in collaboration with IRCAM (french Instiute for Research and Coordination Acoustics / Music). Fourteen such problems have been modelized and solved, which allowed to give a detailed typology. This has been used to conceive and implement a local search algorithm as a library of the Computer Assisted Composition environment OpenMusic. Its architecture allows in particular to define a constraint problem visually, to solve it, and eventually to edit partial or approached results during the resolution process.

---

# Introduction

## 0.1 Introduction (français)

L'informatique musicale est un terme qui regroupe des domaines de recherches assez divers, dont la grande majorité sont tournés vers le traitement du son. Ces applications concernent parfois le processus de composition musicale, à quelque niveau que ce soit : nul besoin de remonter aux Grecs anciens pour imaginer ce que la puissance de calcul d'un ordinateur peut apporter à un compositeur, synthèse de sons nouveaux, dialogue entre interprètes et système temps réel, déplacement du son dans l'espace, etc. Parmi elles, la Composition Assistée par Ordinateur (CAO) a une place un peu particulière. La CAO peut se définir comme l'utilisation de l'ordinateur dans le processus de création musicale à un niveau symbolique (en clair, tout ce qui n'est pas un signal audio). La CAO se situe au niveau de la partition, en désignant abusivement sous ce mot toute notation symbolique de la musique. Il s'agit d'apporter à un compositeur des techniques issues de l'informatique, soit pour répondre à des besoins qu'il peut avoir, soit pour susciter de nouvelles idées musicales. Le lambda-calcul ou la programmation objet ont trouvé par exemple d'élégantes applications musicales dans le logiciel OpenMusic.

Du côté informatique, la programmation par contraintes occupe elle aussi une place à part. Il s'agit de programmer non pas en suivant une méthode de construction d'un résultat, mais en définissant des règles (contraintes), que l'on envoie à un *solver* de contraintes, lequel se charge de trouver une ou des réponses vérifiant les règles. La différence est fondamentale : dans un programme "habituel", celui qui programme doit connaître le procédé de construction du résultat. En programmation par contraintes, celui qui programme donne seulement une information partielle sur le ou les résultats qu'il souhaite obtenir. L'information est traitée par le *solver* de manière opaque pour l'utilisateur. Historiquement reliée à l'intelligence artificielle, la programmation par contraintes s'applique aujourd'hui beaucoup comme une manière de gérer l'explosion combinatoire, les contraintes servant à caractériser des configurations intéressantes dans un espace de recherche exponentiellement grand.

La programmation par contraintes a trouvé en musique une application très naturelle avec l'harmonisation à chant ou basse donnés (voir chapitre 1), où les contraintes sont l'exact analogue des règles de l'harmonie classique. Mais la musique contemporaine ne respecte plus les règles classiques depuis longtemps. Il est impossible aujourd'hui de se référer à de tels traités. Chaque compositeur utilise des structures musicales qui lui sont propres. La programmation par contraintes a-t-elle une place en CAO ? Les travaux précédents suggèrent que oui (voir chapitre 1). Sans entrer dans une analyse nécessairement vaine du processus de composition (ce que nous éviterons à tout prix, car ce serait hors sujet), on peut remarquer que les notions de domaines de valeurs possibles ou de règles sont assez naturelles en musique, même contemporaine (séries dodécaphoniques, mesures et rythmes, modes par exemple).

Une application musicale contemporaine de la programmation par contraintes est donc

nécessairement au cas par cas, comme l'ont été les travaux précédents, résultats d'une collaboration entre un chercheur et un compositeur (voir chapitre 1).

Le but de ce travail de thèse était d'étudier les possibles utilisations des contraintes en CAO, et éventuellement de fournir un système de résolution de contraintes adapté à la composition musicale. Nous avons choisi de commencer par travailler directement avec des compositeurs liés à l'IRCAM, en général familiers d'OpenMusic, mais pas avec la PLC<sup>2</sup>. Cela a permis de modéliser et résoudre une douzaine de problèmes musicaux de contraintes, au cas par cas. A un plus haut niveau, cette étape était cruciale, d'abord pour constater que la PLC passait bien chez les compositeurs, ensuite pour étudier l'usage qui en était fait, le type de problèmes posés, etc. Les problèmes sont détaillés chapitre 2.

Pour la résolution, de nombreuses raisons nous ont poussé vers un algorithme de recherche locale. Ces techniques non-déterministes, apparues récemment pour la résolution de contraintes, reposent sur une amélioration itérative d'une instanciation initiale aléatoire. Elles utilisent les contraintes pour mesurer la qualité d'une configuration. Certains problèmes ont été résolus par une méthode complète. L'ensemble est présenté chapitre 3.

Après cette phase de résolutions au cas par cas, nous avons implémenté un *solver* de contraintes par recherche locale sous la forme d'une librairie intégrée à OpenMusic, OMClouds, présentée chapitre 4. OMClouds a été conçue dans le but de profiter autant que possible des concepts fondamentaux d'OpenMusic. Ainsi, l'écriture des contraintes se fait visuellement, et les résultats peuvent être édités grâce aux objets musicaux d'OpenMusic. D'un point de vue CAO, l'une des principale caractéristiques de la librairie est de ne pas avoir intégré les contraintes dans la partition, comme cela avait été envisagé. Pour l'aspect contraintes, OMClouds a la spécificité d'exploiter les minima locaux comme des solutions approchées, notamment pour les problèmes surcontraints.

Quelques précisions : le lecteur non musicien trouvera en annexe quelques définitions de solfège pour les termes utilisés. Ce n'est pas un cours de solfège ni de musicologie, mais une courte explication des termes musicaux utilisés dans ce document. Les introduction et conclusion de la thèse sont en français, mais les résumés de chapitre sont en anglais uniquement.

## 0.2 Introduction (english)

Applications of computer science to music are very diverse, and encompasses lot of researches on sound processing. Some concern the composition process at various stages : no need to appeal to the Ancient Greeks to imagine what the computer's calculation power can bring to a composer, sound synthesis, dialogues between interpreters and real-time systems, spatial moving of the sound, etc. Computer Assisted Composition (CAC) holds a singular place among these various fields. CAC may be defined as the use of computer techniques in the musical composition process, at a symbolic level (namely, everything but an audio signal). CAC is thus situated at the score level, with an improper definition of score as every symbolic notation of music. The goal is to bring techniques from computer science to the composer, either to meet the needs he may have, or to give some him new ideas. For instance, lambda-calculus or object oriented programming have given very nice applications in music.

From the computer science point of view, Constraint Programming (CP) also holds a particular place. Contrary to the usual programming scheme which consists in following a construction method of the result, CP is about programming by defining rules (constraints), which

---

<sup>2</sup>Programmation Logique par Contraintes

are sent to a solver, which gives one or some answers satisfying the rules. The difference is fundamental : in usual programs, the programmer must know the constructing method of the result. In CP, the programmer only specifies some partial information on the result he/she wants to obtain. This information is computed by the solver, in what appears as a black box to the user. Historically linked to Artificial Intelligence, Constraint Programming is nowadays often applied as a tool to reduce combinatorial explosion. The constraints then characterize interesting configurations in an exponentially big search space.

Constraint Programming has found a very natural application in music with harmonization on a given bass (see chapter 1), where the constraints are the exact translation of the classical harmony rules. But contemporary music has not been following these rules for a long time. It is now impossible to refer to such treaty. Each composer has his own musical structures. Does CP still have something to do in CAC? Previous works suggest that it has (see chapter 1). We will not try to analyze the composition process (on the contrary we will stay far away from this area which is out of our domain), but it can be pointed out that the notion of rule is quite natural in music, even in contemporary music (dodecaphonic series, measures and rhythms, modal music for instance).

A contemporary musical application of CP is thus necessarily adapted to a particular composer, as it has been the case in the previous works (see chapter 1).

The goal of this PhD was to study possible applications of constraint programming in CAC, and possibly realize a constraint system adapted to musical composition. We have chosen to start and work directly with composers mainly from IRCAM. In general, they were familiar with OpenMusic, but not with constraint programming. This phase allowed us to modelize and solve a dozen of musical constraint problems, case by case. This phase was crucial, in order to firstly state that CP was easily accepted by the composers, then to study the use they made of it, the main features of the problems, etc. These problems are detailed in chapter 2.

Concerning the resolution, many reasons led us to choose a local search algorithm. These recently appeared non-deterministic techniques rely on iterative improvement of configurations, randomly initialized. Constraints are used as a measure of the quality of a configuration. Some problems have been solved with classical tools. All of this is presented on chapter 3.

After this phase, we have implemented a constraint solver by local search, in the form of an OpenMusic library, OMClouds, introduced on chapter 4. OMClouds has been designed in such a way that it benefits from fundamental concepts of OpenMusic, as far as possible. So the constraints definition is visual, and the results can be edited in the musical OM objects. From a CAC point of view, one of the main features of the library is not to have integrated the constraints inside the score, as it had first been considered. From the constraint point of view, OMClouds has the specificity to use the local minima as approached solutions, in particular for overconstrained problems.

The non-musician reader will find some rudiments of music in the first appendix. In no way, though, is it a music or musicology course, but a short explanation of the musical terms used in the document.



## Première partie

# Informatique musicale, programmation par contraintes



# Chapitre 1

## Summary of Part I : Computer Music and Constraint Programming

CAC is now a well-established research area, see [9]. It deals with a symbolic representation of music, mainly at the score level (or before), as opposed to sound synthesis, acoustic, etc. CAC provides the composer with computing tools, for him to handle with formal representations of music, and make calculation on these representations. One of the recent CAC softwares is OpenMusic [3], a full visual programming language based on CommonLisp / CLOS, developed at IRCAM. OpenMusic is a functional, object, visual programming language. A set of provided classes and libraries make it a very convenient environment for music composition. OpenMusic is developed jointly by computer scientists and composers, which gave us the opportunity to consult musician-users about the use they could do of CP, before developing a CP system.

On the other hand, Constraint Programming (CP) is also a well established research area. It is an artificial intelligence technique which has been attracting a growing interest for the last decades, both in the academic and industrial areas. CP allows to represent a problem by specifying some partial information on it. It offers a powerful and intuitive way of describing and solving combinatorial problems. A CP problem is structured as a Constraint Satisfaction Problem, with a set of variables (such as unknowns in mathematics), domains of possible values for those variables, and constraints which are predicates stated on the variables. The goal is to find an affectation of values to the variables such that the constraints are satisfied. Many techniques have been developed to solve CSPs. Some of them are called complete, they rely on an exhaustive search of all the possible combinations of values. In order to handle with the combinatorial explosion, high level notions have been introduced to try and reduce the size of the search space, mainly Arc-Consistency, which forbids incompatibilities for the values of the domains. These inconsistent values (values which, once combined to any other one, would lead to a failure) combined can be removed from the domains. Several algorithms have been developed since AC-1-3 [65].

On the other hand, incomplete methods have been used for the last decades in Combinatorial Optimization in order to find optimal or near-optimal solutions. They find their origin in the pioneering work of Lin on the Traveling Salesman Problem [64], see for instance in [1] [54]. The so-called local search techniques encompass a large class of complex methods, the best-known instances being simulated annealing, Tabu search and genetic algorithms, usually referred to as “meta-heuristics”. They work by iterative improvement over an initial state. Consider an optimization problem with cost function which makes it possible to evaluate the quality of a given configuration (assignment of variables to current values) and a transition function that

defines a set of "neighbors" for each configuration. The basic algorithm consists in starting from a random configuration, explore the neighborhood, select an adequate neighbor and then move to the best candidate. This process will go on until some satisfactory solution is found. To avoid being trapped in local optima, adequate mechanisms should be introduced, such as the adaptive memory of Tabu search, the cooling schedule of simulated annealing or similar stochastic mechanisms. Very good results have been achieved by dedicated and finely tuned local search methods for many problems such as the Traveling Salesman Problem, scheduling, vehicle routing, cutting stock, etc. Recently, these techniques have also been mixed either one with the other, or with complete methods, or with operational research, giving some very promising results.

Applying Constraint programming to classical music is a very natural idea, because of its harmonization rules which are constraint-like stated. The textbooks give rules like "no parallel fifth", "opposite motion between two voices", and so on. They are so to say written in a declarative way. For contemporary music, the rules certainly differ, but the principle of stating rules may remain, and some composers already have used constraint solvers. In the field of automatic harmonization by constraints programming, the first solver was proposed by Ebcioğlu [44], followed by Tsang and Aitken [102]. Philippe Ballesta wrote a solver based on Ilog Solver [14]. Pachel and Roy used the particular structures of the tonal music to build a system with BackTalk [80]. All these systems apply on a very specific problem, automatic harmonization in the choral style, and stand on a specific music theory, tonal music.

Two solvers have been developed for contemporary music, PWConstraints [62] and Situation [90]. Both are based on the forward checking algorithm. They are useful for certain types of problems, but still lack the efficiency, the genericity (for PWConstraints) and the ease of use that are crucial to real production situations.

Some applications of Constraint Programming to other fields of Computer Music are to be found in [78], where constraints are used as high level tools to control spatialization, [91] for a model of timed concurrent constraints as "concurrent" musicians in an orchestra, [81] for musical mosaicing, a way of recreating pieces of music from a sample database, and [20] where constraints specify hierarchical temporal relations between some boxes, which can contain sound files for instance.

The applications of CP to music thus cover a very wide range of musical structures and ideas. This is to be linked with the great variety of musical applications of computer science to music. In any way, it can hardly be asserted that constraints are used as a specific tool in music : most of the time, the applications are mainly guided by musical requirements and CP seems to have been chosen as a tool among others.

# Chapitre 2

## Informatique musicale

### 2.1 Ordinateur et musique

Si l'on définit l'informatique musicale au sens large par : n'importe quelle utilisation de l'ordinateur en musique, les applications sont extrêmement nombreuses et il n'est pas question d'en dresser ici une liste exhaustive. Eliminons d'emblée les logiciels "basiques" de traitement du son (peut-être un peu moins connus, mais analogues des logiciels de traitement d'images), comme ceux qui servent simplement à éditer un fichier son, ou à le couper par exemple.

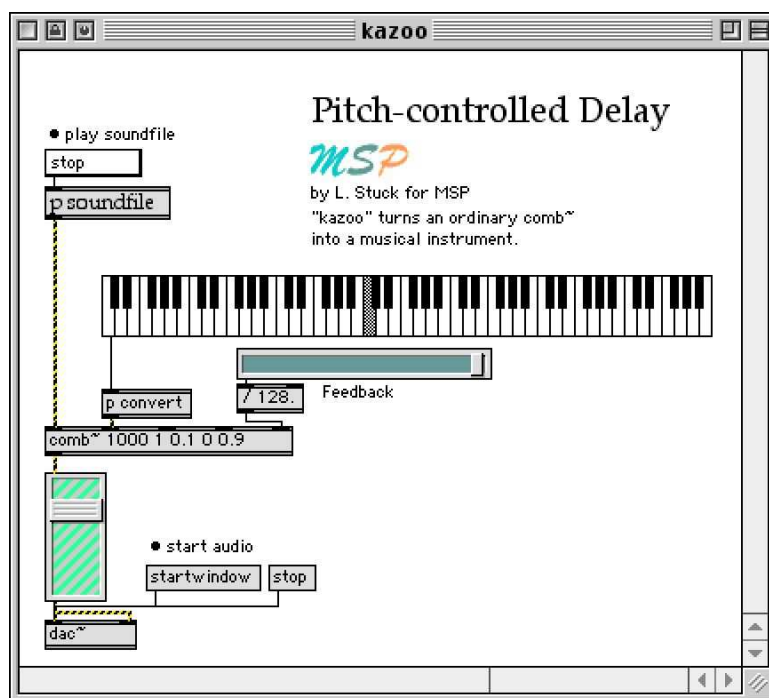


FIG. 2.1 – Un exemple de patch dans le langage Max : les opérations sont effectuées dans des boîtes connectées entre elles, dont certaines représentent visuellement des éléments de studio comme le clavier ou des boutons poussoirs.

Le plus célèbre est certainement Max/MSP. Inventé à l'Ircam par Miller Puckette à la fin des années 80, Max est maintenant pris en charge et commercialisé par David Zicarelli. La

métaphore visuelle est celle du studio électronique dans lequel des signaux voyagent à travers des modules et des connections, voir figure 2.1. Rappelons qu'un studio contient en général un synthétiseur et des instruments enregistrés avec des micros. Les sorties de son (micros, sorties de synthétiseurs...) sont reliés à une table de mixage, système électronique qui permet de récupérer tous les signaux audio, et d'en régler le niveau sonore *via* un certain nombre de curseurs. Les sorties de la table de mixage peuvent par exemple être branchées par exemple sur un système de baffles permet d'écouter le résultat sonore. De sorte que pour un habitué des studios, les actions telles que : connecter des entrées et sorties ou régler des niveaux, sont naturelles (plus que l'utilisation d'un langage de programmation en ligne de code). Max s'inspire de ce schéma avec une interface visuelle, dans laquelle on peut facilement (entre autres) brancher des signaux audio, jouer sur un synthétiseurs, etc.

Le modèle d'exécution de Max obéit à un double modèle : dirigé par les événements pour ce qui concerne le contrôle et MIDI, à flot synchrones pour ce qui concerne le traitement de signal. Le modèle d'évaluation est à sens unique : des messages de contrôle (ou du signal audionumérique) traversent le treillis des connections dans un seul sens (mettons de haut en bas). D'un point de vue technique, Max est d'une extrême efficacité pour le traitement du signal en temps réel, mais l'expressivité en terme de programmation et de structures de données est réduite. C'est donc plutôt un environnement pour la scène. Autres avantages de Max, il est réputé fiable, bien entretenu, bien documenté, d'accès relativement facile pour qui a une culture de studio, notamment grâce à d'excellents tutoriels, et il a de nombreux utilisateurs.

## 2.2 Composition Assistée par Ordinateur

La composition assistée par ordinateur (CAO) est un domaine né dans les années cinquante des premières tentatives de Hiller, Xenakis et autres pour utiliser l'ordinateur en musique. Ils adoptaient une approche symbolique (par opposition à l'analyse et la synthèse du signal audio) généralement considérée comme caractéristique de la CAO. La large diffusion, au cours des années quatre-vingt, de nouvelles technologies dans les langages informatiques (langages à objets, programmation logique, langages multi-paradigmes) et dans les interfaces graphiques a conditionné l'émergence d'une CAO " moderne " dans laquelle les concepts liés à la composition et ceux liés à la recherche informatique se sont quelquefois côtoyés de près. En effet, de la même façon que le choix d'un langage de programmation influence le programmeur dans les représentations qu'il favorise ou défavorise, il joue un rôle dans la formalisation d'une idée musicale, et peut susciter des expressions formelles qui ne seraient pas aisées dans un autre contexte. L'utilisation d'un langage de programmation bien défini oblige le musicien à réfléchir sur le processus même de la formalisation et lui évite de laisser l'ordinateur lui imposer ses choix.

Cette CAO contemporaine a bénéficié de nombreuses contributions parmi lesquelles nous citerons S. Pope [84] qui a proposé plusieurs systèmes basés sur le langage Smalltalk, ouvrant la possibilité pour le compositeur de définir ses propres objets et méthodes et de bénéficier d'une interface graphique de haut niveau ; E. Taube [100] qui a réalisé à Stanford le programme de composition par patterns Common Music ; F. Pachet qui met en oeuvre objets et contraintes dans MusES [77] ; P. Cointe [46] qui a proposé le langage Formes, dans lequel les processus musicaux sont représentés par des objets dont l'ordonnancement temporel est contrôlé par des abstractions appelées moniteurs ; G. Assayag et C. Malherbe [8] et [5] qui ont réalisé le premier environnement de CAO utilisé à l'Ircam, Crime, basé sur LeLisp de l'Inria, et dans

lequel des sous-langages enchâssés permettaient de définir une algèbre compositionnelle avec des sorties imprimées en notation musicale ; F. Courtot, [39][13], qui a réalisé Carla, un langage visuel de programmation logique basé sur Prolog II, dans lequel l'utilisateur définit des types d'objets musicaux par combinaisons de types existants ; Laurson, Duthen et Rueda [62], qui ont mis en oeuvre PatchWork, un langage visuel fonctionnel basé sur Lisp, et dont une idée particulièrement originale était la combinaison de graphes d'appel fonctionnels avec des éditeurs graphiques musicaux permettant de visualiser n'importe quel point du calcul.



# Chapitre 3

## OpenMusic

### 3.1 OpenMusic, introduction générale

OpenMusic (OM) est un langage visuel basé sur les paradigmes fonctionnel et objet, disposant d'un protocole de métaprogrammation et donc d'un certain degré de réflexivité. Ce langage a été développé à l'IRCAM, Institut de Recherche et Coordination Accoustique/Musique, par Gérard Assayag et Carlos Agon (équipe Représentations Musicales). OM compile vers un langage sous-jacent de haut niveau, Common Lisp / CLOS. Ce choix se justifie par le fait que CL/CLOS est un langage fonctionnel et objet puissant, portable, bien défini, disposant d'un modèle formel solide articulé autour du concept de fonction générique. OM a été conçu pour la composition musicale, mais il reste que ce langage visuel ne comporte pas de restriction par rapport à CL/ CLOS, et peut donc être utilisé pour la programmation en général. Nous ne décrivons ici que quelques aspects du langage, voir [3] et [10] pour une présentation plus détaillée et formelle.

L'interface principale dans OM est le workspace, un gestionnaire de fichiers sous forme d'icônes assurant la persistance de l'environnement. Le workspace est continuellement affiché pendant une session OpenMusic. Autre élément constant, le Listener, qui est une fenêtre Lisp dans laquelle sont affichés notamment les résultats des évaluations.

On peut écrire dans OpenMusic des programmes Lisp en lignes de code, ou bien des programmes dans le langage visuel, sous la forme de patches, l'un des objets centraux d'OM. Dans sa forme développée le patch constitue pour le langage visuel une expression correspondant à la notion habituelle de corps de programme, un exemple en est donné figure 3.1. Le patch s'ouvre dans une fenêtre, dans laquelle on peut disposer des icônes qui représentent les instructions du langage (fonctions, méthodes, objets). Ces icônes ont des petits points bleux en haut et en bas, figurant les entrées et sorties des fonctions correspondantes. On peut dessiner des connexions entre ces points bleux pour envoyer la sortie d'une fonction dans l'entrée d'une autre par exemple. Des boutons en haut à gauche de la fenêtre du patch permettent de créer des entrées et des sorties. Ainsi, un programme en OM s'exprime comme un agencement d'icônes interconnectées. L'ensemble forme un graphe d'invocations fonctionnelles. On peut demander l'évaluation en n'importe quel point de ce graphe (sur n'importe quelle boîte). L'évaluation d'une boîte est déclenchée lors de l'évaluation d'une de ses sorties. Il y a ainsi une chaîne d'évaluations qui correspond à l'exécution du programme. La figure 3.2 montre par exemple une définition de la factorielle en OpenMusic.

OpenMusic contient aussi de nombreuses bibliothèques développées par les utilisateurs (compositeurs ou scientifiques), contenant des classes et méthodes adaptées à des problèmes musicaux

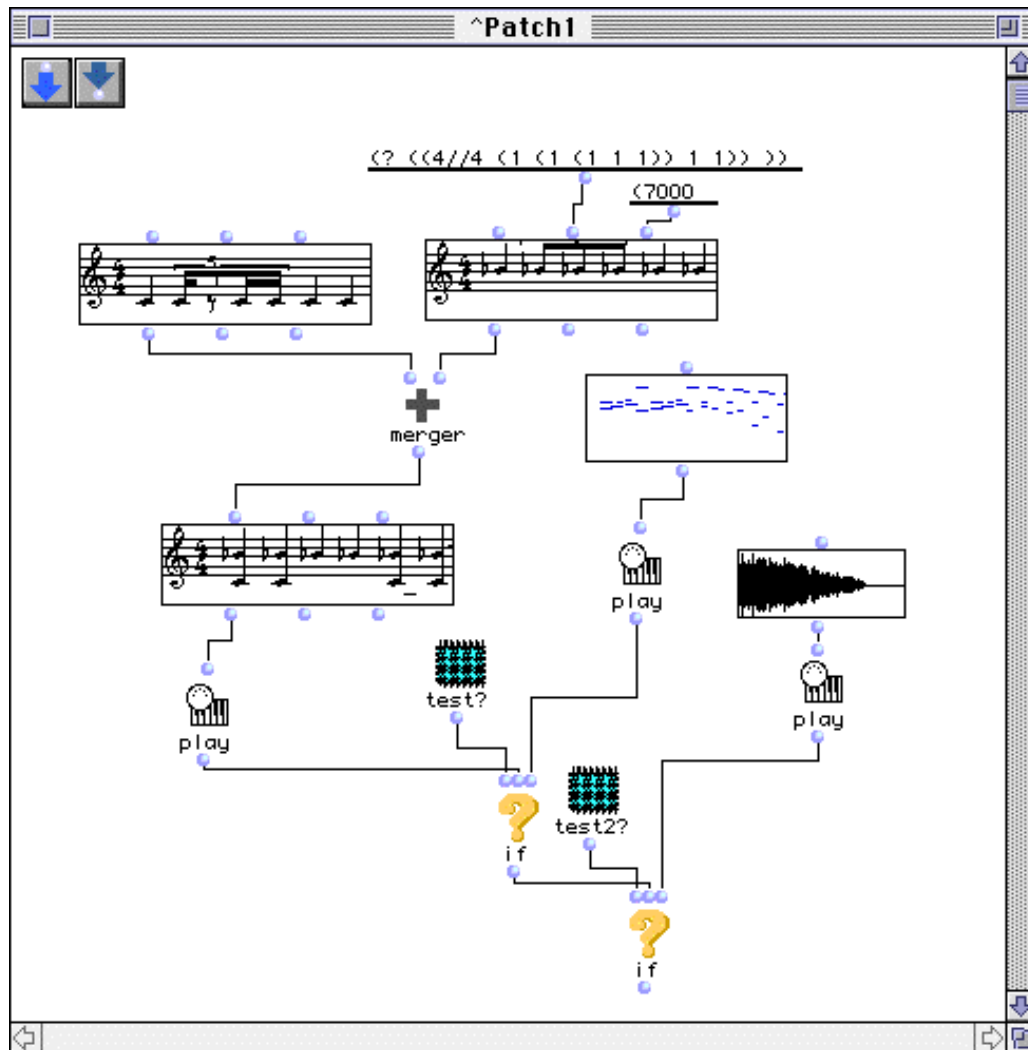


FIG. 3.1 – Exemple de patch OpenMusic.

particuliers. La version 4.6.5 d'OpenMusic, datant d'avril 2003, contient ainsi quarante et une bibliothèques. Trois sont liées aux contraintes (voir ci-dessus) : OMSituation, de Antoine Bonnet et Camilo Rueda, et OMCS, de Mickael Laurson, proposent chacune un solveur de contraintes. Au dessus de OMCS, une bibliothèque due à Örjan Sandred permet d'écrire facilement des contraintes rythmiques, il s'agit d'une bibliothèque de contraintes prédéfinies pour OMCS. Enfin, plusieurs bibliothèques assurent la communication entre OpenMusic et des logiciels de traitement du son (OM-Modalys Om2Csound OMChroma).

## 3.2 Le Langage visuel d'OpenMusic

### Les langages visuels

On peut classer les spécifications de langages visuels en trois grandes familles : approches algébrique, logique et grammaticale [67]. Nous donnons ici un exemple de chaque approche.

Les langages à Icônes basés sur une approche algébrique [24] [25] utilisent des arrangements

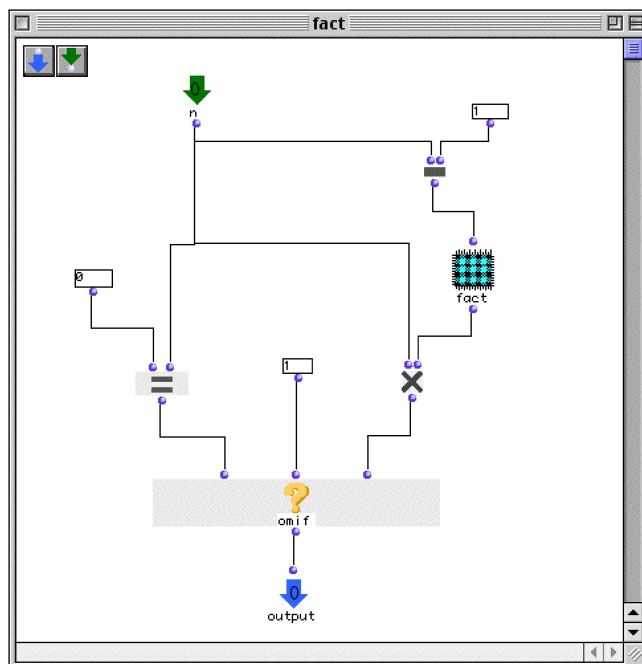


FIG. 3.2 – Définition de la factorielle dans le langage visuel d’OpenMusic. Les connexions dessinées par des lignes correspondent à l’application fonctionnelle. La récursivité est indiquée par le fait que le patch factorielle est dessiné à l’intérieur de lui-même.

d'icônes et d'opérateurs constituant des "phrases visuelles". Ces phrases peuvent alors être subsuées par une nouvelle icône, enrichissant ainsi le vocabulaire. Dans les langages logiques [52] [53], les termes du langage sont définis par des prédicats en logique descriptive décrivant les relations topologiques entre entités visuelles. Le langage Pictorial Janus peut être défini de cette façon. Parmi les avantages de cette approche, on souligne que l'écriture d'un programme est indépendante de l'éditeur, car les spécifications sont d'ordre topologique. Dans l'approche grammaticale [67] et [66], le langage est défini par les dérivations d'un système de réécriture.

Nous remarquons certaines constantes dans les trois approches étudiées. La plus importante semble être la difficulté de définir les frontières entre lexique et syntaxe. Le choix de primitives lexicales peut donc obéir à différents facteurs, entre autres à la fréquence d'utilisation, au degré de formalisation, etc. Après avoir choisi les primitives, il faut aussi faire des choix sur leur structure ; peuvent-elles avoir des attributs ? Si les primitives possèdent des sous-objets ou des attributs, quel type de calcul est-il permis entre les attributs ? On doit aussi décider de l'interprétation des relations spatiales entre objets et de la structure mathématique sous-tendant les expressions visuelles.

Les spécifications visuelles d'OM, données dans les sections suivantes suivent l'approche grammaticale, qui semblait la mieux adaptée pour faire le lien entre le formalisme de type  $\lambda$  & -calcul sous-tendant le langage CLOS et le langage visuel d'OM.

## Spécification lexicale

Le choix des primitives lexicales dans cette spécification nécessite un certain nombre de décisions : une description lexicale de très bas niveau rendrait notre description plus formelle mais aussi plus complexe. Nous avons choisi de faire abstraction de la formalisation de certains

objets de base et d'assumer leur définition de manière axiomatique. Ainsi nous supposons l'existence d'un ensemble d'éléments graphiques de base : l'icône, le rectangle, la ligne et le texte. En général, de telles formes ne se rencontrent pas de manière isolée mais sont plutôt combinées pour former des éléments de plus haut niveau que nous appellerons primitives lexicales. Nous supposons, sans entrer dans les détails, que nous disposons de deux types de relations spatiales de base : Contenant et En contact. Les primitives lexicales sont :

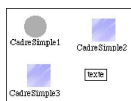
- Cadre de texte : un cadre de texte est défini par un rectangle contenant un texte



- Cadres simples : un Cadre simple est composé d'un icône en contact avec un texte :



- Cadres composés : un cadre composé est constitué d'un rectangle contenant un ensemble de cadres simples, de cadres de texte et éventuellement un ensemble des lignes.



- Lignes : ligne simple, flèche, ligne pointillée.
- Séquences : Une séquence est une forme particulière de cadre composé qui exprime une relation d'ordre entre les cadres simples qui lui appartiennent. Cette relation est représentée par la coordonnée des positions des cadres simples suivant un des axes du rectangle qui constitue la séquence. Ici par exemple on a Cadre1 < Cadre2 < Cadre3.

Un programme en OpenMusic s'exprimera comme un agencement sur l'écran de cadres composés vides ou non.

## Spécification syntaxique

Les entités sont les concepts abstraits de programmation sous-jacents au langage visuel. Elles sont constituées de termes tels que classe, instance, champ, fonction, etc. que nous ne définirons pas et qui sont à considérer dans leur acception habituelle. Chacune de ces entités peut être représentée de manière élémentaire et compacte sous la forme d'une vue ou de manière plus développée et éditable sous la forme d'un container. Les vues d'une entité sont construites à l'aide de cadres simples et de lignes, ou d'autres vues. Les containers sont des cadres composés, auxquels on adjoindra éventuellement des cadres de textes en contact avec le bord supérieur. Plusieurs vues mais un seul container d'une même entité peuvent coexister visuellement à un moment donné.

La notion de base dans OpenMusic est le patch. Le patch réifie le concept d'algorithme. Les patches contiennent des boîtes (icônes) et des connections entre elles. La figure 3.3 montre un patch en OpenMusic.

Les boîtes apparaissent exclusivement dans le corps d'un patch, et sont en effet des invocations : invocation d'une fonction générique, invocation d'une classe, et invocation d'un patch. L'invocation d'une classe résulte par définition en la création d'une nouvelle instance de cette classe. Un patch est une lambda-abstraction simple, à savoir une fonction non typée en entrée. Une boîte d'invocation de patch apparaîtra donc à l'intérieur du corps d'une méthode

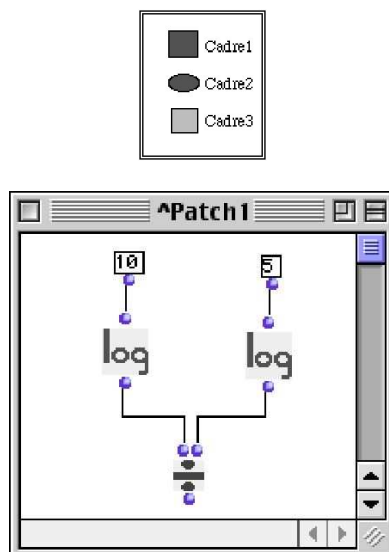
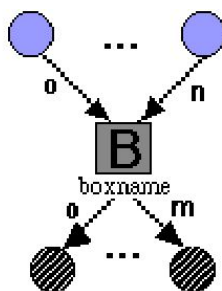


FIG. 3.3 – Un exemple de patch en OpenMusic.

ou bien dans le corps d'un autre patch. Les containers associées à ces boîtes comportent entre autres composants une référence qui pointe vers le container de l'entité associée. Par exemple le composant référence du container de la boîte d'invocation de fonction générique désigne le container de la fonction générique en question, lequel contient des vues de méthodes, lesquelles sont associées aux containers de méthodes qui contiennent le code visuel de ces méthodes.

Les inlets et les outlets des boîtes représentent des entrées et des sorties. L'indice  $n$  permet de garder trace de l'ordre de ces entrée/sorties. La figure 3.4 montre par exemple une vue de boîte qui pourrait correspondre à l'invocation d'une fonction générique à  $n$  arguments et à  $m$  sorties :

FIG. 3.4 – Représentation d'une fonction générique à  $n$  entrées,  $m$  sorties.

### Sémantique opérationnelle

Dans sa forme développée (container de patch) le patch constitue pour le langage visuel une expression correspondant à la notion commune de corps de programme. A l'intérieur d'un patch, on trouve des boîtes et des connexions qui représentent un graphe d'invocations fonctionnelles. On peut demander l'évaluation en n'importe quel point de ce graphe (sur n'importe quelle boîte). L'évaluation d'une boîte est déclenchée lors de l'évaluation d'une de ses sorties

(outlets). Il y a ainsi une chaîne d'évaluations qui correspond à l'exécution du programme. Nous allons donc interpréter les éléments décrits jusqu'ici en termes d'évaluation. La relation d'équivalence définie dans les règles suivantes suppose que les vues multiples d'une même entité sont équivalentes entre elles ; elles sont aussi équivalentes aux containers de la même entité. La sémantique est donnée sous la forme d'un ensemble de règles de réductions qui transforment les expressions graphiques en expressions Lisp. Ces règles utilisent un petit nombre de primitives Lisp telles que `first`, `rest`, `member`, et `funcall`. A titre d'exemple, nous montrons les règles d'évaluation pour une boîte d'invocation de patch, pour l'ensemble complet de règles voir [3].

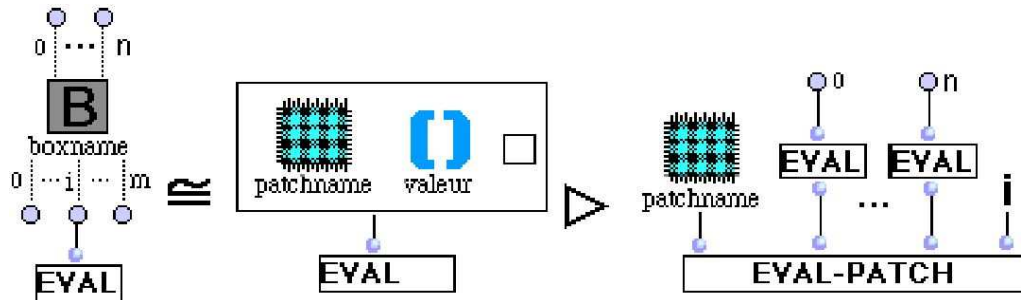


FIG. 3.5 – Invocation de patch.

L'évaluation d'une boîte Invocation de patch par son ième outlet se réécrit en un appel à `EvalPatch` (voir figure 3.5) avec les paramètres suivants : la référence de la boîte, l'évaluation du premier inlet de la boîte, l'évaluation du nème inlet de la boîte et l'indice de l'outlet évalué.

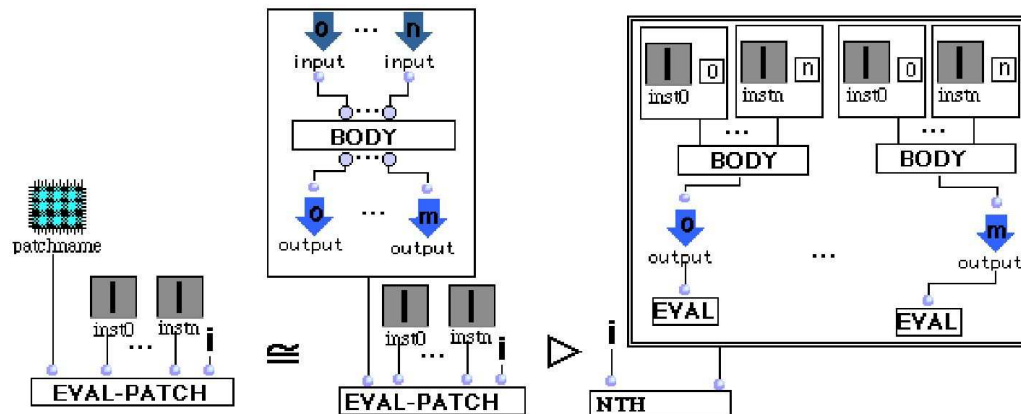


FIG. 3.6 – Patch.

L'évaluation d'un patch (`Eval-Patch`) avec arguments  $inst_0, \dots, inst_n$  et  $i$  se réécrit en le ième élément d'une liste  $(v_0 v_1 \dots v_m)$  correspondant aux évaluations des  $m+1$  output. Pour tout  $input_j$  sa valeur est remplacée par la valeur d'entrée  $inst_j$ .

### 3.3 Implémentation

OpenMusic a été implémenté sur le langage CommonLisp/CLOS sur Apple Macintosh. Le système de méta-programmation de CLOS contient une partie statique consistant en une hié-

rarchie de classes de méta-objets et une partie dynamique ou protocole de méthodes, qui permet la manipulation des méta-objets. Nous entendons par méta-objets les divers éléments du système objet sous-jacent à CLOS, tels que les fonctions génériques, les méthodes, les variables d'instances (slots) et les classes. Les deux tâches principales du méta-programmeur sont : définir des sous-classes des classes de méta-objets et définir ou redéfinir des méthodes pour ces nouvelles classes ou celles déjà existantes [82]. C'est cette technique que nous avons utilisée pour implémenter OpenMusic. OpenMusic constitue donc un enrichissement de la syntaxe et de la sémantique de CLOS par méta-programmation. Cependant, certaines entités visuelles, telles que patch ou boîte, n'ayant pas de correspondant dans CLOS, ne peuvent être constituées par méta-programmation (en effet, il n'y a pas de classe permettant de réifier la notion de programme ou d'invocation fonctionnelle dans CLOS). Nous ne ferons pas une description détaillée de CLOS, le lecteur peut consulter [99] et [82].

Pour implémenter OpenMusic, nous avons décidé d'opérer la même division que CLOS entre une partie statique et une partie dynamique. L'un des problèmes décisifs dans notre conception a été le niveau de détail auquel on définit le protocole ; en général, un protocole très détaillé est peu modulaire ; ainsi les changements doivent être effectués en plusieurs endroits, ce qui affecte la fiabilité. Par contre un protocole peu détaillé rend délicat le choix de l'endroit où introduire les modifications.

### Partie statique

La figure 3.7 illustre la partie statique d'OpenMusic. Les classes de méta-objets CLOS sont représentées par un rectangle gras.

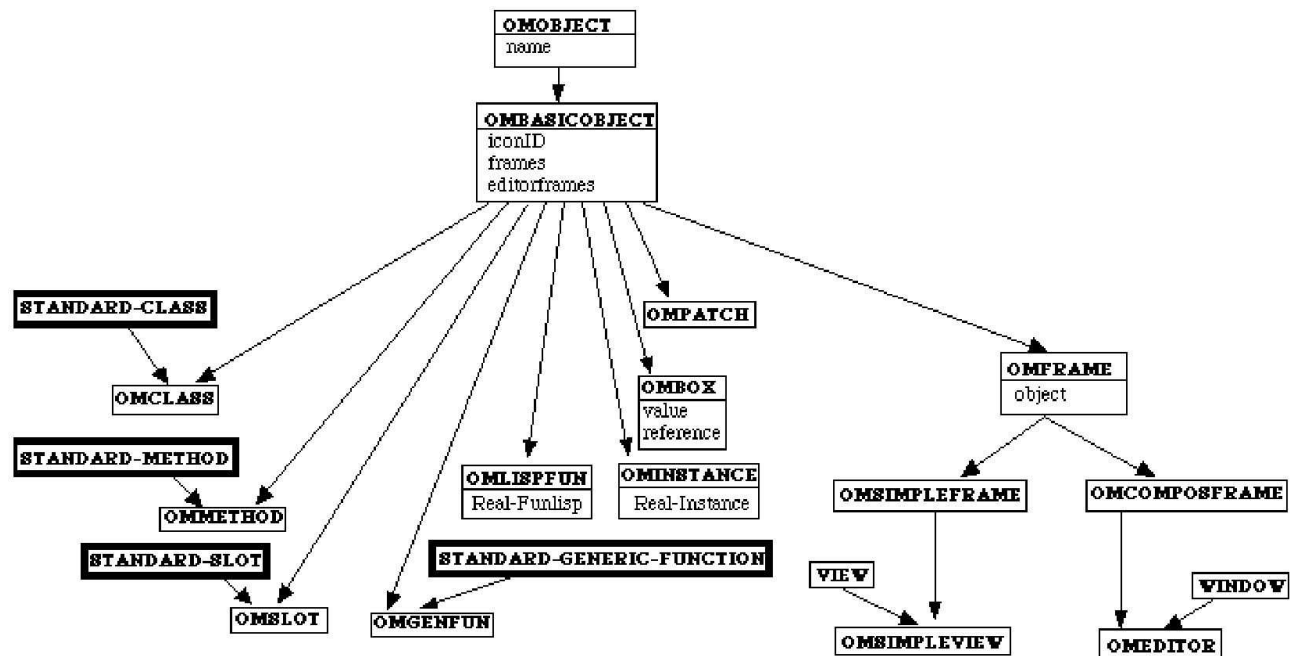


FIG. 3.7 – Hiérarchie de classes de méta-objets dans OpenMusic. dans OpenMusic.

Cette hiérarchie de classes rend compte de tous les objets manipulables dans l'environnement visuel et dans le langage de programmation visuel d'OpenMusic. Les classes fondamentales du système objet visuel d'OpenMusic, OMClass, OMMethod, OMSlot et OMGenFun, héritent

d'une part de la classe `OMObject` et d'autre part d'une classe fondamentale CLOS homologue. Les classes `OMPatch` et `OMBox` réifient respectivement les notions de patch et de boîte inexistantes dans CLOS. La classe `OMLispFun` (qui réifie la notion de fonction Lisp simple  $\tilde{N}$  non générique), ne peut être constituée par méta-programmation, car la classe fondamentale CLOS homologue fonction constitue une primitive qui n'est pratiquement pas sous-classable. Dans ce cas précis, nous opérerons par une sorte de délégation en agissant sur le champ `Real-FunLisp` qui contient un pointeur vers la fonction Lisp. Une dernière classe, `OMInstance`, est introduite afin de visualiser les instances en général. De même que pour les fonctions Lisp, le champ `Real-Instance` d'une `OMInstance` contient un pointeur vers l'instance CLOS. Les huit classes mentionnées, constituent les objets de calcul de notre langage (partie gauche de l'arbre d'héritage). La partie droite de l'arbre d'héritage implémente le paradigme visuel. On peut noter que la classe `OMFrame` hérite aussi de la classe `OMBasicObject`. Cela signifie que le paradigme visuel fait partie du langage, ceci étant fait dans l'intention de permettre à l'utilisateur de modifier et de contrôler ce paradigme. Tout `OMObject` a un nom (`name`) et peut être visualisé sous deux formes, soit comme une vue simple ou comme un éditeur (équivalent au container détaillé plus haut). Un éditeur est une instance de la classe `OMEditor`, tandis qu'une vue est une instance de la classe `OMSimpleView`. Tout objet de visualisation (instance de la classe `OMFrame`) est fortement attaché à un objet de calcul. Ce lien est exprimé par le champ `object` de la classe `OMFrame`. Les entités du langage, à leur tour, possèdent deux champs, `frames` et `editorframes`, qui contiennent respectivement une liste des vue simples et un éditeur qui visualisent l'objet en question.

## Protocole dynamique

Comme on l'a vu, les objets sont composés par un ensemble d'éléments et optionnellement par des relations entre ces éléments. Par exemple, une classe est une liste ordonnée des champs, une fonction générique consiste en un ensemble de méthodes, un patch contient une liste de boîtes connectées, etc. Une fonction fondamentale dans notre protocole est donc la fonction `get-elements`, celle-ci retournant pour chaque `OMBasicObject` la liste de ses éléments, par exemple pour une `OMClass` nous aurons :

`get-elements : OMClass → liste de OMSlots`

Le mécanisme de glisser-déposer est le principal outil d'édition dans OpenMusic. Une opération de glisser-déposer est définie comme une action entre une instance de la classe `OMSimpleView` appelée "source" et une instance de la classe `OMEditor` appelée "cible". Il existe un ensemble de prédicats `allow-drop` qui déterminent si l'opération de glisser-déposer est possible entre une paire (source, cible). Les fonctions `add-element` et `remove-element` ajoutés aux mécanismes de glisser-déposer et de délégation définissent le paradigme d'édition d'objets.

`add-element : OMClass * OMSlot → OMClass`

`remove-element : OMClass * OMSlot → OMClass`

Glisser une vue (source) dans la fenêtre d'un éditeur (cible) produit en général l'appel de la fonction générique `add-element` aux paramètres cible et source. Cette fonction est chargée d'effectuer les modifications visuelles dans l'éditeur ; elle doit aussi déléguer l'invocation de la fonction `add-element` au couple formé par les champs `object` de cible et de source. Cette dernière invocation est chargée de modifier l'objet de calcul visualisé par l'éditeur. Les fonctions suivantes, qui complètent le protocole suivent le même paradigme :

|                                                             |        |
|-------------------------------------------------------------|--------|
| rename : Obj * name                                         | → Obj  |
| Change le nom d'Obj par name                                |        |
| change-icon : Obj * iconID                                  | → Obj  |
| Change l'icône d'Obj par iconID                             |        |
| select : Obj                                                | → Obj  |
| Sélectionne Obj                                             |        |
| unselect : Obj                                              | → Obj  |
| Désélectionne Obj                                           |        |
| moveobject : Obj * position                                 | → Obj  |
| Change la position d'Obj à position                         |        |
| connect : SourceBox * i * TargetBox * j                     | → Bool |
| Connecte la sortie i de SourceBox à l'entrée j de TargetBox |        |
| box-value : Box * i                                         | → Obj  |
| Evalue la i-eme sortie de la boîte Box                      |        |

## 3.4 L'environnement OpenMusic

### Le workspace

L'interface principale dans OpenMusic est le workspace. Le workspace est un gestionnaire de fichiers icônique assurant la persistance de l'environnement, les échanges avec le système hôte se faisant par glisser-déposer. Plusieurs workspaces indépendants peuvent être maintenus, cependant un seul workspace peut être visualisé sous la forme de container à un instant donné, voir figure 3.8. Les folders ou dossiers permettent d'organiser hiérarchiquement l'information.

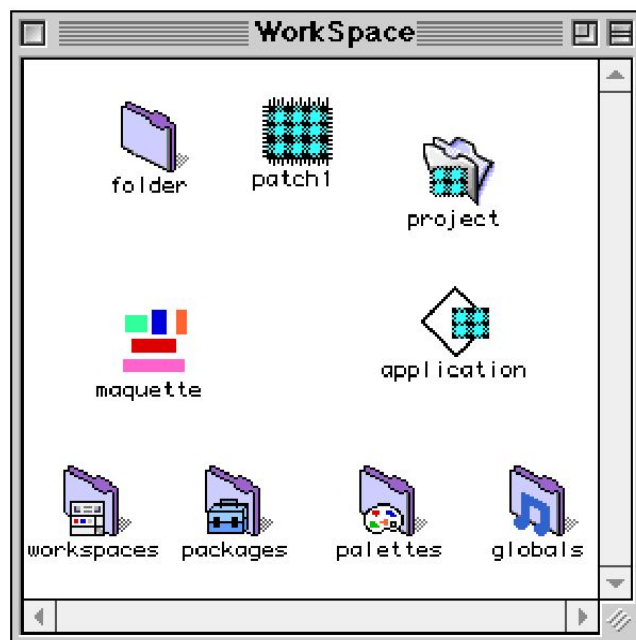


FIG. 3.8 – Un container de workspace.

## Les packages

Les packages sont des dossiers spéciaux qui archivent les classes et les fonctions génériques, à la manière du langage Java. Dans la figure 3.9, le package *music* est ouvert, ainsi que le sous-package *score*. En dessous de *score* apparaissent les classes disponibles dans ce package. La classe *chord*, ouverte, montre les champs de cette classe. Dans la seconde colonne apparaissent les fonctions génériques appartenant à *score*. Dans la dernière colonne, les méthodes appartenant à la fonction générique *concat*, avec leurs types d'entrées sous la forme de mini-icônes. Un autre éditeur existe pour les packages, permettant de visualiser l'arbre de hiérarchie des classes, voir figure 3.10.

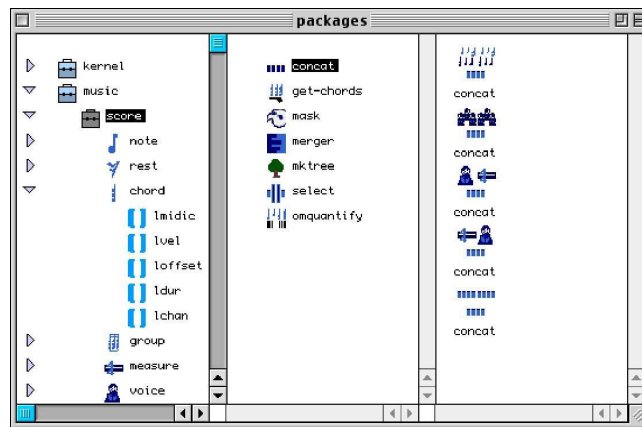


FIG. 3.9 – Un container de packages.

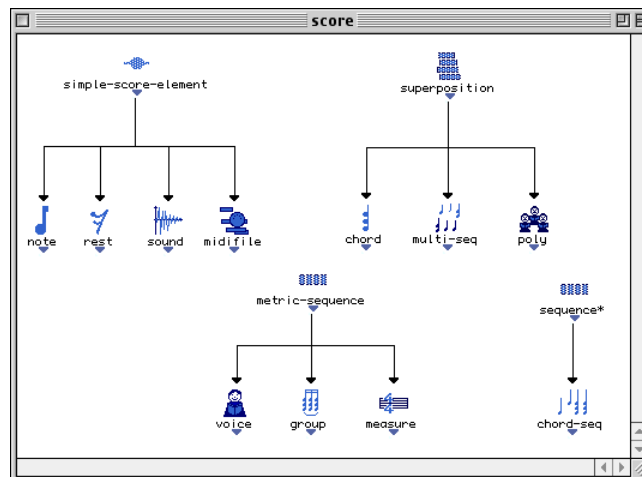


FIG. 3.10 – Arbre des classes musicales

L'un des packages prédéfinis est *music*, qui contient l'ensemble des classes et fonctions musicales d'OM : classes *note*, *silence*, *fichier midi*, *accord*, *suite d'accords*, *polyphonie*, etc. La figure ?? montre la hiérarchie des classes musicales. Un package *user* stocke les classes créées par l'utilisateur (qui peuvent hériter des autres classes OM).

## Classes

Les packages montrent des vues icôniques de classes. Ouvertes, celles ci ont un éditeur qui permet de manipuler les champs. Nous nous intéresserons plus particulièrement aux classes musicales, même si les principes sont les mêmes pour les autres classes.

Les classes musicales prédéfinies sont nombreuses, et leur architecture complexe bien qu'assez naturelle d'utilisation. Chaque classe musicale a un éditeur associé, qui permet de voir l'instance de cette classe sous forme de partition, de l'éditer et de l'écouter. Les plus couramment utilisées sont sans doute chord, chord-seq, voice. La classe chord est utilisée pour des accords, mais aussi par exemple pour une gamme ou une échelle. Elle n'a pas de dimension temporelle, et contient un ensemble de notes simultanées. On peut les voir soit sous la forme d'un accord, soit éclatées par ordre croissant ou décroissant par exemple. La figure 3.11, montre l'éditeur de la classe chord, et la figure 3.12 la liste des slots de chord.

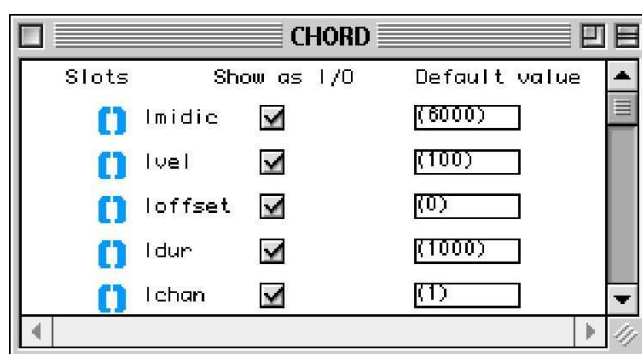


FIG. 3.11 – Editeur de la classe chord

La classe chord-seq est une suite d'accords, elle a une dimension temporelle (les dates d'attaque de chaque accords). La classe voice représente par exemple une mélodie, *ie* une suite de notes donnée avec un rythme formel.

## Sous-classage

Les nouvelles classes sont créées dans le package user. Nous y plaçons un alias de l'icône de classe chord vue précédemment. Les alias permettent de créer des liens invisible entre les différents packages (ici avec le package score) et de morceler ainsi les arbres d'héritage entre ces packages. Ensuite nous créons une nouvelle classe virchord et plaçons une flèche d'héritage issue de chord. Dans le container de virchord, nous créons un champ bpf, en glissant-déposant la classe de même nom (les bpf sont des fonctions linéaires par segment, associées à un éditeur graphique). Nous voudrions que, lors de l'initialisation d'une instance de virchord, le champ

- lmidic Une liste de midicents (hauteur des notes)
- lvel Une liste de vitesses midi
- loffset Une liste d'offsets temporels (appoggiatures)
- ldur Une liste de durées
- lchan Une liste de canaux midi

FIG. 3.12 – Les slots de la classe chord



FIG. 3.13 – Exemple d’héritage, virchord hérite de chord

bpf soit rempli avec une nouvelle instance de la classe bpf, dont les ordonnées seraient les hauteurs de l’accord. Nous voudrions de surcroît, toujours à l’initialisation, qu’une note grave (appelée fondamentale virtuelle) soit calculée et rajoutée aux notes données pour l’initialisation des champs. Il suffira pour cela de définir visuellement une fonction d’initialisation que nous ne détaillons pas ici.

Une fois la nouvelle classe définie, nous pouvons créer par glisser-déposer de cette classe sur une fenêtre de patch, une boîte factory (figure 3.14 en haut à gauche). Cette factory est en fait une boîte de création d’instances virchord. Elle possède tous les inlets (self + champs) et les outlets (self + champs) caractéristiques de la classe virchord. Sur l’inlet correspondant au champ lmidic est connectée une constante contenant une liste de hauteurs en midicents. A l’évaluation de la factory (par clic sur un des outlets), le mini-éditeur occupant le rectangle de la factory est mis à jour avec l’accord nouvellement créé. Cet accord est disponible sur la sortie self, et ses champs sur les autres sorties, pour de futures connections à d’autres boîtes. Un mode d’évaluation particulier (majuscule-clic) permet aussi de créer une instance qui se matérialise sur l’écran et se sépare de la factory, soit l’équivalent visuel d’une variable (boîte en dessous de la factory). A partir de cette instance, un éditeur musical peut être ouvert par double-clic (en dessous de l’instance), ou encore un éditeur objet (à droite). Ce dernier montre la structure stratifiée liée à l’héritage, chaque ensemble de champs étant rangé au niveau de la classe qui le définit. Les valeurs des champs étant des instances, celles-ci peuvent de nouveau être ouvertes par double-clic, éditées dans l’éditeur approprié, changées par glisser-déposer d’une instance venant d’une autre zone de l’environnement, ou encore exportées en les glissant-déposant dans une autre zone.

### Fonctions génériques et méthodes

La fonction générique prédéfinie OM+ additionne 2 objets de type nombre ou liste de nombre (effectuant dans ce cas une opération vectorielle). Son éditeur-container (Figure 3.15) montre la collection de méthodes correspondant aux combinaisons de ces types. Nous allons augmenter cette collection en lui ajoutant une méthode capable d’additionner deux accords virchord (Figure 3.16 à droite). L’icône de la classe virchord est glissée-déposée sur les représentants de classe constituant les entrées de la méthode (en haut à gauche, fenêtre de droite). Ces entrées maintenant typées présentent tous les champs de virchord sous forme d’outlet. Les champs lmidic des deux accords sont concaténés et triés en ordre de hauteurs ascendantes, puis envoyées dans l’entrée lmidic d’une factory virchord. Une nouvelle instance de cette classe est ainsi créée, dont les hauteurs sont l’union des hauteurs des accords d’entrée. Le container de la fonction générique OM+ est mis à jour avec la vue icônique de la nouvelle méthode, dont les types d’entrée sont indiqués par deux mini-icônes (en haut, fenêtre de gauche).

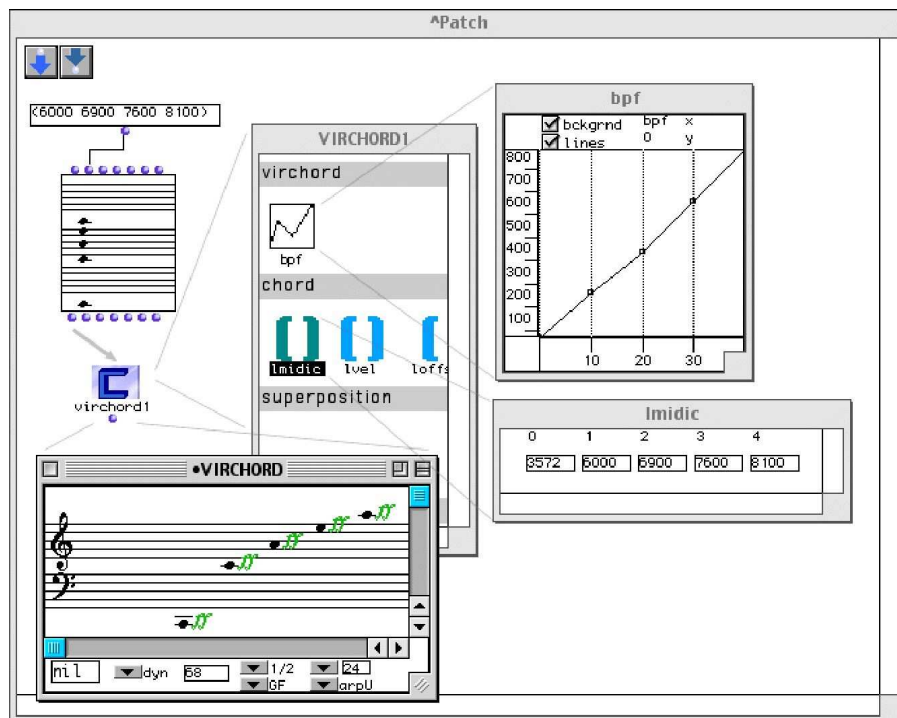


FIG. 3.14 – Création et édition d'une instance de virchord

La Figure 3.17 montre un patch dans lequel deux virchord sont " additionnés " grâce à la fonction générique OM+, qui dispatche dans ce cas vers la nouvelle méthode. Le résultat, une instance de virchord, est directement branché dans l'entrée self d'une factory de la même classe. Il est à noter que l'utilisation directe de l'entrée self inhibe les autres entrées. Cela signifie que l'on désire initialiser la nouvelle instance non pas par des valeurs de champs, mais par la copie d'une instance existante (mécanisme appelé prototype).

### Les éditeurs musicaux

Les figures suivantes montrent quelques uns des éditeurs musicaux d'OM. Ces éditeurs sont des containers particuliers qui, au lieu de montrer la collection des composants d'une classe (en l'occurrence une classe musicale telle que chord ou voice) interprètent ces composants et élaborent une représentation globale, en notation musicale par exemple. D'autres éditeurs du même type peuvent ainsi exhiber une représentation piano-roll ou forme d'onde. La Figure 3.18 montre un éditeur de la classe voice, à partir duquel les accords peuvent être ouverts dans des éditeurs de la classe chord. La Figure 3.19 montre un éditeur de la classe poly.

## 3.5 Un exemple musical

L'approche fonctionnelle est peut être celle qui a été la plus utilisée par les compositeurs. En effet, ce style de programmation donne une réponse à la nécessité de disposer d'objets multidimensionnels et d'appliquer sur eux des opérations qui les modifient ou produisent des nouveaux objets. Dans ce sens, OM peut se concevoir comme une boîte à outils servant à faire des opérations musicales. De grand importance pour le compositeur est la classification des dites

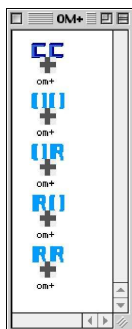


FIG. 3.15 – Les méthodes de la fonction générique om+

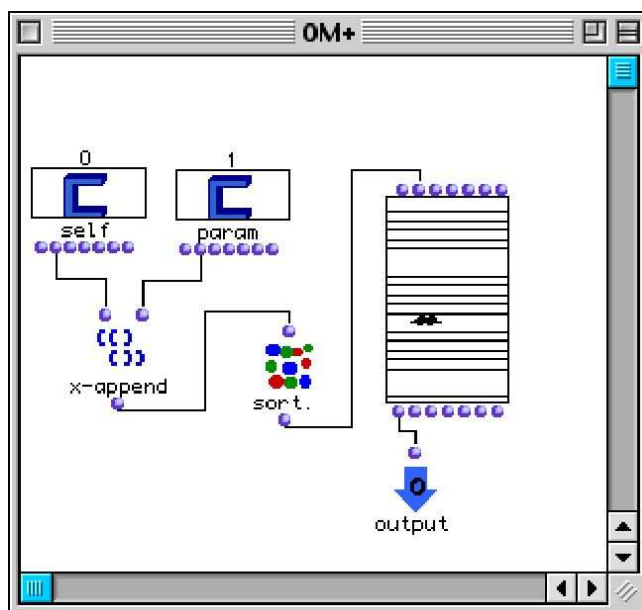


FIG. 3.16 – Création d'un méthode pour om+

fonctions par rapport au type de connaissance musical qu'elles représentent. Nous trouvons dans OM des fonctions orientées intervalles ; des fonctions de traitement harmonique ; d'interpolation d'accords, de motifs et d'objets en général. OM a donné de la force à la démarche fonctionnelle en musique et l'on a très vite vu que ce paradigme pouvait être appliqué au domaine de la synthèse. Tel est le cas dans " L'Amour de loin ", le premier opéra de la compositrice finlandaise Kaija Saariaho [17].

La Figure 3.20 résume le processus de transformation que subissent les matériaux compositionnels dans cette pièce.

Le matériau sonore à la base est composé par des échantillons des voix parlées extraites du livret et des sons instrumentaux.

Ces échantillons sont analysés en utilisant la technique de modèles de résonance. Le résultat de cet analyse est une liste de fréquences, amplitudes et largeurs de bande qui donnent beaucoup d'information sur le timbre des sons. Pour plus d'information sur les modèles de résonances voir [85].

L'étape de filtrage consiste en le croisement d'une partie électronique issue d'une structure

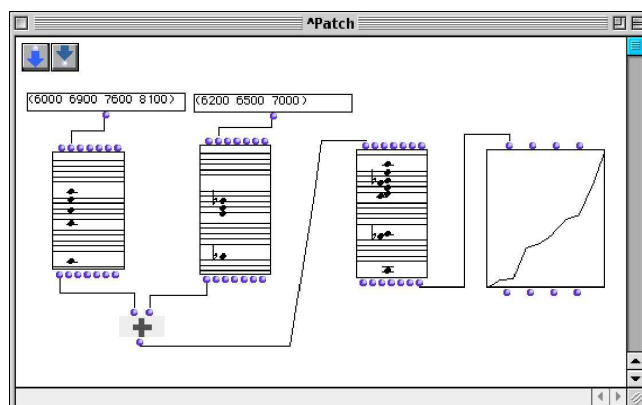


FIG. 3.17 – Addition de deux accords.

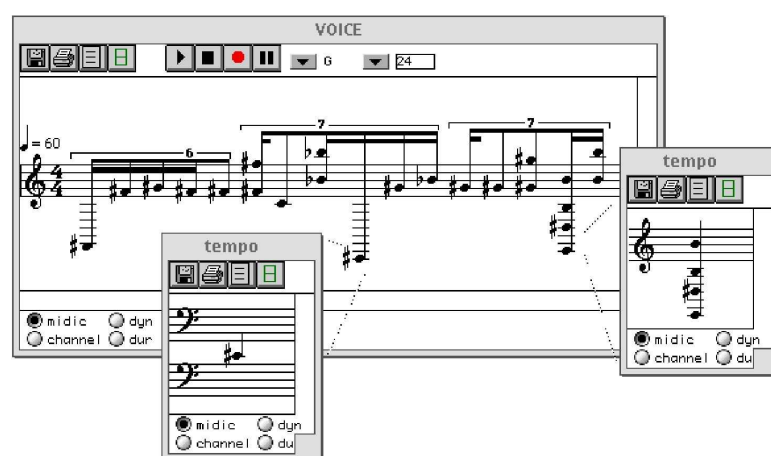


FIG. 3.18 – Editeur musical de la classe voice

harmonique à priori avec un timbre issu de l'analyse par modèles de résonances. La partie électronique est d'abord écrite sous forme de partition. Les sons électroniques reposent sur des structures harmoniques qui sont propres à chaque personnage de l'opéra. La Figure 3.21 présente les accords utilisés dans l'élaboration des sons électroniques pour le personnage Jaufré.

Le patch OpenMusic, Figure 3.22, illustre l'extraction de modèle de résonance pour un échantillon de piano sur la note fa  $\sharp$  0.

Pour chacun des accords associés aux personnages, de nombreux filtrages ont été réalisés. A chacune des fréquences des notes de l'accord d'un personnage sont associées, après un calcul d'interpolation spectrale, l'amplitude et la largeur de bande correspondantes du modèle instrumental. Ce patch OpenMusic effectue le filtrage. Pour toute l'oeuvre, environ trois cents filtres ont été générés. En considérant le nombre d'accords et d'échantillons instrumentaux analysés au regard des trois filtres utilisés lors de la synthèse, il est aisé de voir que le nombre de combinaisons des choix possibles est élevé.

Finalement, le mixage des sons réalisés précédemment est effectué en temps réel à l'aide du **spatialisateur**<sup>1</sup> développé à l'Ircam.

<sup>1</sup>Voir spatialisation dans le glossaire en annexe

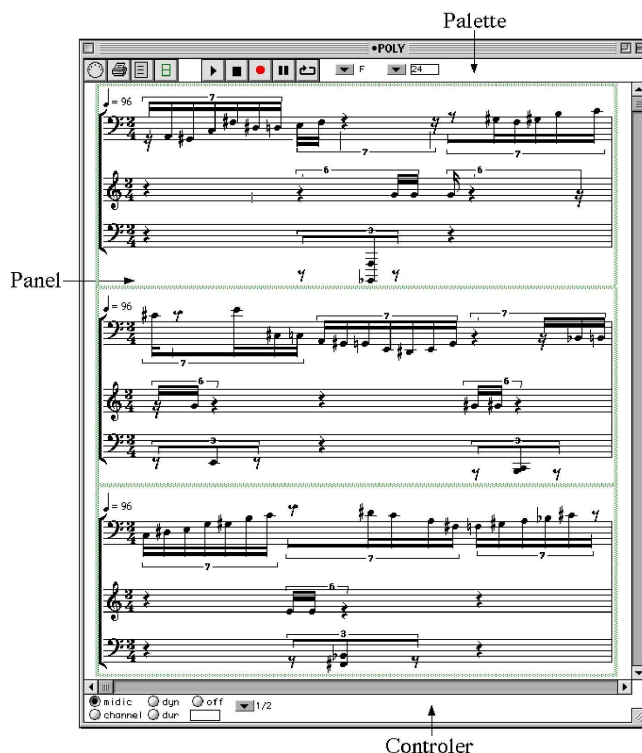


FIG. 3.19 – Editeur musical de la classe poly

### 3.6 Modèle de distribution et méthodologie

OpenMusic est vendu selon un modèle particulier, dans le cadre du Forum IRCAM. Le Forum est un ensemble de logiciels et services proposés aux utilisateurs, en échange d'une inscription valable un an. L'inscription au Forum couvre un CD avec l'ensemble des logiciels développés par l'IRCAM (Modalys pour la synthèse par modèles physiques, jMax pour le temps réel, OpenMusic en composition assistée par ordinateur, etc), avec une assistance technique. En outre, des listes de diffusions pour certains logiciels, dont OpenMusic <sup>2</sup>, permettent de dialoguer avec les autres utilisateurs et si nécessaire les développeurs. Il est par ailleurs demandé aux utilisateurs de donner à l'IRCAM leurs contributions, ce dont sont issues de nombreuses bibliothèques d'OpenMusic. Ce modèle de distribution est donc très différent d'une simple vente de logiciels. Il est adapté à des utilisateurs, musiciens, qui ne sont pas toujours familiers avec l'informatique, ou qui peuvent avoir des questions nouvelles puisqu'ils utilisent a priori souvent des structures de données qui leur sont propres. L'une des conséquences pour les chercheurs des équipes scientifiques est d'avoir un lien assez étroit avec certains utilisateurs, voire de développer des choses au cas par cas, pour nous avec Fabien Lévy, Peter Klanac, Georges Bloch, par exemple.

Autre source de feedback, le cursus IRCAM. Il s'agit d'une formation d'un an proposée à une douzaine de jeunes compositeurs avec notamment des cours d'OpenMusic. A l'issue de leur année, ils doivent écrire une pièce pour un instrument et système électronique. Ces pièces sont jouées en deux concerts à la fin de l'année. Ce cadre nous a permis de travailler avec plusieurs jeunes compositeurs sur des problèmes de contraintes : Mauro Lanza, Geoffroy Drouin, Arturo

<sup>2</sup>openmusic@ircam.fr

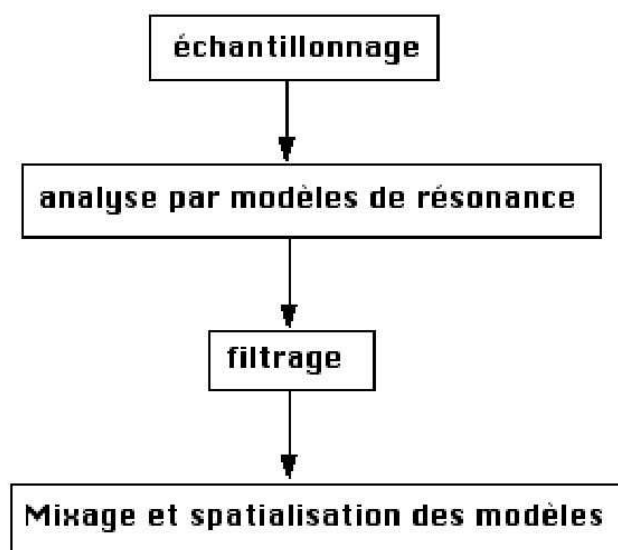


FIG. 3.20 – Transformation des matériaux dans L'Amour de loin

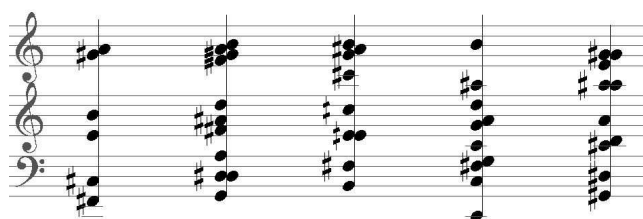


FIG. 3.21 – Accords de Jaufré

Fuentes, par exemple.

On voit ici qu'à la différence d'un laboratoire d'informatique, l'IRCAM mixe autant que possible les musiciens (utilisateurs) et les scientifiques (développeurs). Ce modèle de développement offrait des possibilités d'interaction avec les utilisateurs. Cela nous a permis d'adopter une approche pragmatique, en commençant par collecter plusieurs problèmes musicaux de contraintes, en travaillant avec les compositeurs liés à l'IRCAM, en général déjà habitués à OpenMusic, mais pas nécessairement. Il est important de souligner que ces problèmes sont donc des vrais problèmes musicaux, posés par de vrais utilisateurs-compositeurs, et non des problèmes académiques. Cette méthodologie nous a semblé indispensable pour éviter autant que possible l'effet "un compositeur, un système".

### 3.7 Les utilisateurs, musiciens

Même si OpenMusic est un langage de programmation "comme un autre", il s'adresse d'abord aux compositeurs contemporains. On a ici un modèle d'utilisation très particulier pour deux raisons. D'abord, les utilisateurs ne sont pas des informaticiens, et il n'est pas question de leur pré-supposer des connaissances ni informatiques, ni scientifiques. Il faudra donc en tenir compte dans la conception du système, qui doit être très *user friendly*, à la manière d'OpenMusic.

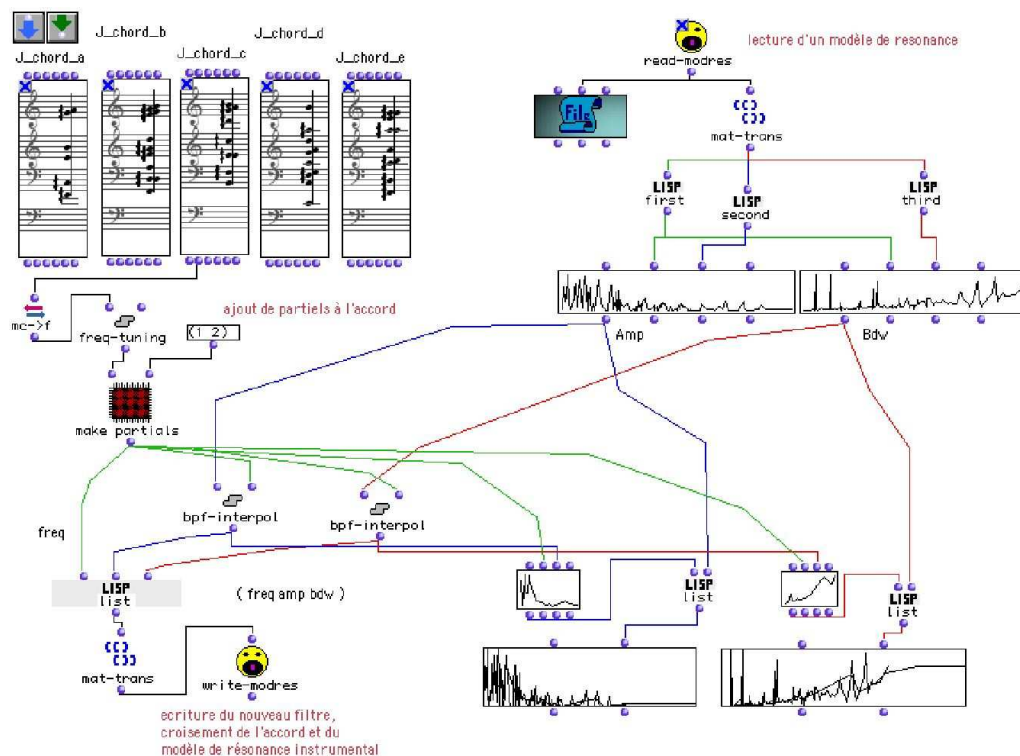


FIG. 3.22 – Analyse par modèles de résonances d'un son de piano

Ensuite, la nature même de ce que l'on cherche à produire pose problème. Avant de fournir un système de contraintes adapté à la CAO, il convient de cerner un peu plus précisément ce que ces deux points imposent au système. Pour cela, nous avons choisi une approche pragmatique, en travaillant d'abord avec des compositeurs sur des CSPs résolus au cas par cas (voir chapitre 3). Cela nous a permis sinon de comprendre la nature exacte de la CAO, au moins de décider comment y intégrer la programmation par contraintes.

A la manière de la Conception Assistée par Ordinateurs pour les architectes ou designers, on considérera ici la CAO comme un ensemble d'outils fournis au compositeur dans le but d'écrire une oeuvre artistique. On s'adresse uniquement aux compositeurs ayant un minimum d'expérience dans OpenMusic, qui savent écrire un patch OM, l'évaluer, etc. Il faut ensuite définir ce que veulent les utilisateurs. Ils utiliseront le futur système pour écrire une oeuvre musicale. Sait-on exactement en quoi cela consiste? Non (ce sont des questions qui font appel à des domaines de recherche dans lequel nous sommes incompetents). Mais, du travail préliminaire, on a constaté que : 1) le compositeur sait rarement lui même au départ ce qu'il attend 2) mais ce n'est pas parce que le but est flou qu'il s'agit de produire "n'importe quoi" : les compositeurs ont un haut degré d'exigence vis-à-vis de ce qu'ils produisent (et, en amont, que le logiciel leur fournit). Le problème est que ce degré d'exigence s'exprime partiellement avec des règles, des calculs, des formalisations quelconques, et partiellement d'une manière qui nous échappe (il faut que ça sonne bien / que ça respecte les structures ou idées musicales / que ça soit nouveau / que ça surprenne, parfois / *ad lib*). Et il est probablement nécessaire que ça nous échappe. Il faut donc respecter un certain degré d'ambiguïté dans l'utilisation du logiciel, fournir des outils ouverts, avec lesquels il doit être facile de faire des essais, de jouer, de tester, de reprogrammer.

Autre point, nous avons souvent constaté en travaillant avec les compositeurs qu'ils ne sont en général pas intéressés par une boîte noire opaque qui leur rende des solutions (quel que soit le niveau technique de la boîte). Ce sont eux qui signent l'oeuvre musicale, et ils souhaitent souvent avoir un contrôle de l'ensemble de ce qu'ils y mettent. En pratique, il faut souvent passer un peu de temps à leur expliquer comment fonctionne le logiciel, pourquoi les réponses qu'ils obtiennent sont ainsi, etc. Cela signifie que l'on doit fournir un système assez transparent, dont les principes doivent être vulgarisables.

En résumé, nous avons à fournir un système de programmation par contraintes dont les spécificités sont assez différentes des habitudes de développement logiciel. Les critères que nous avons retenus sont :

- le système doit être très *user friendly*, avec notamment une programmation visuelle
- le procédé de calcul doit être transparent
- le système doit être ouvert, facilement reprogrammable par exemple
- on doit le voir comme un système non pas qui apporte une solution à un problème de contraintes, mais qui suggère des solutions à la demande

### 3.8 Conclusion sur OpenMusic

OpenMusic est désormais enseigné dans plusieurs universités et conservatoires et utilisé par un nombre croissant de compositeurs et de musicologues. L'utilisation extensive d'OpenMusic par des compositeurs de musique contemporaine a indéniablement eu pour effet de relativiser la complexité de certaines techniques en vogue depuis la révolution dodécaphonique et ses suites. Lorsqu'un ordinateur peut calculer très rapidement toutes les variantes combinatoires issues d'un formalisme donné, par exemple le formalisme sériel, la valeur musicologique ne peut plus venir simplement de ce principe formel, mais de son adéquation réelle à la perception. Corrélativement, l'ordinateur permet d'explorer des champs expérimentaux d'une complexité réelle et qui étaient inaccessibles auparavant, comme par exemple le champ du timbre dans sa représentation spectrale. Parallèlement, OpenMusic a favorisé la démarche expérimentale qui permet de soumettre un très grand nombre d'instances musicales issues d'un formalisme au test de la musicalité. Plus encore, elle autorise l'expérimentation sur les formalismes eux même qui peuvent être testés et en retour modifiés ou abandonnés s'ils ne remplissent pas leurs promesses. Aux notions d'unicité de l'oeuvre et de son créateur, elle tend à substituer l'idée du modèle d'oeuvre et de la coopération entre créateurs et scientifiques. C'est peut être là que réside son aspect le plus original, et donc le plus polémique. Enfin, OpenMusic propose un renouvellement de l'idée de notation qui devient dynamique, incluant les mécanismes génétiques de l'oeuvre. La partition constitue d'une certaine manière sa propre analyse, ce qui est déjà le germe d'un bouleversement des habitudes musicologiques. Nous avons essayé avec OpenMusic de construire un langage qui encapsule de manière consubstantielle la notion de notation, à savoir notation du résultat (une partition musicale) mais aussi notation du processus aboutissant à ce résultat (programme visuel).



# Chapitre 4

## Programmation par contraintes

La Programmation par Contraintes est un paradigme de programmation récent, qui connaît depuis quelques années un succès certain, tant au point de vue théorique et académique qu'au point de vue pratique et industriel. Elle allie en effet la simplicité d'écriture de programmes et la déclarativité d'un langage de haut-niveau à la puissance et à l'efficacité du calcul sur des domaines spécifiques à l'aide d'outils particuliers et optimisés. La notion de contrainte est tout simplement celle d'une relation logique portant sur un ensemble de variables (inconnues d'un problème) et les liant ainsi entre elles. Une contrainte donne ainsi une information partielle sur les valeurs possibles que les variables peuvent prendre, contraignant donc cet espace de valeurs. L'idée de base de la programmation par contraintes est d'intégrer le concept de contrainte dans un langage de programmation de haut niveau pour pouvoir programmer, calculer et raisonner avec cette notion d'information partielle, tout en assurant la cohérence de l'ensemble des contraintes et en réduisant le plus possible les domaines de valeur des variables, c'est à dire l'aspect combinatoire. De nombreuses applications peuvent s'exprimer simplement et efficacement dans le cadre de la programmation par contraintes : problèmes combinatoires, ordonnancement, emplois du temps, simulation, diagnostic de panne, analyse financière, aide à la décision, etc. La programmation par contraintes a aujourd'hui largement dépassé le cadre strictement académique et il en existe de très nombreuses applications pratiques, l'allocation des trains aux quais gare Saint-Lazare à Paris, l'allocation des avions aux portes à l'aéroport de Singapour, l'emploi du temps de certains personnels d'Air France, la planification de tâches de fabrication chez Chrysler, les emplois du temps des infirmières d'un hôpital de Montréal ou pour l'Hospital Authority de Hong Kong qui gère une quarantaine d'hôpitaux publics, l'aide à la conception de pièces mécaniques chez Dassault Aviation, pour n'en citer que quelques exemples.



# Chapitre 5

## Problème de Satisfaction de Contraintes (CSP)

Pour une présentation détaillée de la programmation par contraintes, voir les deux livres [68] et [103].

Toutes ces applications se basent sur une forme normalisée, le format CSP, pour Constraint Satisfaction Problem. C'est un format de représentation pour les problèmes de contraintes ou fortement combinatoires.

### 5.1 Définition

Un problème de satisfaction de contraintes, ou CSP, est donné par :

- un ensemble de *variables*, qui sera noté en général  $V_1 \dots V_n$ . On peut les voir comme l'équivalent de l'inconnue en mathématiques.
- un ensemble de *domaines*, autant que de variables, généralement notés  $D_1 \dots D_n$ . Le domaine  $D_i$  est l'ensemble des valeurs possibles pour la variable  $V_i$ . D'un point de vue théorique et parfois même en pratique, la programmation par contraintes peut traiter des domaines quelconques, par exemple continus (voir pour un tour d'horizon sur les domaines continus, Interval Constraints [18]) ou symboliques. Cependant, c'est inutile dans notre cas et nous aurons toujours des domaines discrets, en général un ensemble d'entiers.
- un ensemble de contraintes,  $C_1 \dots C_p$ . A priori, la notion de contraintes est très large : une contrainte est simplement un prédicat sur les variables, ou d'un autre point de vue c'est une relation qui restreint les valeurs autorisées pour les variables. Mais il s'agit de programmation, et ces contraintes seront évidemment définies dans un langage précis.

Une variable  $V_i$  peut être *instanciée*, ce qui signifie qu'elle a une valeur associée, généralement notée en minuscule  $v_i$ . Une solution au problème est une instantiation qui vérifie les contraintes : chaque variable  $V_i$  a une valeur  $v_i$  et on a  $C_1(v_1 \dots v_n) \cap \dots \cap C_p(v_1 \dots v_n)$ . On appelle instance du problème un cas particulier, quand le nombres de variables, les domaines, et si nécessaire tous les paramètres des contraintes ont des valeurs précises.

Déterminer si un CSP a une solution est dans le cas général un problème de la classe NP<sup>1</sup>. Ceci est lié à l'aspect combinatoire du format CSP. On peut voir un CSP comme décrivant

---

<sup>1</sup>La classe NP, pour Non Polynômial, contient les problèmes de taille  $n$ , tels que l'on ne peut pas en trouver de solution plus vite que  $e^n$ , ie le temps de calcul d'un algorithme de résolution évolue au plus vite comme l'exponentielle de la taille du problème. Elle est opposée à la classe P, ensemble des problèmes pour lesquels on

un espace des possibles, lequel est au départ de taille exponentielle : le domaine de recherche croît en exponentielle du nombre de variable. Les contraintes donnent un moyen de définir une certaine information sur un espace par nature fortement combinatoire.

## 5.2 Modélisation et problème des solutions approchées

La modélisation d'un problème de contraintes est rarement détaillée dans les travaux de recherche. La grande majorité des articles sur le sujet traitent de la résolution de problèmes aujourd'hui bien connus, et bien modélisés. Quelques exceptions : [31] utilise un double modèle pour le problème des nurses <sup>2</sup>. Des contraintes, "channel constraints", assurent la cohérence des deux modèles. L'utilisation de deux modèles en parallèle permet de filtrer plus efficacement les différents types de contraintes. Un autre exemple où la recherche se concentre sur la modélisation elle-même concerne les cassures de symétrie. Un CSP qui présente des symétries est très ennuyeux à résoudre : on risque de passer beaucoup de temps à retrouver une solution identique à une solution déjà trouvée, à une symétrie près. Un des exemples le plus compliqué est le social golfer problem <sup>3</sup>. Ce problème a des symétries dans tous les sens, plus par permutations. Il est donc crucial d'en faire une modélisation telle que le maximum de symétries soient éliminées, mais sans perdre de solutions (voir par exemple les travaux de Harvey, [55], ou [94] pour un modèle avec des contraintes redondantes en local search). Récemment, deux workshops ont été organisés dans la conférence CP <sup>4</sup>, à propos notamment de langages de contraintes et d'outils permettant d'automatiser la reformulation des problèmes.

La première difficulté est dans la formulation même du problème, transformation d'un problème réel, en une formulation façon mathématique, avec toutes les difficultés que l'on sait (l'utilisateur oubliant de préciser certaines choses qui lui semblent évidentes par exemple). Ce sont là des problèmes de communication bien connus que nous aurons systématiquement avec les compositeurs, mais qui ne nous intéressent pas immédiatement.

En revanche, pour un même problème théorique, on peut écrire plusieurs modélisations, dont les caractéristiques influencent grandement la vitesse de résolution, et surtout dans notre cas la définition de ce qu'est une "solution approchée". Ainsi les  $n$  reines. C'est un problème académique classique. Il s'agit de placer  $n$  reines sur un échiquier de taille  $n \times n$ , de telle manière que aucune reine ne puisse manger une de ses consœurs en un coup. Aux échecs, une reine peut se déplacer en ligne, en colonne et en diagonale.

Première modélisation, on prend comme variables des booléens  $B_{i,j}$ . Ici  $i$  est à prendre comme un indice de ligne et  $j$  de colonne.  $B_{i,j}$  vaut vrai si une reine est sur la case  $(i, j)$  de l'échiquier, faux sinon. Les contraintes sont :

- $C_1$  Il faut exactement  $n$  reines sur l'échiquier, ce qui s'écrit  $\sum_{i,j} B_{i,j} = n$

---

peut donner une réponse en temps polynômial.

<sup>2</sup>"nurse rostering problem", il s'agit de trouver un emploi du temps pour les infirmières d'un hôpital, chaque infirmière pouvant travailler dans certaines plages horaires par jour. Les contraintes portent sur des impossibilités verticales, de type "il faut un service minimum dans l'hôpital,  $x$  nurses présentes le matin, la nuit, etc", et horizontales, de type "une infirmière qui a effectué une garde de nuit ne peut pas enchaîner le jour suivant sur une matinée".

<sup>3</sup>Un club de golf accueille  $n$  golfers, qui jouent ensemble par groupes de  $m$ . Le gérant du club souhaite trouver un planning sur  $p$  semaines, tel que chaque joueur joue une fois par semaine, et que deux joueurs différents ne rejouent jamais ensemble pendant les  $p$  semaines.

<sup>4</sup>International Workshop on Modelling and Reformulating Constraint Satisfaction Problems Towards Systematisation and Automation, conférences CP02 et CP03, <http://www-users.cs.york.ac.uk/frisch/Reformulation/02/> et <http://www-users.cs.york.ac.uk/frisch/Reformulation/03/>

- $C_2$  On ne peut pas avoir deux reines sur la même ligne, ce qui donne  $n$  contraintes, une par ligne (d'indice  $i$ ), qui s'écrivent  $\sum_j B_{i,j} \leq 1$
- $C_3$  On ne peut pas avoir deux reines sur la même colonne, ce qui donne  $n$  contraintes, une par colonne (d'indice  $j$ ), qui s'écrivent  $\sum_i B_{i,j} \leq 1$
- $C_4$  On ne peut pas avoir deux reines sur la même diagonale montante, ce qui donne  $2n - 1$  contraintes,  $\sum_k B_{i+k,j+k} \leq 1$
- $C_5$  On ne peut pas avoir deux reines sur la même diagonale descendante, ce qui donne  $2n - 1$  contraintes,  $\sum_i B_{i+k,j-k} \leq 1$

Avec cette modélisation, on a  $n^2$  variables, des domaines de taille 2, et  $1 + n + n + (2n - 1) + (2n - 1) = 6n - 1$  contraintes. L'espace de recherche est de taille le produit cartésien des domaines, soit ici  $2^{n^2}$ , ce qui est considérable. Même  $n = 6$  donne un domaine de recherche de taille de l'ordre de  $10^{10}$ , et on doit à chaque fois calculer les 35 contraintes.

Faire mieux signifie réduire le nombre de variables, ce qui se fait assez naturellement ici en prenant  $n$  variables  $Q_1 \dots Q_n$  de domaines  $\{1 \dots n\}$ , où  $Q_i$  représente la position verticale de la  $i$ -ième reine sur la  $i$ -ième colonne. Avec cette modélisation, il y a toujours exactement une reine par colonne ce qui enlève les contraintes  $C_1$  et  $C_3$ . Les autres contraintes s'écrivent :

- $C_2$  *alldiff*( $Q_1 \dots Q_n$ ) (pas deux reines sur la même ligne)
- $C_4$  *alldiff*( $Q_1 - 1, \dots, Q_n - n$ ) (pas deux reines sur une même diagonale montante)
- $C_5$  *alldiff*( $Q_1 + 1, \dots, Q_n + n$ ) (pas deux reines sur une même diagonale descendante)

L'espace de recherche est maintenant de taille  $n^n$ , ce qui représente un gain considérable, et on n'a plus que trois contraintes à traiter. Dernière possibilité, passer à un CSP sur les permutations (dans le cas où les domaines de toutes les variables sont égaux, soit  $D$ , le domaine de recherche est non plus  $D_1 \times \dots \times D_n$ , mais l'ensemble des permutations des valeurs de  $D$ , de taille  $n!$ ). On cherche une permutation de  $1, 2, \dots, n$ , ce qui force les reines à rester sur des lignes différentes et élimine la contrainte  $C_2$ . Les autres contraintes s'écrivent de la même manière.

Cet exemple illustre à quel point un même problème peut avoir des modélisations différentes. On passe ici d'un problème de taille  $2^{n^2}$  à  $n!$ , par exemple pour  $n = 6$  de  $10^{10}$  à 720, et avec un gain semblable sur les calculs de contraintes. Cependant, ce gain n'est possible qu'en transformant des contraintes dans la modélisation. Ces transformations interviennent de trois manières. Premièrement, le choix même des variables permet d'éliminer des contraintes "structurelles", ici en passant des booléens aux positions entières (une seule reine par colonne disparaît des contraintes). Deuxièmement, les contraintes unaires sont évidemment passées en réductions de domaines, ce que tout le monde fait intuitivement (node-consistency, voir ci-dessous - ce n'est pas le cas ici). Troisièmement, les *alldiff*, contraintes souvent difficiles à traiter, peuvent avantageusement se transformer en permutations. Mais la contrepartie est que ces contraintes sont gommées du problèmes, et qu'on ne peut plus choisir de les abandonner ou de les rendre "soft".

Nous nous trouverons souvent face à un gros problème de modélisation, dû à la recherche de solution approchées. En effet, il peut arriver qu'un utilisateur pose par exemple un problème insoluble, *ie* soit dont on sait démontrer qu'il n'a aucune instantiation solution, soit très long à résoudre et dont on se contenterait bien d'une solution "approximative". Dans ce cas, on a besoin de trouver des solutions approchées. Le sens du mot approché est, en pratique, à définir par l'utilisateur, ce qui est un premier problème. Si l'on suppose que l'utilisateur a choisi par exemple des contraintes "dures" et des contraintes "molles" (voir soft constraints, plus bas), une solution approchée respecte les contraintes dures et "tente de" respecter les contraintes molles. Sinon, une solution approchée peut-être simplement une solution qui respecte "le plus possible" les contraintes selon l'option choisie par l'utilisateur (soit minimiser une certaine erreur, à définir, sur les contraintes, soit minimiser le nombre de contraintes violées, ou toute

combinaison). Quoi qu'il en soit, la notion de contrainte approchée sera à définir proprement.

Or l'étape de modélisation comprend souvent la réduction intuitive de contraintes par l'utilisateur. Reprenons l'exemple des  $n$ -reines : en modélisant chaque reine par un entier, on a automatiquement réduit la contrainte "une reine par ligne". En modélisant le problème avec les permutations comme domaine, on a automatiquement réduit la contrainte "une reine par colonne". D'une manière générale, toutes les contraintes unaires sont automatiquement réduites par n'importe quel utilisateur, et certaines contraintes  $n$  aires comme les alldiff, dès que le nombre de variable sur lesquelles porte le alldiff égale la taille du domaine.

Pour les  $n$  reines, qui n'est pas surcontraint, cela n'a aucune incidence sur la recherche de solutions exactes et ces réductions sont évidemment utiles. Mais si l'on ajoutait une contrainte, et que le problème devienne surcontraint, se poserait le problème du choix des contraintes à rendre "molles", ou de l'option choisie par l'utilisateur pour les solutions approchées. Il faudra donc bien tenir compte de ceci, la recherche de solutions approchées dépend étroitement de la modélisation, puisque deux modélisations équivalentes en solutions exactes ne le sont pas en solutions approchées.

## 5.3 Modélisation en contraintes binaires

Au sujet des problèmes de modélisation, il faut mentionner le cas des CSPs binaires. Pour des raisons qui seront décrites ci-dessous, un CSP dont les contraintes sont binaires (dans laquelle n'apparaissent que deux variables) permet de faire des réductions de domaines. D'un point de vue théorique, n'importe quel CSP peut se traduire de manière équivalente en un CSP binaire (avec exactement les mêmes solutions). Prenons l'exemple d'une contrainte  $X = Y + Z$ . On y ajoute une variable  $U$  définie comme un codage de  $X$ ,  $Y$  et  $Z$ , de telle manière que le domaine de  $U$  vérifie la contrainte  $X = Y + Z$ . Le domaine de  $U$  est au départ le produit cartésien des domaines de  $X$ ,  $Y$  et  $Z$ , moins les valeurs qui ne respectent pas l'égalité. De cette manière la contrainte disparaît, au profit de contraintes permettant de lier  $X$  à  $U$ ,  $Y$  à  $U$  et  $Z$  à  $U$ .

C'est un point important théoriquement, notamment pour les algorithmes de consistance. Cependant, en pratique, c'est rarement intéressant. Pour une étude plus détaillée, voir [12].

# Chapitre 6

## Quelques familles de CSPs

Ici sont décrits brièvement certains CSPs classiques, ou contraintes, dans la mesure où ils nous intéressent dans la suite. Cela ne prétend certainement pas à un catalogue exhaustif de CSPs<sup>1</sup>.

### 6.1 SAT

SAT est sans doute un des problèmes les plus connus, testés et résolus. On définit un problème SAT à partir de clauses logiques, définies comme des disjonctions de variables, des "OU" logiques. Le problème est donné comme la conjonction de  $m$  clauses logiques  $\{C_1 \dots C_m\}$  qui portent sur  $n$  variables booléennes  $\{x_1 \dots x_n\}$ , ce qui s'écrit

$$C_1 \wedge \dots \wedge C_m$$

Dans les clauses, le "NON" logique est bien sûr autorisé, et les variables peuvent apparaître sous la forme positive  $x_i$  ou négative  $\neg x_i$ . Le problème SAT consiste à déterminer si une telle formule logique est satisfiable ou non, et si oui, à donner une affectation de valeurs aux variables pour que la formule soit vraie. (Remarque : n'importe quelle formule logique de la logique propositionnelle peut s'écrire sous la forme d'une formule SAT, qui est sa forme normale conjonctive)

Les problèmes de la famille SAT sont souvent utilisés dans les formes  $k$ -SAT, pour  $k$  un entier, avec comme restriction que les clauses contiennent exactement  $k$  variables. Pour  $k = 2$ , le problème est soluble en temps linéaire. Pour  $k \geq 3$ , le problème est NP-complet. On peut montrer que n'importe quel problème SAT peut se réduire à un problème 3-SAT. Enfin, un problème SAT est plus ou moins "facile" à résoudre selon le nombre de clauses et de variables (pour plus de détails sur SAT, voir par exemple les travaux de Selman [87] ou [109], ou bien les "Ten Challenges Redux" de Kautz présenté à CP03<sup>2</sup>, catalogue de dix questions ouvertes sur le sujet). SAT est donc un problème à la fois élégant dans sa formulation, profondément lié à l'histoire de la logique et de l'informatique, simple à comprendre et cependant extrêmement général puisqu'il s'agit de la satisfiabilité de formules logiques, ce qui explique probablement sa popularité. De très nombreux problèmes ont été réduits à SAT, et il existe un grand nombre d'instances répertoriées, dont certaines viennent de CSPs comme le coloriage de graphes, d'autres sont aléatoires (on distingue les instances structurées, des instances non structurées générées aléatoirement).

---

<sup>1</sup>Il existe par exemple une bibliothèque de CSPs, CSPLib, à l'adresse <http://4c.ucc.ie/tw/csplib/>.

<sup>2</sup><http://www.cs.washington.edu/homes/kautz/talks/index.html>

## 6.2 Tournées de véhicules

Le Travelling Salesman est un voyageur de commerce. Partant d'une ville fixée, il doit effectuer une tournée commerciale dans d'autres villes un peu partout en France. Le problème est d'optimiser son trajet de sorte qu'il parcourre le moins de chemin possible. C'est un problème d'optimisation extrêmement classique. Il appartient à la classe plus générale des problèmes de tournée de véhicules.

Le problème général est donné avec des camions qui partent d'entrepôts pour aller livrer des clients. Il s'agit d'optimiser la longueur de leur trajet. On peut en plus ajouter des fenêtres temporelles pour les livraisons (les clients sont là seulement dans certaines plages temporelles).

## 6.3 Allocation de fréquences

Il s'agit de déterminer les fréquences d'émission d'un ensemble de  $n$  stations radio. Pour éviter des interférences, deux radios ne peuvent pas émettre sur la même fréquence, et elles ne peuvent pas non plus émettre sur des fréquences dont la différence est déjà présente ailleurs. Formellement, on note  $f_i$  la fréquence de la  $i$ -ième radio, qui sont les variables du CSP. Les domaines sont *a priori* entiers, on peut les borner grossièrement si nécessaire. Les contraintes s'écrivent

- $C_1$  alldifferent ( $f_1 \dots f_n$ )
- $C_2$  alldifferent ( $f_i - f_j, i \neq j$ )

La première contrainte pourrait laisser espérer une modélisation en Permut-CSP, mais la taille des domaines l'interdit (il faudrait pouvoir réduire le domaine à  $\{1 \dots n\}$ , auquel cas le problème n'a pas de solutions ce qu'on voit immédiatement par cardinalité de l'ensemble des différences possibles). Un premier problème est de résoudre ce CSP. Un deuxième problème est de trouver le plus petit entier  $p$  tel qu'il existe une solution dans le domaine  $\{1 \dots p\}$ . Il faut au nombre minimum autant de différences qu'il y a de couples dans  $\{1 \dots n\}$ , soit  $C_n^2 = n(n-1)/2$  (ou d'une autre manière, c'est  $\sum_{i=1}^n \sum_{j \neq i} 1$ ).

## 6.4 Problèmes de sac à dos (knapsack problem)

Il s'agit de remplir un sac à dos de volume fixé  $V$ , avec des objets comestibles. L'objet  $i$  a un volume  $v_i$ , et représente une ration d'énergie  $e_i$ . Le problème est de maximiser l'énergie total contenue dans le sac, sans dépasser son volume. Ce problème connaît de nombreuses variantes, multi-critères notamment (on ajoute un poids à chaque objet, à minimiser), ou bien avec des énergies négatives.

Ce problème utilise des contraintes dites de capacité : il s'agit de respecter la capacité du sac à dos, i.e on a une contrainte portant sur la somme des valeurs de certaines variables (celles correspondant aux objets placés dans le sac), ou encore portant sur la somme des valeurs des sous-ensembles des variables. Ces contraintes sont particulières, intuitivement on voit bien pourquoi : elles portent sur des combinaisons partielles des variables. Voir par exemple [93].

# Chapitre 7

## Méthodes complètes

Les méthodes complètes de résolution de contraintes sont utilisées depuis plusieurs décennies. On en décrira ici les principales idées. Pour une présentation plus détaillée, il existe de nombreux excellents tutoriaux, par exemple Bartak [16], ou les tutoriaux d'IC-Park<sup>1</sup>.

### 7.1 Principe général, arbre de recherche

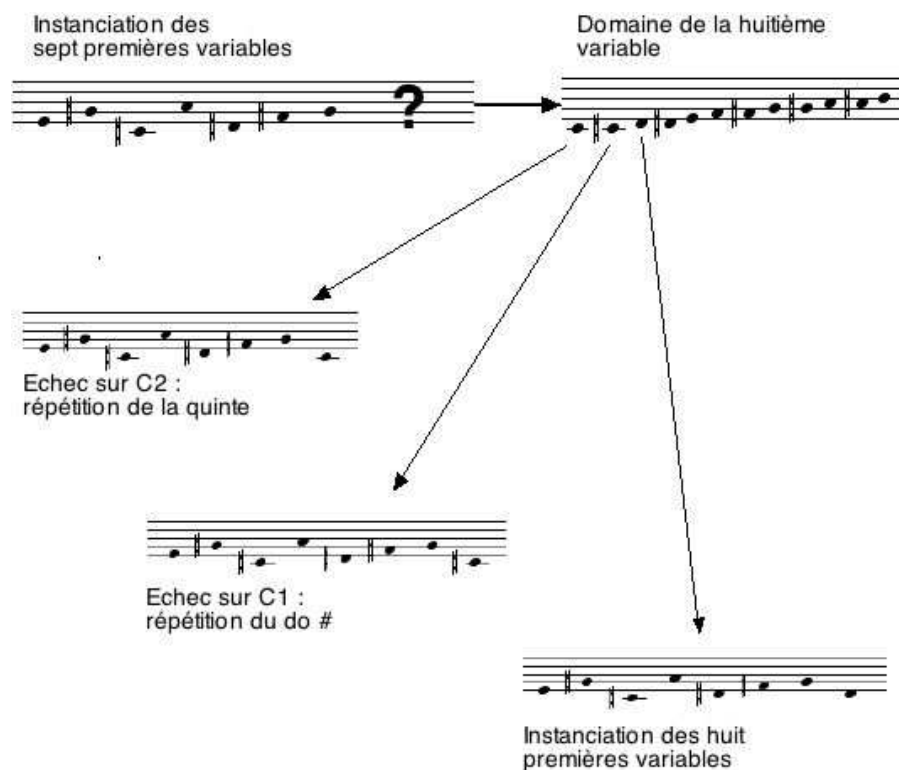


FIG. 7.1 – Les techniques classiques de résolution de contraintes reposent sur des instantiation partielles des variables. Ici pour le problème des all intervals series, on a instancié les sept premières variables de manière à satisfaire les contraintes, et on cherche à instancier la huitième.

<sup>1</sup><http://www-icparc.doc.ic.ac.uk/eclipse/reports/handbook/node1.html>

A moins d'en faire un traitement particulier, les contraintes permettent juste de répondre oui ou non sur des combinaisons de valeurs. L'idée la plus immédiate pour tenter de résoudre un problème de contraintes est donc de faire des essais, et ceci méthodiquement, pour chaque combinaison de valeurs. C'est la technique Generate and Test. A chaque variable est affectée une valeur. On obtient une instanciation  $v_1 \dots v_n$  sur laquelle on peut calculer les contraintes. Si  $C_1(v_1 \dots v_n) \cap \dots \cap C_p(v_1 \dots v_n)$  on a une solution (succès), sinon, on réessaie (échec). On visite ainsi le produit cartésien des domaines jusqu'à trouver une solution. C'est évidemment inefficace. Les contraintes sont calculées au dernier moment, et, la plupart du temps, inutilement. En fait, l'inefficacité vient du décalage temporel entre la vraie raison de l'échec et le moment où il apparaît. Par exemple, si une contrainte  $V_1 = V_2$  est violée dans l'instanciation courante, l'échec ne surviendra que lorsque toutes les autres variables seront instanciées, beaucoup plus tard. Par ailleurs, ce même échec se reproduira quelle que soit les instanciations des autres variables, donc un grand nombre de fois. Finalement, les échecs apparaissent en pratique presque indépendamment de leur cause. Ni la structure de l'arbre de recherche, ni la structure de CSP du problème ne sont exploités.

On représente souvent ce type de résolution de CSP par un arbre dont les noeuds sont les instanciations successives. A la racine, il n'y a rien (aucune variable n'est instanciée). La première branche consiste à instancier la première à la première valeur de son domaine. Au niveau en dessous, la première branche sous la première branche consiste à instancier la deuxième variable à la première valeur de son domaine, etc. La figure 7.2 montre un arbre de recherche en général.

Attention, quelle que soit la technique de résolution choisie, cet arbre est toujours visité *en profondeur* (ne serait-ce que pour des raisons d'espace mémoire). Un parcours en largeur signifierait qu'une même variable a plusieurs valeurs en même temps ce qui n'a aucun sens.

Le Generate and Test consiste à parcourir l'arbre systématiquement depuis la racine jusqu'aux feuilles jusqu'à trouver une solution.

## 7.2 Backtracking (retour arrière chronologique)

Les algorithmes classiques de résolution de contraintes reposent sur le principe du backtracking, ou retour arrière chronologique. Il s'agit toujours d'explorer le domaine de recherche,  $D_1 \times \dots \times D_n$  en testant les contraintes, mais cette fois on effectue ces tests dès que possible. C'est-à-dire, dès que les variables apparaissant dans la contrainte sont instanciées. Une étape de l'algorithme se passe de la manière suivante (on suppose par commodité d'écriture que les variables sont numérotées dans l'ordre d'instanciation, ou bien qu'on instancie dans l'ordre des variables) :

- (i). Etat courant : les  $i$  premières variables sont instanciées,  $V_1 = v_1 \dots V_i = v_i$ .
- (ii). Instanciation de  $V_{i+1}$  à la  $j$ -ième valeur de  $D_{i+1}$ , soit  $v_i^j$
- (iii). Si toutes les contraintes que l'on peut vérifier à ce stade sont vraies, on itère en 1
- (iv). sinon
- (v). Si il reste des valeurs dans  $D_{i+1}$ , soit  $v_{i+1}^{j+1}$  la suivante, on itère en 2
- (vi). sinon, on backtrack sur  $V_i$  que l'on instancie à la valeur suivante de son domaine, ce qui revient à itérer un pas en arrière.

L'ordre d'instanciation est susceptible de modifier la rapidité de l'algorithme. Mieux vaut choisir d'abord les variables de plus petit domaine.

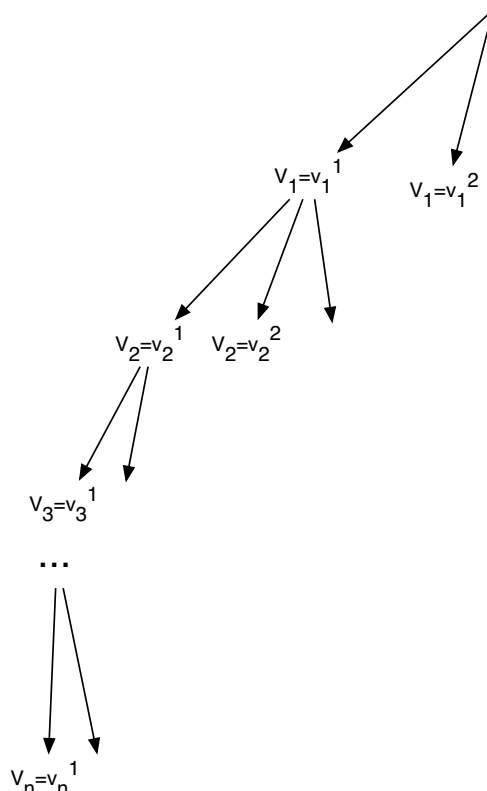


FIG. 7.2 – On représente en général la résolution d'un CSP par des méthodes complète sous la forme d'un arbre dont les noeuds sont les instantiation successives.

Pour le problème des all-intervals series, le principe est montré sur la figure 7.1. Les sept première variables ont été instanciées à mi, sol♯, do♯, la, ré♯, fa♯, sol. La branche suivante de l'arbre consiste à instancier la sixième variable. La première valeur possible du domaine, do, donne un échec sur la deuxième contrainte car la quinte est répétée dans la séquence. On remonte alors dans l'arbre à la septième variable, et on recommence, c'est ce qui est appelé backtrack. La deuxième valeur possible du domaine, do♯, donne un échec sur la première contrainte car do♯est répété deux fois dans la séquence : nouveau backtrack. Pour la troisième valeur possible, ré, on a une séquence qui vérifie (pour l'instant) les contraintes, et la descente dans l'arbre peut continuer.

Avec le backtracking, à chaque fois qu’une instantiation partielle viole une contrainte, le sous arbre qui commence à ce noeud n’est pas visité, ce qui épargne des recherches inutiles. Cependant, le temps de calcul reste exponentiel sauf miracle, et dans le pire des cas, tout l’espace de recherche doit être visité. En tous cas, on a aucune garantie que ce ne soit pas le cas.

Le backtracking ne traite pas du tout les conflits entre variables : lorsqu'une contrainte, par exemple binaire et portant sur  $V_{i_1}$  et  $V_{i_2}$  renvoie la valeur "faux" avec les valeurs  $v_1$  et  $v_2$ , il y a un conflit entre les valeurs courantes  $v_1$  et  $v_2$ . Pour peu que  $V_{i_1}$  soit la première variable instanciée, et  $V_{i_2}$  la dernière, on effectue quantité de recherches inutiles pour l'instanciation de  $V_{i_1}$  à  $v_1$ , puisque le problème n'est détecté que sur les feuilles de l'arbre.

Certaines techniques ont été développées pour raffiner l'algorithme, comme le backjumping qui consiste à remonter non pas à la variable précédente, mais plus haut, directement à la variable supérieure impliquée dans l'échec (ce qui suppose de maintenir pour chaque variable, le nom cette variable supérieure). Ceci pour éviter les backtracks intermédiaires. En réalité, il existe de multiples variantes du backtracking, dont l'efficacité varie selon les problèmes (selon la nature du graphe de contraintes).

En réalité, la structure d'arbre de recherche est une structure bas niveau par nature. De plus, les seuls outils dont on dispose, les contraintes, fournissent par elles-mêmes très peu d'information : vrai / faux. On voit bien qu'écrire un algorithme de résolution passe nécessairement par la définition de concepts de plus haut niveau, pour tenir compte notamment de l'interdépendance des contraintes.

## 7.3 Arc consistance

Hors traitement préalable de l'information fournie par les contraintes, un algorithme de recherche est basé sur des essais plus ou moins efficaces. Les techniques les plus classiques pour accélérer le processus reposent sur la notion d'arc consistance.

**Définition 7.1** Soit un CSP avec les notations précédentes. On suppose les contraintes binaires. Le problème est dit arc consistant si :

$$\forall i, j, \forall \text{ contrainte } C(V_i, V_j), \forall v \in D_i, \exists v' \in D_j \text{ tel que } C(v, v').$$

Cette définition peut sembler un peu austère. Elle est pourtant très naturelle quand elle est présentée à l'envers : un problème est arc consistant si aucune variable n'a de valeur "impossible", ie qui provoquerait systématiquement un échec sur une des contraintes. C'est, autrement dit, une sorte de garantie partielle : chaque valeur possible pour  $V_i$  est garantie par une valeur de  $V_j$ , de sorte qu'en instanciant  $V_i$  n'importe comment, on sait que la contrainte  $C$  peut être vérifiée (indépendamment des autres contraintes).

Bien que difficile à utiliser en pratique, cette définition peut être généralisée dans le cas de contraintes  $k$ -aires. Une contrainte quelconque  $C(V_{i_1} \dots V_{i_k})$  est consistante ssi  $\forall v \in D_{i_1}, \exists v_{i_2} \in D_{i_2}, v_{i_3} \in D_{i_3} \dots v_{i_k} \in D_{i_k}$  tel que  $C(v_{i_1}, v_{i_2} \dots v_{i_k})$ . Un CSP est arc-consistant ssi il est consistant pour toutes ses contraintes.

Remarque : il suffit de compter les  $\forall$  dans la définition pour voir que l'arc consistance est une propriété forte. Mais attention, l'arc consistance n'est pas équivalente à "il existe une solution". On trouve très facilement des problèmes consistants et insolubles, par exemple trois variables  $V_1 \dots V_3$ , de domaine  $D_1 = D_2 = D_3 = \{1, 2\}$ , et comme contraintes  $V_1 \neq V_2, V_1 \neq V_3, V_2 \neq V_3$ . Ici chaque contrainte peut être satisfaite dans son coin, mais l'ensemble qui n'a pas de solution. L'arc consistance ne garantit les contraintes que séparément, et pas ensemble.

L'arc-consistance est une propriété compliquée à mettre en oeuvre, de par sa nature récursive. Il faut filtrer les domaines d'après les contraintes, en trouvant une valeur garantissant la consistance pour chaque contrainte et chaque élément d'un domaine. Mais le problème a alors changé, parce qu'évidemment les domaines ont été restreints. Il se peut que les valeurs garantissant la consistance aient été enlevées dans la suite du filtrage. il faut donc recommencer à vérifier la consistance, jusqu'au point fixe, ie jusqu'à ce qu'une vérification de toutes les contraintes et de toutes les valeurs des domaines n'aie abouti à aucune restriction. Le facteur déterminant pour savoir si l'arc consistance est utilisable est l'arité des contraintes<sup>2</sup>.

<sup>2</sup>nombre de variables apparaissant dans la contrainte

De nombreux algorithmes se sont succédés pour implémenter l'arc-consistance sous des formes diverses. La technique la plus simple est la consistance de noeud (node consistency), élimination des valeurs des domaines inconsistante pour les contraintes unaires. Il s'agit simplement de réduire les contraintes unaires dans les domaines, ce qui peut être fait très rapidement.

Les algorithmes d'Arc-consistance forment toute une famille dont le premier, AC-1, correspond à un filtrage récursif tel que décrit ci-dessus (jusqu'à ce qu'il n'y ait plus de réduction de domaines). AC-3 reprend le principe mais en ne re-vérifiant à l'étape  $i$  que les contraintes binaires susceptibles d'avoir été modifiées à l'étape  $i - 1$ . Pour une comparaison de l'efficacité, voir [65]. AC-4 est une variante d'AC-3 qui évite de recalculer les valeurs des contraintes à l'étape  $i$ . D'autres algorithmes de maintien de consistance ont été développés : AC-5, 6, 7, 2000, 2001, notamment par la société ILOG, voir par exemple [19].

L'arc consistance peut être faite avant de lancer un algorithme de recherche, ou bien pendant la recherche, ce qui est une technique lourde (chaque nouvelle instantiation appelle de nouvelles réductions de domaines). Si la notion de consistance est élégante, l'efficacité des différents algorithmes dépend des problèmes, notamment de la structure du graphe de contraintes.

## 7.4 Forward checking

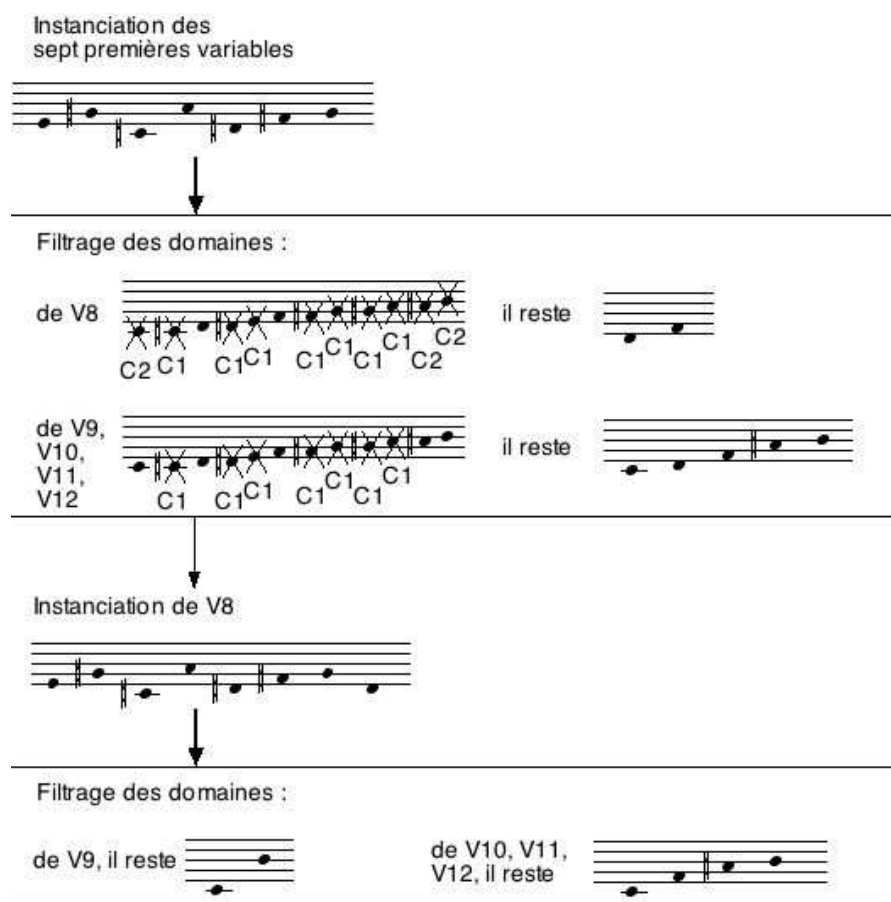


FIG. 7.3 – Un étape de backtracking avec forward checking.

L'arc consistance est une propriété dont on peut dériver toutes sortes d'algorithmes. L'idée générale est de vérifier au cours de la recherche certaines formes de consistance. Si l'on détecte dans les domaines des valeurs inconsistantes, on peut les enlever immédiatement de la recherche car tous les sous arbres de ces valeurs auraient des échecs aux feuilles. Cependant, montrer l'inconsistance d'une valeur pour une contrainte binaire, suppose de visiter tout le domaine de l'autre variable en calculant la contrainte. C'est long. Le risque est de passer plus de temps en calculs de consistance qu'on n'en gagne en enlevant des valeurs. D'où l'idée d'utiliser des consistance partielles, avec un nécessaire compromis entre calcul de consistance et réduction des domaines.

La variante la plus couramment utilisée est le forward checking. Comme son nom l'indique, il s'agit de vérifier les domaines à l'avance. Supposons que  $k$  premières variables sont instanciées. Cela signifie qu'*en arrière*, le problème est cohérent, c'est le principe du backtracking. Mais il se peut aussi que ces  $k$  premières valeurs permettent de déduire des choses pour les autres domaines, ceux des variables qui n'ont pas encore été instanciées. C'est ce qui manque au backtracking : exploiter l'information non seulement à partir de la définition du CSP, mais aussi à partir des instanciations courantes.

Le forward checking consiste à vérifier les domaines des variables  $V_i$  pour  $i \geq k$  dès que possible, dès qu'on a suffisamment d'information pour le faire, en d'autres termes dès que un de ces  $V_i$  apparaît dans une contrainte dont les autres variables sont instanciées. La figure 7.3 en montre une étape pour les all intervals series. En haut, l'état courant de la recherche : les sept premières variables sont instanciées à mi, sol ♯, do ♯, la, ré ♯, fa ♯, sol. Les intervalles successifs sont tierce majeure, quinte juste, quinte augmentée, quarte augmentée, tierce mineure, seconde mineure. Jusqu'ici, pas d'incohérence grâce au backtracking. Le forward checking intervient avant d'instancier la huitième variable.

On vérifie d'abord les domaines de  $N_8$  à  $N_{12}$ . Sur  $N_8$  portent deux types de contraintes : d'abord, sur les notes, on a  $N_1 = N_8$ ,  $N_2 = N_8$ , ...  $N_{12} = N_8$ . Les sept premières contraintes sont binaires et la première variable est instanciée : on peut donc filtrer le domaine de  $N_8$  en éliminant les valeurs incohérentes, soit dans notre cas mi pour  $N_1 = N_8$ , sol ♯ pour  $N_2 = N_8$ , etc. En revanche, les contraintes suivantes, à partir de  $N_9 = N_8$ , ne peuvent pas être exploitées puisque les variables qui y apparaissent hors  $N_8$  ne sont pas encore instanciées. A ce stade, le domaine de  $V_8$  est { do ré fa la ♯ si }.

Il faut aussi traiter les contraintes sur les intervalles. Notre variable apparaît dans deux fois dix contraintes de ce type : d'abord sur l'intervalle avant  $N_8$ ,  $|N_8 - N_7| \neq |N_{j+1} - N_j|$  pour  $j \neq 8$ , et sur l'intervalle après  $N_8$ ,  $|N_9 - N_8| \neq |N_{j+1} - N_j|$ . De cette dernière série on ne peut rien déduire car  $N_9$  n'est pas instanciée. De la première série en revanche, on peut exploiter les contraintes avec  $j \in \{2...7\}$ . Cela revient à interdire les notes qui formeraient à partir de sol ( $N_7$ ) des intervalles apparaissant déjà dans la série partielle. La seconde mineure permet d'enlever le fa ♯ et le sol ♯, la tierce mineure d'enlever le mi et le la ♯, la tierce majeure le ré ♯ et le si, la quarte augmentée le do ♯ et enfin la quinte juste, le do. Les intervalles plus grands ne servent à rien car on sortirait des domaines. Certaines de ces valeurs étaient déjà enlevées par les contraintes  $C_1$ , mais pas toutes et le domaine de  $V_8$  est maintenant réduit à { ré fa }.

Pour les autres variables, on procède de la même manière à ceci près que seule la série de contraintes sur les notes est exploitable. La deuxième série de contraintes contient toujours au moins deux variables non instanciées, dit plus clairement, on ne peut pas savoir quels seront les intervalles après  $N_8$  puisqu'on ne connaît pas encore les notes. Avec la première série de contraintes, on peut filtrer les domaines de  $N_9$  ...  $N_{12}$  à { do ré fa la ♯ si }. Ce n'est qu'après ces divers filtrage que l'on continue les instanciations, ici avec  $N_8$  vaut ré, ce qui permet de filtrer

$N_9 \in \{dosi\}$ , etc. Sans forward checking, on aurait dû essayer inutilement les valeurs do et do #.

lisboete sur le probleme geometrique des molecules le jeu de la vie

Remarque : on a évidemment omis pour des raisons de présentation les réductions de domaines intervenues avant l'étape que nous présentons. En réalité, on démarre les réductions non pas avec les domaines entiers, mais avec des domaines déjà réduits par des applications précédentes du forward checking.

Les coupes dans l'arbre de recherche grâce au forward checking sont de deux natures : d'abord, la simple réduction de domaine élimine bien sûr tous les sous arbres sous les valeurs enlevées. Ensuite, il se peut qu'un filtrage de domaine aboutisse à un domaine vide pour une variable. C'est encore plus radical pour la coupe de l'arbre, puisqu'on a alors un échec et dans ce cas on backtracke directement. On voit nettement ici que le forward checking anticipe les conflits.

Il existe de nombreux algorithmes de forward checking, soit au cas par cas, *ie* de manière adaptée au CSP traité, soit généraux.



# Chapitre 8

## Les soft constraints

Plusieurs cadres ont été proposés pour élargir le champ de la programmation par contraintes, de manière à pouvoir modéliser des préférences (valuations sur les configurations, préférences, contraintes pondérées, le nom change selon les cadres). L'idée générale est de pouvoir traiter l'information plus finement qu'une contrainte à valeur dans  $\{\top, \perp\}$ . L'intérêt est multiple : exprimer des préférences sur l'ensemble des solutions, exprimer des contraintes en leur attribuant des "poids", traiter des problèmes surcontraints en précisant les valeurs de chaque contrainte, ou au contraire définir des critères de choix dans l'ensemble des solutions pour un problème sous-contraint, etc. D'une manière générale, il s'agit d'exprimer au niveau de la modélisation soit des préférences sur des valeurs (l'instanciation  $V_1 = 1, V_2 = 4$  est à proscrire), soit des préférences sur des contraintes (la contrainte  $C_1$  peut être pas satisfaite, un peu satisfaite, très satisfaite...).

Les soft-constraints offrent un cadre général pour représenter toute sorte de modèles de ce type. Chacun est paramétré par des espaces et des lois de composition interne, y compris les CSPs classiques. La question centrale est celle de l'espace dans lesquels les contraintes prennent leurs valeurs. On considère une contrainte comme une fonction donnée par son scope, un sous-ensemble des variables, à valeur dans  $A$ , un ensemble à définir.  $A$  vaut  $\{\top, \perp\}$  dans le cas des CSPs classiques. D'une manière générale, il représente l'ensemble des valeurs possibles pour les contraintes, de sorte qu'il doit vérifier un minimum de propriétés.

Un c-semiring est donné par :

- $A$  un ensemble de valeurs
- $+$  loi de composition interne sur  $A$  commutative, associative et idempotente
- $*$  loi de composition interne sur  $A$  associative et distributive par rapport à  $+$
- $0$  élément neutre pour  $+$ , absorbant pour  $*$
- $1$  élément neutre pour  $*$ , absorbant pour  $+$

Une contrainte  $C$  s'exprime comme une fonction d'un sous-ensemble des variables (noté  $con_c$ ) à valeurs dans un c-semiring  $A$ . La loi  $*$  sert à combiner les contraintes, et la loi  $+$  à les comparer. La valeur  $0$  est la plus mauvaise valeur pour les contraintes, c'est nécessairement l'élément absorbant pour  $*$  parce qu'en combinant des contraintes dont l'une n'est pas satisfaite, l'ensemble ne peut pas être satisfait. La valeur  $1$  est la meilleure valeur pour les contraintes, c'est nécessairement l'élément neutre pour  $*$  parce qu'une contrainte satisfaite peut être combinée avec une autre sans changer la valeur de cette autre contrainte. En fonction des structures choisies, on obtient plusieurs cadres pour exprimer les CSPs.

- Les **crisp CSPs** (le cadre classique) est obtenu avec  $A = \{\top, \perp\}$ ,

$+$  =  $\vee$ ,  
 $*$  =  $\wedge$  la combinaison de contrainte est un et logique,  
 $0$  =  $\perp$ ,  $1$  =  $\top$

- Les **fuzzy CSPs** introduisent le "flou" dans les contraintes, pour qu'une contrainte puisse être partiellement vérifiée. Formellement, les contraintes deviennent des prédicats flous à la manière de la logique floue. L'ensemble des valeurs autorisées est donc continu entre 0 et 1. Cela se représente pour  $A = \llbracket 0, 1 \rrbracket$ ,  $+$  =  $\max$  analogue du ou logique pour la logique floue,  $*$  =  $\min$  analogue du et logique,  $0 = 0$  et  $1 = 1$ .
- Les **CSPs probabilistes** donnent un point de vue différent avec le même ensemble de valeurs. Les contraintes sont vues comme des probabilités : au lieu de considérer les contraintes comme des affirmations, on les voit comme des lois de probabilité sur les instanciations. On prend en conséquence  $A = \llbracket 0, 1 \rrbracket$ ,  $+$  =  $\max$ ,  $*$  =  $*$  combinaison probabiliste,  $0 = 0$  et  $1 = 1$ .
- Dans le cadre des **weighted CSPs ou WCSPs**, les contraintes sont vues comme des préférences à valeurs réelles. Telle contrainte peut être alors deux fois plus vérifiée qu'une autre. On prend alors  $A = \mathbb{R}$ ,  $+$  =  $\min$ ,  $*$  =  $+$ ,  $0 = \infty$  et  $1 = 0$ .

Des algorithmes de consistance ont été donnés pour certains cadres de soft CSPs, voir [21] ou [23].

Les soft constraints sont sans doute aux CSPs ce que la théorie des catégories est à l'algèbre : il s'agit d'abstraire un certain nombre de propriétés nécessaires et suffisantes pour pouvoir définir un problème de contraintes et raisonner avec. Ce cadre n'est probablement pas indispensable en pratique, mais il a l'intérêt de définir proprement les structures de valeurs des contraintes, et de donner une structure minimale (opérations de combinaison) pour pouvoir raisonner avec des contraintes. Les soft constraints ont été utilisées dans des travaux hybrides à la manière de Anglada et al [7], qui définit les Numériques Semiring based CSPs, où les soft constraints servent à exprimer des préférences pour des problèmes la conception de pièces mécaniques. Les NSCSP sont ensuite résolus par transformation en CSPs numériques. Autre exemple, [22] où est présentée une méthode hybride de recherche locale et de soft constraints, avec un système de seuils sur les fonctions de coût pour passer des soft constraints à des techniques de recherche locale.

## 8.1 Langage de contraintes

Jusqu'ici, on s'est concentré sur les techniques complètes de résolution. En pratique, elles se comparent au cas par cas en fonction des problèmes. Un cran au dessus se situent les langages de programmation par contraintes, ou les solvers généraux, *ie* des systèmes qui proposent d'utiliser les contraintes comme un langage informatique, et de résoudre un CSP exprimé dans un langage bien défini.

Il existe des solvers commerciaux dont le plus connu est probablement le très efficace ILOG-Solver<sup>1</sup>. Cependant, leur prix les situe un peu à part de la recherche académique. On peut également citer Eclipse (voir [4]) d'IC-Park ou CHIP (voir [2]) de Cosytech.

Un langage de contraintes académique très élégant est Prolog avec contraintes de l'INRIA<sup>2</sup> (voir [42]). Il est basé sur le langage de programmation logique Prolog, né au départ pour modéliser les langues naturelles. La programmation logique est un domaine de recherche en soi

<sup>1</sup>[www.ilog.com](http://www.ilog.com)

<sup>2</sup><http://gnu-prolog.inria.fr/>

et nous n'entrerons pas dans les détails, voir par exemple [38]. GNU-Prolog avec contraintes se base sur le langage CLP(FD) (Constraint Logic Programming over Finite Domains), voir [34]. Les contraintes s'écrivent dans ce langage comme dérivant d'une seule primitive de contrainte,  $x \text{ in } R$  où  $x$  est une variable et  $R$  un ensemble fini de valeurs, mais dans lequel peuvent intervenir d'autres variables. Les contraintes "habituelles" peuvent s'écrire comme combinaison de telles contraintes, et la propagation s'effectue donc très naturellement.

Prolog est un langage extraordinairement intelligent mais déroutant dans lequel il est difficile d'écrire un programme sans un minimum de connaissances logiques et informatiques. C'est pourquoi il ne s'applique probablement pas dans notre cas, la CAO. D'un point de vue technique, GNU-Prolog avec contraintes n'est en outre pas disponible pour Macintosh.



# Chapitre 9

## Les métaheuristiques

Les métaheuristiques et la recherche locale est aujourd'hui un domaine de recherche bien établi. On trouvera des présentations plus détaillées dans [86], [1] [54], ou plus récemment dans [70].

### 9.1 SAT

Le problème SAT a donné lieu à de nombreuses méthodes de résolution incomplètes, notamment GSAT (voir [95]). L'idée est de partir d'une affectation aléatoire. Si toutes les clauses sont satisfaites, on a une solution. Sinon, on choisit une clause insatisfaite, une des variables de cette clause et on flippe sa valeur (les clauses sont des disjonctions, elles sont vraies quand une ou plus de ses variables sont vraies, et fausses seulement quand toutes ses variables sont fausses, de sorte qu'en modifiant une seule valeur on est certain de satisfaire la clause). On itère jusqu'à obtenir une solution, ou bien à atteindre un nombre maximum d'itérations, et dans ce cas le procédé est répété un nombre fixé de fois. Le fait de relancer l'algorithme régulièrement peut sembler annexe mais c'est en réalité crucial, afin de ne pas tourner trop longtemps dans une même zone de l'espace de recherche (diversification, voir ci-dessous).

Le risque est de se trouver dans des cas qui bouclent, ou bien des cas où satisfaire une clause ne suffit pas à augmenter le nombre total de clauses satisfaites : pas d'amélioration de la qualité de la configuration courante. Le principe de compter le nombre de violations des contraintes, ou le nombre de conflits (le nombre de clauses non satisfaites, ici) est appelée Min Conflict, voir [74].

L'algorithme WalkSAT [11] reprend le même principe en introduisant un paramètre aléatoire de plus : soit  $p$  entre 0 et 1. À chaque itération, la variable choisie est modifiée avec une probabilité  $p$ , ou reste à sa valeur avec une probabilité  $1 - p$ . Cette idée a été connue un succès certain et de nombreuses variantes : UnitWalk est un WalkSAT raffiné pour tenir compte des interdépendances entre les variables [57]. D'autres méthodes ont été développées en hybridant SAT et des méthodes complètes. Gent et al. proposent récemment par exemple dans [49] WalkQSAT, une technique de résolution des QBF, des formules closes avec quantificateurs existentiels et universels, basée en partie sur du backtracking et en partie sur WalkSAT.

WalkQSAT s'appuie sur un arbre binaire qui représente la formule à vérifier (dont les feuilles sont les quantificateurs, par exemple  $\forall x$ , et les branches gauches et droites correspondent à  $x = \top$  et  $x = \perp$ ). Les sous parties de l'arbre qui correspondent aux problèmes SAT (conjonction de disjonctions) sont envoyées à WalkSAT, qui répond soit une instantiation solution, soit *Unknown*.

En fonction de la réponse, WalkQSAT backtrack sur les feuilles antérieures. WalkQSAT est compétitif par rapport à la méthode complète. C'est un bon exemple de l'apport des méthodes incomplètes, soit en tant que telles, soit hybridées à des méthodes de backtracking.

En résumé, toutes ces techniques présentent une différence énorme avec les techniques précédemment décrites, parce qu'elles abandonnent l'idée d'exploiter les contraintes comme définissant une consistance, une sorte de logique à respecter, pour préférer les exploiter comme mesure de qualité d'une configuration. Autre différence notable, au lieu d'envisager tout l'espace des possibles, la famille des WalkSAT se promène de proche en proche dans cet espace. Le cas de SAT se prête particulièrement bien à un tel schéma : à partir d'une clause fausse, un mouvement local (changer une valeur) est garanti rendre la clause vraie. La grande homogénéité du problème (les contraintes ont toutes la même forme) permet aussi de compter simplement les contraintes vraies, et d'en déduire la qualité d'une configuration. Précisons enfin que SAT reste un challenge dans le cas général, voir le discours invité d'Henry Kautz à CP03 qui définit "Ten Challenges Redux", dix problèmes restant à résoudre à propos de SAT : notamment, résoudre les problèmes SAT issus "du monde réel" (les instances provenant de CSPs), mélanger les algorithmes stochastiques et les méthodes complètes, tenir compte des redondances des variables.

De telles méthodes de résolution ont permis de résoudre des instances de SAT jusqu'à des millions de variables. Cependant, les résultats sont très variables selon le type d'instance résolu (structuré ou non, ratio clauses / variables).

## 9.2 La recherche locale

Les méthodes de résolutions regroupées sous ce nom suivent les mêmes principes que GSAT, mais pour des problèmes quelconques. Elles trouvent leurs origines dans les travaux de Lin [64] sur le Travelling Salesman Problem. Elles consistent à prendre le contre pied des méthodes complètes décrites ci-dessus, sur à peu près tous les points. Le principe général est de parcourir l'espace de recherche sans garder de mémoire du chemin visité. Initié aléatoirement, ce parcours doit être guidé vers des solutions. Cela est fait en transformant les contraintes en fonctions de coût, qui donnent une indication sur à quel point les contraintes sont violées. C'est une notion assez naturelle : pour une contrainte  $X=Y$  sur les entiers, il est intuitivement évident que l'instanciation  $X=1, Y=2$  est meilleure que  $X=1, Y=10$ . Cela étant, les pré-requis pour une fonction de coût sont assez faibles : une fonction de coût a pour domaine l'espace de recherche, et vaut zéro quand la contrainte associée est satisfaite. Une même contrainte peut être transformée en plusieurs fonctions de coût, et trouver des fonctions de coût efficaces est dépendant du problème.

L'architecture générale d'un algorithme de recherche locale comprend :

- des fonctions de coût pour chaque contrainte, et une opération de combinaison, qui permette d'obtenir une fonction de coût sur les instanciations pour toutes les contraintes
- une fonction de voisinage, qui associe à chaque instanciation un sous ensemble de l'espace de recherche
- une métaheuristique pour sortir des minima locaux

Les techniques de recherche locale quelles qu'elles soient sont incomplètes. Pour un problème à résoudre, cela signifie qu'on ne peut pas avoir de preuve d'insolvabilité (si le problème n'a pas de solution, les algorithmes bouclent). Pour un problème d'optimisation, on ne peut pas avoir de preuves d'optimalité. C'est la contre partie du fait qu'elles en gardent pas de mémoire de

l'espace de recherche exploré. D'une manière générale, la qualité d'un algorithme de recherche locale se mesure selon deux critères, à nombre d'itérations ou temps de calcul donné : la qualité de la solution trouvée, et le nombre d'optima locaux visités. Quel que soit l'algorithme, il comportera d'une part une manière de viser les optima locaux, et d'autre part une manière de s'égarer ponctuellement.

## Recherche Tabu

La méthode appelée Tabu Search (TS) est due à Fred Glover [50]. Elle connaît actuellement un succès certain, par exemple les cinq algorithmes vainqueurs du challenge ROADEF 2002 étaient tous des algorithmes Tabu ou adaptés de Tabu<sup>1</sup>. La recherche Tabu consiste à garder visiter l'espace de recherche de voisinage en voisinage en cherchant à améliorer la fonction de coût. A partir d'une configuration courante  $c$ , on explore un voisinage  $V_c$  de  $c$  en cherchant à optimiser la fonction de coût  $f$ , soit  $c'$  le meilleur candidat sur  $V_c$  ( $c$  n'est pas considéré dans l'exploration). On bouge alors vers  $c'$ , quel que soit son coût. Les cas où  $f(c') > f(c)$  permettent de s'éloigner des optima locaux.

Le risque est de boucler : si  $c$  est un minimum local, l'étape suivante de la recherche Tabu consiste à choisir  $c'$  voisinage de  $c$ , l'étape suivante consiste à examiner un voisinage de  $c'$  dans lequel il est très probable que  $c$  figure, et sur lequel il est aussi très probable que  $c$  soit le meilleur candidat. L'algorithme entrerait ainsi dans une boucle infinie  $c \rightarrow c' \rightarrow c \rightarrow \dots$ . Pour éviter ceci, La recherche Tabu utilise une liste Tabu qui sert de mémoire à court terme. Soit  $t$  un entier fixé ( $t$  est appelée Tabu tenure). A chaque étape, est maintenue une liste des  $t$  dernières configurations visitées, sont écartées de l'exploration. La Tabu tenure  $t$  devient le nombre minimum d'itérations pour sortir d'un cycle, *ie* si un minimum local est situé au fond d'un "puits" de taille  $t' > t$  (en nombre d'itérations pour sortir du puits), l'algorithme boucle.

En pratique, pour des raisons d'efficacité, ce ne sont pas les configurations elles-mêmes qui sont mémorisées mais par exemple des couples (variable, valeur). Dans ce cas, la liste Tabu contient les couples (variable, valeur) qui ont été perdus dans un mouvement. Si on passe de  $c = (v_1 \dots v_n)$  à  $(v'_1 \dots v_n)$ , on ajoute à la liste Tabu le couple  $(V_1, v_1)$ , et aucune configuration commençant par  $v_1$  ne pourra être retenue pendant  $t$  itérations, jusqu'à ce que  $(V_1, v_1)$  sorte de la liste Tabu. C'est beaucoup plus large que la seule configuration  $c$ .

La liste Tabu risque donc de générer des interdictions très fortes (le nombre de configurations  $N_c$  telles que  $V_1 = v_1$  n'est qu'un ordre de grandeur plus petit que l'espace de recherche, et on en interdit  $t * N_c$ ) Le mécanisme d'aspiration permet de redonner un peu de souplesse dans l'exploration. L'aspiration consiste à autoriser certains mouvements même s'ils sont interdits par la liste Tabu, à condition que le coût s'en trouve amélioré. Quelle que soit la technique d'aspiration choisie, il faut faire attention à ne pas réintroduire un risque de cycle.

Deux mécanismes peuvent être ajoutés à la recherche Tabu. Les deux jouent sur la gestion de la liste Tabu. Premièrement, l'intensification sert à concentrer la recherche vers ce qui est supposé être une bonne solution, ou bien vers des valeurs dont on a pu constater par le passé qu'elles donnaient de bonnes solutions. Deuxièmement, la diversification sert au contraire à éloigner l'algorithme de ce qu'il a déjà visité. Ces deux mécanismes jouent sur la liste Tabu.

Les algorithmes Tabu sont très efficaces sur de nombreux problèmes. Cependant, les options retenues (réglages de paramètres, critères d'aspiration et de diversification) sont à la fois très

---

<sup>1</sup>Le sujet du challenge était un problème de planning, pour un satellite qui prend des photos, avec fenêtres de temps et fonctions de coûts non linéaires, voir <http://www.prism.uvsq.fr/vdc/ROADEF/CHALLENGES/2003/>

importantes pour l'efficacité de la résolution et très dépendants de la nature du problème.

### Le recuit simulé

La méthode de recuit simulé est inspirée de la physique des matériaux, plus précisément thermodynamique de refroidissement des aciers. C'est une technique itérative, qui consiste à chauffer / refroidir plusieurs fois l'acier, dont le désordre interne est mesuré par une certaine fonction d'énergie que l'on cherche à minimiser. Le chauffage est destiné à faire franchir des cols à cette énergie, dans l'espoir qu'elle redescende plus bas (minimum local) dans la phase de refroidissement. Plus le nombre d'itérations est grand, moins l'acier est autorisé (statistiquement) à monter en température. Le but est d'obtenir des aciers d'énergie minimale, donc plus solides.

Soit  $f$  la fonction de coût de l'espace de recherche  $D_1 \times \dots \times D_n$  dans  $\mathbb{R}$  (ou  $\mathbb{N}$ , peu importe). Partant aléatoirement d'une configuration  $c = (v_1 \dots v_n)$ , on considère un voisin  $c'$ , choisi aléatoirement, sur le voisinage de  $c$ . Si  $f(c') < f(c)$ , le coût de  $c'$  est meilleur que celui de  $c$  et on itère le procédé à partir de  $c'$ . Cela revient à considérer que, quand  $f$  est localement décroissante, on se dirige vers une solution (ce n'est pas nécessairement vrai, mais on ne peut pas supposer mieux avec l'information dont on dispose). Mais on autorise aussi les déplacements vers  $c'$  pour  $f(c') > f(c)$  avec une certaine probabilité, qui dépend de la "température", paramètre qui est destiné à descendre au cours de la recherche. De cette manière, l'algorithme est destiné à descendre vers les minima locaux "parfois" au début de la recherche, puis de plus en plus souvent. La possibilité laissée à chaque itération d'augmenter la fonction de coût est destinée à lui faire passer des cols d'énergie. Les algorithmes de recuit simulé sont très dépendants du processus de refroidissement.

## 9.3 Autres

Plusieurs algorithmes de résolution proposés sont des variantes de la recherche locale, dont ils reprennent certains principes (ne pas garder de trace de l'espace visité, considérer les contraintes comme des coûts), sans que l'on puisse pourtant les classer tout à fait comme de la recherche locale, parce qu'ils ne sont pas paramétrables en fonction de voisinage, fonctions de coût, métaheuristique pour sortir des minima locaux.

### Les algorithmes génétiques

Les algorithmes génétiques sont basés sur une sorte de théorie darwinienne et eugéniste de l'évolution. Il s'agit de considérer non pas une configuration, mais un ensemble de configurations, comme une population dont il s'agit d'améliorer la qualité. La notion de qualité est évidemment liée aux contraintes. L'idée d'appliquer de telles techniques à des programmes informatiques est due à Holland [58].

L'ensemble repose sur un codage des configurations sous la forme de "chromosomes", séquences de 0 et de 1 (de vrais et faux pour SAT par exemple). Sur cette population sont définies des opérateurs de croisement, qui d'une certaine manière introduisent la diversification : à partir de deux individus, est créé un nouvel individu qui hérite de caractéristiques de ses deux parents. Autre source de diversification, les mutations introduisent de petites modifications aléatoires dans les chromosomes. Ceci permet à chaque itération d'un algorithme génétique de gérer une

population de chromosomes, en conservant intacte une partie de la population, et en diversifiant une autre partie. On définit aussi un opérateur de sélection qui dans le cas de la résolution de CSPs mesure la qualité d'un chromosome, par rapport aux contraintes du problème. En pratique, selon [54], les algorithmes génétiques sont inefficaces en résolution de contraintes si la gestion de la population est faite aléatoirement. Il faut introduire dans les mécanismes de croisement et sélection des connaissances propres au problème de contraintes à résoudre. Voir [43] pour un exemple d'algorithme génétique et Tabu Search appliqué au coloriage de graphes, où le croisement est défini de manière à optimiser la taille des sous-graphes de même couleur.

### Les fourmis artificielles

Les algorithmes ACO (Ant Colony Optimisation) s'inspirent du comportement d'une colonie de fourmis qui cherchent leur chemin dans un terrain accidenté. Elles parcourent le terrain semi-aléatoirement, en laissant sur leur passage des phéromones que les autres fourmis peuvent détecter. Plus un chemin est court, plus vite il se voit créditer d'un taux de phéromone double par la fourmi qui le choisit (aller / retour). Il est donc préféré par les autres fourmis, ce qui renforce encore son taux de phéromones. L'efficacité collective et la réactivité des fourmis est due à la volatilité des phéromones : les phéromones déposées d'évaporent doucement, ce qui évite à la colonie de se trouver piégée dans un chemin de qualité moyenne.

Les algorithmes ACO suivent le même principe pour résoudre des problèmes de recherche de plus court chemin dans un graphe. Le principe général est le suivant : les fourmis artificielles parcourent le graphe en laissant des phéromones sur leur passage. La probabilité qu'une fourmi choisisse tel noeud suivant dépend de la quantité de phéromones sur l'arête qui mène à ce noeud, et généralement d'heuristiques dépendant du problème, par exemple pour maintenir une solution réalisable : dans le cas du Travelling Salesman Problem, une fourmi se dirige de ville en ville et elle ne peut pas choisir un noeud correspondant à une ville qu'elle a déjà visitée. Dans ce cas, les fourmis gardent une mémoire de leur trajet.

Christine Solnon et al ont développé trois solveurs par algorithmes ACO : Ant-P-solver pour les CSPs sur les permutations ([97]), Ant-solver pour les CSPs ([98]), et Ant-clique pour la recherche de cliques maximales dans des graphes ([48]). Ant-Solver est un système de résolution générique, qui peut traiter n'importe quel CSP. Le graphe construit a comme sommets les couples (variables, valeurs) et des arêtes entre chaque paire de variables distinctes. Les phéromones déposées sur une arête  $(V_i, v_i)$   $(V_j, v_j)$  représentent intuitivement le coût des instanciations où  $V_i = v_i$   $V_j = v_j$ .

## 9.4 Les méthodes hybrides

Il ne s'agit pas ici de dresser une liste exhaustive des méthodes hybrides de résolution de problèmes. D'une part, elles sont beaucoup trop nombreuses, d'autre part, elles sont souvent très spécialisées sur un problème précis et souvent complexes. On donnera seulement quelques exemples de travaux sur les méthodes hybrides. S'il ne s'en dégage peut-être pas encore de "meilleure méthode", ce sont cependant certainement de voies très prometteuses dans le domaine.

Michel Vasquez (voir [105]) propose ainsi une variante de la recherche Tabu mixée avec de l'arc-consistance. Aux domaines des variables est associée une valeur  $u$  pour "Uninstanciated", non instanciée. A la différence de la recherche Tabu où les configurations sont modifiées en chan-

geant une valeur, l'algorithme de Michel Vasquez modifie la configuration courante toujours en instanciant une variable dont la valeur courante est  $u$ . L'originalité de la méthode est d'ajouter après une instanciation (ou un move au sens Tabu) une phase d'arc-consistance, pour retirer de la configuration toutes les valeurs inconsistantes avec la nouvelle instanciation. Retirer signifie ici instancier à  $u$ . Cette méthode a prouvé son efficacité sur des problèmes d'assignation de fréquences (vainqueur du challenge ROADEF en 2001), de Scheduling pour des photographies satellitaires et de positionnement géographique d'antennes (voir [106]).

Gilles Pesant et Michel Gendreau proposent dans [83] une méthode dans laquelle l'exploration du voisinage est faite par une méthode de branch and bound. Narendra Jussien et Olivier Lhomme proposent à l'inverse dans [60] proposent une méthode dans laquelle la recherche locale est effectuée sur des instanciations partielles, testée sur des problèmes de scheduling. Jin-Kao Hao et al ont développé GASAT, une méthode mixant la recherche Tabu avec les algorithmes génétiques. La recherche locale est ici utilisée pour améliorer la qualité des descendants obtenus par cross-over sur des problèmes SAT (voir [15]). Breton et al donnent un modèle multi-agents pour la physique des tas de sable, dans lequel les agents-grains de sable sont des CSPs dont les contraintes sont de rester à l'équilibre.

En résumé, on trouve des méthodes hybrides Recherche Local dans CP, CP dans Recherche Locale, Recherche Locale et Algorithmes Génétiques, Recherche Locale et Agents, et si l'on s'intéresse également à la Recherche Opérationnelle, les applications sont extrêmement variées et nombreuses.

## 9.5 Langages pour la recherche locale

En général, les méthodes de recherche locale sont appliquées au cas par cas, de manière spécifique au problème à résoudre, voire même à l'instance de ce problème. Ainsi une méthode peut résoudre très efficacement des instances de SAT dites structurées, issues d'un problème de coloriage de graphe par exemple, et mal des instances de SAT aléatoires. Non seulement les fonctions de coût sont *ad hoc*, mais c'est aussi le cas des heuristiques, des choix de reset, du tuning des paramètres, etc.

Il existe cependant des modèles permettant d'abstraire des méthodes de recherche locale plusieurs concepts (par exemple, heuristique pour sortir des minima locaux, voisinage, etc), pour obtenir un modèle général de recherche locale dans lequel il suffit de spécifier dans chaque catégorie une méthode particulière. C'est aussi une préoccupation des méthodes hybrides.

Nous avons déjà mentionné plus haut les trois Ant-solvers de Christine Solnon.

Laurent Michel et Pascal Van Hentenrick ont développé un langage de modélisation pour la recherche locale, Localizer (voir [71], [72]). Localizer a pour but de faciliter l'implémentation d'algorithmes de recherche locale. Partant du principe que, précisément, il n'existe pas de cadre général pour implémenter des algorithmes de recherche, Localizer fournit l'architecture nécessaire pour implémenter facilement à la fois les structures de données et les heuristiques choisies. L'architecture générale d'une itération est donnée (par exemple, "tant qu'une solution n'a pas été trouvée et jusqu'à un nombre maximum d'itérations", etc), et l'utilisateur spécifie les voisinages utilisés, le type de mouvements (random, meilleur), ses heuristiques, etc. En particulier, Localizer maintient des invariants, des données dépendant des variables, dont les valeurs sont garanties mises à jour selon les instanciations. Laurent Michel et Pascal Van Hentenryck ont également développé Localizer++ ([73]), une librairie orientée objet pour l'implémentation d'algorithmes de recherche locale. Localizer n'est pas un algorithme de résolution de contraintes,

mais plus généralement une architecture pour écrire des algorithmes de résolution.

Salsa, de Laburthe et Caseau, [61] est un langage de spécification de méthodes de recherches locales et globales (qui compile vers le langage Claire) structuré autour de la notion de point de choix, opération de choix dans un ensemble de décisions à prendre. On peut enfin citer GENET, basé sur une heuristique Min Conflict, voir [40].



# Chapitre 10

## Contraintes en musique

### 10.1 Harmonisation automatique

Le problème de contraintes musicales le plus classique est celui de l'harmonisation automatique. Il s'agit de réaliser un exercice d'harmonisation à quatre voix, à chant donné ou à basse donnée par exemple. Rappelons qu'une harmonie doit respecter de nombreuses règles, consignées dans de multiples traités, de sorte que l'ensemble sonne correctement (selon les critères classiques). Ces règles imposent des structures verticales précises, les accords, dont la forme est codifiée, et restreignent en plus certaines combinaisons de notes.

Historiquement, plusieurs techniques ont été utilisées pour tenter de résoudre ce problème, probabilistes notamment. On peut cependant remarquer que l'harmonisation automatique est tout naturellement posée comme un problème de contraintes. En effet, les règles musicales telles que "pas de quintes parallèles", "pas de croisement de voix", etc, sont déjà données de manière déclarative plutôt que constructive. Il suffit par exemple de prendre comme variables les notes (ou les accords chiffrés par exemple), et comme domaines les tessitures de chaque voix. [79] donne un historique des techniques utilisées pour l'harmonisation.

Le précurseur dans ce domaine est Kemal Ebcioglu qui propose dans [44] un système de composition de chorals dans le style de Bach, baptisé CHORAL. L'entrée du système est une mélodie, à harmoniser pour produire un choral à la manière de Bach. Le système est basé sur le backtracking, avec plus de 300 règles absolues ou heuristiques.

Le même problème est traité par Tsang [102], avec une implémentation *ad hoc*. Les variables, *pulse*, sont formées de plusieurs paramètres : durée, pitches des voix soprano, alto, ténor et basse, accord. Une vingtaine de contraintes expriment les règles musicales. Ce système reste assez lent (cinq minutes pour harmoniser une mélodie de 11 notes sur une station SPARC-1), et gros consommateur de mémoire (70 Méga-octets).

Martin Henz et al ([56]) proposent COMPOzE en 1996, basé sur Oz ([92]), pour construire des progressions harmoniques, *ie* des suites d'accords, à partir de **chiffres**<sup>1</sup> donnés. Les contraintes, par exemple "pas de croisement de voix", sont conservées dans un store sur lequel des propagateurs effectuent les réductions de domaines. Après les réductions de domaines, une variable arbitraire est instanciée, et le procédé est répété jusqu'à l'instanciation de toutes les variables. Selon les auteurs, les résultats sont "simplistes dans le style" à cause de la rigidité de la structure musicale, mais "agréables à entendre".

Philippe Ballesta a réalisé un système basé sur Ilog-Solver [14]. Il construit une représenta-

---

<sup>1</sup>Voir chiffre dans le glossaire en annexe

tion complexe de la note en musique tonale, définie par des attributs, partiellement redondants, tels que la hauteur tonale, l'altération, l'indice d'octave... (ce qui multiplie le nombre de variables). Des contraintes structurelles assurent la cohérence de la représentation. Ce choix de représentation est particulièrement intéressant parce qu'ici les contraintes sont utilisées au niveau même de la représentation musicale, exploitant le fait que la musique est constituée de notions partielles (altération, hauteur, etc). Les contraintes harmoniques sont ensuite définies dans Ilog-Solver sur ces paramètres, avec des techniques de filtrage *ad hoc*. La représentation entraîne une prolifération des variables, et complique l'expression des contraintes. La résolution prend quelques minutes sur SPARC-10.

De même, François Pachet et Pierre Roy (Sony CSL) ont traité le cas de l'harmonisation classique [80]. Implémenté en Backtalk, leur solveur comporte une application musicale pour l'harmonisation automatique, en utilisant les structures musicales propres à la musique tonale. Ce travail s'inscrit dans un cadre plus large de traitement des contraintes sur des objets, dont le principe général est d'effectuer les réductions de domaines à plusieurs niveaux (intra et inter objet). Dans le cas de la musique tonale, les objets choisis sont les accords, dont les chiffres sont restreints ce qui permet un premier filtrage. D'autres contraintes "internes" (portant sur les notes) sont implémentées pour représenter les règles classiques. Cette modélisation à deux niveaux permet de réduire considérablement la combinatoire du problème. Pour une mélodie à une dizaine de notes comme la première phrase de la Marseillaise, le système de Pachet et Roy donne la première solution en moins d'une seconde sur PC Pentium 300 MHz.

La plupart de ces systèmes ont été validés et se comparent dans le cadre d'un problème très précis, l'harmonisation automatique, le plus souvent réduit au style des chorals de Bach. Ce problème a d'ailleurs beaucoup de solutions, voir [41]. Lorsqu'ils proposent un framework plus général, il est lié à la musique tonale. Bien que situés en composition, ces travaux ne concernent donc finalement pas notre sujet, essentiellement parce qu'ils consistent souvent à trouver une modélisation des structures de la musique tonale pour représenter le problème et le résoudre efficacement, qu'il s'agisse des degrés, des accords, etc. Ces représentations ne sont bien sûr pas adaptées à la musique contemporaine.

## 10.2 En contraintes et musique, divers

Il existe également des systèmes utilisant la PLC pour la musique, hors du domaine de la composition. Musicspace, d'Olivier Delerue [78], est un environnement de contrôle de la **spatialisation**<sup>2</sup>. Une fenêtre représente l'espace sonore, dans laquelle l'utilisateur spécifie un point d'écoute  $P$  et des sources sonores  $(S_1 \dots S_n)$ . Les contraintes portent par exemple sur la distance entre  $P$  et  $(S_1 \dots S_n)$ , on peut ainsi imposer une distance minimale entre  $P$  et un  $S_i$ , faire bouger les sources sonores dans des plages définies, les tourner *mute* dans certaines conditions, etc. L'utilisateur définit visuellement les contraintes directement dans la fenêtre. L'ensemble est interactif, et l'utilisateur peut à tout moment rajouter, modifier ou enlever des contraintes. Celles-ci sont toujours géométriques. Pour ces deux raisons Olivier Delerue a développé son propre solveur de contraintes. Musicspace est bien sûr adapté à un usage très précis, mais a la particularité d'être destiné à un utilisateur musicien (ou, en tous cas, de poser le problème d'un système de contraintes musicales destiné à des utilisateurs et non à leur seul concepteur). De ce point de vue, ces travaux apportent la définition visuelle des contraintes et la résolution dynamique avec visualisation simultanée des résultats.

<sup>2</sup>Voir spatialisation dans le glossaire en annexe

Camilo Rueda et Franck Valencia [91] utilisent un calcul concurrent façon  $\pi$ -calcul. L'idée générale est de partir d'un calcul formellement défini pour représenter des structures musicales. Le  $\pi$ -calcul est une extension du  $\lambda$ -calcul qui permet de représenter la concurrence. Un élément de  $\pi$ -calcul est défini par plusieurs processus qui disposent d'un ensemble de canaux pour communiquer. Ils peuvent envoyer et recevoir des messages sur les canaux (les messages pouvant être eux-mêmes des noms de canaux), et se synchroniser. Il existe plusieurs variantes du  $\pi$ -calcul, Rueda et Valencia partent d'un modèle *tcc* (voir [104]) pour construire le *ntcc-calculus* (pour non-deterministic temporal concurrent calculus). L'une des forces du  $\pi$  est que l'on peut à partir d'un process définir une formule de logique temporelle (du type, par exemple, "à partir d'un certain temps, ce process stagne"), et la vérifier automatiquement.

L'une des particularités de ces travaux est le choix des auteurs d'une approche qu'ils appellent "de bas niveau". Ils constatent que les systèmes informatiques destinés à la composition sont en général des adaptations de systèmes informatique existants. Ils préfèrent pour leur part partir non pas d'une implémentation existante, mais d'un modèle formel bien défini, qu'ils adaptent ensuite à la musique.

Ainsi, le  $\pi$ -calcul pourrait naturellement représenter plusieurs musiciens jouant simultanément. Rueda et Valencia donnent l'exemple d'un orchestre dont les solistes auraient une certaine liberté d'improvisation. Ces musiciens peuvent aussi se synchroniser ou bien obéir à un process "chef d'orchestre". La logique temporelle sert à définir des propriétés de terminaison ou de synchronisation qu'il est possible de vérifier automatiquement sur l'orchestre.

Zils et Pachet [81] ont adapté l'algorithme de recherche adaptative (voir ci-dessous) pour faire du "musical mosaicing". A partir d'une base de données de samples, il s'agit de recomposer un morceau donné. Les contraintes portent sur des descripteurs du son, le pitch moyen, le puissance, la percussivité et le timbre global. Elles servent par exemple à imposer certaines continuités.

En utilisant des contraintes pour maintenir dynamiquement la cohérence d'une structure musicale, Anthony Beurivé et Myriam Desainte-Catherine [20] ont réalisé BOXES. Dans ce formalisme, ils définissent des structures musicales temporelles, définies par exemple par un temps d'attaque, un temps de fin et une durée. Une telle boîte peut par exemple contenir un fichier son. Ces structures sont organisées hiérarchiquement, avec comme règle que les enfants sont nécessairement contenus dans les parents (temporellement), ce qui est exprimé grâce à des contraintes. Quand on modifie l'une de ces structures, une première étape consiste à calculer les nouvelles valeurs des différents paramètres (attaque, etc), et une deuxième à vérifier la cohérence. Si celle-ci n'est plus satisfaite, le système résout le problème de contraintes en minimisant un coût donné, par exemple "respecter les durées".

## 10.3 En Composition Assistée par Ordinateur

Les liens entre la musique et la programmation par contraintes datent de fait de plusieurs années. L'utilisation de contraintes en CAO est assez naturelle, puisqu'une contrainte correspond intuitivement à la notion de règle musicale. Des modélisations et résolutions de problèmes musicaux à l'aide de contraintes ont déjà été effectuées, essentiellement dans le domaine de l'harmonisation automatique qui est un problème fortement combinatoire. En musique contemporaine, quelques compositeurs ont déjà utilisé la PLC dans les années 1990 / 2000, notamment Magnus Lindberg (contraintes contrepuntiques), Antoine Bonnet (contraintes harmoniques), Georges Bloch (contraintes compliquées).

L'IRCAM propose déjà dans OM deux solvers de contraintes, Situation et PWConstraints [90].

### Situation

Situation a été conçu par Camilo Rueda en collaboration avec le compositeur Antoine Bonnet [89]. C'est un moteur de contraintes par first-found forward checking, muni d'une interface graphique en OM. Le système est conçu de manière géométrique plane, pour tenir compte des dimensions verticales (harmonie) et horizontales (temporelles) de la partition. L'une des originalités de Situation est la représentation par objets. Un objet est un ensemble de points et de distances. Cela permet d'exprimer à la fois des rythmes ou des accords. Les contraintes portent sur les distances internes (entre points d'un même objet) ou externes (entre des points appartenant à des objets différents).

Situation est efficace mais présente plusieurs inconvénients. En premier lieu, le contrôle visuel du solver est limité. L'écriture des contraintes reste un exercice difficile de par la syntaxe ésotérique du langage. Situation ne permet pas d'approcher une solution (dans le cas d'un problème surcontraint par exemple). Enfin, il gère mal les contraintes globales. Il a essentiellement été validé dans le domaine harmonique.

### PWConstraints

PWConstraints (PWCS) a été conçu par Mikaël Laurson [62]. Le moteur de résolution utilise les algorithmes classiques de forward checking et back-jumping. PWCS travaille sur des séquences, et l'interface de programmation est optimisée pour l'expression de règles sur ces séquences. Les contraintes s'écrivent dans une syntaxe propre, autorisant le pattern-matching. Les règles sont ensuite ré-écrites sous forme de prédicats Lisp. L'une des particularités de PWCS est de distinguer les règles standard des règles "heuristiques". Ces dernières sont ré-écrites non pas comme des prédicats, mais comme des fonctions renvoyant une valeur numérique. Elles ne filtrent pas les résultats, mais expriment des préférences. Les solutions dont la valeur selon ces règles est la meilleure sont favorisées par le moteur. Mikaël Laurson et Mika Kuuskankare [63] ont plus récemment étendu PWConstraints pour la composition instrumentale, par exemple pour donner des doigtés de guitare et de cuivres.

Ces règles heuristiques approchent l'idée de solution approchée, mais leur utilisation dans une stratégie de backtracking n'est pas toujours très heureuse. Par ailleurs, le concept de séquence permet de décrire beaucoup de problèmes musicaux, mais pas tous. Enfin, PWCS traite mal les contraintes globales et l'aspect visuel est inexistant.

## 10.4 Conclusion, la place de la programmation par contraintes

Un bibliographie de la programmation par contraintes en musique ressemble en fait à un panorama tous azimuts, de par la diversité des applications. Tel système porte sur la spatia-lisation, tel autre sur l'organisation temporelle de la partition, en passant par l'harmonisation automatique bien sûr ou le mosaïcing. La diversité des applications est à rapprocher de la diversité des utilisations de l'ordinateur en musique.

En CAO en revanche, on a deux systèmes (trois en comptant BOXES) dont on peut tirer des enseignements. Tous deux ont les avantages de leurs inconvénients : ils sont efficaces. Adaptés à des problèmes précis, ils ont été conçus pour des structures musicales très particulières. Le

langage de contraintes est donc basé sur des structures de données très déterminées. En théorie, il est possible de traiter un large panel de structures, mais en pratique cela suppose quelques contorsions qu'un musicien non informaticien aurait du mal à maîtriser.

Dernier point et non des moindres, peut-on de cette bibliographie attribuer à la PLC une place particulière dans la musique ? Hormis pour l'hamonisation automatique, non. En réalité, les applications présentées ici sont à chaque fois au croisement de la PLC et de nécessités musicales (doigtés, création d'une mélodie, représentation temporelle de la musique...). Souvent, les contraintes ne sont utilisées que comme un algorithme parmi d'autres. En outre, la majorité des systèmes sont utilisés par leurs inventeurs, et pas ouverts à un ensemble d'utilisateurs.



## Deuxième partie

### Modélisation des CSPs musicaux



# Chapitre 11

## Summary of Part II : Musical CSPs modelization

This chapter details some musical constraint satisfaction problems. The first 9 problems are pure CAC problems, which means they have been proposed by composers or musical assistants, except for problems 3 and 9 which are classical musical problems. Problems 10, 11 and 12 come from musical analysis, from works by Marc Chemillier. Problem 13 is a CAC problem, but cannot be easily compared to the other ones. Problem 14 is an instrumental problem, which deals with physiological constraints. We have chosen to classify the CSPs according to their musical nature.

**Harmony** There are three problems dealing with harmonical structures. Problem 1 has been stated by Fabien Levy, composer. It concerns  $n$  spectral chords, which are modeled by three parameters : a virtual fundamental, the interval between two frequencies, and a number of notes. The goal is to find some of these chords such that two successive chords have a fixed number of common notes (optionnally, with the a number of common notes comprised between fixed minimum and maximum / with the same common notes on the whole sequence ). In addition, there are optional constraints ruling the behaviour of the sequence (increasing or decreasing intervals, increasing or decreasing fundamentals). Figure 13.1 gives an example with exactly two common notes, figure 13.2 an example with an increasing fundamentals and decreasing intervals, and figure 13.3 an example with the same common note for the whole sequence.

Problem 2 is rather complex. It has been stated by the composer Georges Bloch. Given a particular rhythmical structure of 6 or 12 voices, the goal is to find a harmony which minimizes a musical distance, the so-called Estrada distance, between two successive chords. Estrada distance measures the differences in the intervals between two chords. A second constraint consist in minimizing a distance (corresponding to the cycle of fifths) between the virtual fundamental of two successive chords. A third constraint is better expressed as a preference, and states that the melodies formed on each voice have to be near to a fixed melodic profile. Finally, a constraint is added in order to avoid trivial solutions (same note everywhere). This problem is overconstrained.

Problem 3 consists in sorting a sequence of chords, such that two successive chords have the maximal number of common notes. As a CSP, this is exactly a Travelling Salesman Problem. Take the cities as the chords, and the distance between two cities as the number of common notes.

It is worth noticing that these three problems have quite similar constraints : minimize a

distance-like function between two successive chords. From this point of view, these problems are very similar. But we see here the main challenge of applying constraint techniques to musical representation : although these three problems are stated on similar structures (sequence of chords), their representations are very different : sets of frequencies, a very complicated structure inherited from some rhythmical properties, and a set of integers.

**Rhythms** Four problems concern rhythmical structures. Problem 4 has been stated by the composer Mauro Lanza. The goal is to find  $n$  rhythmical patterns of fixed lengths, each pattern played on one voice, such that two different voices never play an onset simultaneously for a fixed duration. We can use the Chinese Remainder Theorem to reduce this problem to an array of integer equations, which gives a characterization of the number of solutions : there is no solution unless the greatest common divisor between the lengths of any two voices is greater than the duration.

Problem 5 has been stated by the composer Geoffroy Drouin. It relies on the famous Fibonacci series defined as  $u_0 = 0$ ,  $u_1 = 1$ ,  $u_n = u_{n-1} + u_{n-2}$ . The goal is to find durations which are values of this series, two successive durations being next to each other in the series, and the sum of all the durations being fixed. There are two modelizations for this problem, either taking the variables as the durations, or the variables ranging over  $\{-1, 1\}$ , representing the back or forward move in the indexes of the series.

Problem 6 has been stated by the composer Peter Klanac. He wanted to write a smooth accelerando, but still playable by the interpret, who usually reads the new tempo on the score as a ratio with the current tempo ( $\text{♩} = \text{♩}$ ). Variables are tempi  $t_1 \dots t_n$ , ranging over the integer set from 40 to 250. Constraints rule the ratios between  $t_i$  and  $t_{i+1}$ ,  $t_{i+2}$ ,  $t_{i+3}$ , and  $t_{i+4}$ . The ratios have to be as near as possible of fixed values (0,9, 0,8, 0,75 and 0,666). This problem would be seen as a geometrical series by a mathematician, but it is not, because we have to stay in the integer set. This problem has then no solution, and the goal is to find approximate solutions. Concerning the optimization options, Peter Klanac decided to optimize, in the first place, the number of exact ratios on  $t_i$  and  $t_{i+3}$  (count the number of times when  $t_i/t_{i+1} = 0.75$ , and then to minimize the gap between minimum and maximum of  $t_{i+1}/t_i$  and 0.9. It is important to notice that the constraints are written exactly in the same way, but the notion of approximate solutions are not the same for the composer.

Problem 7 has been proposed by Gilbert Nouno, for a piece with Steve Coleman. It has not been used because of real time requirements. The data are a set of integer durations, which come from a rhythm played on congas by a percussionnist. This durations correspond to a symbolic rhythm, but they may differ from the exact durations. The goal is to find the symbolic rhythm which fits the played duration best.

The observation of these four problems does not allow much deduction : all of them deal with rhythms, but they have very little in common. The musical structures are definitely not the same : patterns, tempi, played durations. We can notice that the question of representing rhythms in CAC is usually a difficult problem. Rhythm is a notion shared by all musicians, but this notion is very contextual, and thus their representations depends on the musical context.

**Melodies** Two problems concern melodies, in the sense of horizontal sequences of notes. Problem 8 has been submitted by Gilbert Nouno, musical assistant of Michael Jarrel. It is about musical gestures, a gesture being either an interval or a short series of intervals (the domains are closed under the opposite sign). A first note is fixed, to which the gestures are

added in order to form a melody. The goal is to find a series of these gestures, such that the resulting notes are in a fixed set. There are two possible modelisations, with the variables as gestures and constraints on notes or with variables as notes and constraints on gestures. The first one is better because of the musical purpose, which is to find gestures, and because we will search for approximate solutions. There are three other constraints. Firstly, in order to avoid trivial solution, we impose that two successive gestures are neither the same nor opposite. Secondly, the percentages of some particular values for gestures may be fixed. Finally, the whole sequence has to correspond to melodic profile, given as a curve, but this can be passed as a domain reduction.

Problem 9 is a well known problem, the all-intervals series. The goal is to find a musical series (permutation of the 12 notes of the chromatic scale, but it can be generalized to a permutation of the  $n$  first integers), such that the intervals heard when the series is played are all different. There is a trivial solution 1 12 2 11 3, and of course we want non-trivial solutions.

These two problems are very interesting, representing the two extremes of musical CSPs : one is academic, well defined, difficult, and the existence of solutions is known. The other one is and looks "real", the modelisation is not obvious, it has many optional constraints, and the number of solutions varies from many to zero. Still, they share the same property to have two possible modelizations, one with notes and one with intervals, modelisations equivalent if we search for exact solutions, but not for approximate solutions.

**Musical analysis** There are three problems, from works with Marc Chemillier, dealing with musical analysis issues. Problem 10 concerns a musical skeleton found by Chemillier in a piece of Ligeti, Melodien (and also in Continuum). A long part of Melodien shows some aggregates which are moving slightly. The analysis found an harmonical structure with some properties, from which the aggregate sequence can be extracted. Here, the variables are the notes of the underlying harmony, and constraints are used to restrict the melodic motions of these notes. Some of these notes are really played in the score, but not all of them.

Problem 11 concerns a very particular property found by Simha Arom in Aka pygmies music in Central Africa, and studied by Chemillier. This is called "rhythmical imparity", and characterized rhythmical series of groups of 2 and 3 time units, played repetitively. Such a sequence is rhythmically impair iff it is impossible to cut it into two equal pieces, see figure 16.3. This is easy to model as a CSP, though enumerating the solutions is not obvious if not by running a CSP program.

Problem 12 is a model given by Chemillier of some musical structures in Nzakara music, in Central Africa. Nzakara people play some vocal pieces accompagnied by a traditional harp, who repeats a canon-like sequence in an *obstinato*. This canon respects some rules : they have two voices, never play the same chords twice in a row, last a fixed duration, and the lower voice is transposed from the upper one, apart from a few notes. Chemillier showed that the number of errors was determined by the duration and the gap of the canon. Here the variables are again the chords, and constraints are the same as mentionned.

The idea of modelling a musical analysis, provided it is expressed in a mathematical-like language, as constraints, can seem straightforward. A musical analysis consists in giving *a posteriori* some rules that the piece respects. Here, there is an interesting parallel between constraint programming and such musical modelling. We could argue that the number of solutions to the CSPs is in some way representative of the accuracy of the analysis : if the CSPs has no solution, the analysis is false, if it has many, then the analysis can apply to many other pieces than the target one, and is not very specific to a particular score, then not very accurate. If the CSPs

has one or a few solutions, then we can argue the analysis is quite specific to the score (see the analysis of Riotte and Mesnage [69], [88]). To go a little further, we can add that the number of constraints gives an indication of the analysis' interest. The limit case would be to modelize a score by simply enumerating its content (constraints such as "first note is this", etc). The less constraints there are, the more high level is the analysis, in some way. it was not our goal to work on musical analysis, by lack of competences, and we do not pretend to have found a universal way of doing analysis, which is a work far too subtle to be represented in such a way. Those remarks are about a particular work, where the parallelism between both representations (CSPs and analysis) applies well.

**Others** Problem 13 deals with enumerating musical scales in quarter tones. The whole set of these is far too big to listen to, and some constraints are added here in order to restrict the enumeration of these scales to the ones of musical interest. The most interesting constraint we have tried is probably the one which forces the scale to have a least number of chords "sounding" in quarter-tones, that is, in which the inner intervals do not reduce as semitones intervals.

Problem 14 is about fingerings for guitar. There are two CSPs, the first one to find fingerings for a single chord for the guitar, the second one to find fingerings for a chord sequence, such that for instance the number of position's changes is minimal. This problem is difficult to formalize because of the great number of parameters, due to the interprets' features (size of the hand, barré or not , etc).

**Special features of these musical CSPs** As a preliminary remark, let us stress the fact that most of these CSPs have been given by composers, or musicians. This confirms the idea that constraint representations has a role to play in music. Main features of the CSPs are summarized tables 18.1 and 18.2. As mentionned, these CSPs show a great diversity both in their constraints and in their musical structures. Every kind of classical constraints primitives, from equality to capacity, appears. Musical structures ranges over the whole set of object which can be found in a score, from notes to tempi, and even out of the score for problems 7 and 14. This is of course due to the fact that every composer thinks with its own musical structures. We can also notice that the variables are often not exactly taken on the score, but above it, as some integers which represent a feature of a musical object (typically, a MIDI value for a note).

During the modellisation process, we often had to add some constraints to ensure that the solutions would not be degenerated (chords with only one note for instance). This is why there is an alldifferent constraint in nearly every CSP.

More important, the goal is not always to find a solution, which may seem unusual to a constraint researcher. Nor is it to answer anything. It is to provide interesting instantiations to the composer. Firstly because most of the CSPs are overconstrained, and have no solutions at all. Secondly, it happens that an approximate solution is musically more interesting than an exact one. We shall never forget the exact place of constraint solving in the compositional process : it is a mere tool, for the composer. It is used at the rough draft stage of the composition, not only because it is CAC, but also because beeing a solution doesn't guarantee beeing musically good. In particular, the solutions are nearly always re-written by hand. So the usual CSP paradigm "problem  $\rightarrow$  solution" is no longer valid, we'd rather consider it as "specify some partial information on musical objects  $\rightarrow$  make suggestions to the composer".

# Chapitre 12

## Introduction, le problème du classement

Dans ce chapitre sont présentés les problèmes de contraintes musicaux étudiés. Y figurent seulement la modélisation, les énoncés, et parfois des commentaires sur la nature du problème (représentation musicale, dénombrement des solutions). Les résolutions seront détaillées chapitre 3.

Les problèmes 1 à 9 sont des problèmes de CAO, c'est-à-dire qu'ils ont été posés par des compositeurs ou assistants musicaux (voir chapitre 1), et utilisés par ces derniers, sauf les problèmes 3 et 9 qui sont des problèmes musicaux classiques. Les problèmes 10, 11 et 12 sont des problèmes d'analyse musicale dus à Marc Chemillier. Le problème 13 est un problème de CAO qui se rattache pourtant difficilement aux autres. Le problème 14 est un problème instrumental avec des contraintes "physiologiques", posé par un compositeur.

A la fois par commodité de lecture, et par souci de leur trouver des caractéristiques communes, ces problèmes sont classés en cinq catégories. Nous avons choisi un classement selon des critères musicaux, nécessairement réducteurs : par exemple, le problème des séries tous intervalles n'est pas à proprement parler un problème mélodique, le problème des durées dans Fibonacci pourrait aussi bien se transposer à des hauteurs qu'à des rythmes, etc. Mais ce classement respecte la volonté initiale du compositeur. Ainsi si ses premiers mots sont "j'ai un problème sur des rythmes", nous considérons avoir affaire à un problème de rythmes même si le CSP finalisé peut représenter aussi bien des accords. Il est important de se tenir à la représentation du compositeur, qui sera l'utilisateur du système. On aurait pu aussi classer en fonction des contraintes utilisées, ce qui aurait été sans doute plus informatique mais on y aurait perdu la spécificité CAO de l'étude.

Ce classement est instructif dans la mesure où il montre bien la difficulté de représentation musicale que l'on rencontre systématiquement. Parfois des points communs apparaissent dans les structures musicales utilisées, et donc dans les modélisations des problèmes, parfois, plusieurs problèmes, semblant de prime abord apparentés, ne peuvent pas du tout se formuler de la même manière.

La plupart de ces problèmes ont été formulés et modélisés plusieurs fois, et souvent le compositeur souhaitait pouvoir disposer de plusieurs variantes (par exemple en enlevant des contraintes). On donne ici la forme stable, celle qui a été utilisée, et parfois les variantes.

Remarque : le terme "mélodique" est utilisé dans un sens abusif pour faire référence à l'organisation horizontale de la musique.



# Chapitre 13

## Problèmes portant sur l'harmonie

### 13.1 Problème 1 : Suite d'accords avec note commune

Le compositeur Fabien Lévy a proposé un CSP portant sur une suite d'accords fréquentiels<sup>1</sup>. Ces accords ont une structure particulière : dans un même accord toutes les notes doivent être équidistantes en fréquence, formant un spectre inharmonique. Les contraintes portent sur le nombre de notes communes entre deux accords successifs, avec plusieurs variantes : exactement une note commune d'un accord à l'autre, au minimum une, voire un encadrement du nombre de notes communes. Une solution est montrée figure 13.2.

Ce qui se traduit formellement : soit  $I$  et  $J$  deux entiers. La suite d'accords est représentée par une liste de  $I$  listes de  $J$  entiers, soit  $(\alpha_i^j)_{j \leq J, i \leq I}$ . Les contraintes s'écrivent :

- Accords en fréquence  $\forall i \leq I, \forall j_1, j_2 \leq J - 1, \alpha_i^{j_1+1} - \alpha_i^{j_1} = \alpha_i^{j_2+1} - \alpha_i^{j_2}$
- Une note commune au moins  $\forall i \leq I - 1, \alpha_i \cap \alpha_{i+1} \neq \emptyset$

La première contrainte se réduit par un changement simple de paramètres, en prenant comme représentation de l'accord non pas l'ensemble de ses notes mais sa fondamentale  $f_i$ , l'intervalle fréquentiel  $t_i$  entre deux notes, et  $J$  le nombre de notes par accord. On a alors  $\alpha_i^j = f_i + t_i * j$  pour  $j \leq J$ .

La deuxième contrainte doit être précisée. D'abord, on ne veut pas d'une suite d'accords constants, qui pourtant satisfont cette contrainte. Par ailleurs, Fabien Lévy souhaitait aussi régler le nombre de notes communes souhaitées d'un accord à l'autre. En notant  $C_i = \#(\alpha_i \cap \alpha_{i+1})$ , la contrainte s'écrit comme un encadrement de  $C_i$  entre deux paramètres fixés  $m$  et  $M$  :  $\forall i \leq I - 1, m \leq C_i \leq M$ . Selon les valeurs de  $n$  et  $M$ , on obtient tous les cas de figures, par exemple un nombre de notes communes fixé sur toute la séquence avec  $m = M$ , ou une garantie que les accords soient différents avec  $m < J$ . Une solution avec  $m = M = 2$  est montrée figure 13.1.

Enfin, le compositeur souhaitait pouvoir disposer de deux contraintes supplémentaires régissant le mouvement soit de la fondamentale, soit des intervalles (croissants ou décroissants). Ces deux contraintes sont moins importantes que les précédentes, et peuvent être violées. Ce qui s'écrit simplement dans la représentation en fréquences :

Intervalles croissants  $t_1 < \dots < t_I$

Fondamentales croissantes  $f_1 < \dots < f_I$

On peut encore varier le CSP en imposant la même note commune sur l'ensemble de la

---

<sup>1</sup>Un problème similaire a été évoqué par un autre compositeur, Kilian Sprotte, mais cela n'a pas donné suite à une modélisation.

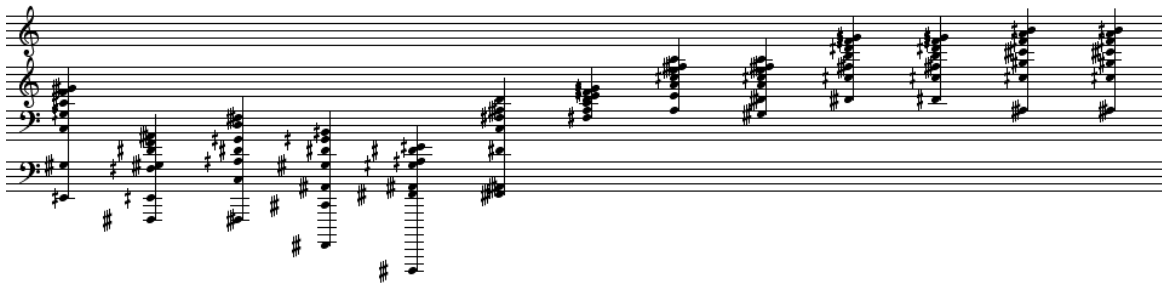


FIG. 13.1 – Une solution avec exactement deux notes communes d'un accord au suivant.

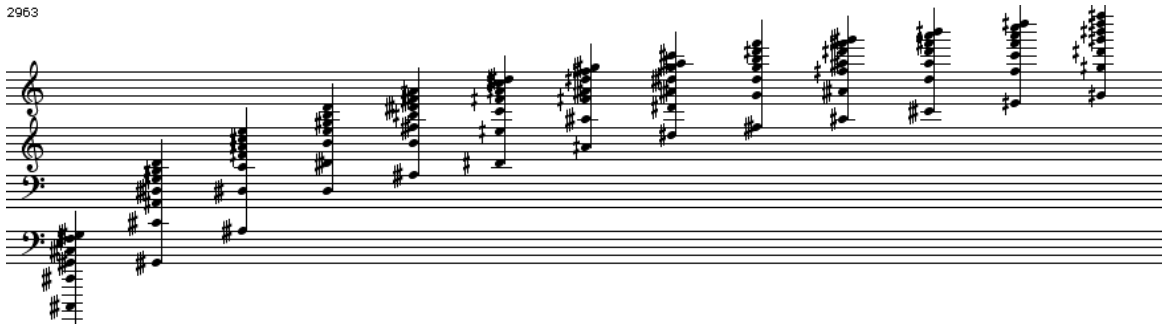


FIG. 13.2 – Une solution avec deux contraintes sur le mouvement global : fondamentale croissante et intervalles décroissants.

séquence, comme montré figure 13.3



FIG. 13.3 – Une solution avec note commune sur tout la séquence.

### Modélisation, résumé

**Données**  $I$  et  $J$  entiers.  $m$  et  $M$  entiers.

**Variables**  $t_1 \dots t_n$  (des intervalles),  $f_1 \dots f_n$  (des fondamentales)

**Domaines** entiers finis

**Contraintes** Soit  $\alpha_i = \{f_i + t_i * j, j \leq J\}$ .

$$m \leq \#(\alpha_i \cap \alpha_{i+1}) \leq M$$

$\mathcal{C}_1$ (option)  $t_i < t_{i+1}$

$$C_3 \text{ (option) } f_i < f_{i+1}$$

## 13.2 Problème 2 : Harmonisation à structure rythmique donnée

Ce problème est dû au compositeur Georges Bloch. On suppose donnée une structure de canon rythmique telle que décrite dans [6], appelée canon rythmique régulier complémentaire de catégorie maximale. Nous ne donnerons ici qu'une description rapide de cette construction.

On suppose donnée une **pulsation**<sup>2</sup>, les attaques (impulsions) se produisant sur des multiples entiers de cette pulsation. Les voix jouent toutes, en boucle, le même motif rythmique. Lorsqu'elles sont toutes entrées, la superposition de toutes les voix forme une pulsation régulière. Autrement dit, pour chaque pulsation, on entend une attaque et une seule. La figure 13.4 montre les entrées pour un canon à 6 voix. Il n'est possible d'avoir de tels canons non triviaux que pour certaines périodes (la plus petite étant 72) et certains nombres de voix (le plus petit étant 6), ainsi que l'a montré le mathématicien Dan Tudor Vuza (voir [107]). La partie qui nous intéresse dans ce CSP est le régime permanent du canon, quand toutes les voix sont entrées et jouent répétitivement leur motif.

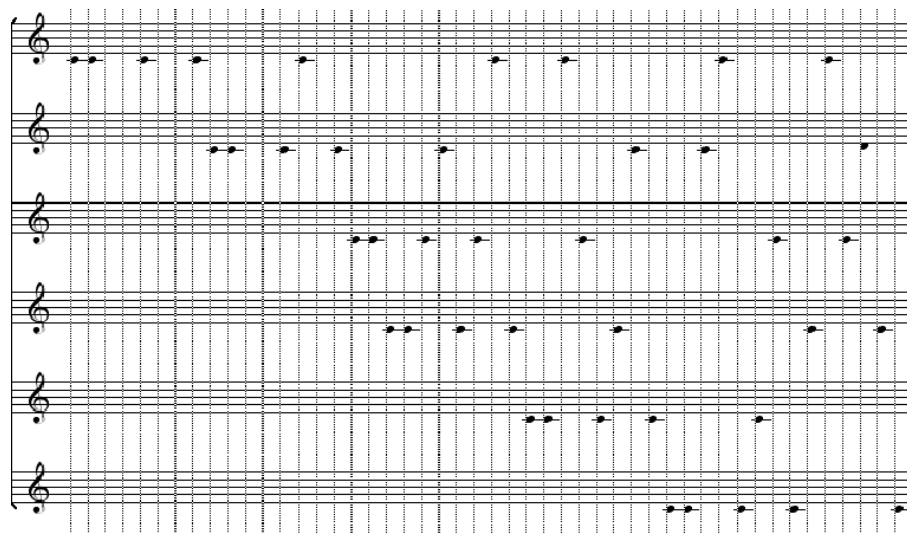


FIG. 13.4 – L'entrée des voix dans un canon rythmique.

Le problème consiste à harmoniser cette structure rythmique. On suppose que chaque note est tenue jusqu'à l'occurrence de la suivante sur la même voix. Ainsi, à chaque pulsation, une et une seule note change dans l'agrégat formé verticalement par toutes les voix. L'harmonie cherchée doit minimiser deux distances musicales, l'une portant sur les intervalles de chaque accord, l'autre sur leur **fondamentale virtuelle**<sup>3</sup>.

On notera  $k$  le nombre de voix, et  $n$  la période du canon. Pour l'écriture du CSP, il n'est pas utile de prendre comme variables toutes les notes jouées puisque l'on sait qu'une seule voix change à chaque pulsation. Il suffit de  $n$  variables  $V_1 \dots V_n$ .

$V_i$  représente la note qui change sur le  $i$ -ième temps.

<sup>2</sup>Voir pulsation dans le glossaire en annexe

<sup>3</sup>Voir fondamentale virtuelle dans le glossaire en annexe

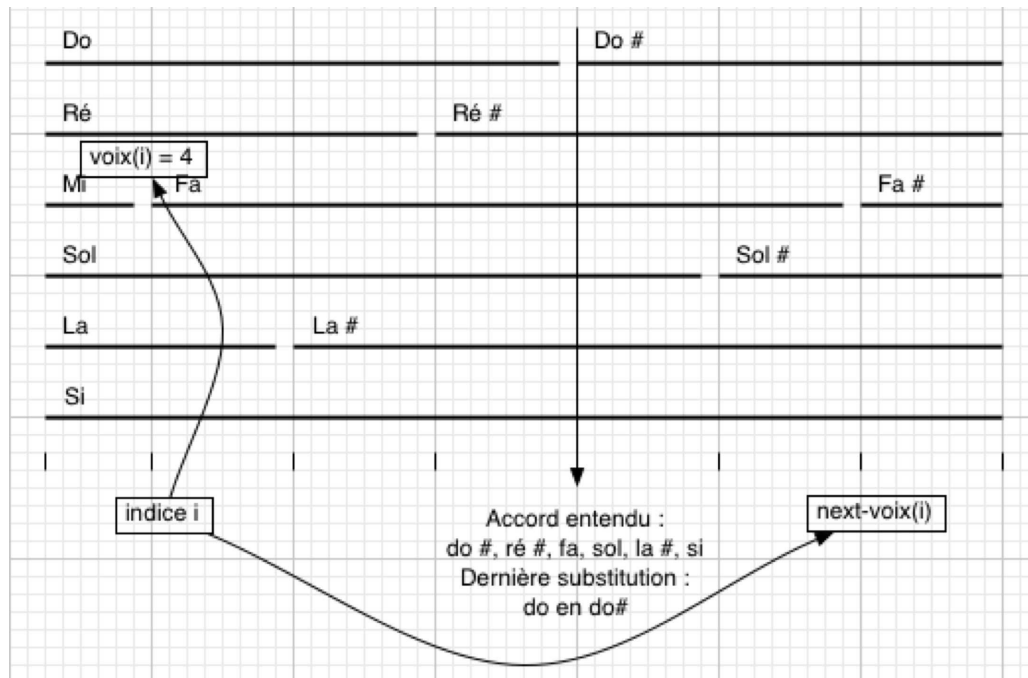


FIG. 13.5 – En régime permanent, l'accord entendu sur un temps est constitué de notes tenues, et d'une note exactement qui vient de changer. Pour un indice temporel  $i$ , les fonctions  $prev - voix(i)$  et  $next - voix(i)$  donnent les indices des changements précédent et suivant sur la voix  $voix(i)$ .

Avec cette représentation, chaque accord est constitué de  $V_i$ , plus les notes jouées par les autres voix, voir figure 13.5. Ces notes sont tenues depuis les pulsations précédentes. Ce sont des variables dont les indices ne sont pas très simple à récupérer. Pour cela, on suppose donnés pour  $i \leq n$  un indice variant dans les pulsations :

- une fonction  $voix$  à valeur dans les indices des voix,  $voix(i)$  est l'indice de la voix qui change sur le temps  $i$ .
- deux fonctions  $next - voix$  et  $prev - voix$  à valeur dans les indices de pulsations.  $next - voix(i)$  donne l'indice de la pulsation sur lequel  $voix(i)$  change à nouveau, ie le changement suivant de  $voix(i)$ , et  $prev - voix(i)$  le changement précédent.
- les décalages temporels de chaque voix à partir d'un temps quelconque, notés  $d_i$ . Il suffit par exemple de prendre les temps d'entrée de chaque voix, avec zéro comme temps de référence.

Ces fonctions suffisent pour représenter les contraintes, et nous pouvons faire abstraction de la structure en canon dans la suite.

**Première contrainte** Avant de poser la première contrainte, Georges Bloch définit une notion de distance entre accords, due à Julio Estrada. On la notera  $E$ . Le mot distance est ici utilisé dans un sens musical et non mathématique. Cette distance ne porte pas directement sur les accords mais sur une donnée intermédiaire, appelée texture. Georges Bloch la définit ainsi :

" Analyse des textures

- (i). Un accord est joué. Les hauteurs constituant cet accord sont rentrées dans une octave. Par exemple entre do4 et do5. On remarque que l'on perd ainsi la notion de **renversement**

4.

- (ii). Les intervalles successifs sont analysés, et notés sous forme de nombre de demi-tons.
- (iii). Un dernier intervalle est ajouté, pour que la somme soit égale à 12 (on remarquera que c'est a priori redondant).
- (iv). Les intervalles sont classés par ordre croissant.

Exemple : l'accord est sol#1, do#7, fa8. On trouve do#-fa-sol#, soit 4 (do#-fa), 3 (fa-sol#), auquel on ajoute 5 (pour trouver la somme 12). Le résultat est la texture 3-4-5. Cette analyse donnerait le même résultat pour un accord parfait mineur, et pour tous les renversements d'accords majeurs ou mineurs de toutes les **tonalités** <sup>5</sup>. La notation musicale utilisée consiste à noter la texture par ordre d'intervalle croissant en commençant par le do du milieu du piano. Par exemple la texture 1-1-3-7 se note do-do#-ré-fa. Cela ne signifie nullement que la texture réelle comporte ces notes, où même les intervalles dans cet ordre." A noter, la distance Estrada fonctionne aussi pour des micro-intervalles. Pour un tempérament donné, le nombre total de notes modulo l'octave est fini, donc le nombre de textures possibles est fini. Par exemple, dans le cas habituel d'une gamme en demi-tons, il est égal à 76. Ce système est donc une des classifications possibles de tous les accords pour un **tempérament** <sup>6</sup> donné.

La distance Estrada est calculée à partir de ces textures. Dans ce CSP, le nombre de notes par accord est constant ce qui simplifie la définition de la distance, qui compte le nombre d'intervalles différents entre les textures :

$$E(Tex_1, Tex_2) = (\text{nombre de notes de l'accord}) \\ - (\text{nombre d'intervalles communs entre } Tex_1 \text{ et } Tex_2) \\ - 1.$$

Par convention  $-1$  est ramené à 0. Parenthèse mathématique, notons que  $E$  ne vérifie pas l'inégalité triangulaire. Comme ce sera souvent le cas, le mot distance a ici un sens musical et constitue une mesure de la similarité perceptive.

La première contrainte consiste à minimiser cette distance entre deux accords successifs de l'harmonie. Pour deux accords de  $n$  notes ayant  $n - 1$  ou  $n$  notes communes, cas du canon rythmique, les seuls valeurs possibles de  $E$  sont 0 ou 2.

Simple à exprimer, cette contrainte est lourde à écrire, parce que l'accord joué sur le  $i$ -ième temps est formé de notes entrées précédemment, d'où l'utilité de la fonction *prev - voix*. Elle s'écrit

$$\text{Minimiser } E(\{V_i\} \cup \{V_{prev-voix(j)}, j \neq voix(i)\}, \{V_{i+1}\} \cup \{V_{prev-voix(j)}, j \neq voix(i+1)\})$$

**Variante :** Georges Bloch a aussi utilisé une variante de ce CSP, dont les solutions exactes sont les mêmes. Il s'agit de transformer légèrement la contrainte pour minimiser la distance Estrada non pas entre deux accords successifs, mais par rapport à une texture de référence fixée. Musicalement, cette variante permet de fixer une même texture sur toute la séquence, et dans le cas de solutions approchées, évite d'avoir une texture de départ très différente de la texture finale. Côté contraintes, cela simplifie le CSP.

<sup>4</sup>Voir renversement dans le glossaire en annexe

<sup>5</sup>Voir tonalités dans le glossaire en annexe

<sup>6</sup>Voir tempérament dans le glossaire en annexe

**Deuxième contrainte** La deuxième contrainte porte sur les fondamentales virtuelles des accords. Georges Bloch utilise comme définition de la fondamentale virtuelle, notée  $F$ , d'un accord  $\{N_1 \dots N_k\}$  :

- (i). Si l'accord formé de  $\{N_1 \dots N_k\}$  contient un intervalle de **quinte** <sup>7</sup> entre  $N_i$  et  $N_j$ , avec  $N_i < N_j$ , alors  $N_i$ .
- (ii). Sinon, et si l'accord contient un intervalle de **tierce majeure** <sup>8</sup> entre  $N_i$  et  $N_j$ , avec  $N_i < N_j$ , alors  $N_i$ .
- (iii). Sinon, la note la plus basse de l'accord.

En réalité, le compositeur souhaitait au départ utiliser une fondamentale virtuelle plus réaliste, comme il en existe dans OpenMusic XXXREF-DE-GERARD. Cela a été abandonné pour des raisons pratiques de temps de calcul. On a cependant défini et utilisé trois variantes de la fondamentale virtuelle : quinte seule puis note basse, quinte puis tierce puis note basse (décrite ci-dessus), et quinte puis quarte puis note basse.

A partir de  $F$ , Georges Bloch définit une seconde distance entre deux accords  $A_1$  et  $A_2$ , notée  $D_f$ . Cette distance porte sur l'intervalle entre les fondamentales de deux accords,  $A_1$  et  $A_2$ . Il s'agit de mesurer l'écart entre  $F(A_1)$  et  $F(A_2)$ . Mais la différence  $F(A_1) - F(A_2)$  n'a pas de sens musicalement (à moins de chercher uniquement des solutions exactes *ie*  $F(A_1) = F(A_2)$ , ce qui ne sera pas le cas, voir ci-dessous). En effet, un écart de seconde (un demi-ton) est considéré comme plus dissonant, donc plus loin, d'un écart de quinte (sept demi-ton). On utilisera donc des tables choisies par Georges Bloch pour mesurer l'écart entre les fondamentales virtuelles. Soit  $d = (F(A_1) - F(A_2)) \bmod 12$ . Ci-dessous, une des tables pour lesquelles le problème a été résolu, qui correspond au cycle des quintes et tient compte des données classiques sur la consonnance des intervalles.

|       |   |   |   |   |   |   |   |   |   |   |    |    |
|-------|---|---|---|---|---|---|---|---|---|---|----|----|
| d     | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| $D_f$ | 0 | 9 | 7 | 6 | 5 | 2 | 8 | 1 | 4 | 3 | 11 | 10 |

La deuxième contrainte consiste à minimiser  $D_f$  entre deux accords successifs, ce qui s'écrit, comme précédemment : minimiser  $D_f(\{V_i\} \cup \{V_{prev-voix(j)}, j \neq voix(i)\}, \{V_{i+1}\} \cup \{V_{prev-voix(j)}, j \neq voix(i+1)\})$

**Troisième contrainte** La dernière contrainte donnée par le compositeur porte sur la structure mélodique du problème. Elle consiste à favoriser un profil mélodique donné, le même pour toutes les voix, décalé de la même manière que le canon rythmique. On fixe un profil en valeurs midi  $p_1 \dots p_m$ , avec  $m$  le nombre d'attaques d'une voix. Le profil est donné en intervalles.  $p_k$  est donc l'intervalle du profil sur le  $k$ -ième changement de notes de la voix considérée. Pour écrire la contrainte, on a besoin de savoir sur un temps  $i$ , d'abord l'indice de la voix qui change donné par  $voix(i)$ , et le nombre de fois où cette voix a changé. Pour simplifier, on effectue un changement d'indices dans les  $p_1 \dots p_m$  en mettant à plat toutes les voix en tenant compte des décalages initiaux, et on notera pour chaque pulsation  $i$ ,  $p'(i)$  l'intervalle du profil attendu à cette pulsation  $i$ . Cette contrainte s'écrit

$$\forall i \leq n, V_i - V_{prev-voix(i)} = p'_i$$

Cette définition n'est pas très satisfaisante parce qu'elle ne spécifie pas la distance utilisée, qui est pourtant essentielle musicalement. Pour des solutions exactes, cela n'aurait pas d'importance, mais nous chercherons nécessairement des solutions approchées (voir ci-dessous). Il est

<sup>7</sup>Voir intervalles dans le glossaire en annexe

<sup>8</sup>Voir intervalles dans le glossaire en annexe

plus pertinent musicalement de la remplacer par une contrainte de minimisation, dans laquelle la distance utilisée est la différence en valeur absolue, qui permet de garantir que l'on "colle" au profil le mieux possible.

$$\forall i \leq n, \text{ minimiser } \sum_{j \leq m} |V_j - V_{prev-voix(j)} - p'_j|$$

**Contraintes ajoutées** Voilà un exemple caractéristique des difficultés à formaliser un CSP musical. La contrainte Estrada impose des solutions semi-triviales : pour annuler  $E$  entre deux accords  $A_1$  et  $A_2$ , il faut que  $A_1$  et  $A_2$  aient les mêmes intervalles. Supposons que  $A_1$  et  $A_2$  aient le même nombre de vraies notes (sans répétitions). Si on se place dans un **ambitus**<sup>9</sup> d'un octave de manière à éliminer les **transpositions**<sup>10</sup>, et que l'on conserve l'ordre des intervalles, on peut tracer la texture sur un cercle de douze unités. On voit alors que la contrainte Estrada revient soit à garder la même note, soit à échanger deux intervalles, *ie* deux possibilités pour la note variable. Dans un ambitus plus grand, il faut ajouter les transpositions de ces deux possibilités. La note différente entre  $A_1$  et  $A_2$  est donc quasiment déterminée, et ceci avec une seule des trois contraintes. En pratique, on obtient souvent des solutions constantes (la même note partout) ou des transposition à l'octave, ce qui n'a pas d'intérêt.

Il faut donc ajouter des contraintes évitant les solutions triviales et chercher des solutions approchées. On peut réclamer soit une "vraie" mélodie, avec les notes successives d'une même voix toutes différentes, soit une "vraie" harmonie, avec les  $k$  notes jouées simultanément toutes différentes. La deuxième solution est doublement mauvaise : du côté contraintes, elle surcontraint le problème. Du point de vue musical, elle n'a pas grand sens, par exemple quand il y a 12 voix. A l'inverse, la première solution est doublement bonne. Elle n'impose qu'une contrainte assez faible, et musicalement, elle impose à chaque instrument de jouer une "vraie" mélodie. On ajoute donc la contrainte :

$$V_{prev-voix(i)} \neq V_i$$

## Modélisation, résumé

**Données**  $d_i$  décalages des entrées des voix,  $voix$ ,  $prev - voix$  et  $next - voix$  fonctions pour gérer les indices des voix,  $E$  distance Estrada,  $D_f$  distance entre fondamentales virtuelles,  $p_i$  un profil mélodique en intervalles

**Variables**  $V_1 \dots V_n$  (des notes)

**Domaines** entiers finis (des hauteurs midi)

**Contraintes** Soit  $A_i = \{V_i, V_{prev-voix(j)}, j \neq voix(i)\}$ .

minimiser  $E(A_i, A_{i+1})$

**variable de  $C_1$**  minimiser  $E(A_i, A)$  pour  $A$  accord fixé

$C_2$  minimiser  $D_f(A_i, A_{i+1})$

$C_3$  minimiser  $\sum_{j \leq k} |V_{j+d_i} - p_j|$

$C_4$  (préférence)  $V_{prev-voix(i)} \neq V_i$

---

<sup>9</sup>Voir ambitus dans le glossaire en annexe

<sup>10</sup>Voir transposition dans le glossaire en annexe

### 13.3 Problème 3 : Classement d'accords

On part d'une séquence de  $n$  accords,  $A_1 \dots A_n$ , par exemple tirés aléatoirement dans un ambitus donné. Le but est de trouver une permutation des  $A_i$  maximisant le nombre de notes communes entre deux accords successifs. On notera  $C_{i,j}$  le nombre de notes communes entre  $A_i$  et  $A_j$ ,  $C_{i,j} = \sharp A_i \cap A_j$ . Le domaine est l'ensemble des permutations des  $A_i$ , ce qui élimine une contrainte alldiff. La seule contrainte s'écrit alors : maximiser  $\sum_{1 \leq i \leq n-1} C_{i,i+1}$

Cette contrainte, portant sur toute la séquence, n'impose en rien une continuité musicale entre deux accords successifs. Il arrive que l'on trouve une solution optimale pour le nombre de notes communes sur toute la séquence, mais avec des "trous", ie zéro note commune entre deux accords successifs. Pour garder une continuité musicale, on peut imposer en plus que deux accords successifs partagent au moins une note commune, ce qui ajoute  $n - 1$  contraintes  $C_{i,i+1} \geq 1$ .

Ce problème est évidemment équivalent au classique Travelling Salesman Problem, en prenant comme villes les accords et comme distance entre les villes  $i$  et  $j$   $N - C_{i,j}$  avec  $N$  suffisamment grand ( $N \geq \max_{1 \leq i,j < n} C_{i,j}$ ). Pour un ordre donné sur les accords, la somme des notes communes entre deux accords successifs est égale à  $N * n$  moins la distance du chemin correspondant entre les villes. Maximiser les notes communes revient alors à minimiser la distance totale parcourue.

#### Modélisation, résumé

**Données**  $n$  accords  $A_1 \dots A_n$

**Variables**  $A_1 \dots A_n$

**Domaines** permutations de  $A_1 \dots A_n$

**Contraintes**  $C_1$  maximiser  $\sum_{i \neq j} \sharp A_i \cap A_j$

$C_2$  (optionnelle)  $\sharp A_i \cap A_j \geq 1$

### 13.4 Conclusion

Ces trois problèmes sont assez semblables dans leurs contraintes. Le coeur du problème est à chaque fois d'optimiser une fonction ressemblant à une distance (même si ce n'en est pas une au sens mathématique) entre les variables. On en retiendra qu'il faut pouvoir facilement traiter un problème d'optimisation pour un solver en CAO.

En revanche, on voit déjà apparaître l'une des difficultés de la représentation musicale : on a ici trois problèmes portant sur des suites d'accords, une notion commune et immédiate pour tous les musiciens, et pourtant ces mêmes suites d'accords conduisent à trois modélisations différentes. Dans le problème 3, la modélisation la plus simple convient bien (ensemble d'entiers dans des domaines finis). Dans les problème 1 et 2, il serait possible de choisir cette modélisation mais cela multiplierait le nombre de variables et ajoute des contraintes. Les modélisations choisies consistent à réduire des contraintes structurelles, ou des contraintes portant sur la représentation musicale du problème (accords en fréquence, structure de canon rythmique).

# Chapitre 14

## Problèmes portant sur le rythme

### 14.1 Problème 4 : Rythmes asynchrones

Ce problème a été soumis par le compositeur Mauro Lanza, qui cherchait à composer un quasi-canon rythmique. On a  $n$  voix jouant simultanément. Une unité de temps est donnée, et les rythmes seront comptés comme des multiples de cette unité. La  $i$ -ième voix répète un motif rythmique de période  $T_i$ , formé d'un nombre fixé d'**onsets** <sup>1</sup>.

Il s'agit de trouver les motifs pour que jamais deux voix ne jouent simultanément deux onsets, et ce pendant une durée  $D$  fixée (on aura nécessairement  $D \leq ppcm(T_1 \dots T_n)$ , et les  $T_i$  sont en général choisis premiers entre eux, pour des raisons détaillées ci-dessous). La figure 14.1 montre une solution approchée et une solution exacte, avec des  $T_i$  de 8, 14 et 12 unités de temps - ici la double croche - pour une meilleure commodité de lecture.

On notera  $l_1 \dots l_n$  les longueurs des rythmes de chaque voix,  $\alpha_1 \dots \alpha_n$  les nombres d'onsets dans chaque voix, et  $D$  la durée. On note les variables  $(V_1^1 \dots V_{\alpha_1}^1) \dots (V_1^n \dots V_{\alpha_n}^n)$ , où  $V_j^i$  représente le  $j$ -ième onset de la  $i$ -ième voix, ie la date à laquelle la voix  $i$  frappe son  $j$ -ième rythme, comptée en nombre d'unités de temps depuis le début de la partie considérée. Le domaine de  $V_j^i$  est  $[1 \dots l_i]$ .

Les contraintes s'écrivent :

- Motif rythmique : il faut que le motif  $i$  ait exactement  $\alpha_i$  temps frappés. Donc, à l'intérieur d'un même motif, on doit imposer que les onsets soient différents.

Pour  $i \leq n$  et  $j_1 \neq j_2$

$$V_{j_1}^i \neq V_{j_2}^i$$

Ou bien

$$alldiff(V_1^i \dots V_{\alpha_i}^i)$$

- Non simultanée : deux voix différentes  $i_1$  et  $i_2$  ne doivent pas frapper un rythme en même temps. Il ne suffit pas que les variables soient différentes, car il faut tenir compte de toutes les répétitions. La variable  $V_i$  représente en fait les onsets joués sur les temps  $V_i, V_i + l_i, V_i + 2 * l_i, \dots V_i + k_i * l_i$  où  $k_i$  est le plus grand entier tel que l'onset joué à la  $k_i$ -ième période reste plus petit que  $d$  ( $k_i$  est fixé à une unité près, il peut varier de 1 selon la valeur des  $V_i$ , voir figure 14.2). Cela s'écrit :

Pour  $i_1 \neq i_2, k_1 \neq k_2$  et pour tout  $p_1, p_2$  tels que  $V_{k_1}^{i_1} + p_1 * l_1 \leq d$  et  $V_{k_2}^{i_2} + p_2 * l_2 \leq d$ ,

$$V_{k_1}^{i_1} + p_1 * l_1 \neq V_{k_2}^{i_2} + p_2 * l_2$$

---

<sup>1</sup>Voir onset dans le glossaire en annexe



FIG. 14.1 – A gauche, une solution approchée. L'erreur est encadrée en gris, les voix inférieures et supérieures frappant un onset sur ce temps. A droite, une solution exacte.

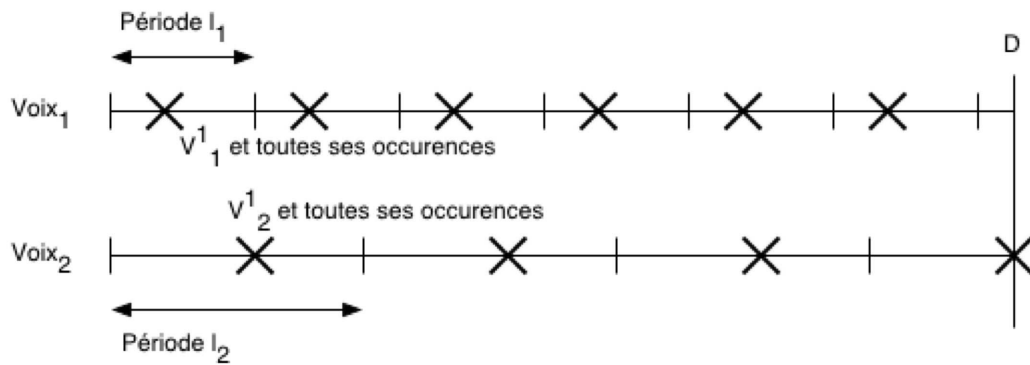


FIG. 14.2 – Pour interdire aux deux onsets  $V_1^2$  et  $V_2^1$  d'être joués simultanément, il faut tenir compte de toutes leurs occurrences jusqu'à  $D$ . Le nombre d'occurrences d'un onset  $V_i^j$  sur la voix  $i$ , noté  $k_i$ , dépend de la valeur de  $V_i^j$ , car sauf exception  $l_i$  de divise pas  $D$  et il reste donc à la fin une période inachevée dont seule une partie sera jouée.

ou plus simplement

$$(V_{k_1}^{i_1} + p_1 * l_1 \neq V_{k_2}^{i_2})[l_2]$$

En pratique, le compositeur souhaite non pas zéro onset simultané, mais le minimum possible, sachant qu'en fonction du nombre de voix et d'onsets le problème peut ne pas avoir de solution. Dans ce cas, on cherche donc des solutions approchées.

Ce problème n'admet pas toujours de solutions. De manière évidente, dès que par exemple on demande plus d'onsets frappés que  $D$ , le principe des tiroirs impose deux onsets simultanés. On peut étudier plus précisément le problème, en particulier l'existence de solutions, grâce au théorème des restes chinois.

**Rappel** le théorème des restes chinois (Chinese Remainder Theorem)

**Théorème 14.1 (de Bézout)** *Pour tous  $n_1, n_2 \in \mathbb{N}$ , il existe deux entiers uniques  $\gamma_1, \gamma_2$  tels que  $\gamma_1 * n_1 + \gamma_2 * n_2 = \text{pgcd}(n_1, n_2)$ . On appelle  $\gamma_1, \gamma_2$  les coefficients de Bézout de  $n_1, n_2$ .*

**Théorème 14.2 (des restes chinois)** Soit pour  $n_1, n_2, a_1, a_2 \in \mathbb{N}$ , le système d'équations

$$\begin{cases} x \equiv a_1[n_1] \\ x \equiv a_2[n_2] \end{cases}$$

Soit  $\gamma_1, \gamma_2$  les coefficients de Bézout de  $n_1, n_2$ .

Ce système admet exactement une solution dans  $\llbracket 1 \dots \text{ppcm}(n_1, n_2) \rrbracket$ , soit  $x_0 = \frac{\gamma_1 n_1 a_2 + \gamma_2 n_2 a_1}{\text{pgcd}(n_1 n_2)}$ . L'ensemble des solutions est  $\{x_0 + k * \text{ppcm}(n_1, n_2), k \in \mathbb{N}\}$ .

On ne redonnera pas ici les démonstrations, ultra-classiques de l'arithmétique. Le calcul des coefficients de Bézout est donné par l'algorithme d'Euclide (le même que celui qui calcule le pgcd de deux entiers) qui consiste en une suite de divisions entières. Sa complexité est en logarithme du plus grand entier, donc extrêmement rapide.

Une manière simple de représenter ce théorème est de considérer un calendrier dans lequel sont placées deux fêtes périodiques. L'une a eu lieu l'année  $a_1$  et revient toutes les  $n_1$  années, l'autre a eu lieu l'année  $a_2$  et revient toutes les  $n_2$  années. Connaître l'année où ces fêtes auront lieu simultanément (si cela arrive) équivaut à résoudre le système ci-dessus.

**Le théorème des rythmes chinois** Sous la forme du système d'équations, ou bien du calendrier, le théorème chinois est très proche d'un sous cas du CSP des rythmes asynchrones : pour  $n = 2$  et  $\alpha_1 = \alpha_2 = 1$ , les fêtes du théorème chinois deviennent les onsets du CSP. Résoudre l'équation du théorème chinois revient à trouver une configuration du problème qui viole la contrainte de non-simultanéité, ie dans laquelle les deux onsets tombent au même instant. En effet, violer la contrainte revient à chercher  $V_1$  et  $V_2$  tels que

$$V_1 + p_1 * l_1 = V_2 + p_2 * l_2$$

$$\text{avec } V_1 \leq l_1, V_2 \leq l_2$$

Ceci est équivalent à résoudre

$$\begin{cases} x \equiv V_1[l_1] \\ x \equiv V_2[l_2] \end{cases}$$

avec la condition supplémentaire  $x \leq D$ . Par le théorème chinois, il existe une unique solution  $x_0$  au système dans  $\llbracket 0, \text{ppcm}(l_1, l_2) \rrbracket$ . De plus,  $x_0$  est donné par la formule  $x_0 = \frac{\gamma_1 l_1 V_2 + \gamma_2 l_2 V_1}{\text{pgcd}(l_1 l_2)}$ , où  $\gamma_1$  et  $\gamma_2$  sont les coefficients de Bezout de  $l_1$  et  $l_2$ . Ces deux valeurs étant données avec le problème donc fixées, il suffit de chercher  $V_1$  et  $V_2$  tels que  $x_0 < D$ , cette condition étant équivalente à la négation de la contrainte. D'où une condition très simple sur  $V_1$  et  $V_2$  pour que la configuration  $(V_1, V_2)$  soit une solution :

$$\gamma_1 l_1 V_2 + \gamma_2 l_2 V_1 > \text{pgcd}(l_1, l_2) * D$$

**Généralisation à plusieurs voix** Il faut évidemment généraliser l'étude du CSP au cas réel, avec (1) un nombre quelconque de voix, (2) un nombre quelconque d'onsets par voix. Malheureusement, la généralisation n'est pas la même que le théorème chinois à  $p$  équations, lequel donne une solution au système  $x = V_i[l_i]$  pour tous les  $i \leq p$ . Cela reviendrait à généraliser

sur (1) en trouvant un onset *commun à toutes les voix*. Au contraire, nous cherchons à trouver des onsets différents *deux à deux*, soit à résoudre pour chaque  $i_1 \neq i_2 \leq n$  le système

$$\begin{cases} x = V_{i_1}[l_{i_1}] \\ x = V_{i_2}[l_{i_2}] \end{cases}$$

Le problème complet est équivalent à résoudre le système suivant. On note  $\gamma_{i,j}$  le coefficient de Bézout de  $l_i$  avec  $l_j$ , ie  $\gamma_{i,j}l_i + \gamma_{j,i}l_j = \text{pgcd}(l_i, l_j)$ .

$$\forall i, j \leq n, \forall k \leq \alpha_i, \forall k' \leq \alpha_j \left\{ \frac{\gamma_{i,j}l_i V_{k'}^j + \gamma_{j,i}l_j V_k^i}{\text{pgcd}(l_i, l_j)} > D \right.$$

Cette généralisation n'a donc qu'un intérêt modéré. Elle permet de revenir à un système d'équations entières, mais pas meilleur que la modélisation en CSP. Il est inenvisageable de fournir ce type d'outils dans un système de résolution adapté à la CAO pour des raisons évidentes (compliqué, structure musicale trop spécifique). On peut cependant en retenir une explication à la nécessité de choisir les  $l_i$  premiers entre eux, ce qui, fait notable, avait d'ailleurs été donné comme condition par Mauro Lanza lorsqu'il avait exposé son CSP. En effet, dès que  $D > \text{ppcm}(l_i, l_j)$  pour n'importe quels  $i, j$ , le système n'a pas de solutions exactes. Plus précisément, on a un minorant du nombre d'erreurs en comptant le nombre de couples  $(i, j)$  tels que  $D > \text{ppcm}(l_i, l_j)$ . D'où l'intérêt de choisir grands les ppcm deux à deux des  $l_i$ , pour obtenir de meilleures solutions.

### Modélisation, résumé

**Données**  $n$  entier,  $l_i$  entiers premiers entre eux,  $\alpha_i$  entiers

**Variables**  $V_i^j$  pour  $i \leq n$  et  $j \in \llbracket 1 \dots \alpha_i \rrbracket$

**Domaines** entiers finis, pour  $V_i^j \llbracket 1 \dots l_i \rrbracket$

**Contraintes**  $C_1$  Pour  $i \leq n$  et  $j_1 \neq j_2$   $\text{alldiff}(V_1^{i_1} \dots V_{\alpha_i}^{i_{\alpha_i}})$

$C_2$  Pour  $i_1 \neq i_2$ ,  $k_1 \neq k_2$  et pour tout  $p_1, p_2$  tels que  $V_{k_1}^{i_1} + p_1 * l_1 \leq d$  et  $V_{k_2}^{i_2} + p_2 * l_2 \leq d$ ,

$$V_{k_1}^{i_1} + p_1 * l_1 \neq V_{k_2}^{i_2} + p_2 * l_2$$

## 14.2 Problème 5 : Durées dans Fibonacci

Ce problème est dû à Geoffroy Drouin, compositeur du Coursus de l'IRCAM en 2002/2003. Il tourne autour de la célèbre suite de Fibonacci, définie par

$$u_0 = 0$$

$$u_1 = 1$$

$$u_n = u_{n-1} + u_{n-2}$$

On suppose donnés une unité de temps et un entier  $D$ . Le but est de trouver  $n$  durées qui sont des valeurs de la suite, telles que deux durées voisines soient voisines dans la suite (en avant ou en arrière), et que la durée totale soit égale à  $D$ .

Plusieurs modélisations sont possibles. La première consiste à prendre comme variables des entiers, et comme domaines les valeurs de la suite de Fibonacci jusqu'à un nombre suffisamment grand. Elle est équivalente à une modélisation par indices, où sont choisis les durées comme variables  $V_1 \dots V_n$ , repérées plus simplement par leurs indices, avec comme domaine les entiers

$j$  tels que  $u_j \leq D$ . On a alors deux contraintes :  $|V_i - V_{i+1}| = 1$  pour la proximité dans la suite et  $\sum_{i=1}^n u_{V_i} = D$  pour la durée totale.

La deuxième modélisation réduit la première contrainte. Mais elle suppose de singulariser la première variable, ou bien de connaître la valeur de départ dans la suite de Fibonacci, soit  $j_0$  son indice. On prend comme variables  $V_1 \dots V_n$  le saut dans la suite. Le domaine est alors  $\{-1, 1\}$ . On fixe  $V_0 = j_0$  par commodité mais ce n'est pas une variable. Reste alors comme seule contrainte  $\sum_{i=1}^n u_{\sum_{1 \leq j \leq i} V_j} = D$ .

### Modélisation, résumé

**Données**  $D$  entier,  $(u_i)_{i < N}$  des termes de la suite de Fibonacci jusqu'à  $N$  arbitrairement grand

**Variables**  $V_i$  pour  $i \leq n$

**Domaines**  $\{-1, 1\}$

**Contraintes**  $C_1 \sum_{i=1}^n u_{\sum_{1 \leq j \leq i} V_j} = D$

## 14.3 Problème 6 : Tempi et accelerando

Ce problème est dû au compositeur Pierre Klanac. Les variables sont des **tempi**<sup>2</sup>  $t_1 \dots t_n$ , dans une gamme donnée de 40 à 250. Les contraintes portent sur les rapports entre tempi successifs : on veut un *accelerando*, en fixant les rapports entre  $t_i$  et  $t_{i+1}$ , mais aussi entre  $t_i$  et  $t_{i+2}$ , entre  $t_i$  et  $t_{i+3}$ , et  $t_i$  et  $t_{i+4}$ . Avec les valeurs choisies par le compositeur, les contraintes s'écrivent :

- $C_1 : t_{i+1}/t_i = 0.9$
- $C_2 : t_{i+2}/t_i = 0.8$
- $C_3 : t_{i+3}/t_i = 0.75$
- $C_4 : t_{i+4}/t_i = 0.666666$

Evidemment, dans les réels, fixer  $t_{i+1}/t_i$  revient à fixer automatiquement les autres rapports. Un mathématicien considérerait même le problème comme une série géométrique en fonction de  $t_1$  avec  $t_i = 0,9^{i-1} * t_1$ , ce qui fixerait les autres rapports à  $0,9^2 = 0,81$  dans  $C_2$ ,  $0,9^3 = 0,729$  dans  $C_3$  et  $0,9^4 = 0,6561$  dans  $C_4$ . Mais ceci ne s'applique pas puisque le domaine est entier. Ce problème est transformé en problème d'optimisation, pour trouver des entiers dont les rapports soient le plus proches possibles des valeurs souhaitées.

Plus précisément on classe les solutions de la manière suivante, à la demande du compositeur : d'abord le plus grand nombre de rapports exacts sur  $t_{i+3}$  et  $t_i$ , ensuite le plus petit écart entre le minimum et le maximum de  $t_{i+1}/t_i$ .

### Modélisation, résumé

**Données**

**Variables**  $t_i$  pour  $i \leq n$

**Domaines** entiers finis (par exemple de 40 à 250)

**Contraintes**  $C_1 \ t_{i+3}/t_i = 0.75$

$C_2 \text{ minimiser } |\max_i(t_{i+1}/t_i) - 0.9|, |\min_i(t_{i+1}/t_i) - 0.9|$

---

<sup>2</sup>Voir tempo dans le glossaire en annexe

## 14.4 Problème 7 : Suivi de tempo

Ce problème a été proposé par Gilbert Nouno, mais pas utilisé à cause d'impératifs temps réel. Il s'agissait de reconnaître des motifs rythmiques joués sur des congas par un percussionniste cubain, dans le cadre d'une improvisation avec le saxophoniste Steve Coleman. Le percussionniste jouait sur des congas équipés de capteurs MIDI, d'où il fallait récupérer un rythme symbolique approximé.

Le problème est donc de trouver un **tempo**<sup>3</sup> à partir de rythmes réellement joués par un musicien. La donnée est une suite de durées entières,  $d_1 \dots d_n$ . Elle n'a a priori aucune propriété particulière, même si l'on sait qu'il s'agit d'un rythme joué, donc que c'est l'approximation d'une donnée structurée, et que les rapports entre les  $d_i$  sont, grossièrement, entiers et petits. Par exemple, on pourrait avoir en entrée les durées 515 – 498 – 1020, qui pourrait correspondre au rythme "**deux croches - noires**"<sup>4</sup>, soit un rythme exact de 500 – 500 – 1000 sur une pulsation de 500 millisecondes.

On cherche à trouver le tempo et le rythme sous-jacent. C'est un exemple de problème musical bien posé, mais informatiquement délicat. Par nature, ce problème n'admet pas de solution unique. Caricaturalement, si le musicien a joué très rubato ou swingué, il peut avoir ajouté à la durée notée de la partition une durée presque égale.

D'abord on fixe un ensemble de rapports temporels autorisés, soit  $R$ . Par exemple, par rapport à une noire, 2 représente une blanche,  $1/3$  une croche de triolet, etc. Les variables sont  $V_1 \dots V_n$ , durées du rythme symbolique recherché.  $V_i$  est une approximation du  $d_i$  correspondant, donné avec le problème. Les  $V_i$  doivent être multiples du tempo exprimé comme une durée en millisecondes. Aux  $V_i$  peuvent correspondre plusieurs tempi, on prendra  $\text{pgcd}(V_i, i \leq n)$ .

La première contrainte impose au motif  $V$  d'être proche du rythme réellement joué : minimize  $\sum_i |V_i - d_i|$ . La deuxième contrainte restreint le type de rythmes symboliques. Sans elle, n'importe quelle suite d'entiers peut s'écrire comme un rythme symbolique, évidemment beaucoup trop compliqué pour avoir un sens musical. Pour tout  $i \leq n$ , on impose que  $V_i / \text{gcd}(V_j, j \leq n)$  soit dans  $R$ . Cela revient à autoriser seulement les valeurs rythmiques données dans  $R$  (blanche, croche de triolet, etc). A noter, on utilise le pgcd pour avoir la plus petite division rythmique perçue, parce que le minimum ne convient pas : le minimum des  $V_i$  est la plus petite durée entendue dans le rythme. Mais pour  $V_1 = 100$  et  $V_2 = 101$ , le minimum vaut 100 et n'a aucun sens pour  $V_2$ . Le pgcd (ici de 1) mesure au contraire la plus petite différence de durée entre les rythmes, autrement dit la plus petite durée audible.

**Variante** On peut élargir cette contrainte en fixant seulement le dénominateur de  $V_i / \text{gcd}(V_j, j \leq n)$  dans  $R$ , pour exprimer  $R$  plus simplement en division du temps. Par exemple, ajouter 3 à  $R$  revient à autoriser tous les multiples de la croche de triolet, comme "croche - noire en triolet". Musicalement, cela apporte une souplesse assez naturelle, considérant que c'est surtout la division du rythme symbolique qui en mesure la complexité.

### Modélisation, résumé

**Données**  $R$  un ensemble de rationnels (rapport rythmiques),  $d_1, \dots, d_n$  entiers (durées)

**Variables**  $V_1 \dots V_n$

**Domaines** entiers finis, de 0 à  $\max_i(d_i)$

<sup>3</sup>Voir tempo dans le glossaire en annexe

<sup>4</sup>Voir rythmes dans le glossaire en annexe

**Contraintes**  $C_1$  minimize  $\sum_i |V_i - d_i|$

$C_2$   $V_i / \gcd(V_j, j \leq n) \in R$

**variante de  $C_2$**  dénominateur de  $V_i / \gcd(V_j, j \leq n) \in R$

## 14.5 Conclusion

Ces quatre problèmes rythmiques permettent de déduire peu de choses. Les problèmes 4 et 5 sont au même niveau musical (représentation d'un rythme). Mais le problème 6 porte sur les tempi et pas exactement sur une structure rythmique. Le problème 7 est en dehors du cadre habituel de la partition puisqu'il prend des données extérieures (un flux d'onsets joués).

Remarque : la notation et la gestion des rythmes sont des points particulièrement difficiles d'une manière générale en informatique musicale. C'est probablement là que se pose le plus fortement le problème de la représentation informatique d'une structure musicale. Le rythme est une structure complexe, notamment à cause de sa nature stratifiée (voir [76]).

C'est est une notion commune à tous les musiciens, mais très contextuelle, qui intègre beaucoup de savoir faire et un facteur humain important. Par exemple, pour un rythme donné, une opération aussi simple que "ajouter une croche" peut signifier : allonger la mesure en changeant sa signature, ajouter une croche sur un temps en divisant de manière adéquate, faire une valeur ajoutée au sens de Messiaen, *ad lib*. Tout ce qui concerne le rythme est fortement contextuel (telle opération sera évidente dans le cadre de la musique classique mais pas contemporaine par exemple). Notre conclusion pour le solver est : pour les rythmes, il faudra supposer que les utilisateurs sauront trouver les modélisations adaptées. On ne peut pas leur donner de facilités de programmation. En pratique, il s'agira heureusement toujours de durées entières, ce qui permet de rester d'un point de vue CSP sur les domaines finis.



# Chapitre 15

## Problèmes portant sur une mélodie

### 15.1 Problème 8 : Suite de gestes

Ce problème est dû à Gilbert Nouno, pour le compositeur Michael Jarrel. Il s'agit de composer une mélodie à l'aide de gestes. Un geste est soit un intervalle musical, soit une courte suite d'intervalles. A partir d'une note choisie, la mélodie est formée d'une suite de ces gestes, à l'exclusion d'autres notes. Le problème est de trouver une telle mélodie, dont les notes restent dans un domaine fixé, un **agrégat**<sup>1</sup>  $H$  de référence donné au départ. Les deux contraintes principales portent l'une sur les intervalles entre deux notes successives, l'autre sur les notes jouées. Cela conduit à deux modélisations fondamentalement différentes.

La première consiste à prendre comme variables les notes, et contraindre les intervalles entre deux notes successives. La contrainte portant sur l'agrégat est alors automatiquement réduite dans le domaine, mais la contrainte sur les gestes est délicate à traiter (notamment pour tenir compte des gestes composés de plusieurs intervalles). La seconde consiste à prendre comme variables les intervalles, et à contraindre les notes résultantes dans un domaine fixé, avec une difficulté symétrique dans l'expression des contraintes. On peut remarquer la symétrie entre les deux modélisations en CSP et les deux points de vue musicaux.

Comme l'intention première du compositeur est de travailler sur des gestes, on retiendra la seconde modélisation. Les variables  $V_1 \dots V_n$  sont des intervalles ou suites d'intervalles. Le domaine est donné par  $I$ , sous forme d'intervalles **MIDI**<sup>2</sup>, et tel que chaque intervalle apparaisse toujours montant et descendant  $\{Int_j, -Int_j, j \leq p\}$  (en pratique  $\{(3, 8), (-3, -8), (11), (-11), (6), (-6)\}$ ). On fixe la note de départ en hauteur midi, soit  $D$ , et l'agrégat de référence  $H$  est représenté comme un ensemble de hauteurs midi. La contrainte sur l'agrégat s'écrit :

$$(D + \sum_{1 \leq i \leq n} V_i) \in H$$

Cette contrainte est une sorte de contrainte de capacité, à ceci près qu'elle n'implique pas des variables de décision.

Sur ce modèle de bases, le compositeur souhaite ajouter des contraintes évitant des mélodies trop évidentes. Une deuxième contrainte restreint les répétitions. On interdit que deux gestes successifs soient égaux ou opposés :

$$|V_{i+1}| \neq |V_i|$$

La troisième contrainte impose des proportions d'apparition de chaque intervalle. On fixe  $P_{Int_j}$ , pour  $j \leq p$ , entier représentant le nombre de fois où  $Int_j$  doit apparaître.

---

<sup>1</sup>Voir agrégat dans le glossaire en annexe

<sup>2</sup>Voir MIDI dans le glossaire en annexe

$$\#\{i, |V_i| = Int_j\} = P_{Int_j}$$

Si tous les  $P_{Int_j}$  étaient fixés et de somme égale à  $n$ , on passerait évidemment à un problème de permutation. Mais d'une part, ce n'est pas le cas ; d'autre part, on peut ainsi traiter des proportions qui ne forment pas une vraie probabilité ( $\sum_j P_{Int_j} \neq n$ ).

La dernière contrainte impose un mouvement mélodique global. Elle peut être simplement passée dans une restriction de l'**ambitus**<sup>3</sup> de l'agrégat  $H$  quitte à découper le CSP en sous-CSPs.

### Modélisation, résumé

**Données**  $D$  entier (note),  $H$  ensemble d'entiers (harmonie),  $P_{Int_j}$  pour certains  $j$ , entiers

**Variables**  $V_1 \dots V_n$

**Domaines** entiers finis (intervalles)

**Contraintes**  $C_1$   $(D + \sum_{1 \leq i \leq n} V_i) \in H$

$$C_2 \quad |V_{i+1}| \neq |V_i|$$

$$C_3 \quad \#\{i, |V_i| = Int_j\} = P_{Int_j}$$

## 15.2 Problème 9 : Séries tous intervalles

Il s'agit de trouver une permutation  $\sigma$  des  $n$  premiers entiers, telle que les  $|\sigma_{i+1} - \sigma_i|$  soient une permutation des  $n - 1$  premiers entiers. Si on prend  $n = 12$ , on obtient une série **dodécaphonique**<sup>4</sup> au sens musical (motif dans lequel toutes les notes de la gamme chromatique apparaissent exactement une fois). C'est alors un problème musical classique, décrit notamment dans [75]. La contrainte sur les différences successives impose que les intervalles aussi apparaissent exactement une fois chacun lorsque l'on joue la série : on entendra exactement une seconde diminuée (intervalle de 1 demi-ton), une seconde augmentée (intervalle de 2 demis-tons), une tierce mineure (3 demi-tons), etc. Cela s'écrit :

$$alldiff(|\sigma_{i+1} - \sigma_i|, i \leq n)$$

Les all-intervals series ont notamment été résolues avec Ant-P-Solver, de C. Solnon [97], et ILOG-Solver par J.-F. Puget<sup>5</sup>. Ce problème est également répertorié dans CSPLib<sup>6</sup>. Elles sont assez proches des problèmes d'allocation de fréquences.

La variante musicale du CSP diffère légèrement, car on considère que la série doit être jouée en boucle, et donc l'intervalle entre  $\sigma_1$  et  $\sigma_n$  compte également. Le *alldiff* sur les intervalles devient alors impossible puisque les intervalles ont au plus  $n - 1$  valeurs. La solution naturelle est de faire jouer deux fois la quarte augmentée, puisqu'elle est son propre renversement.

Quelques remarques sur ce problème :

- (i). Pour la variante musicale, une permutation circulaire sur une solution est encore une solution, ce qui est naturel puisque l'on considère la solution comme jouée répétitivement, avec la quarte augmentée répétée. Ce n'est pas le cas pour le problème original.
- (ii). Pour mémoire, il existe une solution triviale au problème des all intervals series, en zig-zag, avec  $1 \ n \ 2 \ (n - 1) \ 3$  etc, dont les intervalles sont ordonnés  $(n - 1)$ ,  $-(n - 2)$ ,  $(n - 3)$ ,  $-(n - 4)$ , etc. On cherche les autres solutions, ce qui peut rajouter une contrainte.

<sup>3</sup>Voir ambitus dans le glossaire en annexe

<sup>4</sup>Voir dodécaphonisme dans le glossaire en annexe

<sup>5</sup><http://www-users.cs.york.ac.uk/tw/csplib/prob/prob007/puget.pdf>

<sup>6</sup><http://www-users.cs.york.ac.uk/tw/csplib/index.html>

- (iii). La difficulté du problème vient de la nature des contraintes. On a ici deux all différents emboîtés. Un alldifferent peut se réduire en permutations, mais pas deux.

### Modélisation, résumé

**Données**

**Variables**  $\sigma_1 \dots \sigma_n$

**Domaines** entiers, de 1 à 12 (généralisables à un entier quelconque)

**Contraintes**  $C_1$  *alldiff* ( $|\sigma_{i+1} - \sigma_i|, i \leq n$ )

**variante de  $C_1$**   $\{|\sigma_n - \sigma_1|, |\sigma_{i+1} - \sigma_i|, i \leq n\} = \{1, 2, 3, 4, 5, 6, 6, 7, 8, 9, 10, 11\}$

## 15.3 Conclusion

Ces deux problèmes réunis sont intéressants à double titre. D'abord, ils sont très dissemblables, l'un académique, propre, difficile, l'autre "réel", avec plusieurs variantes, à la formulation un peu floue, et dont le nombre de solutions varie de beaucoup à pas du tout. ils représentent probablement les deux extrêmes de ce que l'on devra pouvoir traiter. Ensuite, ils portent tous deux sur une simple suite de notes, et tous deux sur les intervalles entre ces notes. Ils ont la même dualité, avec une modélisation possible en variables = notes, et une modélisation possible en variables = intervalles. L'une ou l'autre modélisations sont équivalentes pour des solutions exactes.



# Chapitre 16

## Problèmes d'analyse musicale

### 16.1 Problème 10 : Textures de Ligeti

Ce problème, dû à Marc Chemillier [27], est lié à plusieurs oeuvres de Ligeti ("pattern meccanico", [33]). Il s'agit de retrouver une **harmonie**<sup>1</sup> sous-jacente à la partition.

Est donnée une séquence d'**agrégats**<sup>2</sup>  $A_i$  ayant chacun au plus  $k$  notes. On cherche une polyphonie sous-jacente à  $k$  voix, notée dans un tableau  $X_{i,j}$ , où  $i$  désigne l'accord et  $j$  la voix. Les contraintes s'écrivent :

- (i). Chaque  $A_i$  est inclus dans la ligne correspondante  $X_i$ .
- (ii). Les voix dont les notes n'apparaissent pas dans les  $A_i$  doivent rester sur la même hauteur,  $X_{i,j} \notin A_i \Rightarrow X_{i,j} = X_{i-1,j}$ .
- (iii). Les mouvements mélodiques de chaque voix sont limités à zéro, plus ou moins un demi-ton ou moins un ton,  $X_{i,j} - X_{i-1,j} \in \{-2, -1, 0, 1\}$  (en valeur midi).
- (iv). En plus, les  $X_i$  doivent former un "vrai" accord (pas de notes égales), on ajoute donc une contrainte imposant la croissance stricte pour chaque  $X_i$ .

Dans la partition, les agrégats sont joués en courts motifs répétés, comme dans la pièce pour harpe Continuum. Nous étudions une texture du début de Melodien, pièce pour orchestre. Ici,  $k$  vaut 10. La figure 16.1 montre les notes jouées dans les agrégats en noir, et la structure sous-jacente que nous cherchons est représentée par les lignes brisées recouvrant ces notes.

Ce problème rappelle fortement le CSP d'harmonisation automatique, dans le style des chorals de Bach par exemple. La ligne mélodique du choral est remplacée par la séquence d'agrégats, et la polyphonie à quatre voix par celle à dix voix. Mais les contraintes sont très similaires puisqu'elles formalisent des restrictions du mouvement mélodique. D'un point de vue musical, la principale différence vient du fait que les voix d'un choral sont destinées à être jouées, tandis qu'ici on cherche des notes cachées : il s'agit d'une structure implicite, sous-jacente à la partition. D'un point de vue contraintes, il faut souligner quelques différences importantes : l'harmonisation automatique est connue pour admettre un grand nombre de solutions [41], et on peut abstraire certains objets musicaux comme par exemple les degrés harmoniques, ce qui permet de réduire l'espace de recherche [80]. Au contraire, le nombre de solutions est ici très réduit, deux dans le cas de Melodien.

---

<sup>1</sup>Voir harmonie dans le glossaire en annexe

<sup>2</sup>Voir agrégats dans le glossaire en annexe

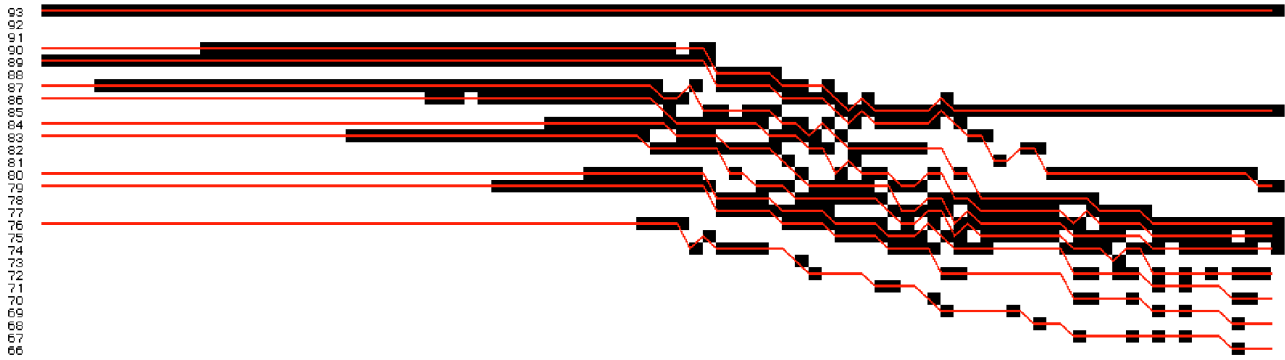


FIG. 16.1 – Exemple d'analyse des textures du début de *Mélodien*. Les notes jouées sont en noire et les textures sous-jacentes en rouge.

Pour *Melodien*, on a 49 colonnes et 10 notes par colonne, ces notes restant dans un ambitus de 66 à 93.

Il y a plusieurs manières de représenter ce problème. La plus simple serait de prendre  $49 \times 10$  variables et de résoudre un CSP à 500 variables, sur un domaine de taille 27. Pour réduire le nombre de variables, on peut aussi ne prendre comme variables que les notes cachées, 83 dans le cas de *Melodien*, et écrire les contraintes sur ces seules notes. En effet, les autres variables sont presque déjà instanciées dans la partition. Mais ce n'est pas satisfaisant car cela ne permet pas de vérifier si le processus de modélisation est valable pour toute la partition. Une bonne solution intermédiaire est de découper le CSP colonne par colonne : chaque CSP porte donc sur 10 variables, avec des domaines petits (de taille 4 puisqu'on passe à la ligne  $i + 1$  la contrainte de mouvement mélodique à partir de l'instantiation trouvée à la ligne  $i$ , ce qui revient à un filtrage dans une méthode classique). D'un point de vue CSP, on perd de l'information de cette manière, puisque cela revient à bloquer la combinatoire entre les ligne. En pratique, cela n'a pas d'importance car par la remarque suivante, il y a peu de solutions. De cette manière, nous pouvons vérifier la validité du modèle musical sur toute la partition.

Un algorithme spécifique décrit dans [28] montre que l'analyse de cette texture n'a que 6 solutions. Plus précisément, il y a trois transitions possibles de la ligne 21 à la ligne 22 et deux de la ligne 36 à la ligne 37. La figure 22.9 montre ces deux possibilités, qui donnent en valeur midi, et en notant entre parenthèses les notes cachées :

$$\begin{aligned} X_{36} &= ((67)(70) 72 73 75 76 77 80 85 93) \\ X_{37} &= (67 70 72 74(75) 76 77 80 85 93) \end{aligned}$$

$$\begin{aligned} X_{36} &= ((67)(70) 72 73 75 76 77 80 85 93) \\ X_{37} &= (67 70 72(73)74 76 77 80 85 93) \end{aligned}$$

### Modélisation, résumé

**Données**  $A_1 \dots A_n$  accords de  $k$  notes au plus

**Variables** pour chaque  $i \leq n$ ,  $X_i^1 \dots X_i^k$

**Domaines** entiers de 66 à 93 (hauteurs midi)

**Contraintes**  $C_1$   $A_i \subset X_i$

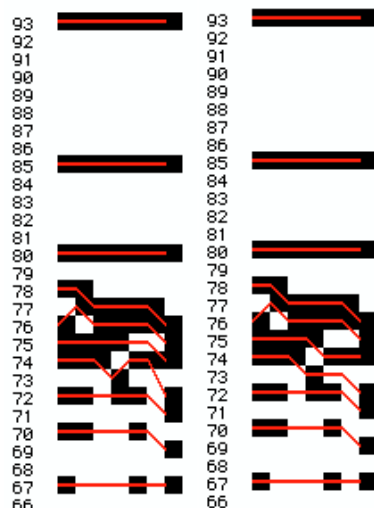


FIG. 16.2 – Deux solutions entre les lignes 36 et 37 de Melodien

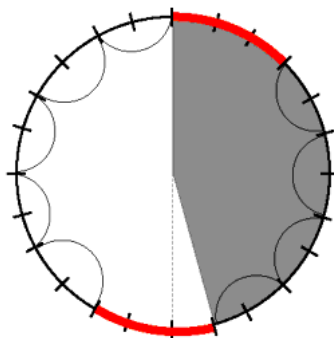


FIG. 16.3 – Il est impossible de couper le cercle en deux parts égales, ni sur l'axe vertical, ni sur aucun autre axe.

$$C_2 \quad X_i^j \in A_i \wedge X_i^j = X_{i-1}^j$$

$$C_3 \quad X_i^j - X_{i-1}^j \in \{-2, -1, 0, 1\}$$

$$C_4 \quad X_i^j < X_i^{j+1}$$

## 16.2 Problème 11 : Imparité rythmique

Ce problème est dû à Simha Arom et Marc Chemillier. Il porte sur des structures rythmiques du répertoire des pygmées Aka en Centrafrique. Les Aka jouent en boucle des formules rythmiques sur une pulsation régulière, avec des accents frappés tous les 2 ou 3 temps. Ces accents forment des groupes de 2 ou 3 unités de temps, par exemple 3 2 2 2 2 3 2 2 2 2 2. Cette formule respecte une propriété que Simha Arom appelle "imparité rythmique".

La figure 16.3 montre une formule dessinée sur un cercle. On voit qu'il est impossible de couper ce cercle en deux en restant dans les temps frappés. Cette dissymétrie est intrinsèque à la formule et aux autres formules jouées dans cette partie de la Centrafrique.

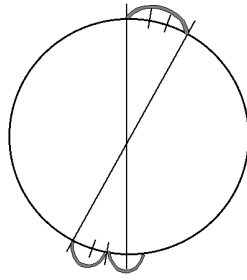


FIG. 16.4 – Il est impossible de couper le cercle en deux parts égales, ni sur l'axe vertical, ni sur aucun autre axe.

La modélisation en CSP est immédiate. On note  $n_2$  le nombre de groupes de deux et  $n_3$  le nombre de groupes de trois, fixés. Les variables sont notées  $V_1 \dots V_n$ , et l'on se place sur le domaine des permutations de l'ensemble constitué de  $n_2$  deux et  $n_3$  trois.

La contrainte est alors très simple à exprimer. Par commodité, on oublie les modulus dans la notation, en supposant que les motifs sont joués indéfiniment.  $\forall k \leq n$ ,  $\sum_{i=k}^{i=k+n/2-1} V_i \neq \sum_{i=k+n/2}^{i=k-1} V_i$

Ce CSP pose un problème de dénombrement intéressant (voir les travaux avec Marc Chemillier, [29], en annexe). Première remarque, par parité de  $2 * n_2 + 3 * n_3$ , on a nécessairement un nombre impair de groupes de 2, noté  $n_2$ , et un nombre pair de groupes de 3, noté  $n_3$ . Le problème est de dénombrer, pour  $n_2$  et  $n_3$  fixés, les formules vérifiant la propriété d'imparité, à permutations près. Hors permutations et sur un seul axe, il suffit de compter le nombre total de formules, égal au nombre de manières de placer  $n_2$  (ou  $n_3$ ) groupes dans le cercle :  $C_{n_2+n_3}^{n_2}$ . Dont on enlève le nombre de formules qui se coupent sur l'axe vertical. On note  $n_t = 2 * n_2 + 3 * n_3$  le nombre total d'unités de temps. Une telle formule s'obtient en plaçant à droite  $i$  groupes de 2 et  $j$  groupes de 3, de telle manière que  $2 * i + 3 * j = n_t/2$ , et en plaçant à gauche le reste des groupes de 2 et de 3. Soit un nombre de :  $\sum_{P(i,i',j,j')} C_{i+j}^i * C_{i'+j'}^{i'}$  avec  $P(i,j) = (i + i' = n_2) \wedge (j + j' = n_3) \wedge (2 * i + 3 * j = n_t)$ . Ce qui donne le nombre de formule vérifiant l'imparité rythmique sur l'axe vertical, ie une seule des contraintes. La généralisation serait facile s'il ne fallait pas éliminer les permutations.

Mais le problème est d'étendre à tous les axes en éliminant les permutations, puisque les formules sont jouées en boucle. On peut remarquer que les groupes de 3 sont nécessairement "face à face", intuition compliquée à formaliser mais facile à voir sur un dessin 16.4. Une modélisation avec des mots de Lindon qui permet le dénombrement dans certains cas, voir les travaux avec Marc Chemillier [29]. Une étude de ces mêmes propriétés avec des outils différents est donnée dans [101].

Remarque : Pour des cercles de moins de 24 unités de temps, presque toutes les formules possibles sont jouées par les peuples africains.

### Modélisation, résumé

**Données**  $n_2, n_3$  entiers

**Variables**  $V_1 \dots V_n$

**Domaines**  $\{2, 3\}$

**Contraintes**  $C_1 \sum_{i=k}^{i=k+n/2-1} V_i \neq \sum_{i=k+n/2}^{i=k-1} V_i$

## 16.3 Problème 12 : Canons Nzakara

Ce problème est proposé par Marc Chemillier [26]. On veut reproduire des formules jouées sur une harpe à cinq cordes par le peuple Nzakara en Afrique Centrale. Les formules sont jouées en **ostinato**<sup>3</sup>, des poèmes nzakara étant chantés sur chacune d'elles.

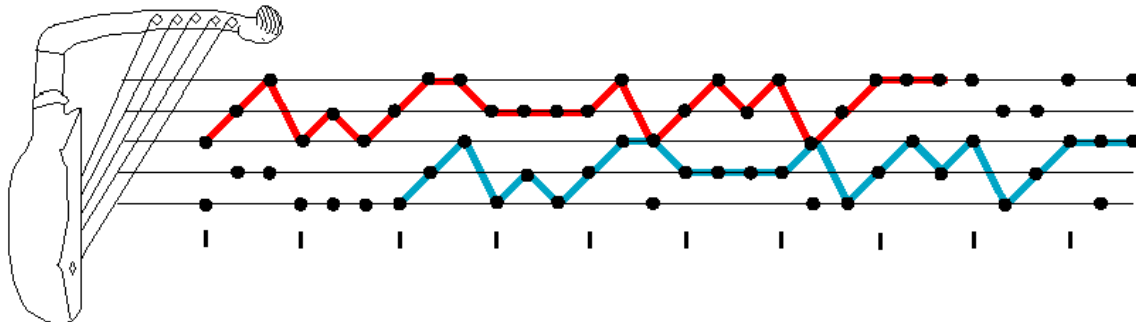


FIG. 16.5 – Un canon de la catégorie limanza, pour  $n = 30$  et  $p = 6$

Pour la modélisation, on représentera les notes Nzakara par une approximation midi : 60, 62, 64, 67, 70. Les formules sont construites à partir d'accords de deux notes, dont les valeurs sont  $\{(60, 64), (60, 67), (62, 67), (62, 70), (64, 70)\}$ . On prend  $n$  variables  $V_1 \dots V_n$  dont les domaines sont les accords nzakara. On notera  $V_i$  la note inférieure de l'accord  $i$  et  $\overline{V}_i$  la note supérieure. La structure des formules est représentée par deux contraintes.

La première impose un canon mélodique transposé. La transposition est donnée par une fonction  $t$  :  $t(60) = 64$ ,  $t(62) = 67$ ,  $t(64) = 70$ . Les formules Nzakara reprennent sur la voix inférieure la mélodie de la voix supérieure, décalée de  $p$  accords ( $p$  entier fixé), et transposée par  $t$ .

$$\underline{V_{(i+p \bmod n)}} = t(\overline{V}_i)$$

Avec seulement cette contrainte, les formules répétitions d'un même motif sont solutions. Or non seulement on n'en veut pas, mais les Nzakara ne font même jamais de répétition d'un même motif. On ajoute donc la contrainte  $V_{(i+1 \bmod n)} \neq V_i$ . Mais ce n'est pas suffisant, car ici on n'évite pas les formules consitutes de sous-formules (par exemple de longueur  $p$  si  $p|n$ ). Pour cela, on rajoute une troisième contrainte :  $V_{(i+p \bmod n)} \neq V_i$ .

Les valeurs Nzakara pour  $n$  et  $p$  sont  $n = 10$ ,  $p = 4$  ou  $n = 20$ ,  $p = 4$  (catégorie ngbakia) et  $n = 30$ ,  $p = 6$  (catégorie limanza, voir figure 16.5). Ainsi posé, ce CSP n'a pas de solutions. Le nombre minimal d'erreurs est égal à  $pgcd(n, p)$  [26].

### Modélisation, résumé

**Données**  $t$  définie par  $t(60) = 64$ ,  $t(62) = 67$ ,  $t(64) = 70$

**Variables**  $V_1 \dots V_n$

**Domaines**  $\{(60, 64), (60, 67), (62, 67), (62, 70), (64, 70)\}$

**Contraintes**  $C_1$   $\underline{V_{(i+p \bmod n)}} = t(\overline{V}_i)$

$$C_2 \quad V_{(i+1 \bmod n)} \neq V_i$$

$$C_3 \quad V_{(i+p \bmod n)} \neq V_i$$

<sup>3</sup>Voir ostinato dans le glossaire en annexe

## 16.4 Conclusion

Voilà trois problèmes qui auraient trouvé leur place respectivement en harmonie, rythme, harmonie (sans d'ailleurs modifier les conclusions de ces deux paragraphes). Mais l'utilisation des contraintes en analyse musicale est un cas à part, ne serait-ce que parce qu'il s'agit d'une application nouvelle, qui appelle quelques commentaires (voir [30] pour plus de détails).

Il s'agit ici de CSPs permettant de modéliser l'analyse d'une (ou plusieurs) séquence musicale par un ensemble de contraintes dont la séquence analysée est une solution. Dans cette optique, le CSP proposé comme modèle est d'autant plus "proche" de la séquence analysée qu'il admet relativement peu de solutions, et qu'il capte ainsi une part de la spécificité de cette séquence. Le nombre de solutions d'un CSP modélisant une analyse musicale peut ainsi être vu comme un critère permettant d'évaluer la fidélité d'un modèle à la réalité musicale qu'il s'efforce de décrire : moins il y a de solution, plus le modèle est fidèle. Le cas limite est celui d'un CSP admettant comme solution unique la partition analysée (situation qui correspond aux "modélisations de partition" pratiquées par André Riotte et Marcel Mesnage [69], [88]).

Cette façon de voir l'analyse musicale permet de définir la notion de *fidélité* d'une analyse par rapport à la pièce musicale qu'elle modélise. Mais on ne dit rien sur la nature de l'analyse elle-même. Il va de soi que la qualité d'une analyse ne peut se réduire à la notion de fidélité que nous proposons ici, mais prend en compte d'autres aspects. Par exemple, une analyse d'une pièce musicale qui est codée informatiquement par une série de valeurs  $x_1, x_2, x_3, \dots$  (codes midi ou autres) peut consister en des variables  $X_1, X_2, X_3, \dots$  dont l'instanciation sera entièrement déterminée par une contrainte de la forme  $(X_1 = x_1) \wedge (X_2 = x_2) \wedge (X_3 = x_3) \wedge \dots$ . Une telle analyse est évidemment très fidèle. Mais elle est en même temps assez peu "analytique", car essentiellement *descriptive*.

Bien que cela ne soit pas immédiatement le propos, on peut préciser l'opposition entre "analytique" et "descriptif", en se plaçant au niveau du langage de contraintes utilisé. Ce langage est défini formellement à partir de variables ( $X_1, X_2, X_3, \dots$ ), de connecteurs logiques ( $\vee, \wedge$ ), de quantificateurs, et d'un ensemble de symboles non-logiques comprenant des relations (par exemple  $<$ ), des fonctions (par exemple  $+$ ), et des constantes (par exemple des codes midi). L'ensemble des *termes* du langage est le plus petit ensemble d'expressions contenant les variables et les constantes, et fermé par l'application des fonctions. Le langage de *contraintes* est le plus petit ensemble contenant les termes, et fermé par la combinaison de termes par relations ou connecteurs, et par la quantification des termes. En explicitant les éléments qui interviennent dans cette construction, on peut distinguer différentes classes de langages de contraintes (comme on le fait dans la distinction entre logique du premier et du deuxième ordre), et ceci conduit à définir différentes classes d'analyses. Ainsi, on pourra dire, par exemple, qu'une analyse est d'autant plus descriptive, que son langage de contraintes utilise plus de *constantes*.

# Chapitre 17

## Autres problèmes

### 17.1 Problème 13 : Echelles en quarts de ton

Ce problème a été initialement suggéré par Jean-Michel Bardez (Société Française d'Analyse Musicale). Le but est de produire des **échelles**<sup>1</sup> en **micro-intervalles**<sup>2</sup>, par exemple en quarts de ton. On s'intéressera ici aux échelles en quarts de ton définies de la façon suivante : l'échelle doit contenir au minimum un quart de ton et un demi ton, et pas d'autres intervalles. En particulier, pas de trois quarts de ton, même si ce qui suit pourra être étendu à n'importe quel micro-intervalle.

L'intérêt musicologique de l'utilisation du quart de ton n'est pas à prouver. L'intérêt d'utiliser des contraintes pour trouver de telles échelles est lié au caractère fortement combinatoire du problème (c'est un des rares cas où les contraintes sont à prendre dans un sens purement combinatoire). Ce n'est bien sûr pas difficile de produire une échelle en quarts de ton, ou quelques-unes. Un dénombrement simple<sup>3</sup> permet de montrer qu'il y a 75025 échelles en quarts de ton avec la définition ci-dessus. Mais il est inenvisageable d'écouter toutes ces échelles, sans parler de les "essayer". Les contraintes permettent de générer des sous-ensembles de ces échelles, vérifiant des propriétés choisies par l'utilisateur. Les contraintes sont ici à voir comme une manière de restreindre intelligemment l'espace des possibles. L'espace de recherche est à voir une sorte de base de données générée à la demande.

On prend avec  $n$  variables de domaines  $\{1/2, 1/4\}$ , qui représentent des intervalles de demi-ton et de quart de ton. (à noter le domaine peut être étendu, avec  $3/4$ , ou  $5/4$ ).

Une première contrainte impose que l'échelle soit d'une octave, la somme des variables doit être égale à 6. Problème, cela suppose d'avoir un nombre variable de variables. La modélisation la plus simple est de prendre le nombre maximum de variables et de rajouter 0 au domaine. Ce nombre maximum est de 24 dans le cas des quarts de ton (6 tons pour l'octave \* 4 quarts de ton par ton). En notant les variables  $(V_1 \dots V_{24})$ , la contrainte s'écrit alors :

$$\sum_{i=1}^{24} V_i = 6$$

Etant entendu que les contraintes servent ici non pas à *modéliser* un problème, mais à

---

<sup>1</sup>Voir échelle dans le glossaire en annexe

<sup>2</sup>Voir micro-intervalle dans le glossaire en annexe

<sup>3</sup>On cherche le nombre de  $v_1 \dots v_{24}$  à valeurs dans  $\{0, 1, 2\}$  dont la somme vaut 24. Dans le cas où il y a  $i$  variables de valeur 2, la somme égale à 24 impose exactement  $24 - 2i$  variables de valeur 1 et les autres à 0, soit au total  $24 - i$  variables non nulles. Il reste à placer les 1 et les 2 ce qui donne le coefficient du binôme  $C_{(24-i)}^i$  possibilités. Le dénombrement s'obtient en sommant les cas de  $i$  à 0 (seulement des 1) à 12 (seulement des 2), soit  $\sum_{i=0}^{12} C_{24-i}^i$ .

identifier dans cet ensemble les échelles qui respectent des propriétés fixées, nous proposons un ensemble de contraintes qui peuvent être posées séparément ou bien combinées *ad lib*.

- Symétrie :  $(V_n \dots V_1) = (V_1 \dots V_n)$
- Transposition limitée :  $\exists i < n$  tel que  $(V_i \dots V_n V_1 \dots V_{i-1}) = (V_1 \dots V_n)$
- Motif : l'échelle contient un motif fixé  $a_1 \dots a_k$  ( $a_1 \dots a_k \subset (V_1 \dots V_n)$ )
- Proportions minimum de demis ou quarts de ton,  $m$  fixé.  $\#\{i | V_i = 1/4 \text{ ou } 1/2\} \geq m$
- Harmonie en quarts de ton : cette contrainte, la plus intéressante, est plus complexe. Elle porte sur les futures harmonies à construire sur l'échelle. Elle a été introduite pour la raison suivante. Comme  $1/4 + 1/4 = 1/2$ , on peut trouver des échelles en quarts de ton sur lesquelles la plupart des accords construits soient en demi ton. En pratique, après avoir fait quelques tests aléatoires sur certains résultats, nous avons constaté qu'il arrivait souvent que l'oreille ne perçoive pas les quarts de ton dans les accords (ou bien, peut-être, les "rectifie" d'instinct). Plu précisément, l'oreille identifiait immédiatement les quarts de ton lorsque l'échelle était jouée comme une suite de notes, mais beaucoup plus difficilement lorsque l'on faisait jouer quelques accords ? La couleur harmonique de ces échelles en quarts de ton apparaissait alors comme tempérées. D'où l'idée de rajouter une contrainte pour imposer ce que l'on pourrait appeler une "couleur quarts de ton" aux accords construits dans l'échelle. D'abord, on choisit un **chiffre**<sup>4</sup> de référence, soit  $d_1 \dots d_k$ , par exemple 0, 4 et 7 par analogie avec l'accord parfait majeur. On réalise ce chiffre sur tous les **degrés**<sup>5</sup> de l'échelle et on impose qu'un nombre minimum des accords obtenus contiennent au moins un intervalle en quarts de ton.

En notant  $Int_i$  les intervalles de l'accord  $\{V_i, (\sum_{j=i}^{i+d_1} V_j) \dots (\sum_{j=i}^{i+d_k} V_j)\}$ , et avec  $m$  fixé la contrainte s'écrit :

$$\#\{i | \exists \alpha \in (2 * \mathbb{N} + 1) \alpha/4 \in Int_i\} \geq m$$

On obtient ainsi des dénombrements d'échelles. Ainsi il y a 4272 solutions en imposant un parfait majeur (0, 4, 7) en quarts de ton sur au moins huit degrés, 126 solutions pour la même contrainte sur au moins douze degrés, et pas de solution pour un accord de dominante (0, 4, 7, 10) en quarts de ton sur au moins douze degrés.

## 17.2 Problème 14 : Doigtés pour la guitare

Ce problème a fait l'objet d'une collaboration avec deux compositeurs et guitaristes de l'IRCAM, Mikhaïl Malt et Arturo Fuentes. Au premier abord l'énoncé est simple : il s'agit de trouver des doigtés pour jouer à la guitare une harmonie donnée. Mais en fonction du but recherché, les modélisations possibles sont nombreuses.

Avant d'entrer dans les détails, il faut préciser l'application d'un tel système à la CAO. Un système pour donner automatiquement des doigtés de guitare trouverait probablement des applications pédagogiques voire grand public. Ce n'est pas notre but, pour de nombreuses raisons techniques. Nous resterons volontairement dans le cadre de l'aide à la composition. Dans ce cadre, un système de doigtés (quel que soit l'instrument utilisé) ne doit pas tellement servir à imposer des doigtés pour une partition, ce qui reste évidemment du domaine de compétence de l'instrumentiste, mais plutôt à savoir si une partition est jouable ou non. Un compositeur qui écrit pour la guitare est rarement guitariste lui-même, et risque donc d'intégrer à sa partition

<sup>4</sup>Voir chiffre dans le glossaire en annexe

<sup>5</sup>Voir degré dans le glossaire en annexe

des accords injouables. Selon que le système trouve facilement des doigtés pour les accords, ou non, le compositeur a une indication pour savoir si la partition est jouable ou pas.

Il faut ensuite définir exactement l'objet de l'étude. Première question, que vaut-il mieux traiter : une mélodie (où l'on cherchera alors probablement à limiter les changements de **position** <sup>6</sup>), un accord, une suite d'accord ? Pour la mélodie, un tel système serait utile seulement dans un cas précis : limiter les changements de position (ce qui s'appliquerait d'ailleurs aux autres instruments à cordes). Mais les doigtés dans le cadre mélodique sont extrêmement personnels, surtout pour des musiciens de la qualité des interprètes de contemporain. En outre, on pourra toujours voir la mélodie comme un cas particulier de suite d'accords. En résumé, un tel système pour une mélodie serait peut-être intéressant pour donner des doigtés en déchiffrage (première lecture de la partition), probablement pas plus. Les cas d'un accord et d'une suite d'accords sont plus intéressants, car il existe des accords injouables à la guitare. D'ailleurs, très souvent les partitions fournies aux guitaristes ne donnent que les chiffres des accords et c'est à l'instrumentiste de les réaliser : il s'agit donc de reprendre ce principe en étendant la gamme d'accords traités à la musique contemporaine. On travaillera avec des suites d'accords.

Deuxième question, la nature de l'entrée. On peut choisir de traiter des accords donnés *in extenso*, sous la forme d'un ensemble de hauteurs MIDI par exemple. Cela laisse très peu de latitude pour les doigtés, qu'il s'agisse d'un accord ou d'un enchaînement. Ou alors des accords donnés sous la forme d'un **chiffre** <sup>7</sup>, à la manière notamment des accords de jazz, do cinquième par exemple. Cela sous-entend que les octaves sont autorisées et qu'il s'agit de trouver une réalisation de l'accord. <sup>8</sup> Dans ce cas, il faudra préciser si toutes les notes de l'accord doivent être jouées ou bien s'il suffit de rester dans cette harmonie (avec des solutions possibles comme do six fois pour do cinquième!), ou encore imposer un nombre minimum de notes de l'accord effectivement présentes dans l'accord joué.

Troisième problème, les nombreux paramètres intrinsèques au problème. La **scordature** <sup>9</sup> peut changer, il faudra tenir compte de la taille de la main, autoriser ou non les barrés, et l'on peut varier *ad lib*. Il faut laisser à l'utilisateur le choix du plus grand nombre de paramètres possibles.

**Doigtés pour un seul accord** Certaines contraintes physiques ou physiologiques sont évidentes, même si elles dépendent de la modélisation choisie. Il faut exprimer qu'une seule corde joue une ou zéro note, que la main a une taille restreinte (paramétrable, que l'on peut simplement exprimer en nombre de frêts), et que la main a au maximum quatre doigts. Cette dernière assertion pose problème dans le cas où d'un barré. Premièrement, la main n'a alors plus que trois doigts libres. Deuxièmement, on ne peut pas jouer de notes en dessous du barré. Troisièmement, dans le cas d'un accord complet, ce ne sont plus les cordes à vide qui sonnent mais les notes du barré. Le barré partiel pose encore d'autres problèmes. Ceci sera détaillé plus bas, on commencera par détailler le problème sans prendre en compte les barrés.

On peut choisir aussi de différencier les doigts, ou non. Distinguer les doigts complique un peu le problème, et n'offre qu'un intérêt restreint. Partant du principe que les doigtés de guitare notés sur les partitions ne distinguent pas les doigts, et donc qu'un guitariste est habitué à lire

<sup>6</sup>Voir position dans le glossaire en annexe

<sup>7</sup>Voir chiffre dans le glossaire en annexe

<sup>8</sup>Cela présente quelques ressemblances avec l'harmonisation automatique XXXREF-Chapitre1, avec deux différences notables : la condition des doigtés jouables, et le fait que le cadre harmonique est beaucoup plus large que l'harmonie classique.

<sup>9</sup>Voir scordature dans le glossaire en annexe

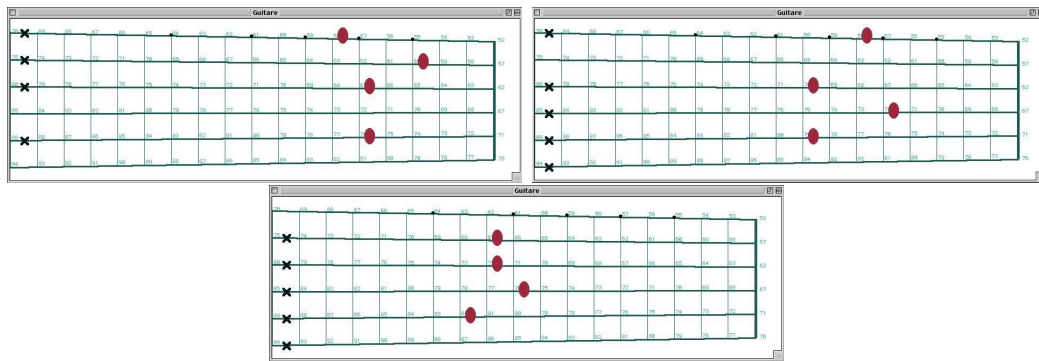


FIG. 17.1 – Plusieurs réalisations et doigtés possibles pour l'accord de do septième (do mi sol si bémol).

des doigtés "indistincts" à partir desquels il trouve lui-même la configuration qui lui convient le mieux, nous choisissons de ne pas différencier les doigts.

Pour toutes les raisons développées jusqu'ici, ce CSP peut être modélisé de plusieurs manières différentes. En résumé, les possibilités suivantes doivent idéalement être offertes :

- (i). Taille de la main paramétrable
- (ii). Sordature paramétrable
- (iii). **Ambitus**<sup>10</sup> paramétrable (la sonorité d'une note jouée en bas du manche peut être très différente de celle d'une note jouée en haut du manche)
- (iv). Possibilité d'autoriser ou d'interdire les barrés
- (v). Toutes les cordes jouées ou seulement certaines cordes en imposant un minimum de notes jouées
- (vi). Accord donné *in extenso*, ou bien comme chiffrage avec possibilité de renversement et transposition à l'octave
- (vii). En fonction des deux paramètres précédents, autorisation ou non de doubler certaines notes

Nous ne détaillerons qu'un cas, le suivant : donner des doigtés pour un accord, décrit par une note et un ensemble d'intervalles. Par exemple, l'entrée est do4, et les intervalles tierce quinte septième. On cherche alors une réalisation de l'accord, qui soit jouable à la guitare (voir plus loin), et telle que toutes les notes jouées appartiennent à l'accord, et que toutes les notes de l'accord soient présentes dans les notes jouées au moins une fois, à une **transposition**<sup>11</sup> d'un octave près. Les cordes à vide sont autorisées. A noter, la note donnée en entrée n'est pas nécessairement la basse de l'accord joué, les renversements sont autorisés. Ainsi la figure 17.1 montre différentes solutions admissibles au problème de do septième.

On prendra en entrée un chiffrage donné sur une forme un peu particulière : une note de référence donnée en hauteur MIDI  $N$ , et des intervalles  $(i_1 \dots i_k)$  entiers, donnés en nombre de demi-tons. Il est facile d'exprimer ainsi les chiffrages de la musique classique, et cela permet de représenter n'importe quel accord, ce qui est nécessaire pour l'application à la CAO. On cherche un doigté dont les notes sont soit dans l'accord qui correspond à ce chiffrage, soit

<sup>10</sup>Voir ambitus dans le glossaire en annexe

<sup>11</sup>Voir transposition dans le glossaire en annexe

transposées d'une octave. La note de référence n'ayant pas valeur de basse, voir ci-dessus, elle n'est pas comptée dans les notes jouées de sorte qu'il faut ajouter 0 aux intervalles pour l'obtenir. L'accord de do septième sera ainsi codé (60 (0 4 7 10)). Est aussi donnée en paramètre la scordature, sous la forme des notes jouées à vide  $S_1...S_6$ .

On prend comme variables les 6 cordes, soit  $V_1...V_6$ . La valeur d'une variable indique le nombre de **frêts**<sup>12</sup> avant la note jouée sur cette corde. Les domaines sont initialement l'ensemble des positions possibles  $\llbracket 1...18 \rrbracket$  auquel on ajoute 0 pour corde à vide, et une constante  $\epsilon$  pour corde non jouée. On considérera par commodité que  $\epsilon < 0$  (il suffira de l'implémenter à  $-1$ ). Le domaine des variables est donc au départ  $\{0, 1, ...18, \epsilon\}$ . La note jouée sur la corde  $i$  vaut en hauteur midi  $S_i + V_i$ . Enfin, on notera  $R = \{S_i + V_i \geq 0\}$  l'accord effectivement joué par le doigté donné (en supprimant les  $\epsilon$ ).

Première opération, réduire la contrainte "appartenir à l'accord", en filtrant les domaines de manière à éliminer les notes impossibles. Les domaines considérés sont donc dépendants de l'instance du problème, avec  $D_i$  égal à l'ensemble des positions dont la note résultante est dans l'accord, plus  $\epsilon$ .

A noter, on aurait pu choisir comme variables les quatre doigts et comme domaines les frêts de toutes les cordes. L'inconvénient de ce choix aurait été d'ajouter une contrainte délicate à écrire, pour éviter que deux doigts soient placés sur la même corde. Par ailleurs, il aurait fallu rajouter des variables pour les cordes à vide, ce qui n'aurait pas eu grand sens.

Certaines autres contraintes sont indiscutables. Dans le cas simplifié où les **barrés**<sup>13</sup> sont interdits. Première contrainte, la main d'un guitariste ayant quatre doigts, il faut interdire les configurations où plus de quatre  $V_i$  sont plus grands que 1 :

$$\#(V_i > 0) \leq 4$$

Deuxième contrainte, la main a une taille restreinte et ne peut pas jouer simultanément les premiers dix-huitième frêts. Il faut tenir compte ici du rétrécissement des frêts quand on monte dans les positions. Une mesure au double décimètre donne les rapports de taille entre les frêts, que l'on transforme en une fonction  $f$  indiquant pour une position donnée le nombre de frêt couverts par la main. Le cas le plus fréquent, où la main a une taille de trois frêt en bas du manche, est donné tableau 17.2 (la taille de la main est paramétrable sans changer la structure du CSP). En considérant que c'est le doigt le plus bas qui donne la position, la contrainte s'écrit alors :

$$\max(|V_i - V_j|) \leq f(V_1)$$

Troisième contrainte, il faut jouer l'accord. Les cas pathologiques comme la répétition d'une même note sur toutes les cordes ont en partie été éliminés par la deuxième contrainte, la guitare étant accordée par **quartes**<sup>14</sup>. A moins d'une scordature exceptionnelle, on ne considère pas nécessaire d'ajouter une contrainte pour limiter les redondances. Cependant on impose que toutes les notes de l'accord apparaissent bien dans la réalisation au moins une fois :

$$\forall j, (N + i_j) \in R$$

---

<sup>12</sup>Voir frêt dans le glossaire en annexe

<sup>13</sup>Voir barré dans le glossaire en annexe

<sup>14</sup>Voir intervalles dans le glossaire en annexe

| Position | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |
|----------|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|
| Frêts    | 3 | 4 | 4 | 4 | 5 | 5 | 6 | 6 | 6 | 7  | 7  | 6  | 5  | 4  | 3  | 2  | 1  | 0  |

FIG. 17.2 – Une fonction  $f$  donne le nombre de frêts couverts quand la main monte dans les positions. Ici les valeurs de  $f$  pour une main d'une taille de trois frêts en position zéro,

**Doigtés pour une suite d'accords** On notera  $V_i^j$  la variable représentant la  $i$ -ième corde dans le  $j$ -ième accord, et pour une instanciation donnée,  $v_p^j$  le minimum en  $i$  de  $v_i^j$ , qui donne la position de la main sur le manche pour le doigté  $v^j$ .

Une contrainte s'impose naturellement pour des raisons physiques : la main ne peut pas se déplacer sur le manche plus vite que la musique, il faut donc limiter les changements de positions. Cela dépend évidemment à la fois de la dextérité du guitariste et de la vitesse d'enchaînement des accords, on aura donc un paramètre  $s$  à choisir et à limiter le changement de position de \*saut\* par temps. Attention cependant, les variables instanciées à 0 pour "corde à vide" ou à  $\epsilon$  pour "corde non jouée" peuvent avoir des déplacements de plus de  $s$ . Ce qui s'écrit, pour chaque  $i$  :

$$\text{si } V_i^j \text{ et } V_i^{j-1} > 0, \text{ alors } |V_i^j - V_i^{j-1}| \leq s$$

Ensuite, la facilité d'enchaînement est aussi fonction des positions de la main. Ici encore il y a plusieurs possibilités selon les préférences du musicien. On peut par exemple préférer laisser le plus possible de doigts à la même place. Deuxième possibilité, limiter les changements de position, avec encore deux cas : soit préférer de petits changements de position, soit limiter le nombre de changements quitte à en avoir de grands. En pratique, après consultation de plusieurs guitaristes, le constat est que ceci est très personnel et dépendant du contexte, de sorte qu'il est illusoire de normaliser cette contrainte et que diverses possibilités doivent être proposées selon les cas.

Quoi qu'il en soit, il s'agit toujours de minimiser une (sorte de) distance d'un accord à l'autre, ie à chercher un plus court chemin dans la séquence des doigtés pour chaque accord. On notera  $d$  cette distance, avec comme possibilités pour  $d$  :

- (i). les doigts bougent le moins possible :  $d_1(v^j, v^k) = \#(v^j \cap v^k)$
- (ii). la main change de position le moins souvent possible :  $d_2(v^j, v^k) = 0$  quand  $v_p^j = v_p^k$ , et 1 sinon
- (iii). les changements de position sont le plus petits possibles :  $d_3(v^j, v^k) = |v_p^j - v_p^k|$
- (iv). si le barré est autorisé, on ajoutera un cas  $d_4$  pour préférer rester sur un barré, c'est-à-dire ne pas sauter d'un barré à un barré sur un frêt différent (voir ci-dessous)

Evidemment ces distances peuvent être combinées, par exemple la somme de  $d_1$  et  $d_2$  donne un instrumentiste qui préfère bouger la main le moins souvent possible, ou encore "si  $d_4 > 0$  alors  $d_3$  correspond à un instrumentiste qui veut absolument tenir ses barrés et sinon, ne pas trop bouger, etc. On ajoute finalement une contrainte :

$$\text{minimize}(d(V^j, V^{j-1})$$

.

**Cas des barrés** Le problème du barré est un peu compliqué à écrire. On peut le traiter en pratique en relâchant la contrainte sur le nombre de doigts : au lieu d'interdire plus de quatre

doigts, on autorise six doigts quand trois d'entre eux sont posés sur la position la plus basse. Cela complique l'écriture de la contrainte mais pas le problème. Par commodité d'écriture on peut ajouter une variable  $B^j$  qui indique premièrement s'il y a un barré, deuxièmement sur quelles cordes (de deux à six). Cette variable est unifiée à : si plus de 4  $V_i^j$  sont positifs,  $B^j$  vaut l'indice de la corde la plus basse qui n'est pas jouée à vide. Sinon,  $B^j$  vaut 0 (pas de barré pour ce doigté).

Pour les doigtés de plusieurs accords, on utilise les  $B$  pour donner les distances.  $d_4$  consiste à pénaliser les sauts de barrés :

$$d_4(v^j, v^k) = \text{si } B^j > 0, B^k > 0 \text{ et } v_p^j \neq v_p^k \text{ alors } 1 \text{ sinon } 0$$



# Chapitre 18

## Résumé

Le tableau 18.1 reprend les caractéristiques de chacun des CSPs musicaux exposés. On constate que le nombre de variables et la taille des domaines restent toujours assez petits pour de la recherche opérationnelle, mais trop grands cependant pour que les problèmes soient résolus à la main : ce sont bien des problèmes combinatoires. La nature musicale des problèmes est extrêmement variée, on y trouve la plupart des structures musicales traitées en CAO. Le nombre de solutions est variable et dépend des instances, mais on a plusieurs problèmes surcontraints. Enfin, le type de problème donné dans la dernière colonne indique ce que l'utilisateur souhaite : une solution, la solution optimale, ou toutes les solutions. Tous les cas de figures apparaissent. Cependant, le cas où il faut toutes les solutions reste limité à l'analyse musicale (problèmes 10, 11, 12), ou au problème très particulier des échelles (problème 13).

Le tableau 18.2 résume les types de contraintes utilisées dans les différents problèmes : égalité, différent, inégalité, appartenance, cardinalité, minimisation ou maximisation, et enfin fonction dans la dernière colonne qui ne désigne pas à proprement parler un type de contrainte, mais qui est cochée lorsque la contrainte fait appel à une fonction annexe coûteuse à calculer, la fondamentale virtuelle par exemple. Ce tableau permet de constater que l'on trouve tous les types de contraintes classiques dans ce panel de problèmes musicaux. A noter, la forte présence de "différent" ou alldifferent, qui s'explique par le fait qu'une structure musicale quelle qu'elle soit est souvent non dégénérée. Mais si très souvent dans les problèmes académiques les alldifferent peuvent se réduire en passant à des CSPs sur les permutations, ici ce n'est que rarement le cas parce que la taille du domaine n'est pas égal au nombre de variable (ou combinaison de variable) apparaissant dans le alldifferent.

| CSP | Var    | Dom      | Cont  | Nature      | Solutions | Type         |
|-----|--------|----------|-------|-------------|-----------|--------------|
| 1   | 60     | 1000     | 2 / 4 | Acc freq    | Oui       | Résolution   |
| 2   | 144    | 24       | 4     | Mélodies    | Non       | Résolution   |
| 3   | 50     | Perm     | 2     | Accords     | Non       | Optimisation |
| 4   | 25     | 40       | 2     | Rythmes     | +/-       | Résolution   |
| 5   | ?      | $\infty$ | 1     | Durées      | +/-       | Résolution   |
| 6   | 25     | 144      | 3     | Tempi       | Oui       | Optimisation |
| 7   | 20     | ?        | 1     | Durées      | Oui       | Optimisation |
| 8   | 40     | 6        | 4     | Mélodies    | Non       | Résolution   |
| 9   | 20     | Perm     | 1     | Notes       | Oui       | Résolution   |
| 10  | 10/500 | 4        | 4     | Accords     | Oui       | Génération   |
| 11  | 30     | Perm     | 1     | Rythmes     | Oui       | Génération   |
| 12  | 30     | 5        | 2     | Accords     | Non       | Résolution   |
| 13  | 24     | 3        | ?     | Intervalles | Bcp       | Génération   |
| 14  | ?      | 20       | ?     | Doigtés     | Bcp       | Rés / Opt    |

FIG. 18.1 – Principales caractéristiques des CSPs musicaux modélisés ci-dessus.

| CSP | = | $\neq$ | $\leq$ | $\in$ | Card | Min/Max | Fonction |
|-----|---|--------|--------|-------|------|---------|----------|
| 1   |   |        | X      |       | X    |         |          |
| 2   | X | X      |        | X     |      | X       |          |
| 3   |   |        | X      |       |      | X       |          |
| 4   |   | X      |        |       |      |         |          |
| 5   | X |        |        |       |      |         | X        |
| 6   | X |        | X      |       | X    | X       |          |
| 7   |   |        | X      | X     |      | X       | X        |
| 8   |   | X      |        | X     | X    |         |          |
| 9   |   | X      |        |       |      |         |          |
| 10  | X | X      |        | X     |      |         |          |
| 11  |   | X      |        |       |      |         |          |
| 12  | X | X      |        |       |      |         |          |
| 13  | X | X      |        | X     | X    |         |          |
| 14  |   |        | X      | X     | X    | X       |          |

FIG. 18.2 – Les types de contraintes utilisées.

# Chapitre 19

## Conclusions

Avant toute observation sur la nature de ces CSPs, il est important de rappeler que presque tous ont été posés par des musiciens, qu'ils soient compositeurs (problèmes 1, 2, 4, 5, 6, 7, 8) ou analystes (10, 11, 12). Cela confirme l'hypothèse initiale de ce travail, qui était que les contraintes trouvent une place naturelle en musique, que ce soit en CAO ou en analyse, plus généralement que pour la seule harmonisation automatique. L'observation de l'ensemble de ces problèmes permet de tirer plusieurs conclusions.

### 19.1 Variété multiple

Il faut d'abord souligner leur grande variété, à plusieurs niveaux. Premièrement, si d'un point de vue CSP on a toujours des variables entières de domaines finis, les structures musicales utilisées sont, elles, très différentes : accords, notes, rythmes, tempi, harmonie, gestes. Les tempi sont aisément modélisables par une suite d'entiers, les accords et les rythmes réclament une structuration plus fine, et on a parfois affaire à une modélisation très complexes comme dans le cas des canons rythmiques. Deuxièmement, les contraintes sont en général riches. Certains CSPs se réduisent à un système d'équations linéaires, mais la plupart comportent des combinaisons de contraintes d'égalités, inégalités, des contraintes plus délicates à traiter comme le alldifferent, et des contraintes de haut niveau comme des contraintes de capacité ou de cardinalité (voir tableau). Enfin, le but lui-même varie : on peut avoir des problèmes d'optimisation, de résolution (trouver une solution ou quelques unes), génération (trouver toutes les solutions). Plusieurs problèmes (Mauro Lanza, Nzakara) sont même surcontraints, et le but est alors de trouver une solution approchée.

### 19.2 Variables

Les variables ne se situent pas au niveau de la partition, mais le plus souvent en amont. Elles servent à représenter une caractéristique d'un objet musical : valeur MIDI pour une note, durée pour un rythme par exemple. Il y a toujours une étape de transformation voire de calcul pour passer des valeurs des variables à un objet musical que l'on peut dessiner, ou informatiquement éditer, sur une partition. Autre point, on a toujours des CSPs dont les variables sont homogènes, au sens où elles représentent les mêmes objets musicaux et les contraintes s'appliquent de la même manière sur toutes les variables. Les seuls exceptions sont les CSPs modélisés comme des intervalles (ou décalage dans la suite de Fibonacci), où une note de départ doit être choisie, et

les accords définis comme des paramètres fréquentiels. Le premier cas, il a été décidé de fixer la première valeur, ce qui convenait bien aux compositeurs. On pourra donc considérer avec des variables homogènes, ce qui épargne par exemple de les nommer.

### 19.3 Contraintes "oubliées"

En général, les CSPs ainsi présentés contiennent une ou deux contraintes de plus que la proposition initiale du compositeur. En effet, très souvent, celui-ci omet certaines contraintes qui sont intuitives pour le musicien. Ainsi un accord, sans considérations sur les voix, est *a priori* constitué de notes toutes différentes, et même strictement ordonnées (on identifie do-mi-sol et mi-do-sol), comme pour un rythme exprimé en onsets. De sorte qu'il a souvent fallu ajouter des contraintes pour formaliser le problème musical en CSP, par rapport à la proposition initiale du compositeur.

### 19.4 On ne veut pas de solution

Très important, le but n'est pas toujours de résoudre le CSP. En informatique, un solveur de contraintes est utilisé pour donner une ou des solutions à un problème combinatoire. Ce principe, problème  $\rightarrow$  solution, ne s'applique pourtant pas vraiment dans notre cas. D'abord, dans le cas des CSPs surcontraints, une stricte résolution consisterait à répondre "non". Ce n'est évidemment pas satisfaisant. Mais plus précisément, d'une part les solutions des CSPs en CAO sont toujours réécrites par le compositeur en fonction de critères esthétiques non formalisables, et d'autre part une solution approchée mais "qui sonne bien" sera préférée à une solution exacte "qui ne sonne pas". Pour citer Mauro Lanza, (problème 4) "Dans 90 pour cent des cas, la solution avec le moins d'erreurs était la bonne". Il faut donc, plutôt que résoudre classiquement le CSP, considérer le solveur comme faisant à la demande des propositions de solutions, éventuellement partielles ou approchées.

### 19.5 Solutions approchées

Si l'on intègre l'idée de solutions approchées, il faut le faire avec soin. La spécificité musicale peut donner quelques surprises à l'informaticien. Par exemple, le problème 6 montre qu'une même contrainte, l'égalité, peut se transformer en plusieurs fonctions d'optimisation très différentes. Un autre exemple se retrouve plusieurs fois, à propos d'une distance à optimiser. La notion de distance est naturelle en musique, même s'il ne s'agit pas nécessairement d'une distance selon la définition mathématique. Transformer une contrainte d'égalité en minimisation d'une distance ne peut pas se faire hors contexte. S'il s'agit d'intervalles, les musiciens sont habitués à tenir compte de données classiques sur la consonnance (cf problème 2). S'il s'agit de durée en revanche, une distance euclidienne convient (cf problème 7). Dans tous les cas, il ne suffit pas pour obtenir une solution approchée de relacher la contrainte n'importe comment.

Troisième partie

Recherche adaptative et applications  
musicales



# Chapitre 20

## Summary of Part III : Adaptive Search and Musical Application

This chapter is about the local search algorithm which has been chosen for the musical application, and presents the works done on the musical CSPs of chapter 2. Adaptive Search has been proposed by Philippe Codognet, see [36] for some recent developments. It follows local search main principles : random initialization, iterative improvement of the configurations. The notion of error function is refined at the variable level : for each variable, a cost or error function is defined, which represents the error of the current value of this variable with regard to the constraints. This error can be for instance the sum of the errors of the constraints where this variable appears. This allows to choose at each iteration a variable among the worst variable, and to modify in order to improve the configuration. Thus the neighborhood function is the domain of one of the worst variable. Adaptive Search is a metaheuristic, parametrized by a cost for each variable, a cost for the whole configuration (which may simply be aggregated from the variable's costs). A Tabu-like mechanism is added to prevent being trapped in local minima. When the moves of the worst variable does not allow to improve the cost, it is marked tabu for a fixed number of iterations.

Some of the musical CSPs have been solved with complete techniques based on Screamer [96], others with adaptive search. Let us just mention some conclusions from this works. Problem 2 has led to an musically interesting use of local search : the CSP is overconstrained ("very" overconstrained). The composer gathered some approximate results and sorted them according to their cost with regard to the latest constraint about a melodic profile. They were placed on the score sequentially, which is possible because the rhythmical structure in canon is supposed to be played repetitively. Generally, all the overconstrained CSPs were still solved with approximate results which were used by the composers.

Problems 4 and 8 have been integrated in OpenMusic with a particular architecture, were an *ad hoc* class was defined, which inherited from an "adaptive search" class and from a musical OpenMusic class, in order to benefit from the musical editors. But this was not possible to generalize with a reasonable ease of use.

Problem 7 has been solved by an algorithm adapted from AS but a little different, relying on a parallel argument both in mathematics and music. The main idea is to use the gcd of the whole sequence of played note as the least audible rhythmical value such that the rhythm can be written as multiples of a quarternote. At each iteration, this gcd is improved by local moves of the variables, and when it is not possible, the allowed moves are increased. It always terminates.

Problem 8 shows a great difficulty because of the capacity-like constraint. The cost for this constraint is hard to define because it may create artificial local minima, without any coherence with the problem structure.

# Chapitre 21

## Recherche Adaptative

### 21.1 Introduction

Comme cela a été présenté dans la partie 1, les méthodes heuristiques ou incomplètes sont utilisées depuis quelques décennies pour résoudre les problèmes fortement combinatoire. Elles s'opposent diamétralement aux méthodes "classiques", qui explorent systématiquement le domaine de recherche, avec des techniques pour restreindre cette exploration. Les méthodes complètes sont essentiellement structurées autour d'une notion de haut niveau définie sur le format CSP, l'arc-consistance. Pour résumer très grossièrement leurs principales caractéristiques (sauf cas particuliers) : elles conservent en mémoire l'espace visité, elles donnent toujours une réponse ou prouvent l'optimalité, elles sont par nature exponentielles en temps de calcul, elles sont déterministes.

Les méthodes incomplètes se placent sur un terrain très différent. Elles ne gardent pas de mémoire de l'espace visité (ou alors à très court terme), elles peuvent ne pas répondre, elles sont par nature indéterministes, et surtout, elles ont prouvé être extrêmement rapides sur de nombreux problèmes. Pour une description des métaheuristiques les plus connues, voir la partie 1.

Les méthodes incomplètes reposent sur l'idée de mesurer la qualité d'une instanciation par rapport aux contraintes, avec une fonction de coût sur l'espace des instanciations <sup>1</sup>. Il faut aussi définir une notion de voisinage sur l'espace de recherche, *ie* pour chaque instanciation, on suppose connaître un ensemble d'instanciations voisines. Par exemple, pour le problème SAT, le voisinage d'une instanciation peut être l'ensemble des instanciations qui ne diffèrent que par une valeur.

L'architecture générale est la suivante : à partir d'une instanciation aléatoire, explorer itérativement le voisinage, calculer le coût des configurations voisines, et si cela permet d'améliorer le coût, bouger vers ce voisin à partir duquel on répète le procédé. Ceci aboutit soit à une configuration solution de coût zéro, soit à une configuration telle qu'il n'est pas possible d'améliorer le coût sur son voisinage, un minimum local. Pour éviter d'y rester coincé, les algorithmes de recherche locale incluent des mécanismes tels que la mémoire Tabu, le refroidissement pour le recuit simulé, des resets aléatoires, etc (voir partie 1). Il s'agit de métaheuristiques, qui doivent être adaptées à chaque problème particulier (choix des fonctions de coût, paramétrages divers). Les algorithmes qui en découlent ont depuis quelques années prouvé leur très grande efficacité

---

<sup>1</sup>à valeur dans un espace à définir, en général les entiers, voir chapitre sur les soft constraints, là où les contraintes ne sont qu'à valeur dans  $\{\top, \perp\}$

sur des problèmes de grande taille.

La Recherche Adaptative est une heuristique de recherche locale proposé par Philippe Codognet [35] ou [37]. Elle appartient à la famille des GSAT [95], Walksat(OIP) [11], et Wsat(OIP) [108]. Les coûts y sont sauf cas particulier calculés à la Min Conflict [74]. Mais la recherche adaptative est une métaheuristique basée sur les variables dans le format CSP, ce qui permet de traiter des problèmes hétérogènes.

La recherche adaptative a donné des résultats très prometteurs sur des problèmes classiques comme les  $n$ -reines ou les carrés magiques, voir [36]. Daniel Diaz propose en *freeware* un code C et des benches pour la recherche adaptative<sup>2</sup>.

## 21.2 La Recherche Adaptative, présentation générale

L'idée intuitive est de diriger la recherche par des coût non pas sur les configurations, mais sur des variables. La structure du format CSP permet de calculer un coût pour une configuration, par exemple le nombre de contraintes violées, mais avec une analyse un peu plus fine, on peut aussi calculer des coûts au niveau de chaque variable, par exemple le nombre de contraintes violées dans lesquelles la variable apparaît.

On peut le voir comme une manière de projeter un coût global au niveau des variables (en prenant seulement les contraintes dans lesquelles la variable apparaît, ou bien à l'inverse comme une manière de collecter directement l'information au niveau des variables (à partir de quoi on en déduit un coût global, en sommant par exemple). Ce deuxième point de vue est assez naturel car il y a gros à parier dans un algorithme de recherche locale, avec des contraintes quelconques, qu'*avant* de donner un coût pour la configuration, on soit passé par de telles fonctions.

Ceci permet de sélectionner à chaque itération une plus mauvaise variable à modifier. On suppose qu'en changeant la valeur de la variable qui entre dans le plus grande nombre de contraintes violées, le coût total de la configuration a de grandes chances d'être amélioré.

La méthode de Recherche Adaptative (ou Adaptive Search, AS) prend en entrée un problème au format CSP avec les notation habituelles, variables  $V_1 \dots V_n$  de domaines  $D_1 \dots D_n$  avec comme contraintes  $C_1 \dots C_p$ . L'idée directrice de l'AS est une variante de celle de la recherche locale : en recherche locale, on transforme les contraintes de manière à définir un coût pour chaque configuration. En recherche adaptative, on transforme les contraintes de manière à définir un coût pour chaque variable dans chaque configuration.

Soit  $c = (Q_1 \dots Q_n)$  une configuration du problème des  $n$ -reines. On peut définir assez naturellement le coût de la configuration comme le nombre total de reines en prise. Le coût d'une reine dans la configuration peut par exemple se définir par : le coût pour  $Q_i$  pourrait être naturellement le nombre d'autres reines en prise avec  $Q_i$ . On sait alors exactement quelles reines sont "plutôt bien" ou "plutôt mal" placées dans la configuration.

Formellement, cela revient à projeter le calcul du coût global sur les variables. Pour pouvoir calculer le coût global d'une configuration  $c$ , il a fallu calculer un coût pour chaque contrainte, soit  $f_C$  la fonction de coût associée à la contrainte  $C$ . On attribue à  $V_i$  la somme<sup>3</sup> des coûts des contraintes dans lesquelles  $V_i$  apparaît. Pour les  $n$ -reines par exemple, la reine  $Q_{i_0}$  apparaît dans les contraintes  $Q_{i_0} + i_0 \neq Q_j + j$  et  $Q_{i_0} + i_0 \neq Q_j - j$  avec  $j \neq i_0$ , ce qui revient exactement à compter le nombre de reines en prise sur les diagonales montantes et descendantes de  $Q_{i_0}$ .

<sup>2</sup><http://contraintes.inria.fr/diaz/adaptive/>

<sup>3</sup>somme, ou maximum, ou une autre opération raisonnablement monotone

Cette opération permet de déterminer un ensemble de "plus mauvaises variables", celles dont le coût est le plus élevé de la configuration, soit  $E_c$ . On choisit une variable  $V_{pire}$  aléatoirement dans  $E_c$  (hors Tabu, voir ci-dessous). La valeur de  $V_{pire}$  est remplacée par toutes celles de son domaine  $D_{pire}$ . La meilleure valeur de  $D_{pire}$  est retenue, c'est-à-dire celle qui optimise le coût de la configuration totale (et non le coût de  $V_{pire}$  - ce qui n'aurait pas de sens).

Toujours dans le cas des  $n$ -reines, l'algorithme part d'une configuration aléatoire avec une reine par ligne. Une étape de l'algorithme consiste à chercher la reine qui est en position de manger le plus de ses consœurs (calcul des coûts pour chaque variable). Cette reine est choisie pour bouger sur sa ligne, dont on lui fait essayer toutes les autres cases (exploration du domaine). La position où la reine mange le moins possible de ses consœurs est retenue (choix de la meilleure valeur pour le coût global).

[42] détaille cet exemple avec des benches excellents : l'algorithme répond presque en temps linéaire, allant jusqu'à résoudre des instances de taille 10000, là où une méthode complète (ILOG-Solver) échoue à partir d'instances de taille 200. La recherche adaptative se compare également favorablement avec des benches en Localizer (source, [73]) sur des machines comparables (Pentium 800 pour Localizer, Pentium 733 pour la recherche adaptative), avec 0,15 secondes contre 5,04 sur une instance de taille 1024, et 1320 secondes contre 10096 sur une instance de taille 65536.

Pour éviter de se trouver piégée dans un minimum local, la méthode ajoute un mécanisme "à la Tabu", sous la forme d'une mémoire adaptative. Si, dans la dernière étape, aucune valeur de  $D_{pire}$  ne permet d'améliorer le coût de la configuration, alors la variable  $V_{pire}$  est marquée Tabu et écartée de la recherche pendant un nombre fixé d'itérations, soit  $l_{tabu}$ . Précisément, cela revient à explorer  $E_c$  sauf les variables marquées Tabu.

## 21.3 Algorithme

La recherche adaptative est une métaheuristique dont le schéma général est donné ci-dessous, indépendamment du problème.

**Entrée** Un problème au format CSP, avec des fonctions de coût, soit :

- des variables  $V_1 \dots V_n$  de domaines entiers, ou en tous cas finis
- des contraintes  $C_1 \dots C_p$  avec des fonctions de coût associées
- une opération de combinaison des coûts sur les variables (la plupart du temps, la somme des coûts des contraintes dans lesquelles la variable apparaît)
- une fonction de coût globale (la plupart du temps, la somme des coûts des contraintes)

**Sortie** une solution ou un minimum local

### Algorithme

**Initialisation** à une configuration aléatoire

**Répéter**

**Calculer** le coût de chaque variable dans la configuration

**Sélectionner**  $V_{pire}$  une variable aléatoirement parmi les variables de coût le plus élevé, non marquées Tabu

**Calculer** le coût global de la configuration en remplaçant  $V_{pire}$  par les valeurs de son domaine

|    |    |    |    |    |
|----|----|----|----|----|
| 23 | 6  | 19 | 2  | 15 |
| 4  | 12 | 25 | 8  | 16 |
| 10 | 18 | 1  | 14 | 22 |
| 11 | 24 | 7  | 20 | 3  |
| 17 | 5  | 13 | 21 | 9  |

FIG. 21.1 – Un carré magique de côté 5. Les sommes sur les lignes, colonnes et diagonales sont constantes, égales à 65.

**Si** il existe une valeur améliorant le coût global, affecter  $V_{pre}$  à cette valeur

**Sinon** marquer  $V_{pre}$  Tabu

**Jusqu'à** annuler le coût global (solution), ou bien atteindre un nombre maximum d'itérations

## 21.4 Exemple pour les carrés magiques

Le problème des carrés magiques consiste à placer les  $n^2$  premiers entiers sur un carré de côté  $n$ , de telle manière que les sommes des nombres sur une même colonne, ligne ou diagonale soient égales. Ce qui détermine la somme, qui vaut  $n(n^2 + 1)/2$ <sup>4</sup> mais que l'on notera  $s$  par commodité, la valeur de  $s$  étant connue. Le tableau 21.1 donne un exemple de carré magique.

Pour un algorithme de recherche adaptative, on peut prendre les opérations suivantes :

**Variables** les  $n^2$  premiers entiers, avec comme domaine les permutations (domaine pour une variable : permutation avec toutes les autres)

**Contraintes** les équations de somme sur les lignes, colonnes et diagonales. A une équation  $V_{i_1} + V_{i_2} + \dots + V_{i_n} = s$ , on associe comme coût  $v_{i_1} + v_{i_2} + \dots + v_{i_n} - s$  sans valeur absolue

**Opération de combinaison** la valeur absolue de la somme des erreurs des contraintes dans lesquelles la variable apparaît, (et pas la somme des valeurs absolues, ceci afin d'éviter de cumuler artificiellement des erreurs de signes opposés)

**Coût global** somme des valeurs absolues des erreurs sur les contraintes

Cela donne un algorithme qui, partant d'une configuration aléatoire des  $n^2$  premiers entiers dans le carré, suit itérativement une étape telle que détaillée figure 21.2 (exemple tiré de [42]). Cet exemple est une étape de l'algorithme en temps normal, c'est-à-dire sans intervention du mécanisme tabu : la variable sélectionnée permet une modification qui améliore le coût global, et un swap est effectué.

Si la modification n'avait pas permis d'améliorer le coût global, la variable sélectionnée aurait été marquée tabu, et l'algorithme aurait continué sans elle pendant un certain nombre d'itérations (on aurait considéré une autre variable de plus mauvais coût s'il y en avait eu une, ou dans le cas de l'exemple, la deuxième variable de coût le plus mauvais qui vaut 21 ici).

Si aucune variable n'avait permis d'améliorer le coût global, *ou bien si* la liste tabu avait été pleine (condition moins forte quand la longueur tabu n'est pas égale au nombre de variables), alors tout ou partie des variables aurait été ré-initialisé aléatoirement.

<sup>4</sup>Soit  $s$  la somme qui doit être égale partout. En sommant sur les  $n$  colonnes, on obtient  $s \times n$ . En sommant sur tout le carré, on obtient  $\sum_{i=1}^{n^2} i$ , soit  $n^2(n^2 + 1)/2$ . Les deux sont égaux d'où  $s$ .

| Calcul des coûts<br>sur les contraintes                                                                                                                                                                                                                                                                                                     |                                                                                   | Calcul des coûts<br>sur les variables,<br>plus chère = 23                                                                                                                                                                                                         | Calcul du coût global<br>pour chaque swap,<br>meilleur swap = 33                                                                                                                                                                                                                      | Nouvelle configuration<br>après swap                                                                                                                                                                                                                                         |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <div> <div> <div>11</div><div>7</div><div>8</div><div>15</div> </div> <div> <div>16</div><div>2</div><div>4</div><div>12</div> </div> <div> <div>10</div><div>6</div><div>5</div><div>3</div> </div> <div> <div>1</div><div>14</div><div>9</div><div>13</div> </div> </div> <div> <div>4</div><div>-5</div><div>-8</div><div>9</div> </div> | <div>-8</div> <div>7</div> <div>0</div> <div>-10</div> <div>3</div> <div>-3</div> | <div> <div>8</div><div>2</div><div>1</div><div>8</div> </div> <div> <div>4</div><div>8</div><div>16</div><div>9</div> </div> <div> <div>6</div><div><b>23</b></div><div>21</div><div>1</div> </div> <div> <div>1</div><div>2</div><div>5</div><div>9</div> </div> | <div> <div>39</div><div>54</div><div>51</div><div><b>33</b></div> </div> <div> <div>53</div><div>67</div><div>61</div><div>41</div> </div> <div> <div>45</div><div><b>57</b></div><div>57</div><div>66</div> </div> <div> <div>77</div><div>43</div><div>48</div><div>41</div> </div> | <div> <div>11</div><div>7</div><div>8</div><div><b>6</b></div> </div> <div> <div>16</div><div>2</div><div>4</div><div>12</div> </div> <div> <div>10</div><div><b>15</b></div><div>5</div><div>3</div> </div> <div> <div>1</div><div>14</div><div>9</div><div>13</div> </div> |

FIG. 21.2 – Une itération de recherche adaptative pour le problème des carrés magiques, qui permet de passer d’une configuration de coût global 57 à une configuration de coût global 33. A gauche, la configuration courante, sur laquelle on calcule les erreurs pour les lignes, colonnes et diagonales. Ensuite, ces erreurs sont additionnées pour chaque variable, *ie* chaque case du carré, ce qui permet de sélectionner la plus mauvaise variable, en gras. Enfin, sont calculés les coûts globaux quand on permute hypothétiquement cette plus mauvaise variable avec chacune des autres, ce qui permet de choisir la permutation qui améliore le plus le coût global, en gras.

Les carrés magiques, comme les  $n$ -reines, se prêtent bien à ce type d’algorithme. D’abord, la modélisation avec comme mouvements possibles d’une variable (voisinage), les permutations avec les autres variables, élimine une contrainte *alldiff*. Ensuite, intuitivement, le fait de calculer les coûts au niveau des cases du carré est pertinent : une variable de fort coût calculé est aussi une variable qui dépasse beaucoup dans le carré (w.r.t. les sommes), et que l’on doit donc intuitivement bouger pour améliorer la situation globale.

En pratique, les benches donnés dans [42] sont très compétitifs. La recherche adaptative résout des instances jusqu’à 50 lignes et colonnes, là où des méthodes complètes (GNU-Prolog ou ILOG-Solver) s’arrêtent à des instances de taille 10. La recherche adaptative se compare favorablement à une métaheuristique tabu programmée en Localizer, donnée dans [73] (qui explore à chaque itération *tous* les échanges de deux variables sur le carré, soit un ordre de grandeur de plus que la recherche adaptative). Sur des machines comparables (Pentium 800 pour Localizer, Pentium 733 pour la recherche adaptative), la recherche adaptative est environ cent fois plus rapide : 3.4 secondes contre 313.2 pour une instance de taille 10, 203.4 secondes contre 23158 pour une instance de taille 50.

## 21.5 Fonctions de coût

L’une des choses les plus délicates est de trouver des fonctions de coût adaptées à chaque problème, en particulier les fonctions projetées sur les variables. Ce problème est accentué dans le cas d’un solver générique, où l’on veut une bibliothèque de fonctions de coût, compatibles entre elles.

Pour une même contrainte  $C$ , il existe une infinité de fonctions de coût  $f_C$ . La seule chose requise est que le coût vaut 0 quand la contrainte est satisfaite. Mais la réciproque n’est même pas nécessaire, cf l’exemple des carrés magiques détaillé ci-dessus où l’on admet des coûts négatifs pour chaque contrainte, ce qui autorise une variable de coût zéro non pas comme zéro (sur les lignes) plus zéro (sur les colonnes), mais comme par exemple  $-3$  (sur les lignes) plus 3

(sur les colonnes). De sorte qu'il s'agit de trouver pour chaque problème des fonctions de coût efficaces.

### 21.5.1 Une bonne fonction de coût

Pour une même contrainte, certaines fonctions seront meilleures pour tel problème et d'autres pour tel autre, comme le montrent les deux exemples suivants, tirés de [42].

Un exemple flagrant est le cas de la contrainte  $alldiff(V_1...V_n)$ . Intuitivement, on lui associe comme coût pour  $V_i$ , le nombre d'occurrences de la valeur de  $V_i$  dans le  $n$ -uplet  $(V_1...V_n)$ , moins un. Dans le cas des  $n$ -reines, cela revient à compter le nombre de reines en prise, et cela fonctionne bien, cf ci-dessus.

Mais le problème des séries tous intervalles (problème 9), donne des résultats différents. Dans une modélisation en permutations, avec variables  $V_1...V_n$ , il ne reste que la contrainte sur les intervalles,  $alldiff(|V_2 - V_1|...|V_n - V_{n-1}|)$  à prendre en coût<sup>5</sup>. Toutes les variables apparaissent de la même manière dans la contrainte.

Le coût total de la configuration peut se calculer selon les intervalles manquant dans  $1 \dots n - 1$ . On considère qu'une solution présente tous les intervalles, et que l'erreur d'une configuration est le nombre d'intervalles qui ne sont pas présents dans la configuration. Il faut d'abord calculer pour chaque intervalle  $j$  son nombre d'occurrences, soit

$$O_j = \#\{i \in [1, n - 1], |V_{i+1} - V_i| = j\},$$

d'où le coût global qui s'écrit

$$\sum_{j=1}^{n-1} O_j - 1.$$

Ceci revient à compter comme coût de la configuration le nombre d'intervalles manquants, soit le même type de fonction de coût pour la contrainte  $alldiff$  que dans le problème des  $n$ -reines, où l'on compte la redondance des valeurs apparaissant dans le  $alldiff$ .

Pourtant, [42] propose une autre fonction de coût plus efficace. Il s'agit de pondérer dans le coût global les intervalles par leurs tailles. Cela s'écrit

$$\sum_{j=1}^{n-1} (\text{si } O_j = 0 \text{ alors } j \text{ sinon } 0)$$

. L'idée sous-jacente est qu'il est plus difficile dans ce problème en particulier de placer les grands intervalles. Musicalement, un intervalle d'une **seconde**<sup>6</sup> ou d'une tierce peut se placer à peu près n'importe où dans la série, sur n'importe quel **degré**<sup>7</sup> sauf le dernier. Mais, un intervalle d'une quinte augmentée ne peut se placer que sur les deux premiers degrés, ou bien en diatonique sur le do (intervalle do sol $\sharp$ ), le do $\sharp$ (la), le ré (la $\sharp$ ) et le ré $\sharp$ (si). Quant à la septième augmentée, elle ne peut être placée que sur le do (intervalle do si).

On voit bien que les intervalles grands sont plus durs à placer, ce qui se traduit dans la fonction de coût par une pondération de l'occurrence d'un intervalle par sa taille. Les benches

<sup>5</sup>On considère la variante non musicale, non cyclique, sans ajouter la quarte augmentée, sans tenir compte de  $|V_n - V_1|$ .

<sup>6</sup>Voir intervalle dans le glossaire en annexe

<sup>7</sup>Voir degré dans le glossaire en annexe

donnés montrent de très bonnes performances de cet algorithme notamment contre ILOG-Solver. La recherche adaptative résout des instances de taille 24 en moins d'une seconde.

Cet exemple montre que la contrainte alldifferent qui pourtant n'a rien d'exotique peut se traduire en coût de plusieurs manières différentes. Ce sera également le cas des contraintes égalité ou inégalité, avec des coûts soit qui mesurent la distance à une solution, soit à la Min-conflict en 0 ou 1.

Enfin, dans l'application musicale, il arrivera que l'on choisisse des fonctions de coût mesurant par exemple une distance musicale, comme dans le problème 2 où le compositeur traduit une contrainte d'égalité en une distance selon le cycle des quintes. Musicalement, dans le cadre classique par exemple, une seconde est plus loin de l'unisson qu'une quinte, même si les valeurs MIDI sont plus proches dans le cas de la seconde que de la quinte.

En résumé, trouver seulement une fonction de coût pour une contrainte est déjà dépendant de la structure du problème considéré ainsi que de paramètres musicaux que l'on ne peut pas automatiser.

### 21.5.2 Combiner les coûts

Le problème suivant est qu'il ne suffit pas de pouvoir comparer deux fonctions de coûts pour deux contraintes avec une relation d'ordre. Le problème 8 en donne un exemple, ce qui sera détaillé ci-dessous. Il s'agit de contraintes de capacités sur des variables qui ne sont pas de décision, autrement dit, de contraintes sur les sommes partielles des variables. Quelle que soit l'option choisie, on aboutit à un problème de cohérence. Ceci est dû au fait que la variable en position  $i$  apparaît dans  $n - i$  contraintes, de telle manière que la première variable est fortement contrainte, la variable, une fois moins contrainte, ainsi de suite.

Chaque variable est donc contrainte à un ordre différent, décroissant sur la séquence, ce qui pose un problème pour définir les coûts : faut-il tenir compte de ceci dans les coûts, avec un domaine de valeurs beaucoup plus grand pour le coût des premières variables que des dernières (ceci revient en pratique à construire la séquence à partir de la première variable, variable par variable) ? Faut-il niveler les coûts artificiellement ou les moyenner (ceci aboutit à des minima locaux artificiels et qui donnent un phénomène à la *heavy-tails* de Gomes [109] avec des sous-problèmes de complexité presque égales au problème initial) ? En pratique, pour des problèmes de sac à dos par exemple, les coût utilisés dans la littérature (et qui correspondent à nos coût *globaux*) sont de type : "valeurs des objets choisis dans le sac" moins la capacité du sac. Nous n'avons pas trouvé mieux.

Dans le cas de l'application musicale, on aura souvent un problème qui trouve sa source dans la notation en midicents. Les hauteurs notes sont codées en midicents par multiples de 100 dans le **tempérament**<sup>8</sup> classique. Pour une contrainte d'égalité entre deux notes,  $N_1 = N_2$ , on peut choisir comme coût naturellement la distance entre les valeurs des notes, soit  $|n_2 - n_1|$ . Dans ce cas, le pas du domaine de valeur de la fonction de coût est de 100 également, avec des coûts qui se comptent en 1000 comme ordre de grandeur. Cela n'a pas de sens par rapport à des coûts de contraintes différent par exemple, de domaine de valeurs  $\{0, 1\}$ . En pratique, cela revient à hiérarchiser les coûts ou créer des islands, voir ci-dessous.

On peut définir la compatibilité des fonctions de coût intuitivement comme suit : deux fonctions de coût sont compatibles si l'une ne domine pas l'autre, *ie* si leurs domaines de valeurs sont identiques ou si une variable peut être alternativement sélectionnée en raison d'une erreur

<sup>8</sup>Voir tempérament dans le glossaire en annexe

sur l'une ou l'autre alternativement. En d'autres termes, il en faut pas qu'une fonction de coût crée un minimum local artificiel. Les coûts à la Min-conflict (pour une variable, le nombre de contraintes violées dans laquelle elle apparaît) sont en général bien adaptés dans le cas de problèmes hétérogènes.

### 21.5.3 Hiérarchiser les coûts

On peut cependant choisir de multiplier une fonction de coût pour forcer la contrainte à être vérifiée. Supposons avoir deux fonctions de coût de domaines de valeur 0 et 1,  $f_1$  et  $f_2$ , et  $X$  une variable apparaissant dans les deux contraintes. Supposons maintenant que la contrainte 1 soit impérative, mais que la contrainte 2 soit une préférence. On pondère  $f_1$  par 2. Dans ce cas, si  $X$  est sélectionnée comme variable à modifier, son domaine est exploré de manière à minimiser  $2f_1 + f_2$  (en réalité le coût global correspondant, il s'agit bien sûr d'un cas simplifié). Il est évident que c'est une valeur du domaine de  $X$  vérifiant la contrainte 1 qui sera choisie, sauf s'il n'en existe pas ou s'il existe une valeur qui vérifie les deux contraintes. La recherche se fait en pratique avec comme voisinage le domaine de  $X$  filtré par  $C_1$  (même si ce n'est pas un vrai filtrage).

Cela rappelle une idée de Jimmy Lee, qui étudie dans [47] *islands* de valeurs pour la recherche locale. Une île pour une contrainte  $C$  est un ensemble de configurations tel que l'on puisse s'y déplacer par des mouvements locaux, en respectant toujours la contrainte  $C$ . Le problème pour raisonner sur les *islands* identifié par Lee est que l'intersection de deux *islands* n'en est pas une (c'est évident si on pense à la définition comme à une définition de connexité).

Dans le cas des *islands* comme des fonctions de coût pondérées, il est intéressant de pouvoir préciser des contraintes dures de préférences, mais le risque d'un point de vue recherche locale est de perdre la diversification.

### 21.5.4 Générer les coûts

Tous ces problèmes se retrouvent quand il s'agit de générer automatiquement des fonctions de coût, ce qui sera notre but dans le cadre d'un solveur par recherche adaptative. Alors que l'on vient de noter que les bonnes fonctions de coût sont fortement dépendantes de la nature du problème, et ceci pour plusieurs raisons différentes, il s'agit de donner automatiquement des fonctions de coût valables dans n'importe quel cas. Générer automatiquement les fonctions de coût passe par un compromis entre toutes ces remarques.

Finalement, nous avons retenu deux possibilités, des coûts un peu adaptés, et des coûts à la Min-conflict qui ont l'avantage de minimiser l'ensemble des désagréments. Tout ceci est détaillé dans la partie 4 sur la librairie OMClouds.

## 21.6 Paramétrage

La recherche adaptative compte un certain nombre de paramètres à régler.

### 21.6.1 Complexité

Supposons que le calcul du coût d'une variable soit en  $a$  opérations, et que le calcul du coût global de la configuration soit en  $b$  itérations. Une étape de l'algorithme suppose de calculer

le coût de toutes les variables  $a * n$ , plus le coût global de la configuration  $b$ , plus les coûts des remplacements de  $V_{pre} \# D_{pre} * b$ . Soit au total  $n * a + (1 + \# D_{pre}) * b$  opérations. Dans le cas, probable, où l'opération de combinaison est la somme, on a  $b = n * a$ , et le nombre d'opérations est de  $n * a * (\# D_{pre})$ . Cela signifie que chaque itération prend un temps linéaire en la taille des domaines, en le nombre de variables et en le calcul du coût d'une variable.

### 21.6.2 Minimum local

Le fait de progresser variable par variable permet de représenter commodément l'AS dans le graphe de la fonction de coût globale. Il faut faire attention à distinguer deux types de minima locaux. A chaque fois qu'une variable  $V$  est marquée Tabu dans l'AS, cela signifie que l'on a un minimum local de la fonction de coût globale sur la droite  $V$  (géométriquement, sur la section du graphe de la fonction par le plan vertical passant par la configuration courante et de direction l'axe des  $V$ ). De la même manière, pour plusieurs variables marquées Tabu, on a un minimum local de la fonction de coût dans le sous espace défini par ces variables.

A supposer que la longueur Tabu le permette, quand toutes les variables sont marquées Tabu, on a un minimum local de la fonction de coût globale au sens habituel du terme (sur tous les axes). Ce n'est que dans ce cas que la configuration trouvée est minimale quand on modifie la valeur d'une des variables.

### 21.6.3 Liste Tabu

La liste Tabu n'en est pas tout à fait une au sens de la recherche Tabu. On peut distinguer deux cas : premièrement, un couple  $(V_1, v_1)$  est visité puis on passe à un couple  $(V_2, v_2)$  qui améliore le coût global (cas standard, il faut l'espérer). Deuxièmement, de  $(V_1, v_1)$  on passe à  $(V_2, v_2)$  qui n'améliore pas le coût global. Le marquage Tabu de  $(V_1, v_1)$  sert ici à ne pas boucler en revenant à  $(V_1, v_1)$ . La liste Tabu de la recherche adaptative ne contient que les variables du deuxième cas.

### 21.6.4 Longueur Tabu

La longueur Tabu conditionne la ténacité de l'algorithme et la valeur des minima locaux. Plus la longueur Tabu est grande, plus l'algorithme cherche autour de configurations supposées bonnes. Avec une longueur Tabu infinie (ou  $\geq n$ ), on tente d'améliorer le coût de toutes les variables, et dans ce cas on obtiendra de "vrais" minima locaux de la fonction de coût. Une longueur Tabu de  $t$  garantit que les minima locaux trouvés sont optimaux sur les  $t$  dernières variables.

### 21.6.5 Aléa

Il y a deux sources de hasard dans la recherche adaptative. Premièrement, on choisit  $V_{pre}$  aléatoirement dans  $E_c$ . Deuxièmement, on re-initialise aléatoirement les valeurs quand un nombre maximum d'itérations est atteint.

Le reset est essentiel pour faire fonctionner l'algorithme, comme d'ailleurs en recherche locale, voir les travaux de Gomes et al [51] pour une étude précise de l'importance des resets dans le cas de phénomènes de *heavy tails*, sous problèmes dont la résolution prend exponentiellement plus de temps à résoudre que tous ceux qui ont déjà été rencontrés (pour une technique complète

appliquée à SAT). On peut reseter toutes les variables ou une partie d'entre elles. Nous avons tenté d'autres options sur le problème 4 des rythmes asynchrones, en essayant de structurer les reset autour de la liste Tabu : seules les variables de la liste Tabu étaient réinitialisées (manière de diversifier la recherche ailleurs que là où le mouvements locaux ne permettaient pas d'améliorer le coût). Les résultats n'ont pas été bons.

### 21.6.6 Opération de combinaison

L'opération de combinaison retenue aura toujours été l'addition. Il se peut cependant qu'il vaille mieux choisir, comme dans le cas des séries tous intervalles, des fonctions *ad hoc*.

# Chapitre 22

## Résolution des CSPs musicaux

Ici est présentée la résolution des CSPs exposés au chapitre 2. Nous conservons les mêmes notations.

### 22.1 Problème 1 : Suite d'accords avec note commune

On prend évidemment la représentation en fréquences. Pour les contraintes, le plus simple est de définir une simili-distance  $d$  entre deux accords, soit par le nombre de notes communes, soit par une distance en valeur absolue. Dans le cas où on veut une note commune par exemple, il suffit de prendre  $d(\alpha_{i_1}, \alpha_{i_2}) = \min_{j_1, j_2} |\alpha_{i_1}^{j_1} - \alpha_{i_2}^{j_2}|$ . L'inconvénient (propre à la représentation choisie, qui minimise le nombre de variables) est évidemment que l'on doit calculer les accords à partir des paramètres de fréquence à chaque fois que l'on calcule la distance. Ce problème a également été résolu en Screamer.

### 22.2 Problème 2 : Harmonisation à structure rythmique donnée

Ce problème a été résolu en recherche adaptative.

D'un point de vue technique, inutile de s'attarder sur la résolution de ce CSP, rendu délicat par la nature complexe des structures musicales utilisées. Notons juste qu'à chaque itération, le nombre de calculs à effectuer est énorme (fondamentales virtuelles, textures, distances), et que le simple calcul d'une contrainte est donc complexe et long. L'ensemble est donc très lent même si la convergence en nombre d'itérations peut être rapide. C'est un problème qui se retrouve souvent pour des applications musicales. Le plus intéressant ici est plutôt l'utilisation très particulière faite par le compositeur de l'algorithme, qui montre bien comment un utilisateur musicien peut jouer avec le système qu'on lui fournit quitte à le détourner de sa fonction première.

Premièrement, la nature cyclique du problème permet de juxtaposer des solutions dans la partition. Georges Bloch a fait tourner le programme de résolution en récupérant des solutions (forcément approchées), qu'il a classées ensuite selon leur qualité telle que mesurée par le coût global, puis accolées dans la partition. C'est pour Georges Bloch une manière de montrer musicalement le processus en marche, avec une harmonie qui progresse continuellement vers la meilleure solution.

Deuxièmement, Georges Bloch a joué sur l'initialisation. Il a par exemple fixé l'initialisation à un parfait canon, à la fois rythmique (assuré par la représentation du problème) et surtout mélodique. L'initialisation vérifie alors parfaitement la dernière contrainte. La résolution consistait à détériorer cette structure de canon parfait pour respecter les autres contraintes.

Enfin, le compositeur a beaucoup exploité le système de poids et de coûts. Il explique avoir testé plusieurs combinaisons de pondération sur les fonctions de coûts. Finalement, il a même utilisé le coût de la dernière contrainte au carré (ce qui est évidemment possible, cela fait partie des fonctions de projection cohérentes pour la recherche adaptative).

## 22.3 Problème 3 : Classement d'accords

OM contient déjà un algorithme basé sur le TSP Approximation Algorithm, en théorie des graphes. Cet algorithme a le mérite d'être extrêmement rapide. L'inconvénient est qu'il est conçu de manière à optimiser le nombre total de notes communes (ou la distance totale parcourue), avec des solutions bonnes dans l'ensemble mais qui parfois ont des trous, *ie* deux accords successifs sans aucune note commune. Musicalement, cela peut poser un problème, si le but est d'obtenir une continuité dans un matériau au départ aléatoire. La programmation par contraintes apporte ici plus de souplesse, avec la possibilité d'ajouter une contrainte telle que celle décrite ci-dessous.

L'inconvénient d'utiliser une méthode incomplète est évidemment de ne pas pouvoir prouver l'optimum (en réalité, de ne même pas savoir quand arrêter la recherche, à part avec un nombre d'itérations maximum). Avec l'Adaptive Search, nous avons testé deux stratégies. La première repose sur un marquage des minima locaux : on stocke en permanence le meilleur minimum local, soit *optimal*, qui contient à la fois la valeur et son coût, et on fixe un nombre  $N$ . Dès que *optimal* est atteint  $N$  fois, on renvoie *optimal*. Cela donne des résultats très satisfaisants, mais c'est beaucoup trop lent. La deuxième possibilité est beaucoup plus simple et fonctionne bien : il suffit de garder le meilleur minimum trouvé en  $N$  essais ( $N$  resets). C'est le critère standard du nombre d'itérations maximum, à ceci près qu'il fait abstraction du nombre d'itérations pour une résolution. Nous avons testé cette approche *vs* une exploration complète sur des petites valeurs pour lesquelles une vérification est encore faisable. Sur 50 essais avec 8 accords de 6 notes tirées aléatoirement parmi 30 possibilités, la meilleure solution a été trouvée 39 fois et les autres étaient seulement à une note de différence de l'optimum, ce qui est largement suffisant pour notre application. Par ailleurs, la contrainte imposant une note commune au minimum fonctionne bien expérimentalement.

## 22.4 Problème 4 : Rythmes asynchrones

La contrainte pour avoir un vrai rythme peut être intégrée dans la même fonction de coût que la contrainte de non-simultanéité. Pour la variable  $V_j^i$ , on compte 1 à chaque fois qu'une autre variable tombe en même temps, qu'il s'agisse d'une autre variable de la même voix (contrainte "vrai rythme") ou d'une autre voix (contrainte "non-simultanéité"). Bien sûr, on fait ce décompte sur toute la durée, ce qui revient à une heuristique Min-Conflict en éclatant les alldifferent. En pratique, il faut faire attention à rester exactement sur  $d$ , sachant que les  $l_i$  ne divisent pas  $d$  en général. La contrainte tient donc compte de la valeur de la variable  $V_i$  (voir énoncé du problème).

La fonction de coût utilisée est la suivante :

```
(defun contrainte-min-sup (i j chord motif)
  (let* ((pre-k (multiple-value-list (floor *duree* (car (last motif))))))
    (vrai-k (+ (car pre-k)
               (if (<= (nth j motif) (cadr pre-k)) 1 0)))
    (onset (nth j motif))
    (longueur (1- (length chord)))
    (long-i (car (last motif))))
  (+ (contrainte-inf i j motif onset long-i)
     (max 0 (-
              (loop for k from 0 to vrai-k
                    sum (loop for l from 0 to longueur
                              sum (nb-occ (modulo (+ onset
                                                       (* k long-i)
                                                       (car (last (nth l chord))))
                                                       (nth l chord))))
              (1+ vrai-k))))))
```

Ce problème a donné lieu à une première version de la visualisation des résultats intermédiaires dans OpenMusic, sous une forme assez différente de celle d'OMClouds. Cette architecture n'a pas été conservée. Le principe en est le suivant : quand un utilisateur définit un problème de contraintes, il a en tête une structure musicale donnée (accord, rythme, etc). L'utilisateur sait probablement déjà comment représenter cette structure musicale dans OpenMusic, sous la forme d'une classe OM. Par ailleurs, l'aspect contraintes et recherche locale suppose d'éditer des résultats intermédiaires de cette structure. Traduit dans un langage objet, on veut hériter d'un côté, de la classe OpenMusic, de l'autre, d'une classe à définir qui permette d'éditer les résultats intermédiaires, d'où l'architecture suivante.

On définit une classe *ch-solver*, classe qui représente la visualisation d'un problème en recherche adaptative. Cette classe n'a qu'un seul slot, *seuil*, dont la valeur est celle en dessus de laquelle les résultats intermédiaires ne sont pas affichés. Cette classe a besoin d'une boîte particulière dans laquelle seront actualisés les résultats intermédiaires (voir les boîtes OpenMusic, chapitre 1) *chsolver-box*, qui hérite des boîtes OpenMusic *OMBoxEditCall*. La classe *ch-solver* :

```
(defclass! ch-solver ()
  ((seuil :initform 0 :initarg :seuil :accessor seuil)))
(defmethod get-type-of-ed-box ((self ch-solver)) 'ch-solverbox)
(defclass ch-solverbox (OMBoxEditCall) ()) Ces boîtes ont une évaluation presque identique à celle des boîtes habituelles à ceci près qu'il faut éditer les résultats partiels en cours d'évaluation. Pour ceci est définie une méthode update-solver que l'on ajoute dans la méthode d'évaluation pour ch-solverbox. En tous, deux méthodes sont propres à ch-solverbox, omng-boxvalue et update-solver
(defmethod update-solver ((self ch-solverbox) voices)
  (beep)
  (setf (inside (value self)) (var->tree voices))
  (update-if-editor self))
```

Du point de vue musical, la structure utilisée est bien représentée dans OM par la classe *poly*, qui est une *omstandardclass*. Elle sert à représenter des polyphonies, *ie* des superpositions de voix, une voix étant donnée par des notes et un rythme. Le slot *voices* de *poly* contient la liste des différentes voix.

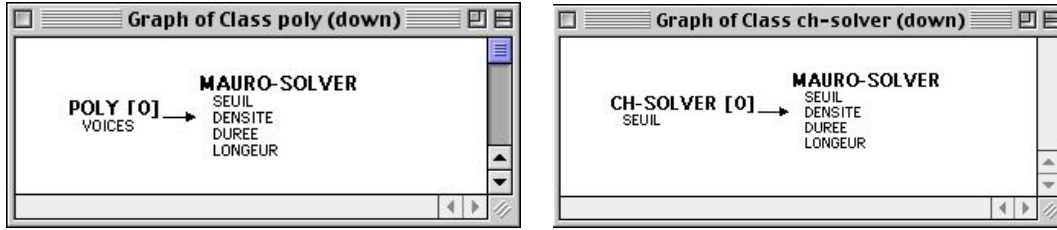


FIG. 22.1 – L’héritage de la classe **mauro-solver**. Du côté musical, la classe **poly** qui sert à représenter les polyphonies. Pour la recherche adaptative, la classe **ch-solver** qui permet notamment d’éditer les résultats intermédiaires.

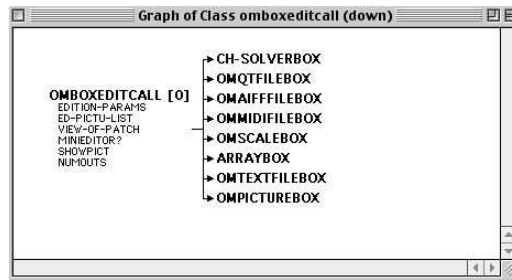


FIG. 22.2 – La classe **mauro-solver** a comme boîte la classe **ch-solverbox**, qui hérite de la classe OpenMusic **OMBoxEditCall**. On définit pour **mauro-solver** les méthodes **omng-boxvalue** et **update-solver** pour ré-évaluer les instances de la boîte à chaque minimum local.

On définit la classe **maurosolver** comme héritant des deux classes **poly** et **ch-solver**. Reste à lui ajouter comme slots les paramètres utiles au problème, ici, les longueurs des motifs de chaque voix, la durée totale, et un paramètre densité qui n’a pas été donné dans la modélisation, et qui représente la densité totale d’onset sur toute la séquence. Une densité de  $p$  signifie qu’il y a en moyenne un onset par  $p$  unités de temps. Ce paramètre représente le nombre de variables<sup>1</sup> simplement et de manière intuitive, à condition de supposer que les onsets sont répartis uniformément sur les voix.

```
(defclass! mauro-solver (ch-solver poly)
```

```
  ((longueur :initform 0 :initarg :longueur :accessor longueur)
```

```
  (duree :initform 0 :initarg :duree :accessor duree)
```

```
  (densite :initform 0 :initarg :densite :accessor densite)
```

```
  (:icon 486 ))
```

Reste à définir la méthode d’initialisation de la classe **mauro-solver**, ce qui consiste à calculer le nombre de variables, appeler la résolution en recherche adaptative, et forcer celle-ci à mettre à jour la boîte correspondante à chaque minimum local. Le résultat final est visualisé dans un éditeur de **poly**, voir figure 22.3

Cette architecture est satisfaisant en théorie, parce que l’on tire profit de l’aspect objet d’OpenMusic, et surtout des classes musicales proposées dans le logiciel. Le double héritage de la classe **mauro-solver** est assez élégant mais en pratique, on ne peut pas conserver la séparation entre ce qui concerne la musique (**poly**) et ce qui concerne la résolution de problèmes (**ch-solver**). Pour afficher les résultats partiels, il faut que la méthode **update-solver** ait accès non seulement aux valeurs calculées par la résolution, mais aussi aux structures musicales utilisées, ici une

<sup>1</sup>Pour une densité de  $d$ , le nombre de variables pour la voix  $i$  est  $l_i/(*dn)$ .

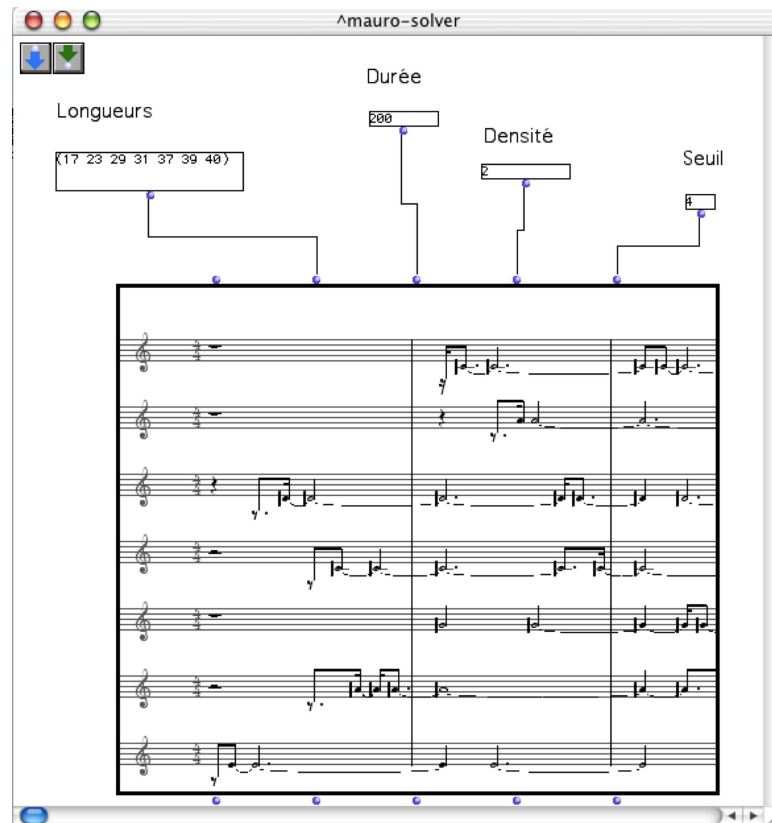


FIG. 22.3 – Le problème des rythmes asynchrones dans OpenMusic. Les résultats sont édités à travers une classe ad hoc, qui hérite de la classe `poly` d'OpenMusic. Visuellement, rien ne la distingue d'un `poly` (c'est le même éditeur), sinon les entrées qui sont les paramètres du problème de contraintes.

liste de `voices`. A l'inverse, l'implémentation de la résolution transmet à chaque minimum local non pas les valeurs des variables, mais une structure musicale, la superposition de rythmes qui en résulte. On a ainsi des croisements dans les deux sens.

Autre problème, la modélisation du problème de contraintes n'est pas nécessairement la même que celle de la classe musicale OpenMusic utilisée. C'est le cas ici avec les rythmes (ce sera d'ailleurs exactement le même cas pour tous les autres problèmes rythmiques). La classe `poly`, via la liste des `voices`, utilise évidemment la représentation OpenMusic des rythmes, en arbres, que nous ne détaillerons pas mais qui est très compliquée. La modélisation du problème utilise elle une représentation beaucoup moins riche, sous la forme d'une suite de durées. Il faut donc écrire une fonction pour passer de l'un à l'autre. Nous ne la donnons qu'à titre indicatif;

```
(defun var->tree (l)
  (let* ((l-deployee (deploye l))
        (treelist (mapcar #'(lambda (x) (mktree (om/ (cons (- (car x)) (x->dx x)) 16) '(4 4)))
                          l-deployee)))
    (loop for i from 0 to (1- (length treelist))
      collect (let ((voix
                    (make-instance 'voice
                                   :tree (nth i treelist))
```

```

:chords
  (let ((note (* 100 (om-random 60 72))))
    (make-instance 'chord
      :Lmidic (loop for j from 0 to *duree* collect note)
      :Lchan (loop for j from 0 to (1- (length (nth i l-deployee)))
        collect (if (verif-min-sup i j l-deployee) 2 1))))
:tempo 100)))

```

Peu importe l'implémentation de la fonction elle-même, on constate qu'il serait abhorrant de demander à chaque utilisateur d'écrire ce genre de fonction, en ligne de codes qui plus est, pour chaque problème défini.

En résumé, cette architecture présente un maigre avantage théorique et de nombreux inconvénients pratique, notamment de réclamer beaucoup de programmation pour chaque problème particulier.

## 22.5 Problème 5 : Durées dans Fibonacci

Ce problème a été résolu avec la librairie OMClouds (voir chapitre 4). Les deux modélisations ont été implémentées. La figure 22.4 montre la définition du CSP avec comme variables le décalage entre les indices : une seule contrainte, mais il faut entrer en paramètres la valeur de l'indice de départ. La figure 22.5 montre le CSP avec comme variables les indices.

## 22.6 Problème 6 : Tempi et accelerando

Ce problème a été résolu en Screamer (backtracking, avec un filtrage *ad hoc* sur la contrainte  $C_1$ ). On cherche à optimiser le nombre de rapports exacts, mais plus précisément on classe les solutions de la manière suivante, à la demande du compositeur : d'abord le plus grand nombre de rapports exacts sur  $t_{i+3}$  et  $t_i$ , ensuite le plus petit écart entre le minimum et le maximum de  $t_{i+1}/t_i$ . Les solutions trouvées ont été trouvées correctes par le compositeur. On obtient des solutions avec 20 tempi, une erreur cumulée sur  $t_{i+1}/t_i$  de 0,9375, 8 rapports exacts sur  $t_{i+2}/t_i$  et 8 rapports exacts sur  $t_{i+3}/t_i$ .

## 22.7 Problème 7 : Suivi de tempo

Ici, nous n'avons pas vraiment utilisé l'algorithme de recherche adaptative mais un algorithme *ad hoc*, qui d'ailleurs suit un raisonnement parallèle d'un point de vue mathématique et musical. On raisonne sur la plus petite durée commune à l'ensemble de la donnée. Soit  $d_{min} = pgcd(d_1 \dots d_n)$ . Cette valeur correspond au minimum pour pouvoir écrire l'ensemble des durées en valeurs entières, ou bien au tempo minimum tel que la partition s'écrit avec des noires, ou multiples de noires. Au départ, ce tempo est irréaliste, puisque beaucoup trop grand, avec une division du temps beaucoup trop petite.

On cherche donc à déplacer les  $d_i$  le plus légèrement possible, de manière à maximiser le  $pgcd$  de toute la séquence, et à diminuer le tempo jusqu'à obtenir des valeurs raisonnables. Il est évident qu'à terme, on aura tous les  $d_i$  égaux. Il faut donc choisir de s'arrêter lorsque les rapports entre les  $d_i$  appartiennent à l'ensemble  $R$  fixé, par exemple 1, 2 et 4 si on autorise des noires, des croches et des doubles croches. Pour les déplacements, ils doivent être aussi

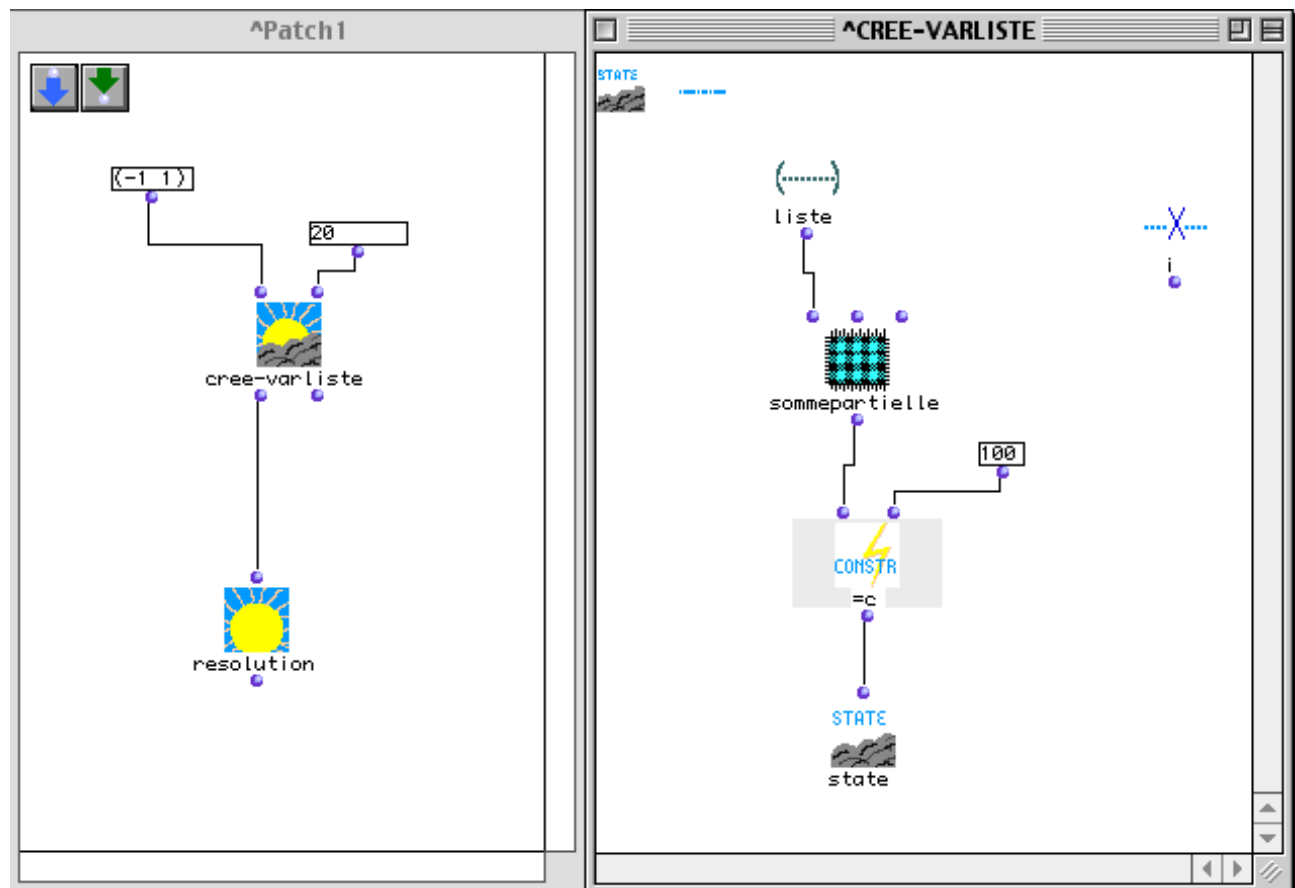


FIG. 22.4 – Le problème des durées dans Fibonacci dans OMClouds, avec la modélisation par décalage d’indices. La fenêtre de gauche montre le patch pour la définition et résolution du CSP, et celle de droite la définition des contraintes dans la boîte cree-varliste. Le patch appelé somme-partielle calcule la somme de termes de Fibonacci à partir d’un indice de départ et d’une suite de décalage d’indices. Ici, la seule contrainte porte sur la somme des termes de Fibonacci.

petits que possible, et évidemment multiples de  $d_{min}$  (on ne peut pas faire moins sinon le *pgcd* augmente).

En pratique, on calcule pour chaque  $d_i$  un coût,  $\prod_{j \neq i} pgcd(d_i, d_j)$  par exemple. La variable de coût maximal est choisie pour modification. On lui ajoute  $d_{min}$  ou  $-d_{min}$  et on sélectionne la valeur qui maximise le *pgcd* de toute la séquence. On ajoute un mécanisme Tabu, et les resets consistent à augmenter le nombre de déplacements autorisés (ce qui ajoute un paramètre interne  $p$ , initialisé à 1, et l’étape de modification consiste en fait à ajouter  $-p * d_{min}$ ,  $-(p-1) * d_{min}$ ,  $\dots$ ,  $(p-1) * d_{min}$ ,  $p * d_{min}$ ). Cet algorithme fonctionne bien en pratique. Il a en outre l’intérêt de toujours terminer (on peut donner une preuve de terminaison en raisonnant sur le *pgcd*).

## 22.8 Problème 8 : Suite de gestes

Les fonctions de coût pour ce CSP posent quelques problèmes, avec les mêmes options que dans sa modélisation, variables hauteur ou intervalles. En prenant les gestes comme variables, l’écriture du coût de la contrainte harmonique est délicate, puisqu’elle ne porte pas vraiment sur

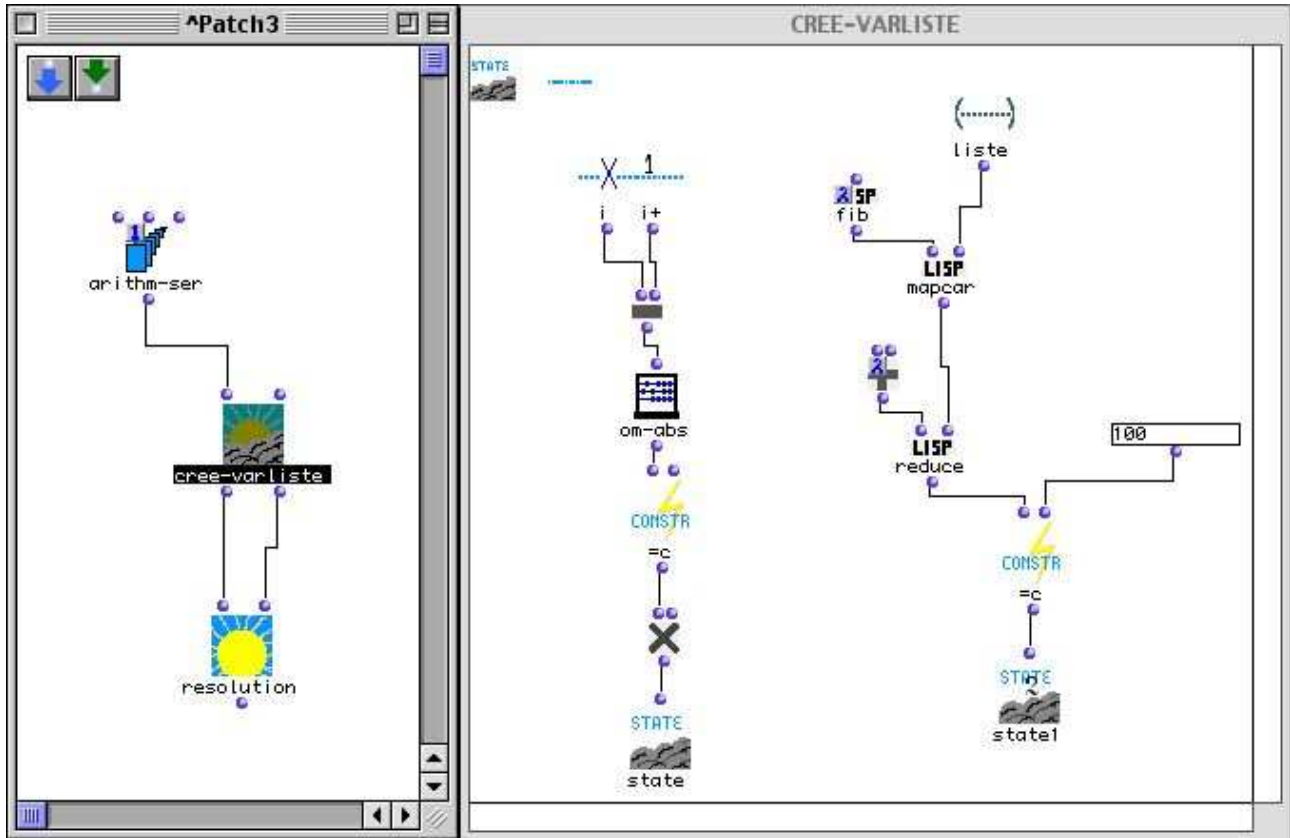


FIG. 22.5 – Le problème des durées dans Fibonacci dans OMClouds, avec la modélisation par indices. La première contrainte impose des indices décalés de 1 ou  $-1$ , la deuxième porte sur la somme des termes. A noter, ici comme au dessus, on a besoin de l'entrée *liste* dans le patchd e contraintes, parce que la contrainte porte sur la somme partielle des  $i$  premiers termes de la suite de Fibonacci.

l'intervalle  $V_i$  mais sur la note résultant du choix de la note de départ, et des intervalles situés avant  $V_i$ . On la notera  $N_i = (\text{Depart} + \sum_{1 \leq i \leq n} V_i)$ . La contrainte analogue à une contrainte de capacité pose des difficultés.

En appliquant strictement le principe de projection des coûts, on votti apparaître un effet indésirable : cela revient à placer les variables dans l'ordre croissant, celles du début étant largement favorisées. Supposons avoir une distance d'une note à l'harmonie, notons la  $dist$ , le coût de  $V_i$  est égal à la somme des  $dist(N_1) \dots dist(N_{i-1})$ , plus  $dist(N_i)$ , donc les coûts sont croissants en  $i$ . On revient alors sans le vouloir d'une méthode par contraintes à une méthode constructive. Expérimentalement, cela ne fonctionne pas.

Une première alternative est de moyenner par  $i$  (nombre de variables qui apparaissent dans le coût de  $V_i$ ), en prenant  $(\sum_{j < i} dist(N_j))/i$  comme coût pour  $V_i$ . Mais les coûts des  $V_i$  ne sont alors plus réellement comparables, ce qui provoque des boucles dont l'algorithme ne peut plus sortir par le mécanisme Tabu normal. Prenons pour l'exemple des coûts Min-conflict, 0 si la note est dans l'harmonie et 1 si elle n'y est pas.  $V_1$  a comme spectre des coûts possibles  $\{1/n, 2/n, \dots, 1\}$ ,  $V_2$  a  $\{1/(n-1), 2/(n-1) \dots, 1\}$ , et  $V_i$  a  $\{1/(n-i), 2/(n-i) \dots, 1\}$ , jusqu'à  $V_n$  qui a  $\{0, 1\}$ .

Ces coûts sont comparables en théorie, mais en pratique, cela n'a pas de sens, parce que

les différents  $V_i$  on des spectres différents. Supposons par exemple que  $V_1$  se trouve à son coût minimum non nul, c'est-à-dire  $1/n$ .  $V_i$  ne peut pas appartenir à l'ensemble des plus mauvaises variables tant que les coûts des autres variables sont non nuls, puisque le plus petit coût possible pour les autres variables est  $1/(n-1)$ . La seule manière de sélectionner  $V_1$  pour une modification est d'attendre d'annuler les autres coûts. Dans ce cas, les erreurs sont  $1/n, 0, 0, \dots, 0$ , coût de la configuration de  $1/n$ , et  $V_1$  est sélectionnée. Mais il est très improbable de lui trouver une meilleure valeur, puisque la seule meilleure valeur possible est 0, cas d'une solution, or en modifiant  $V_1$ , on modifie aussi toutes les autres notes résultantes (dans lesquelles  $V_1$  apparaît), dont il est très improbable qu'elles soient dans l'harmonie. Le mécanisme vaut aussi pour tous les autres  $V_i$ , dans une moindre mesure : en résumé, pour pouvoir changer un  $V_i$  de coût faible, d'abord il faut minimiser ses successeurs, puis la modification de  $V_i$  entraîne une dégradation des coûts de ces mêmes successeurs.

Non seulement on aboutit à un minimum local quasi certainement à chaque fois que  $V_1$  a un coût minimal, mais en plus on l'atteint après avoir effectué une résolution sur un ordre de grandeur à peine moindre. On observe expérimentalement que cela ne fonctionne pas. En pratique, l'algorithme avec cette fonction de coût ne renvoie pas de réponse pour des petites instances même pour de petites valeurs.

Troisième possibilité, on considère que l'intervalle  $V_i$  qui mène à  $N_i$  n'a de poids dans la résolution que si la note précédente était déjà dans l'harmonie. Sinon, cela n'a pas de sens d'essayer de faire revenir  $N_i$  dans l'harmonie *via*  $V_i$ , puisque  $V_{i-1}$  sera probablement modifiée dans une itération ultérieure. On annule donc tous les coûts des  $V_i$  pour lesquels  $N_{i-1} \notin Harm$ . L'idée est séduisante, mais cela revient à dire que les bonnes solutions sont celles qui ne sont pas dans l'harmonie, d'où un nouvel échec à la résolution.

En fait, la solution qui fonctionne le mieux est la plus simple, on prend  $dist(N_i)$  comme coût pour  $V_i$ .

La deuxième contrainte est beaucoup plus simple à traiter de par la modélisation choisie. Il suffit de prendre 1 si  $|V_i| = |V_{i-1}|$ , 0 sinon.

La troisième contrainte, qui porte sur la cardinalité, se traduit en coût assez naturellement : chaque intervalle  $V_i$  se voit affecter  $(Card\{j, |V_j| = Int_i\}) - P_{Int_i}$ . Notons que cela fonctionne également quand les  $P_{Int_i}$  ne représentent pas une vraie probabilité : si sur une séquence de 40 gestes on veut 10 fois l'un et 20 fois l'autre, cette définition du coût favorisera les solutions approchées avec des rapports 1/3 et 2/3.

En pratique, ce problème a été d'abord résolu selon le même principe que le problème 4 des rythmes asynchrones. On définit une classe **gilbert-solver** qui hérite de la même classe **ch-solver** pour la recherche adaptative et de la classe musicale **chord-seq**. C'est une des classes les plus utilisées d'OpenMusic, qui sert à représenter une suite d'accord, ici la suite de notes définies à partir des gestes. Il faut de la même manière définir une fonction **var->chord-seq**, analogue de la fonction **var->tree**.

## 22.9 Problème 9 : Séries tous intervalles

La contrainte sur les intervalles peut être passée de deux manières différentes. La plus naturelle consiste à prendre comme coût d'une note la redondance des intervalles qu'elle forme avec ses voisines, moins 1. Mais on peut aussi pondérer par la taille des intervalles, avec  $f_{global} = \sum_{i \leq n-1} |\sigma_i - \sigma_{i+1}|$ , voir [35]. Cette fonction est expérimentalement beaucoup plus efficace. Cela s'explique bien intuitivement, puisque les grands intervalles sont plus difficiles à placer que les

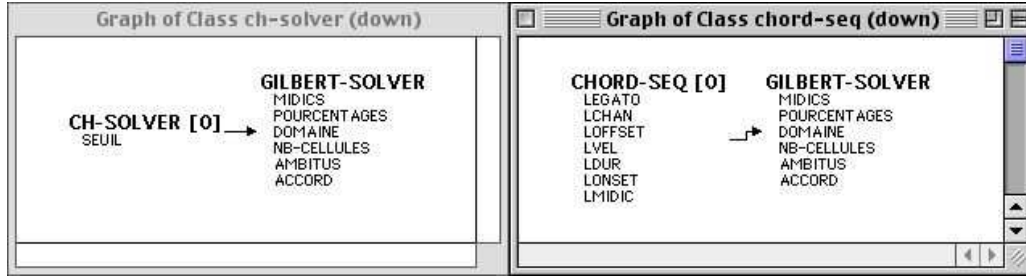


FIG. 22.6 – La classe `gilbert-solver` hérite des classes `ch-solver` pour la recherche adaptative et `chord-seq` pour la représentation musicale des résultats. Cette classe est un peu plus complexe que `mauro-solver`, parce qu’il faut fixer plusieurs paramètres pour le problème de contraintes. Les slots `pourcentages`, `domaine`, `nb-cellules`, `ambitus` et `accord` sont les paramètres du problème, voir chapitre 2. Le slot `midics` peut sembler redondant avec le slot `lmidic` de `chord-seq`, mais ce n’est pas le cas, il sert à stocker les valeurs midi des notes jouées (le problème portant sur des intervalles).

petits.

Sur les all-intervals series, on a testé différentes longueurs tabu, en division du nombre de variables. Les résultats sont donnés ???. Ils montrent que l’on ne doit pas nécessairement poser la longueur Tabu égale au nombre de variables. Ici, une longueur Tabu de  $n/3$  fait converger plus vite. En revanche, pour des valeurs plus petites, l’algorithme ne converge pas après trois millions d’itérations.

## 22.10 Problème 10 : Textures de Ligeti

Pour Melodien, on a 49 lignes et 10 notes par ligne, ces notes restant dans un ambitus de 66 à 93 en valeurs midi. Nous avons choisi les fonctions de coût suivantes, en reprenant la notation et la numérotation des contraintes ci-dessus, avec  $i$  l’indice dans les colonnes et  $j$  dans les lignes :

- (i).  $f_1(i, X_{i,j}) = k - \text{Card}\{l \in [1, k], X_{i,l} \notin A_i\}$
- (ii).  $f_2(i, X_{i,j}) = \sum_{l < j} \max(0, 1 + X_{i,l} - X_{i,j}) + \sum_{l > j} \max(0, 1 + X_{i,j} - X_{i,l})$
- (iii).  $f_3(i, X_{i,j}) = \text{si } X_i \in A_i \text{ alors } 0 \text{ sinon } |X_{i,j} - A_{(i-1),j}|$
- (iv). la dernière contrainte peut être passée en réduction de domaine

Il y a plusieurs manières de représenter ce problème. La plus simple serait de prendre  $49 * 10$  variables et de résoudre un CSP à 500 variables, sur un domaine de taille 27. C’est se compliquer inutilement la vie. Pour réduire le nombre de variables, on peut aussi ne prendre comme variables que les notes cachées, 83 dans le cas de Melodien, et écrire les contraintes sur ces seules notes. En effet, les autres variables sont presque déjà instanciées dans la partition. Mais ce n’est pas satisfaisant car cela ne permet pas de vérifier si le processus de modélisation est valable pour toute la partition. Nous avons choisi une solution intermédiaire, en résolvant le CSP ligne à ligne : chaque résolution porte donc sur 10 variables, avec des petits domaines de taille 4 puisqu’on passe à la ligne  $i + 1$  la contrainte de mouvement mélodique à partir de l’instantiation trouvée à la ligne  $i$ , ce qui correspond à un filtrage. D’un point de vue CSP, on perd de l’information de cette manière, puisque cela revient à interdire les backtracks à une ligne

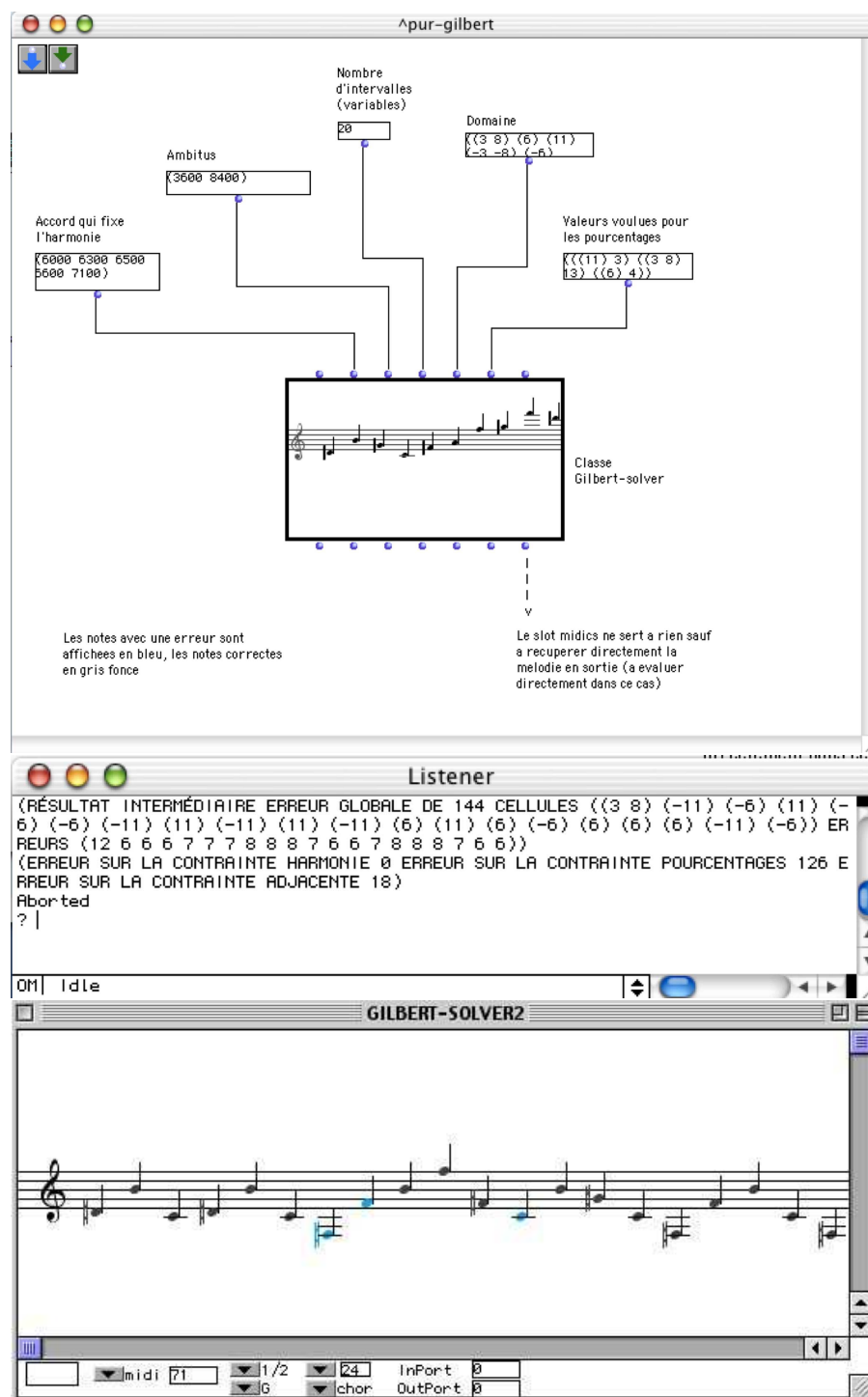


FIG. 22.7 – Le problème sur les suites de gestes a d’abord été résolu dans un cadre *ad hoc*, avec une classe `gilbert-solver` dont le patch montre une instance. Cette classe hérite de la classe musicale `chord-seq`. On lui ajoute des slots pour représenter le problème de contraintes. Le problème est surcontraint, mais les résultats intermédiaires sont édités dans le patch et le Listener affiche les erreurs sur les différentes contraintes (qui n’ont de sens que relativement entre elles, parce que les coûts ne sont pas de type min-conflict). Quand on ouvre l’accord, les notes avec une erreur sont affichées en bleu, les notes sur lesquelles il n’y a pas d’erreur en gris foncé.

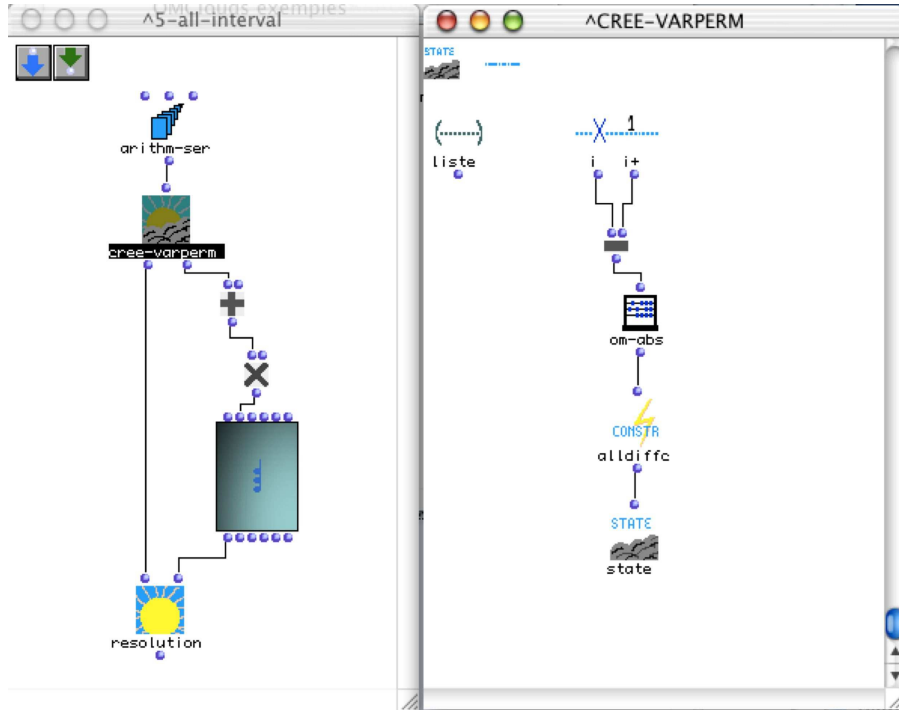


FIG. 22.8 – Le problème des séries tous intervalles dans OMClouds. A droite la contrainte, cela correspond à la fonction de coût qui compte le nombre de redondances dans le all-different sur les intervalles.

antérieure. En pratique, cela n'a pas d'importance car on trouve très facilement les solutions, sinon nous pourrions fixer un nombre maximum d'itérations pour l'algorithme, décider qu'on a un échec quand il est atteint et backtracker à la ligne précédente. Cette manière nous permet de vérifier la validité du modèle musical sur toute la partition. En pratique, le temps de calcul est assez stable, environ 7 secondes sur un Mac G4.

Un algorithme spécifique décrit dans [28] montre que l'analyse de cette texture n'a que 6 solutions. Plus précisément, il y a trois transitions possibles de la ligne 21 à la ligne 22 et deux de la ligne 36 à la ligne 37. La figure 22.9 montre ces deux possibilités, qui donnent en valeur midi, et en notant entre parenthèses les notes cachées :

$$\begin{aligned}
 X_{36} &= ((67)(70) 72 73 75 76 77 80 85 93) \\
 X_{37} &= (67 70 72 74(75) 76 77 80 85 93) \\
 X_{36} &= ((67)(70) 72 73 75 76 77 80 85 93) \\
 X_{37} &= (67 70 72(73)74 76 77 80 85 93)
 \end{aligned}$$

Les fréquences d'apparition de ces deux solutions, figure 22.10, dans la résolution par recherche adaptative ne sont pas équiprobables.

## 22.11 Problème 11 : Imparité rythmique

La fonction de coût pour  $V_1$  compte simplement 1 si on a une séparation en deux au niveau de  $V_1$ , 0 sinon. Formellement, toujours en oubliant les modulo, on compte 1 quand  $\sum_{j=i}^{j=i+n/2-1} V_j \neq$

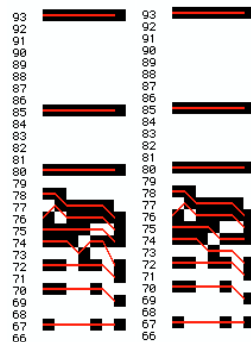


FIG. 22.9 – s entre les lignes 36 et 37 de Melodien

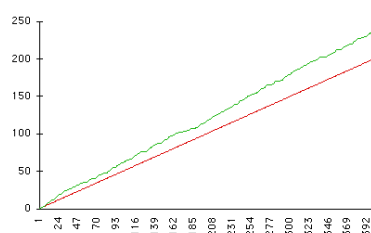


FIG. 22.10 – Les apparitions de l'un ou l'autre solutions sont presque équiprobables.

$\sum_{j=i+n/2}^{j=i-1} V_j$ . Pour ce problème, nous avons utilisé la recherche adaptative pour compter le nombre de solutions, ou au moins pour en avoir une approximation selon  $n_2$  et  $n_3$ . La question du dénombrement rigoureux du nombre de solution reste ouverte. Un dénombrement dans certains cas a été donné dans [29]. Nous avons obtenu les valeurs suivantes, avec  $n_2$  en abscisse et  $n_3$  en ordonnée. Selon le principe des images à collectionner dans les tablettes de chocolat, il faut  $n * \log(n)$  essais en moyenne pour trouver  $n$  solutions, nous avons effectué 1000 tirages pour les nombres inférieurs à 100 et 10000 tirages pour les nombres plus grands. Ces résultats ont été confirmés *a posteriori* par Louis-Martin Rousseau de l'Université de Montréal qui a résolu le problème sous ILOG-Solver. Ce tableau est remarquable si l'on regarde les différences entre les colonnes successives, qui se comportent comme les suites des puissances des entiers.

|    | 1 | 3  | 5  | 7   | 9   | 11  | 13   | 15   | 17   |
|----|---|----|----|-----|-----|-----|------|------|------|
| 2  | 1 | 1  | 1  | 1   | 1   | 1   | 1    | 1    | 1    |
| 4  | 1 | 2  | 3  | 4   | 5   | 6   | 7    | 8    | 9    |
| 6  | 1 | 4  | 7  | 12  | 19  | 26  | 35   | 46   | 57   |
| 8  | 1 | 5  | 14 | 30  | 55  | 91  | 140  | 204  | 285  |
| 10 | 1 | 7  | 26 | 66  | 143 | 273 | 476  | 776  | 1197 |
| 12 | 1 | 10 | 42 | 132 | 335 | 728 | 1428 | 2586 | 4389 |

## 22.12 Problème 12 : Canons Nzakara

Première solution, utiliser un Prolog. L'écriture des contraintes est très simple, il faut juste définir un prédicat *transpo* pour la fonction de transposition, avec *transpo*[60, 64], *transpo*[62, 67]

et *transpo*[64, 70]. Comme nous savons qu'il n'y a pas de solutions, il faut transformer un peu la modélisation du CSP. On peut par exemple ajouter un prédicat *transpo*(60, 70), qui modélise "l'erreur" faite par les Nzakara. La solution nzakara est alors trouvée en une demi-heure. Mais l'ajout d'un *transpo* n'est pas très satisfaisante, car rien *a priori* ne garantit que le nombre de *transpo*(60, 70) sera minimal.

Nous avons également résolu ce CSP en recherche adaptative avec de bons résultats. Les contraintes se traduisent facilement avec une heuristique Min-Conflict. Pour la première contrainte, le coût est de 1 si  $V_{(i+p \bmod n)} = t(\bar{V}_i)$  et 0 sinon. Pour la deuxième contrainte, on prend 1 de pénalité chaque fois que  $V_{(i+1 \bmod n)} = V_i$  ou  $V_{(i+p \bmod n)} = V_i$ . D'après [26], il n'y a que deux solutions, dont l'une est dégénérée (elle est formée de deux fois la même sous-séquence). La figure 22.12 montre les fréquences d'apparition des solutions : sur 500 tests, la solution Nzakara (non-dégénérée) apparaît avec une probabilité de 0,32, ce qui est cohérent et montre que les fonctions de coûts choisies ne sont pas biaisées.



FIG. 22.11 – Deux solutions, pour  $n = 10$  et  $p = 4$

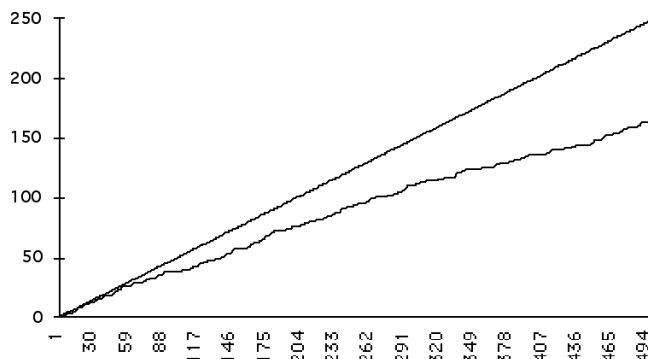


FIG. 22.12 – Fréquence d'apparition des deux solutions pour  $n = 10$  et  $p = 4$ .

## 22.13 Problème 14 : Doigtés pour la guitare

Même s'il existe une implémentation qui tourne, l'étude de ce problème n'est pas entièrement terminée, il y manque une expertise musicale (il faudrait regarder précisément avec un guitariste). Il a été implémenté dans OpenMusic avec les variantes décrites dans le chapitre 2, notamment : jouer toutes les cordes ou seulement une partie d'entre elles, dans ce cas jouer au minimum toutes les notes de l'accord de départ, possibilité de faire des barrés, ou non. La taille de la main, la scordature et la taille du manche (position maximale autorisée) sont paramétrables. En outre, deux représentations sont utilisées, l'une ou l'accord est donné comme un

ensemble de hauteurs midi et dans ce cas il s'agit de jouer exactement ces notes, ce qui laisse très peu de possibilités, l'autre ou l'accord est donné sous un codage de **chiffage**<sup>2</sup> *ad hoc*, noté avec une valeur de basse de référence et les intervalles par rapport à cette basse. Dans ce cas, on autorise à octavier les notes, ce qui de fait autorise les **renversements**<sup>3</sup>. Plusieurs fonctions de coût ont été utilisées (selon les options choisies, toutes de type Min conflict. Par exemple :

```
(defun cont-guitare (j accord)
  (let* ((note (nth j accord))
        (doigts (mapcar 'cadr accord))
        (doigtsjoues (remove nil doigts)) ici est calculé l'accord réellement joué, sans les cordes à
        vide
        (notes (versnotes accord)) et les notes joués à partir des positions
        (prov (minim notes))
        (mini (car prov)) position du premier doigt, le plus bas sur le manche
        (nbmini (cadr prov))) nombre de doigts dans cette position
        (if (null note)
            (loop for el in *accordcible*
                  count (not (membre el notes))) ici est imposé que toutes les notes soient jouées au
            moins une fois
            (+
             (if (zerop (cadr note)) 0
                 (max 0 (- (loop for el in doigtsjoues
                                count (not (zerop el))) 4))) au maximum 4 doigts posés
             (if (zerop (cadr note)) 0
                 (loop for j1 from 0 to (1- (length accord))
                       when (and (not (null (nth j1 accord))) (/= j j1))
                       sum (let ((autrenote (nth j1 accord))) et pour les doigts posés, la taille de la main
                             est limitée
                             (if (zerop (cadr autrenote)) 0
                                 (bvn (>= (abs (- (cadr autrenote) (cadr note))) *main*))))))))))
```

Pour les doigtés d'une suite d'accords, on a implémenté plusieurs distances : maximiser le nombre de doigts restant à la même position, minimiser le nombre de changements de position. La plus intéressante est sans doute : minimiser le nombre de changements de position, et s'il y a un barré, interdire les "glissandi de barrés", *ie* interdire un barré pour l'accord suivant (à noter, en fonction des choix de l'instrumentiste, on peut aussi bien les favoriser). L'ensemble fonctionne de la manière suivante : le programme est lancé pour chercher des doigtés,  $k$  fois pour chaque accord. Ensuite, en reprenant ces ensemble de  $k$  doigtés possibles comme domaines, on relance une recherche adaptative pour trouver des doigtés sur la séquence. Cela fonctionne bien, même s'il serait aussi intéressant d'implémenter une recherche complète pour la première phase.

Deux interfaces graphiques ont été développées. La première représente le manche de guitare tel que montré dans le chapitre 2, voir figure 17.1. Pour visualiser les enchaînements, on garde l'accord précédent dans une couleur plus pâle, voir figure 22.14. Ceci peut être utilisé dans OpenMusic comme indiqué figure ?? pour éditer les accords réellement joués, qui ont le même

<sup>2</sup>Voir chiffage dans le glossaire en annexe

<sup>3</sup>Voir renversement dans le glossaire en annexe

chiffage à partir de la basse et modulo l'octave, mais qui peuvent cependant être assez différents.

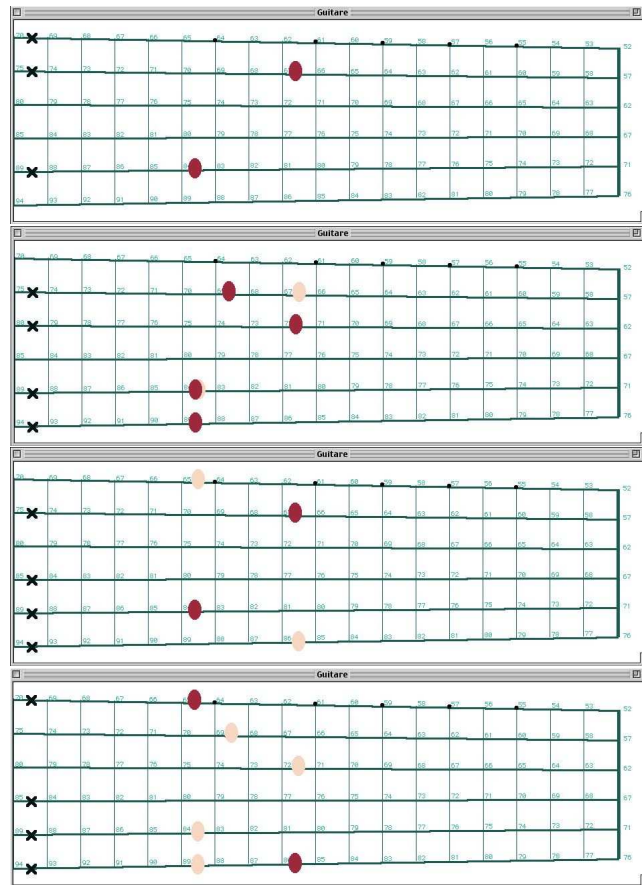


FIG. 22.13 – Solution pour une cadence I IV V I. Les doigtés sont affichés sur un dessin de manche de guitare. On passe d'un doigté au suivant en cliquant dans la fenêtre (et alors, les accords sont joués en MIDI). Le doigté en rouge foncé est le doigté courant, ceux en rose clair indiquent le doigté de l'accord précédent.

La deuxième interface graphique est plus orientée vers les standards des partitions. Elle consiste à utiliser les **extras** d'OpenMusic, objets attaché à des éléments de la partition pour ajouter n'importe quelle information à celle-ci. On définit une classe **tab-extra** (pour tablature), qui a un éditeur visuel associé comme montré figure 22.15. Nous avons tenté de respecter les notations habituelles, notamment les chiffres romains pour les positions, les liaisons pour les barrés. Ces objets peuvent être attachés à des accords de la partition.

## 22.14 Conclusion

Ces problèmes ont été résolus en grande majorité par une méthode de recherche locale, que l'on peut voir comme une technique de résolution de contraintes ou bien d'optimisation. Certains ont été résolus par une méthode complète de résolution de contraintes. Ces résultats appellent un commentaire, sans doute pas tant sur les techniques de résolution employées (voir

chapitre suivant), mais sur la démarche elle-même. En effet, comme souligné à la fin du chapitre 2, les buts recherchés dans la résolution de ces CSPs diffèrent selon les utilisateurs : optimisation, résolution de contraintes, etc.

Les problèmes 3, 4, et le deuxième niveau du problème 14 (doigtés d'un accord à l'autre) sont indubitablement des problèmes d'optimisation globale et ont naturellement été traités comme tels par de la recherche locale. Si l'on considère la programmation par contraintes comme un paradigme de programmation spécifique, autrement dit, comme une manière de "programmer de l'information partielle", les problèmes 1, 2, 9, 10, 11, 12 et le premier niveau du problème 14 (doigtés pour un accord) sont des CSPs purs : à chaque fois, on spécifie certaines règles à respecter, et il s'agit d'obtenir une solution, sur un espace combinatoirement grand.

Les autres problèmes ont certes été résolus comme des CSPs, il faut cependant reconnaître que c'est un peu arbitraire. En particulier, le problème 5 sur les durées dans Fibonacci aurait certainement trouvé des solutions algébriques. Le problème 6 est plus subtil, c'est de l'optimisation multicritères, d'une certaine manière. Le problème 8 a posé beaucoup de problèmes en tant que CSP, et, même si une autre solution n'est pas évidente, on aurait cependant probablement pu le résoudre avec un algorithme qui construise la séquence pas à pas.

En résumé, la plupart de ces problèmes sont des problèmes de contraintes ou d'optimisation bien posés. En outre, la programmation par contrainte apporte une grande souplesse notamment pour combiner les contraintes en ajouter ou en retirer par exemple. Cela a été très souvent utilisé, au point que d'ailleurs nous ne savons souvent pas quel modèle aura été retenu par le compositeur. C'est probablement la caractéristique de la programmation par contraintes qui a été la plus utilisée et appréciée : définir des relations partielles entre des variables, et combiner facilement ces relations entre elles.

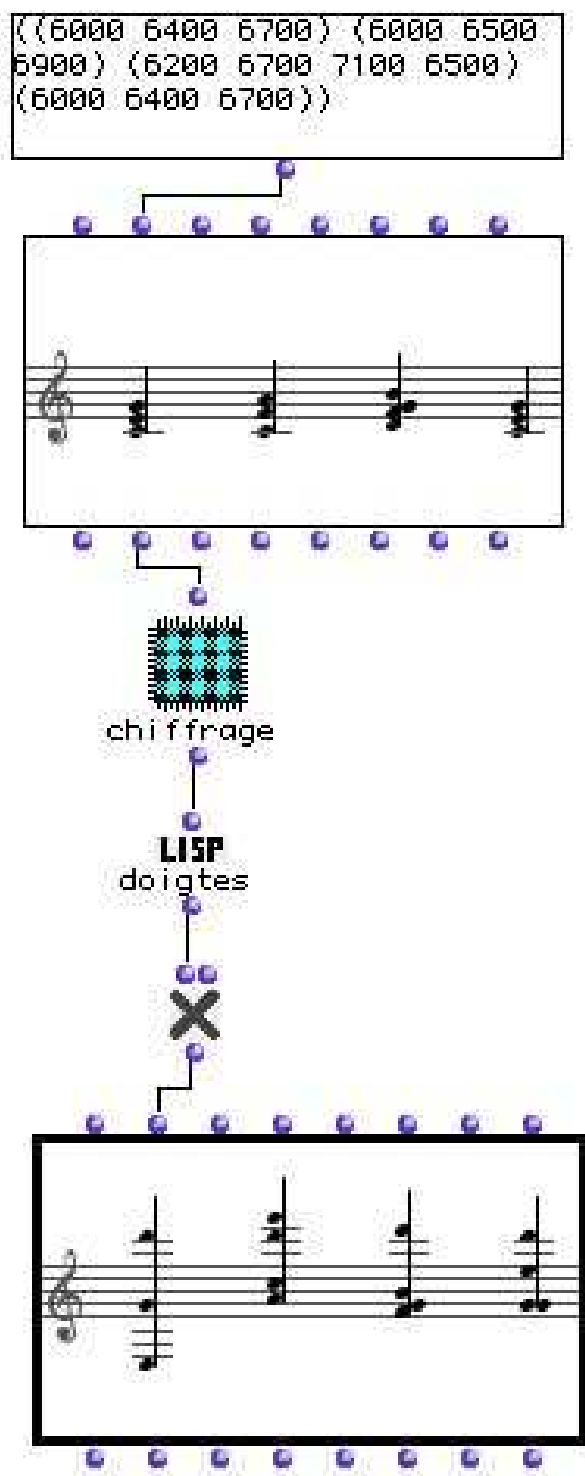


FIG. 22.14 – Solution pour une cadence I IV V I. Dans OpenMusic, les accords sont définis dans le chord-seq du haut, d'où sont extraits les chiffres. La réalisation du doigté (notes réellement jouées) est visualisée dans le chord-seq du bas.

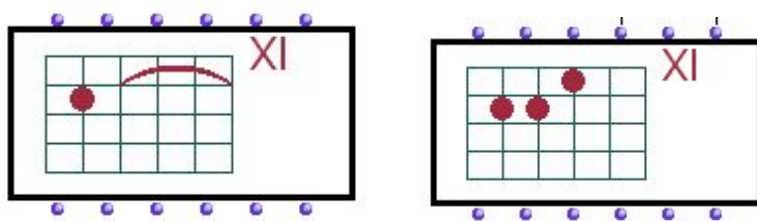


FIG. 22.15 – Une classe `tab-extra` est définie pour représenter des tablatures. Les éditeurs visuels ont été écrits de manière à respecter la notation habituelle. A gauche, un doigté avec un barré sur les quatre premières cordes. A droite, un doigté de onzième position.



## Chapitre 23

# Recherche Adaptative et probabilités

On peut le constater facilement en pratique, la recherche adaptative est très efficace. Que peut-on en dire à un niveau théorique ? En particulier, est-il possible de décrire, de manière probabiliste, le comportement de l'algorithme, par exemple en fonction de la distribution aléatoire de départ et de reset ? A première vue, ce n'est pas évident. Le cadre de la recherche adaptative étant celui de la recherche locale, on hérite des notions de voisinage, diversification, calcul des coûts, etc. Mais cela ne donne pas un modèle de fonctionnement. A notre connaissance, il existe peu de recherches sur des modèles pour la recherche locale, à l'exception de modèles de Markov à la [45], ou bien de Gomes qui se place sur le terrain statistique pour étudier le phénomène des *heavy tails* sur un problème de carrés latins<sup>1</sup> [51] (mais en recherche complète).

Une première manière d'aborder le problème serait de considérer la recherche adaptative comme un algorithme de steepest descent, avec un ordre adapté. Assez naturellement, le calcul des erreurs pour chaque variable suggère la définition d'un ordre partiel sur l'ensemble configurations  $\prec$  comme suit :  $c = (v_1 \dots v_n) \prec c' = (v'_1 \dots v'_n)$

$\Leftrightarrow$

$$\forall i, e_i \leq e'_i \text{ et } \exists i_0, e_{i_0} < e'_{i_0}$$

L'idée sous-jacente est de structurer l'ensemble des configurations avec cette relation dont on vérifie aisément qu'elle est un ordre partiel. L'ensemble des plus petits éléments de cet ordre est exactement l'ensemble des solutions (si on se limite aux erreurs positives pour les variables, ce qui sera toujours le cas, la seule exception étant les carrés magiques de Philippe Codognet). On voudrait pouvoir représenter une descente de l'algorithme comme une chaîne de cet ordre : partant aléatoirement, l'algorithme descendrait jusqu'à un minimum local, lequel serait un minimum de l'ordre considéré, à condition que la Tabu tenure soit maximale, *ie* égale au nombre de variables.

Cela ne fonctionne pas pour une raison clef : si une étape de l'algorithme fait passer de  $c$  à  $c'$ , rien ne garantit que  $c \prec c'$ . En effet, la condition d'amélioration n'est pas les coûts des variables mais le coût global de la configuration. Le coût des variables ne sert qu'à la sélection de  $V_{pre}$ . Il se peut qu'en modifiant la valeur de  $V_{pre}$ , le coût global diminue mais qu'il existe une autre variable (ou même  $V_{pre}$  elle-même) dont le coût augmente, et les deux états ne sont pas comparables.

---

<sup>1</sup>Les carrés latins ressemblent aux carrés magiques, mais en plaçant seulement les entiers de 1 à  $n$  dans les cases avec  $n$  le côté du carré, de telles manières que chaque colonne et chaque ligne contienne des nombres tous différents. Gomes et al s'intéressent au problème de compléter un carré latin quand une partie est déjà instanciée, identifiant notamment des cas critiques, avec des instances difficiles à résoudre et faisant apparaître les *heavy tails*.

Une autre idée serait de modéliser la recherche comme une chaîne de Markov, (voir par exemple dans [45] pour la Recherche Tabu). Une chaîne de Markov d'ordre  $k$  est une suite d'états  $u_t$ , où  $t$  représente le temps, telle que  $u_{t+1}$  ne dépend que de  $u_t, u_{t-1} \dots u_{t-k}$ . C'est une suite d'états à mémoire limitée. Le lien avec la Recherche Tabu est naturel, puisque la liste tabu est précisément une mémoire des  $k$  derniers états. Par ailleurs, les aspects probabilistes correspondent bien. Une suite de Markov est codée par un graphe d'états, et la probabilité de passer d'un état à un autre est fixée (ce qui donne une matrice de transition  $A$ , indicée par les états,  $A_{i,j}$  est égal à la probabilité de passer de l'état  $i$  à l'état  $j$ ). Cela modélise parfaitement les déplacements aléatoires de voisinage en voisinage. On obtient la probabilité d'un déplacement de  $k$  itérations avec les puissances de  $A$ , la matrice de transition étant  $A^k$ .

L'intérêt d'utiliser les chaînes de Markov est que sous certaines conditions, en connaissant la loi de probabilité sur l'espace de départ, on peut calculer une loi de probabilité stationnaire (probabilité de se trouver dans les états à l'infini). Cela signifierait, pour la recherche tabu, de calculer la probabilité de trouver des solutions, en fonction de la distribution de départ. Mais évidemment cela ne fonctionne pas, pour une raison intuitive évidente (si on sait trouver la probabilité d'avoir des solutions, on devrait savoir trouver des solutions), et pour une raison simple, le nombre de lignes et de colonnes dans la matrice de transition d'une telle chaîne de Markov est égal au nombre d'état, soit exactement la taille de l'espace de recherche. De sorte que calculer la matrice de Markov est déjà de complexité exponentielle par rapport à la taille du problème, sans parler de l'élever au carré, etc.

Il serait intéressant de continuer cependant dans cette voie en contournant le problème, en se basant sur une ou des chaînes de Markov agrégées. Notamment, dans le cas particulier de la recherche adaptative, en définissant une chaîne de Markov pour chaque variable, dont le nombre d'états est la taille du domaine, donc raisonnable. Un état contient l'erreur courante de la variable. Les probabilités de transition sont inconnues au départ, en particulier on ne peut pas les extraire naturellement de la structure du problème. Il existe plusieurs algorithmes (avant arrière par exemple) pour retrouver les paramètres de la chaîne de Markov à partir d'une suite d'états envoyés par cette chaîne (ce type de techniques a trouvé beaucoup d'applications dans le traitement de la voix naturelle par exemple ou dans l'analyse d'un génôme). Ces algorithmes pourraient fonctionner pendant le processus de recherche, et donner les probabilités de transition pour chaque variable. De la sorte, on pourrait envisager de calculer une probabilité d'annuler l'erreur de chaque variable, et en combinant les probabilités ("annuler l'erreur globale" est un et logique de "annuler les erreurs des variables", ce qui donne une formule de Poincaré), l'erreur totale. Cela repose sur deux présupposé que les erreurs se comportent comme une chaîne de Markov, ce qui est vrai pour les états mais pas nécessairement des erreurs (cependant les erreurs sont entièrement déterminées par les configurations qui elles se comportent comme une chaîne de Markov). En résumé, ceci est une piste de recherches qui semble intéressante.

## Quatrième partie

### OMClouds



# Chapitre 24

## Summary of Part IV : OMClouds

This last chapter presents the OMClouds library, which implements the adaptive search algorithm in OpenMusic. OMClouds is distributed with OpenMusic since april, 2003, version 4.6.5 and up.

**Technical choices** About the CSP modelisation, the variables are supposed to be homogeneous, that is, we suppose that the constraints state in the same way on all the variables. Thus, variables don't need to be named, and are represented by their indexes, as if they were placed in a list. Constraints hold on some variables located by the gap in their indexes. So the constraints are in the form of "minimize a distance between variable  $i$  and variable  $i+1$ ". There are three kind of CSPs : lists (variables are placed in a list), cycles (variables are in a list that is supposed to be repeated cyclically), and permutations (domains are the permutations of the variables). The interface is visual as in OpenMusic. OMClous has generally been made in order to respect the main features of OM as far as possible.

To take profit of OM being object oriented, we have chosen to define a CSP object (three of them actually, one for each kind of CSP), which contains all the information needed to represent, solve, and edit the CSP : current configuration, domains, constraints, and also tabu list, errors, global error, and so on. The main functions about the resolution are thus generic functions, which can be redefined for new CSPs structures, or in order to modify some heuristic choices like the neighborhood exploration for instance.

**CSP definition** A problem is defined in an OM box, called for instance `cree-varliste` in case of a list of variables (the architecture is comparable for the three kinds of CSPs). This box is an instance of `asboxlist`, a subclass of the OM class `box-with-patch`, which has an attached patch. When opened, the patch of any `asbox` shows some predefined icons representing the current variable, the whole variable list, and an output, plus two buttons to add either variables, or outputs, see figure 27.2. This allows to define the constraint visually, in OpenMusic manner, by using the constraint primitives (see figure 27.3), and connecting them to the "state" outputs. In this case, the constraint are translated into cost functions following tabular 27.4. A min-conflict like translation has also been implemented. Finally, it is also possible for the user to write his own cost-functions, just by connecting them to the state outputs.

**Resolution** The `asboxes` have two outputs. The first one gives the instance of the corresponding CSP class, and is intended to be directly connected to the `resolution` box. In this configuration, the resolution box simply takes the CSP object and applies adaptive search until

the global error is below a certain  $\epsilon$ , zero by default. When a solution is found, the values are given in the output of the **resolution** box.

The second way of using OMClouds allows to deal with approximate solution, during the search process. Again, the **resolution** box is an instance of the **box-resol**, subclass of the Open-Music **OMBoxCall** class. The evaluation method has been redefined for this **box-resol** boxes, in order to deal with the second output of a **asbox** class. This second output is set by the resolution box to the values of the last local minimum encountered during the search process. This allows to connect these values to any kind of OM calculus or musical editors. Then, during the search process, the resolution box re-evaluates its second input each time a new local minimum is found. This activates the evaluation of everything that is connected to it, from the second output of the **asbox**. What is important here is the fact that the local minimum can be shown during the search process, and processed or edited by any kind of OM calculation. According to the notes on the musical CSPs, OMClouds does not simply give the intermediate values, but also leaves the possibility to process them so that they can be edited in a score like editor.

The main classes used in OMClouds are shown on figure 27.8.

We have measured some performances in OMClouds on the all-intervals series,  $n$ -queens, and asynchronous rhythms problems. The results are far slower than the one presented in [42], with an implementation in C of the same algorithm. Several factors explain this, among others, we have chosen to generate the cost-functions and pass them as parameters, which forbids optimization on these, although they are often calculated in a single step of the program. Anyway, the results are sufficient for our application.

# Chapitre 25

## Choix techniques

La librairie OMClouds a été écrite d’après les conclusions de l’étude de la douzaine de CSPs des chapitres précédents. Nous détaillons ici les choix techniques à la fois d’interface, de modélisation et d’implémentation.

### 25.1 Modélisation

On supposera les variables homogènes, *ie* les contraintes portent sur toutes les variables de la même manière et les variables ont toutes le même domaine. De la sorte, les variables ne sont pas nommées, et il suffit d’en définir le nombre. Comme les variables ne sont pas nommées, on les repère par leurs positions relatives. Une contrainte est définie par défaut sur la variable courante, la  $i$ -ème, et une ou plusieurs des variables qui la suivent,  $i + 1$ ,  $i + 2$  ...  $i + k$ . Il sera aussi possible de définir les contraintes globales.

La restriction ainsi apportée par rapport au format CSP général est en théorie forte. La restriction sur les domaines n’a que peu d’importance (il existe d’ailleurs une version d’OMClouds avec des domaines différents), mais la restriction sur les contraintes interdit par exemple les CSPs définis *in extenso* à la SAT. Formellement, même en réordonnant les variables ou redéfinissant les contraintes, il n’est pas possible de réduire n’importe quel CSP à un CSP avec cette restriction.

Mais en pratique, la plupart des problèmes musicaux du chapitre 2 sont modélisables de la sorte, et même la plupart des problèmes académiques classiques ( $n$ -reines, allocation de fréquence par exemple). Entre parenthèse, dans la librairie OMClouds, un utilisateur averti peut tricher sur les indices pour ajouter des contraintes sur une variable particulière.

Le langage de contraintes doit d’abord contenir les primitives énumérées tableau 18.2, mais il est aussi possible d’écrire facilement ses propres fonctions de coût (distances à minimiser par exemple). Enfin, il est possible de pondérer les contraintes.

### 25.2 Interface

L’interface doit être visuelle, pour respecter la programmation en OpenMusic et les habitudes des utilisateurs. Dans OMClouds, qui n’est évidemment pas une librairie de programmation fonctionnelle, on conserve cependant une partie des principes et des méthodes du langage visuel, à deux niveaux.

Premièrement, les contraintes ou fonction de coûts sont définissables comme des fonctions sur les variables (même s'il peut s'agir d'un prédicat), et OMClouds reprend sur ce point exactement la sémantique visuelle d'OpenMusic. En revanche, ce ne sont pas des fonctions OpenMusic, notamment parce qu'elles ne sont pas destinées à être appelées dans d'autres patches. Le patch de définition des contraintes n'est donc pas une instance du patch OM, mais d'une sous classe du patch OM.

Deuxièmement, dans l'architecture elle-même, il faut séparer clairement les étapes de modélisation et de résolution. La principale raison en est l'édition des résultats partiels, optima locaux rencontrés au cours de la recherche, qui sont par nature situés après la modélisation, mais avant la résolution. OMClouds utilise deux boîtes OpenMusic, l'une pour la résolution, l'autre pour la modélisation. Ces deux boîtes sont connectées par un lien qui correspond à l'invocation fonctionnelle normale, mais on peut y ajouter une connexion spéciale (voir ci-dessous).

## 25.3 Objets

OpenMusic est orienté objet. Une première possibilité aurait été de redéfinir ou d'utiliser les objets musicaux d'OM pour définir les variables, à la manière de Roy (voir chapitre 1 ou [79]). Mais Roy utilisait ces structures fortement définies d'abord, parce qu'il se situait dans un cadre très précis où les structures musicales utilisées étaient connues à l'avance, ce qui n'est pas notre cas, et pour faire des réductions de domaines par consistance, ce qui ne correspond pas tellement à notre approche de recherche locale (même si on pourrait le faire en traitement préalable). En choisissant de définir des variables notes, accords ou autres, d'une part on perd un certain degré de généralité (limitation des structures musicales traitées, impossibilité d'intégrer les structures définies dans OM ou dans d'autres bibliothèques), d'autre part on s'oblige à définir un langage de contraintes au cas par cas.

Nous avons choisi une approche diamétralement opposée. Partant du principe que l'objet central du solver est le CSP (variables, domaines, contraintes et toutes informations utiles à la résolution, l'édition de solutions approchées ou exactes, etc), OMClouds est structurée autour d'un objet CSP. Une instance de cet objet doit contenir toute l'information nécessaire à la fois à la résolution, et à l'édition de résultats approchés. L'avantage de cette approche dans le cas particulier de la recherche locale est d'avoir en un même objet le CSP lui-même, la configuration courante et toutes les informations utiles à la résolution d'une manière très souple. On peut aussi définir plusieurs types de CSPs de manière transparentes pour l'utilisateur, ce qui répond à l'objectif de modélisation en liste, cycle ou permutation. La résolution devient simplement une fonction générique, dont les méthodes sont adaptées au type de CSP. Enfin, ce n'est pas implémenté dans OMClouds car ce n'en était pas le but, mais on peut ajouter d'autres types de résolution pour une même définition de CSP.

## Chapitre 26

# Méthode complète, intégration de Screamer dans OpenMusic

L'un des tout premiers travaux a été d'importer dans OpenMusic un solveur écrit en Common Lisp, Screamer [96]. Screamer ajoute à Common Lisp une forme d'indéterminisme via deux constructions, *either* et *fail*, qui introduisent le point de choix dans le langage. (*either*  $e_1 \dots e_n$ ) évalue d'abord  $e_1$ , puis si l'évaluation de  $e_1$  donne un *fail*,  $e_2$ , etc. Screamer propose en outre un solveur avec propagation.

La version 4.0 d'OM possède maintenant une librairie Screamer, qui introduit des fonctions permettant le backtracking. Ces fonctions ont été redéfinies pour s'intégrer à la programmation visuelle d'OM. L'indéterminisme posant problème pour un langage fonctionnel, nous avons ajouté au langage une nouvelle classe de patches, qui correspondent aux fonctions déclarées indéterministes par Screamer. Ils sont notés avec un point d'interrogation, voir figure 26.1. A la fermeture d'un patch, la fonction de génération de code d'OpenMusic collecte toutes les boîtes présentes dans le patch. Si l'une d'elles est une fonction non-déterministe de la librairie Screamer, ou bien contient une fonction non-déterministe, le patch est déclaré non déterministe, ce qui modifie son évaluation, et affiche un point d'interrogation sur l'icône du patch.

Lors d'une évaluation à la sortie d'un patch indéterministe, OM ajoute automatiquement l'une des trois fonctions de Screamer qui appellent une résolution. L'utilisateur peut, dans les Préférences, choisir le mode de résolution qui lui convient : une seule solution, toutes les solutions, énumérer les solutions. Dans les deux premiers cas, le résultat est directement renvoyé. Dans le deuxième cas, apparaît un Nondeterministic Listener, voir figure 26.3, fenêtre affichant la solution courante, avec les mêmes éditeurs qu'OM (notamment musicaux, ce qui signifie que l'on peut écouter). Le Nondeterministic Listener permet aussi de choisir de passer à une autre solution ou de garder la solution affichée.

Pour la déclaration de variables, l'utilisateur peut appeler directement les primitives Screamer *a-member-of* et *an-integer-between* qui font ce que leurs noms indiquent. Nous avons ajouté *list-of-members-of* et *list-of-integers-between*, qui font également ce que leurs noms indiquent. Plus intéressant, nous avons également ajouté *a-chord-in*, qui contient une contrainte prédéfinie de manière à produire des valeurs strictement croissantes uniquement : en effet, un accord est constitué *a priori* de notes toutes différentes et dont l'ordre ne compte pas. Enfin, *list-of-chord-in* comporte cette même contrainte prédéfinie, plus une entrée optionnelle : celle-ci permet de poser des contraintes sur chaque accord de la séquence. Ces cinq fonctions sont identifiées par une icône ad-hoc, voir figure 26.2. Dès qu'une de ces quatre fonctions est appelée dans un patch, le patch est classé non-déterministe. Les contraintes sont écrites comme des prédicats

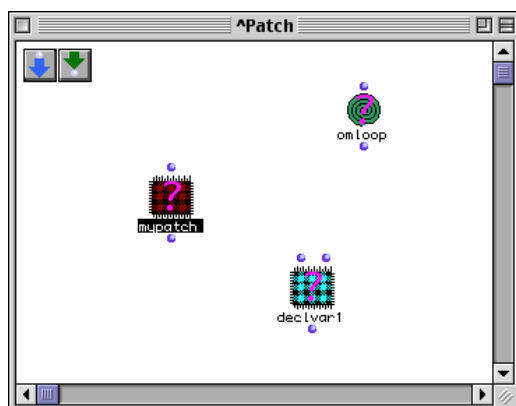


FIG. 26.1 – Exemples de patches non-déterministes. Un point d’interrogation est ajouté automatiquement sur les patches qui contiennent un point de choix. L’évaluation en sortie est modifiée pour ajouter une primitive Screamer : une solution, toutes les solutions, ou éditer les solutions.

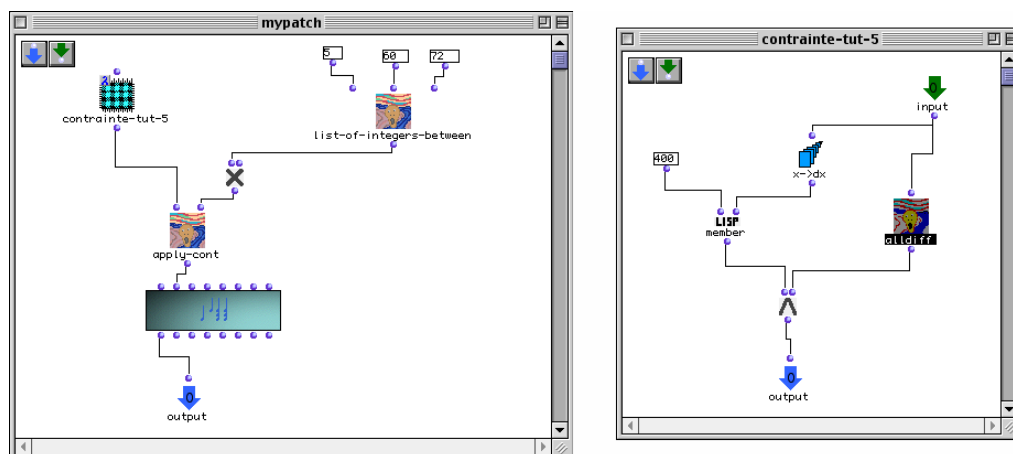


FIG. 26.2 – Un exemple de patch utilisant Screamer pour calculer des valeurs midi pour un chord-seq. A droite, la contrainte utilisée, exprimée comme un préicat OM. Elle impose que toutes les notes soient différentes, et que l’accord contienne une tierce majeure.

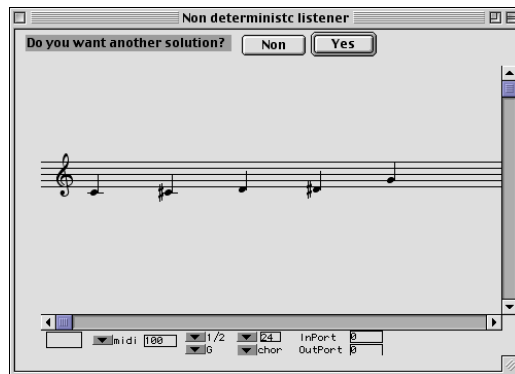


FIG. 26.3 – L'énumération des solutions a lieu dans le Nondeterministic Listener, avec les éditeurs OpenMusic habituels, ici pour la classe `chord-seq`.

OM, et sont traduites en Screamer à l'évaluation. Elles sont passées au solver via une fonction `apply-cont`, voir figure 26.2. Elles s'intègrent donc de manière naturelle au langage visuel.

Avec *either* et *fail*, seule une minuscule partie de Screamer est dans OpenMusic. L'inconvénient est énorme, puisque c'est à l'utilisateur d'écrire les réductions de domaines associées aux contraintes ou de placer les points de choix. Il aurait été intéressant d'intégrer l'ensemble de Screamer dans OpenMusic, mais impossible pour des raisons techniques (à cause d'un bug dans la gestion la liste des fonctions non-déterministes de Screamer, appelée à la fermeture des patches).

La librairie intégrant Screamer n'est donc pas réellement intéressante pour les utilisateurs d'OpenMusic, à moins qu'il s'agisse de spécialistes. L'intérêt de ce travail n'est donc pas réellement de résoudre des CSPs musicaux mais plutôt de tester une première approche pour exprimer les contraintes visuellement. En pratique, nous avons constaté que Screamer a été utilisé surtout pour des problèmes combinatoires (générer toutes les combinaisons possibles, souvent même sans aucune contrainte).



# Chapitre 27

## OMClouds

OMClouds est une librairie intégrée dans la version 4.6.5 d'OpenMusic, disponible au forum d'avril 2003.

### 27.1 Modélisation du problème

Les problèmes sont définis comme instances de trois classes *ad hoc*, qui contiennent l'ensemble des données nécessaires à la modélisation et à la résolution. OMClouds fournit trois sortes de CSPs :

- **liste** les variables sont placées simplement dans une liste
- **cycle** les variables sont toujours placées dans une liste, mais on considère qu'elles forment un cycle, donc la dernière variable a pour suivante la première variable. Cela revient à considérer les indices modulo le nombre de variables.
- **permutation** les variables sont placées dans une liste, mais le domaine de valeurs possibles est l'ensemble des permutations sur les variables. Dans la résolution, l'étape "exploration du domaine de  $V_+$ " est remplacée par "exploration quand on permute  $V_+$  avec chaque autre variable".

Pour pouvoir modéliser les trois sortes de problèmes, on a en fait trois classes : **asvarliste**, **asvarcycle**, et **asvarperm**. Dans la suite, on détaillera la modélisation pour **asvarliste**. L'architecture d'OMClouds est semblable pour les trois classes.

Leurs slots sont :

- (i). **valeurs** les variables étant toujours instanciées, on stocke ici l'état courant du système
- (ii). **domaine** qui contient le domaine des variable (sauf pour **asvarperm**)
- (iii). **erreurs** stocke les erreurs variables par variable
- (iv). **erreur-global** stocke l'erreur globale de la configuration (somme des **erreurs**)
- (v). **tabu** stocke la liste des variables marquées tabu, et le nombre d'itérations avant qu'elles ne sortent
- (vi). **longueur** nombre de variables
- (vii). **contraintes** contient la définition des fonctions de coût pour les variables
- (viii). **cont-glob** contient la fonction de coût pour l'ensemble de la configuration, somme des **contraintes**

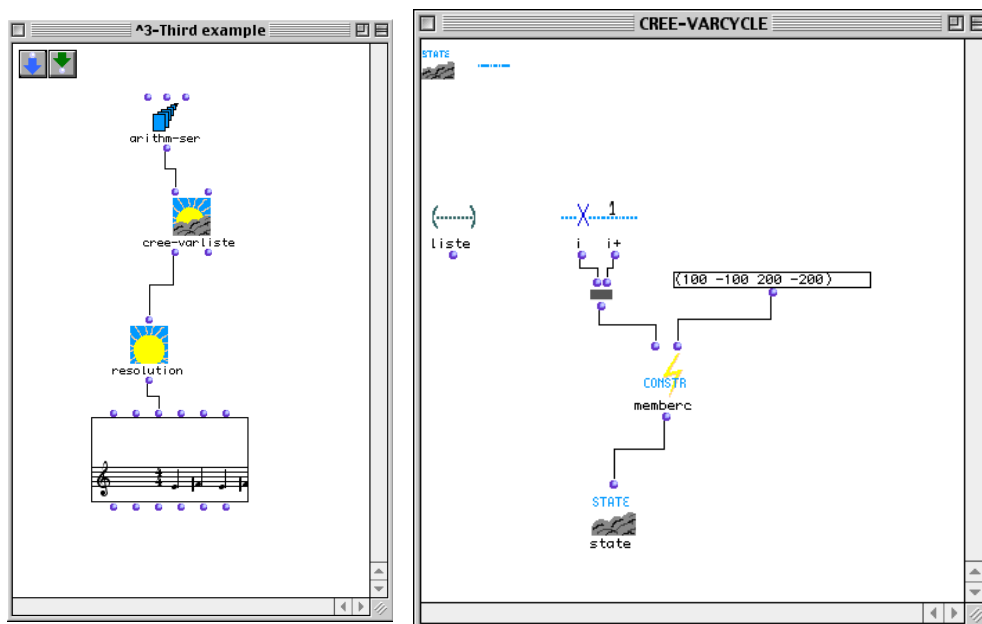


FIG. 27.1 – Un problème de contraintes dans OMClouds. L'instance d'**asvarliste** est créée par la boîte **cree-varliste**, avec en entrée le domaine, ici une liste de valeurs midi, et le nombre de variables. La définition des contraintes, à droite, est faite dans le patch associé à cette boîte (ici, il s'agit d'une liste de hauteurs midi dont les intervalles doivent être dans un ensemble fixé). Le **asvarliste** est ensuite passé à la boîte résolution, qui donne une solution en sortie.

OMClouds permet de créer visuellement, à la manière d'OpenMusic, des instances de ces classes pour modéliser un problème musical, avec les fonctions **cree-varliste**, **cree-varperm** et **cree-varcycle**. Elles sont représentées visuellement par des boîtes particulières, instances de la classe **asbox**, qui ont notamment un patch associé pour la définition des contraintes. Les deux entrées représentent les domaines et le nombre de variables. Ces boîtes ont deux sorties, la première renvoie une instance de la classe concernée, la deuxième peut être utilisée pendant la résolution et renvoie uniquement les valeurs courantes des variables (voir ci-dessous).

Le patch des contraintes est défini dans un patch associé à l'instance d'**asbox** considérée. L'objet OpenMusic dont héritent **asbox** et ses sous classes **asboxliste**, **asboxperm**, **asboxcycle** est **box-with-patch** (sous classe de **OMBoxcall**), qui a un slot patch dans lequel est stocké le nom de la fonction associée. **box-with-patch** est par exemple utilisée pour les boucles : en OpenMusic, on écrit une boucle dans le langage visuel en ouvrant un patch à partir de la boîte **omloop**, dans lequel est écrit le corps de la boucle, et qui génère, à sa fermeture, le code adapté. Ici c'est le même principe.

Enfin, la fonction d'évaluation d'OpenMusic (**omng-box-value**) a une méthode pour chacune des trois **asbox**, de manière à générer l'instance du csp correctement. Ceci est une différence notable par rapport à OpenMusic. Dans le langage visuel, en général, une instance d'une classe est créée par une "factory", boîte qui a toujours comme premières entrée et sortie l'instance de la classe, puis comme entrées et sorties suivantes certains des slots de la classe avec leurs valeurs par défaut. Ici, seuls deux slots du csp apparaissent en entrée, les autres sont calculés par **omng-box-value**. Pour les sorties, la première sortie renvoie l'instance du csp comme en OM, la deuxième est assez particulière, voir ci-dessous. La fonction **omng-box-value** effectue aussi l'initialisation aléatoire des variables.

## 27.2 Définition des contraintes

Les contraintes sont définies dans le patch (classe OM `varpanel`) attaché à la boîte `asbox`. Ce patch est le même pour les trois types de CSP (même si bien sûr les contraintes s'appliquent différemment). Il contient plusieurs entrées et sorties prédéfinies, qui sont des sous-classes des classes input et output d'OpenMusic.

L'entrée `var-i` est unique. Elle représente la variable courante, nous noterons  $i$  son indice dans la suite. On peut ajouter des entrées `var-i+` avec un bouton adéquat (en haut à gauche). Elles représentent les variables suivant la variable courante. On peut ajouter autant de `var-i+` que nécessaire, qui représenteront successivement les variables d'indices  $i + 1$ ,  $i + 2$ ,  $i + 3$ , etc. Ainsi, pour écrire une contrainte sur deux variables qui se suivent, on ajoutera la variable d'indice  $i + 1$  pour poser la contrainte entre `var-i` et l'instance de `var-i+` ainsi créée, qui représente la variable d'indice  $i + 1$ .

OMClouds intègre plusieurs primitives de contraintes : `=c`, `equalc` pour l'égalité, `<c`, `<=c`, `notequalc` pour les inégalités, `andc`, `orc` pour les opérateurs logiques, `minimizec`, `memberc`, `alldiffc` et `cardc` pour des contraintes de plus haut niveau. Elles sont associées à des fonctions de coût prédéfinies.

Les sorties `varstate` servent à poser les contraintes. On peut en ajouter autant que nécessaire. Il suffit de connecter la contrainte à la sortie. Une remarque importante : les contraintes peuvent être des fonctions prédéfinies mais ce n'est pas obligatoire. Un utilisateur peut vouloir définir ses propres fonctions de coût, par exemple un cas très fréquent est de minimiser une certaine distance sur les variables. Il lui suffit alors de programmer visuellement la distance dans le patch de contraintes et de la connecter à une sortie `varstate`.

A la fermeture du patch, OpenMusic appelle la méthode `Compile-Patch`, notamment pour générer le code associé. Ici le patch n'est pas un patch classique mais une instance de `Patchforvarperm`, pour laquelle `compile-patch` a une méthode définie. Ceci afin de réaliser deux opérations : d'abord, traiter le cas des entrées et sorties propres à `Patchforvar*`, ensuite, truquer la génération de code pour les primitives de contraintes.

D'abord, les entrées doivent être traduites en indices sur les contraintes. Ceci est fait en sous classant les classes OpenMusic des entrées de patch. Ceci sera détaillé plus bas.

Il faut aussi traiter les cas limites, puisque  $i + \dots$  ne doit pas dépasser la longueur de la liste, ce qui arrive toujours en fin de liste. Il faut ici distinguer deux cas selon qu'il s'agit d'un cycle ou bien d'un perm/liste. Ce n'est pas fait au niveau des symboles mais directement dans la fonction de coût générée. En pratique, s'il y a le plus grand indice des `var-i+` est  $k$ , les  $k$  premières variables se voient affecter des coûts décalés. Par exemple, pour une contrainte portant sur les variables  $i$  et  $i + 1$ , les deux premières variables auront le même coût. C'est un gros inconvénient, mais en pratique cela ne semble pas gêner trop souvent la résolution. En fait, comme la variable de plus mauvais coût est choisie aléatoirement parmi toutes les variables de plus mauvais coût, cela ne doit pas avoir une trop grande incidence.

On commence par appeler la génération normale de code d'OpenMusic, qui renvoie le corps de la fonction "normale" telle qu'écrite dans le patch, soit `Body`. Au lieu de placer directement `Body` comme corps de fonction, on ajoute un morceau pour décaler les indices au début.

Enfin, la fonction de coût générée est définie en Lisp, appelée par un nom nouveau que l'on affecte au slot `contraintes` du csp correspondant. Le slot `cont-glob` a toujours comme affectation la somme des appels de contraintes sur tous les indices des variables (la somme est donc l'opération de combinaison choisie dans OMClouds).

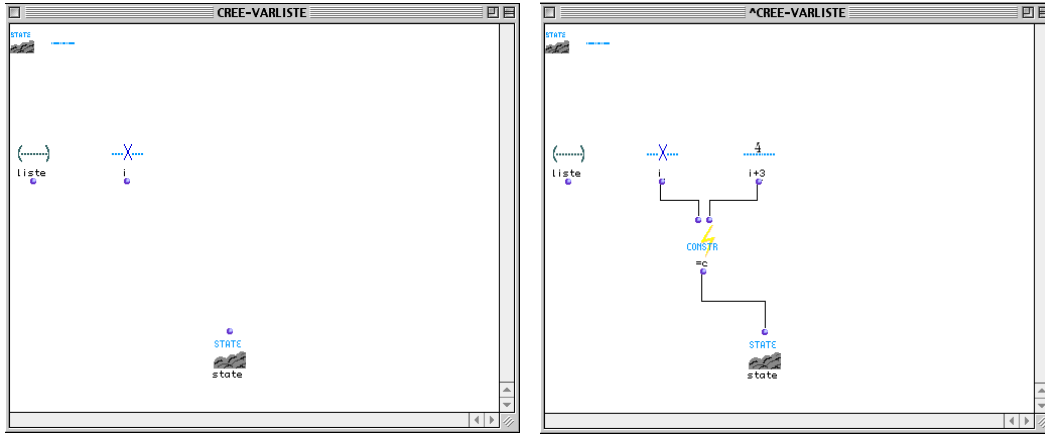


FIG. 27.2 – Définition des contraintes, à gauche, le patch par défaut. A droite, un exemple avec comme seule contrainte que chaque variable  $V_i$  soit égale à  $V_{i+4}$ .

## 27.3 Génération des coûts

A la fermeture d'un patch de contrainte, une fois réglés les problèmes d'entrée, sortie et définition de fonction, a lieu en général une transformation des primitives de contraintes en fonctions de coût. Toutes les primitives de contraintes sont montrées figure 27.3.

Transformer automatiquement les contraintes en coût est très difficile. Pour une seule contrainte, c'est relativement simple. On utilise une grammaire de la forme du tableau 27.4. Cela correspond en général à compter le nombre de conflits, en tous cas pour les contraintes `equalc`, `/=`, `notequalc`, `evenc`, `oddc`, `alldiffc` et `cardc` (qui est une contrainte "exactly"). Les contraintes arithmétiques dans le cas de nombres ont des fonctions de distances simples, c'est le cas de `=c`, `<c`, `<=c`.

Il y a cependant un gros problèmes pour combiner les contraintes, celui de la normalisation. On ne maîtrise pas l'image de la fonction de coût, qui dépend non seulement de son expression mais aussi du domaine des variables de son scope. Pour deux contraintes  $C_1$  et  $C_2$ , si  $C_1$  est une égalité sur les entiers et  $C_2$  un différent, et si le domaine est l'ensemble des multiples de 100 entre 6000 et 7200 (ce qui n'est pas un cas pathologique, c'est le domaine de valeur de la gamme **chromatique**<sup>1</sup>), on aura comme image de  $f_{C_1}$  l'ensemble  $\{0, 100, 200, 300 \dots 1200\}$  et comme image de  $f_{C_2}$  l'ensemble  $\{0, 1\}$ . Il est délicat alors de comparer les coûts, et la sélection de la plus mauvaise variable aussi bien que les modifications de valeurs ne se feront que pour  $C_1$  sans tenir compte de  $C_2$ .

C'est un problème complexe. Toutes les solutions naturelles sont mauvaises, au moins dans certains cas. Une première solution serait de calculer les coûts sur un échantillon d'instanciation, puis de moyenner par les valeurs obtenues (soit le maximum, soit la moyenne). Dans le cas précédent, on aurait une nouvelle fonction de coût pour  $C_1$  qui ressemblerait à  $f'_{C_1} = 1/1200 f_{C_1}$ . Cela ne résout le problème des images qu'en apparence. Les images seraient des intervalles beaucoup plus semblables  $\{0, 1/1200, 2/1200 \dots 1\}$  et  $\{0, 1\}$  mais la comparaison n'a toujours pas de sens, parce que toutes les valeurs de l'image de  $f_{C_1}$  autres que 0 et 1 sont identiques au sens de  $f_{C_2}$ . Une deuxième possibilité serait de prendre systématiquement le nombre de conflits comme fonction de coût, de manière à avoir des valeurs homogènes. C'est implémenté dans

<sup>1</sup>Voir gamme dans le glossaire en annexe

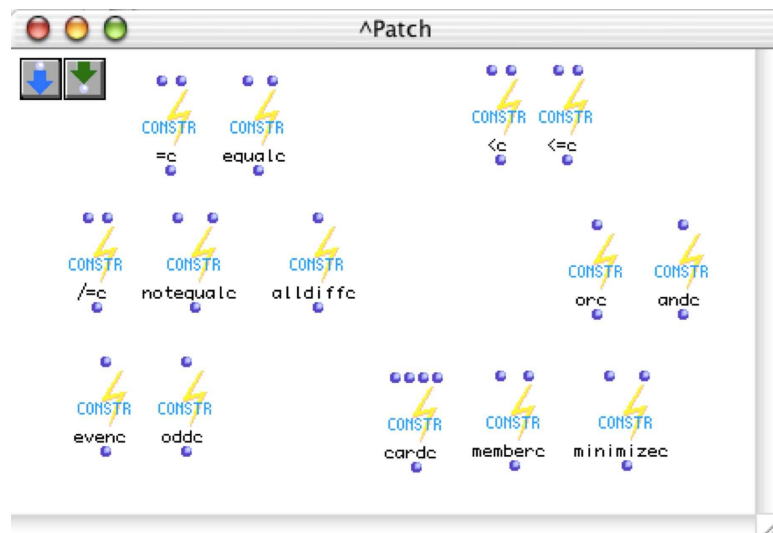


FIG. 27.3 – OMClouds intègre des primitives de contraintes de plusieurs types : égalités, inégalités, différent, opérateurs logiques, parité, cardinalité et appartenance.

OMClouds. Cependant, on perd forcément sur les problèmes avec des contraintes arithmétiques.

Indépendamment du problème de normalisation, trouver les bonnes fonctions de coût est difficile à automatiser, parce qu'il est difficile de trouver des règles générales. Le problème 8 sur les gestes est un bon exemple de cette difficulté et pourtant c'est indéniablement un problème combinatoire bien posé. Le problème 9 sur les all-interval series est un autre exemple : les fonctions de coûts "habituelles" fonctionnent, mais une petite modification améliore beaucoup la convergence. Le problème 6 donne aussi un exemple de trois contraintes identiques, qui ont pourtant des sémantiques différentes, parce que l'on cherche des solutions approchées. C'est cette fois le voeu de l'utilisateur qui intervient (ce qui devient intermédiaire avec un problème de modélisation).

En résumé, le choix effectué dans OMClouds est le suivant : par défaut, les fonctions de coût sont celles du tableau 27.4 (le nombre de conflits, implémenté, n'est pas accessible simplement à l'utilisateur). A charge pour l'utilisateur d'observer les minima locaux et de pondérer, ou modifier, ces fonctions selon les résultats. Les opérations "pondérer" ou "écrire une fonction de coût from scratch" sont extrêmement simple dans OMClouds.

Enfin, pour pondérer les fonctions de coût, il suffit d'ajouter un \* (*via* la fonction `OpenMusic om*` par exemple) à la sortie de la contrainte.

## 27.4 Résolution

L'algorithme de recherche adaptative est implémenté dans la méthode **résolution**. Elle a par défaut une entrée, qui prend une instance d'**asvarliste**. Ses autres entrées sont optionnelles. La deuxième sert à l'édition de résultats approchés, voir ci-dessous. La troisième est la longueur tabu  $l_{tabu}$ , qui est fixée par défaut au nombre de variables. On peut la changer soit pour accélérer la résolution, soit pour forcer la recherche autour des minima locaux. La quatrième est la condition d'arrêt  $\epsilon$ , qui est fixée par défaut à 0. On peut la changer soit pour obliger le solver à ne pas terminer, et a renvoyer indéfiniment des solutions (il suffit de la mettre à  $-1$

| Contrainte $C$                           | Coût $f_C$                                                        |
|------------------------------------------|-------------------------------------------------------------------|
| $x_1 =_c x_2$                            | $ x_1 - x_2 $                                                     |
| $x_1 \text{ equal } x_2$                 | 0 si $x_1 \text{ equal } x_2$<br>1 sinon                          |
| $x_1 <_c x_2$                            | $\max(1, x_1 - x_2)$                                              |
| $x_1 \leq_c x_2$                         | $\max(0, x_1 - x_2)$                                              |
| $x_1 \neq_c x_2$                         | 1 si $x_1 = x_2$<br>0 sinon                                       |
| $x_1 \text{ notequal } x_2$              | 1 si $x_1 \text{ equal } x_2$<br>0 sinon                          |
| $C_1 \text{ and } C_2$                   | $\max(f_{C_1}, f_{C_2})$                                          |
| $C_1 \text{ or } C_2$                    | $\min(f_{C_1}, f_{C_2})$                                          |
| $\text{even } x$                         | 0 si $x$ est pair<br>1 sinon                                      |
| $\text{odd } x$                          | 0 si $x$ est impair<br>1 sinon                                    |
| $\text{minimize } x$                     | $x$                                                               |
| $\text{all diff } x$                     | nombre de redondances dans $x$                                    |
| $\text{card } i \text{ l } \text{el } n$ | $ ( \text{nombre d'éléments de la } l \text{ valant } el ) - n  $ |

FIG. 27.4 – Transformation des contraintes en fonctions de coûts.

par exemple), soit au contraire pour arrêter la recherche à partir d'un certain seuil si l'on sait que le problème n'a pas de solutions. La dernière entrée optionnelle est le nombre d'itérations avant de réinitialiser le solveur.

L'algorithme est implémenté sous la forme d'une suite de fonctions génériques dont les méthodes sont propres à l'objet asvar (liste, perm ou cycle). La fonction résolution :

- vérifie la condition d'arrêt
- réinitialise aléatoirement les variables
- appelle la fonction générique une-descente
- itère

La fonction une-descente :

- vérifie la condition d'arrêt ou le nombre d'itérations avant de dépasser un nombre fixé
- met à jour la liste tabu et l'objet csp (en calculant toutes les erreurs, en appelant les coûts stockés dans les slots de l'objet)
- calcule un pire indice parmi ceux qui ne sont pas tabu
- si ce pire-indice est vide (liste tabu pleine), sort avec l'objet courant
- sinon, calcule la meilleure modification de la variable de pire indice
- si cela améliore le coût global, itère
- sinon, marque la variable de pire indice tabu et itère

Dans tout ceci, les fonctions suivantes sont génériques : une-descente, mise-à-jour, pire-indice, modification-de-moindre-coût, modification-aléatoire. Dans certains cas, est utilisée une fonction générique print-var pour afficher les résultats intermédiaires (valeurs et erreurs) dans le Listener. De cette manière, il est possible de modifier certains choix heuristiques, par exemple l'exploration du voisinage ou les réinitialisations aléatoires.

Plus intéressant, la première entrée optionnelle sert à exploiter les résultats partiels ren-

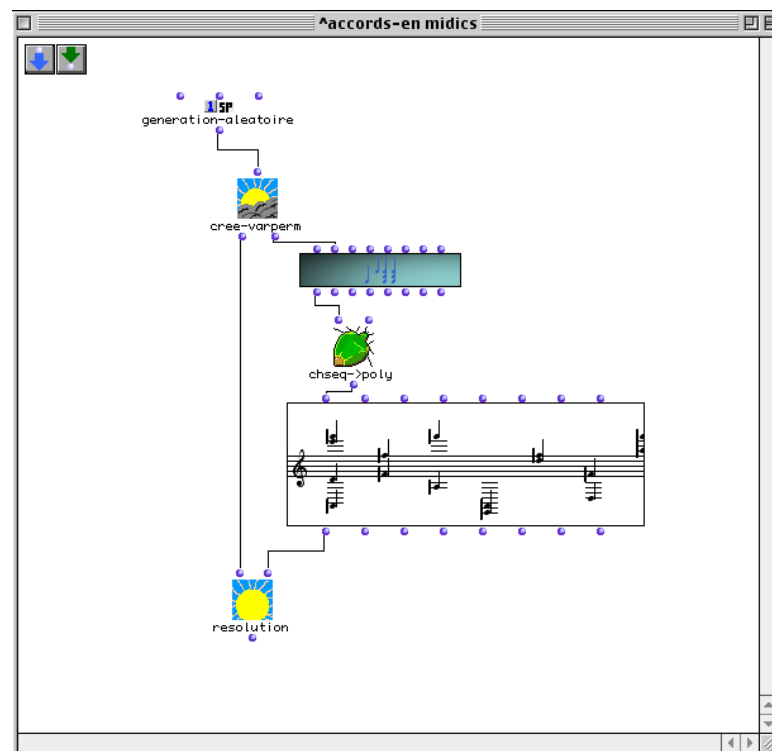


FIG. 27.5 – OMClouds permet d'exploiter des résultats partiels. Ici le problème du classement d'accords. On récupère les variables, suites de valeurs midi, que l'on place dans un objet chordseq. La fonction chordseq->poly représentée par l'insecte sert à lier les notes pour mieux voir le résultat. L'ensemble de ce calcul est actualisé à chaque minimum local.

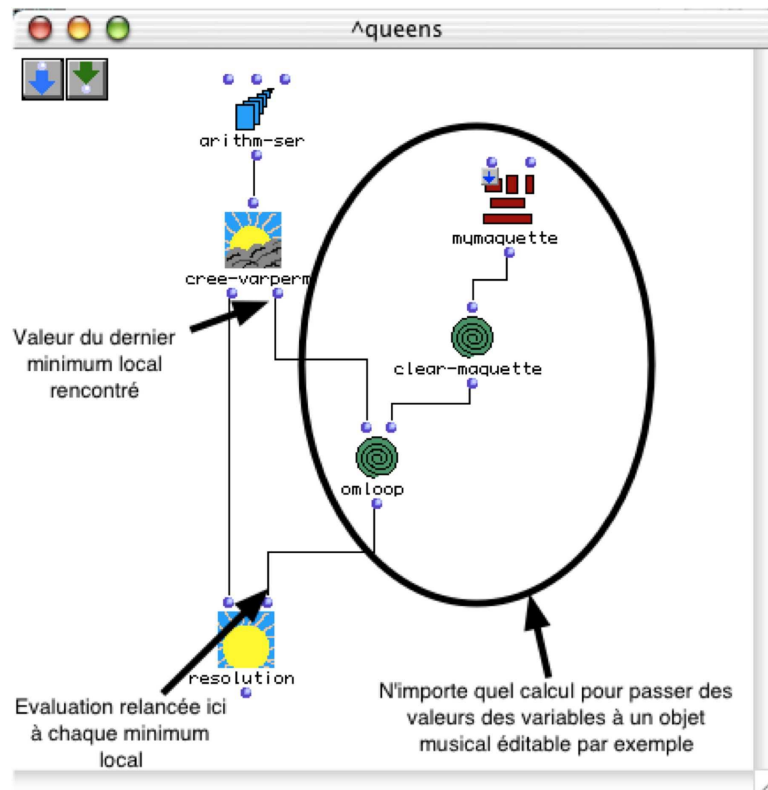
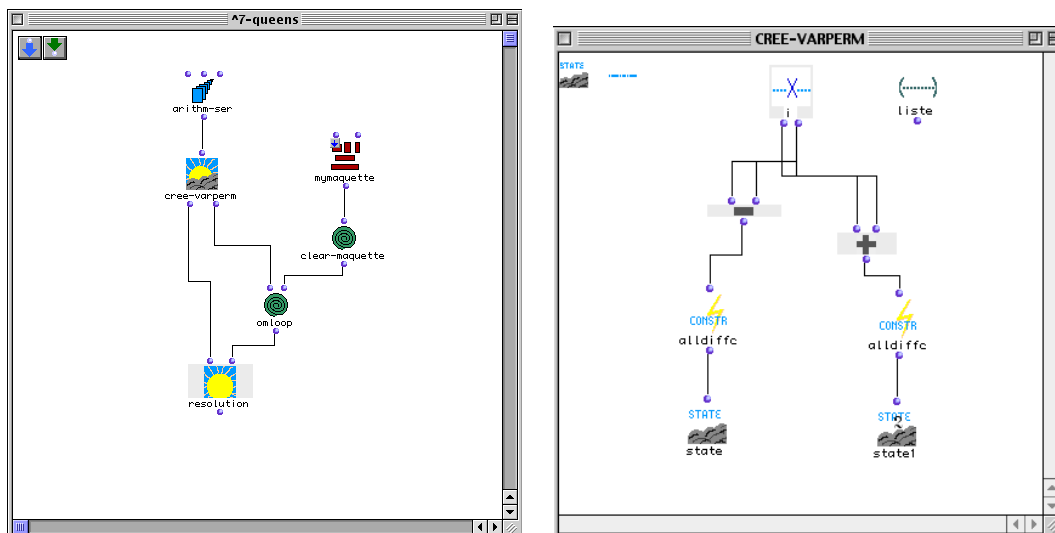


FIG. 27.6 – Pour éditer, visualiser, écouter un résultat à partir des variables, l'utilisateur peut ajouter un calcul qui sera effectué à chaque minimum local.

contrés lors de la recherche. Quand un minimum local est trouvé, la boîte **résolution** fait deux mises à jour : d'abord, elle actualise les "meilleures valeurs" dans la deuxième sortie de la boîte **asvarliste**, ensuite elle réévalue l'entrée optionnelle.

Ceci est implémenté en modifiant la fonction **om-ng-box-value** d'OpenMusic, qui calcul en général la valeur d'une boîte et lance la chaîne d'évaluation sur les boîtes qui sont reliées à ses entrées. Dans le cas d'une boîte OpenMusic normale (classe **OMBoxcall**), et d'une évaluation normale, **omng-box-value** récupère les entrées, récupère la référence de la boîte (le slot **référence** d'une boîte OpenMusic contient le nom de la fonction définie pour cette boîte), et calcule la valeur de la ou des sortie(s). Pour une boîte de la fonction **résolution** (classe **box-resol**, sous classe de **OMBoxcall**), c'est entièrement différent, parce qu'il ne s'agit pas d'une évaluation fonctionnelle. De sorte que la fonction normalement stockée dans **reference** n'est jamais appelée, et l'algorithme est implémenté directement dans **omng-box-value**.

L'implémentation est sous la forme donnée ci-dessus dans le cas où une seule entrée de la boîte est utilisée. Si la deuxième entrée optionnelle est utilisée, on ajoute un mécanisme pour relancer les évaluations sur cette deuxième entrée. Cela suppose de modifier aussi la valeur de l'instance du csp dans la boîte de définition. L'architecture dans ce cas doit suivre le modèle montré figure 27.6, l'idée étant qu'entre le csp modélisé, et ce que l'utilisateur souhaite entendre ou visualiser, il y a souvent un calcul. Par exemple, un csp portant sur des notes sera modélisé avec des hauteurs midi, des entiers, qu'il faut ensuite mettre en entrée d'une instance des classes **note** ou **accord**. Il peut aussi s'agir de calculs plus complexes comme montré figure 27.5, où, dans l'exemple du classement d'accord (Problème 3), une fonction de la librairie Repmus est utilisée

FIG. 27.7 – Les  $n$ -reines dans OMClouds

pour lier les notes communes entre les accords de manière à voir et entendre immédiatement la qualité de la solution trouvée.

En pratique, à chaque fois que la fonction *une-descente* termine, plusieurs opérations sont effectuées : d'abord, le slot *value* de la boîte de définition de contraintes (instance de *asbox-liste*, sous classe de *asbox*, sous classe de *box-with-patch*) est mis à jour avec comme valeur la liste des valeurs des variables dans le minimum local. La méthode *omng-box-value* pour les *asbox* est écrite de manière à ne jamais recalculer cette sortie, et à renvoyer toujours soit les valeurs à l'initialisation, soit ces valeurs de minimum local. Ensuite, la méthode *omng-box-value* pour la boîte de résolution affiche un certain nombre d'informations sur le minimum local dans le Listener, avec la fonction générique *print-var* (par défaut, les valeurs, les erreurs et l'erreur globale). Enfin, elle relance une évaluation sur l'entrée de gauche. De cette manière, tous les calculs sont ré-effectués jusqu'à la deuxième sortie de *asbox-liste*.

## 27.5 Détail d'un exemple

L'exemple choisi est le problème classique des  $n$ -reines. Rappelons qu'il s'agit de placer  $n$  reines sur un échiquier de taille  $n$  par  $n$ , de telle manière que deux reines ne puissent pas se manger avec les règles habituelles des échecs.

Ce problème est modélisé en CSP sur les permutations de  $n$  variables  $Q_1 \dots Q_n$ , où  $Q_i$  représente la position de la reine située sur la  $i$ -ième colonne (voir Chapitre 1). Les contraintes sur les diagonales qui s'écrivent  $alldiff(Q_i + i)$  et  $alldiff(Q_i - i)$ .

L'utilisateur commence par faire un *cree-varperm*. Cette fonction est représentée par une boîte *as-box*, sous-classe de la classe OpenMusic *box-with-patch* (utilisée par exemple pour les *omloop*). Les définitions des deux classes sont (on omettra les slots de documentation par soucis de lisibilité)

```
(defclass box-with-patch (OMBoxcall)
  ((patch :initform nil :accessor patch)))
(defclass asbox (box-with-patch)
```

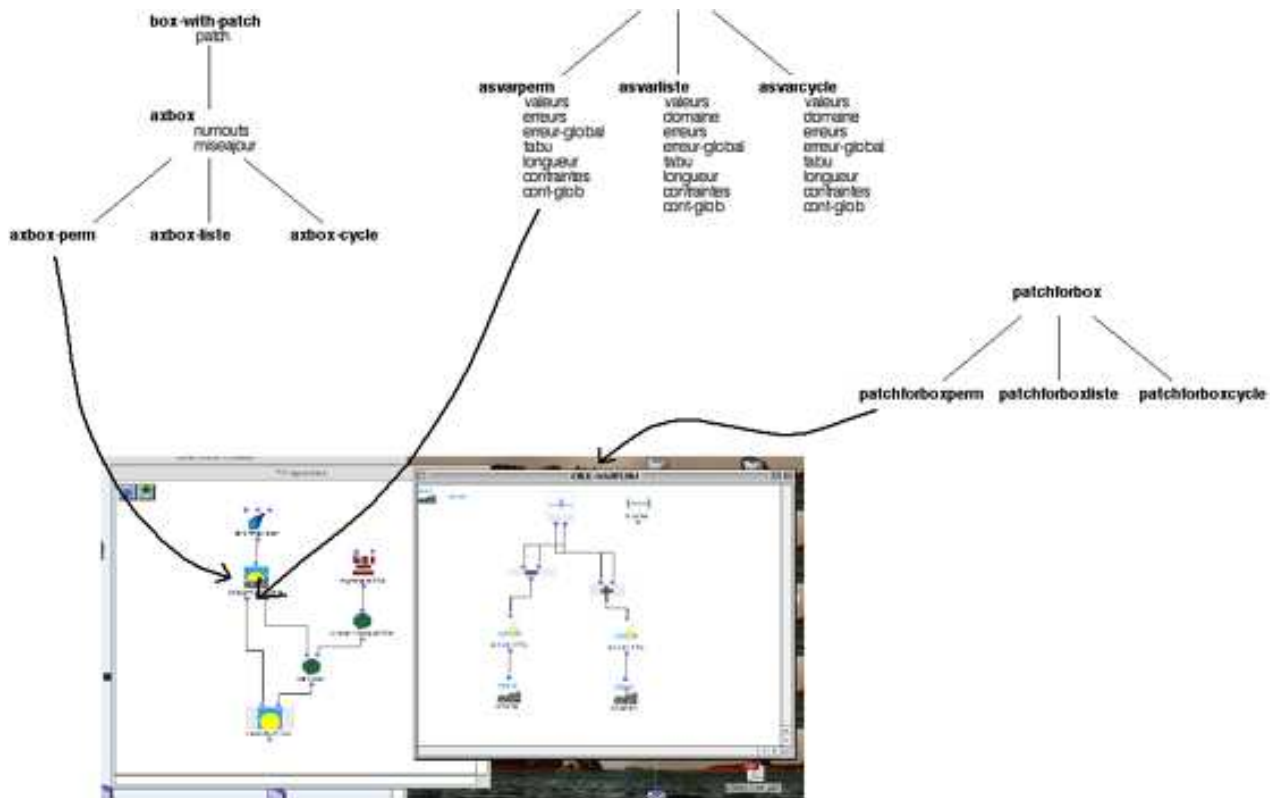


FIG. 27.8 – Les principales classes définies pour OMClouds.

```
((numouts :initform 2 :accessor numouts)
 (miseajour :initform nil :accessor miseajour)))
```

Le slot `patch` contient l'instance du `patchforvarperm` créé en même temps que `cree-varperm`. C'est celui qui est ouvert quand l'utilisateur double clique sur la `as-box` correspondante. Tous les patches OpenMusic contiennent un éditeur qui contient lui même un panel. Ces deux classes sont redéfinies pour `patchforvarperm` de manière à pouvoir spécifier les entrées et sorties propres à la définition des contraintes (classes `vareditor` et `varpanel`). Donc, lorsque l'on ouvre le `patchforvarperm`, lui est associée un `varpanel`, laquelle a des méthodes permettant d'afficher les entrées et sorties correspondant aux contraintes.

Par défaut, le `varpanel` contient une entrée représentant la variable courante (instance de `var-i`, sous-classe de `omin`), une entrée représentant la liste de toutes les variables (instance de `varliste`, sous-classe de `omin`), et une sortie pour poser les contraintes (instance de `varstate`, sous-classe de `omout`). Leurs définitions sont :

```
(defclass var-i (omin)
 ((sorties :initform 1 :accessor sorties)
 (nomliste :initform nil :accessor nomliste)))
(defclass varliste (omin) ())
(defclass varstate (omout) ())
```

La classe `omin` a un slot `in-symbol` qui contient un symbole quelconque (généré, unique), qui servira à identifier dans le code le paramètre correspondant. Dans le cas habituel, chaque entrée représente un paramètre de la fonction définie par le patch, donc chaque entrée a son `in-symbol`. Dans le cas des définitions de contraintes, c'est entièrement différent puisque la fonction définie

| Taille $n$ | tabu = $n$ |       | tabu = $n/3$ |       | tabu = $n/2$ |       |
|------------|------------|-------|--------------|-------|--------------|-------|
|            | Iter       | Temps | Iter         | Temps | Iter         | Temps |
| 10         | 295        | 0,3   | 276          | 0, 19 | 219          | 0,2   |
| 12         | 1879       | 2,5   | 1514         | 1,8   | 1059         | 1,4   |
| 14         | 8991       | 16    | 7014         | 13    | 8290         | 16    |
| 16         | 116635     | 332   | 54317        | 149   | 64697        | 183   |
| 18         |            |       | 593429       | 2337  |              |       |

FIG. 27.9 – Résultats sur le problème des séries tous intervalles dans OMClouds. Les temps sont en secondes. Moyenne sur dix résolutions.

par le patch a toujours deux paramètres exactement,  $i$  et  $l$ . Les deux entrées **var-i** et **varliste** se voient affecter deux symboles uniques que nous appellerons *i-symbol* et *liste-symbol* dans la suite, de la même manière que les **omin**. Pour **var-i+** en revanche, le *in-symbol* doit représenter  $i + \dots$  dans le code généré. En tant que sous-classe de **omin**, **var-i+** a un slot indice qui stocke son indice, et que l'on utilise pour connaître les  $\dots$  dans  $i + \dots$ .

Le slot *in-symbol* de **var-i+** est donc affecté à 'I-Symbol + l'indice de l'instance de **var-i+**, et c'est ce que apparaîtra dans la génération de code.

La contrainte  $alldiff(Q_i + i)$  suppose de connaître la valeur de  $Q_i$ , que l'on a en sortie de **var-i**. Il faut, en plus, l'indice  $i$ . On l'obtient en tapant le caractère **n** dans **patchforvarperm**, ce qui a pour effet de mettre à 2 le slot numouts de toutes les instances de **var-i** (et **var-i+** s'il y en a mais ce n'est pas le cas dans les reines). L'action contraire est obtenue en tapant **N**.

Pour obtenir  $Q_i + i$ , il suffit de connecter les deux sorties de **var-i** à une boîte **om+**. Le **alldifferent** est dans la liste des primitives de contraintes sous le nom **alldiffc**. On obtient une boîte **alldiffc** à laquelle il faut connecter la sortie de la boîte **om+**. On a alors écrit  $alldiff(Q_i + i)$ , qu'il reste à connecter à la sortie du patch, instance de **varstate**, voir figure 27.7.

## 27.6 Performances

Les performances mesurées dans OMClouds sont, d'une manière générale, suffisamment bonnes pour notre application, mais cependant beaucoup plus mauvaises que celles données par exemple dans [42].

### 27.6.1 Séries tous intervalles

Sur le problème des séries tous intervalles, les fonctions de coût générées automatiquement par OMClouds sont un peu différentes des fonctions "biaisées" de [42]. Le tableau 27.9 donne les résultats mesurés sur dix essais, avec une longueur tabu variable, et toutes les variables ré-initialisées quand la liste tabu est pleine. Les temps sont donnés en secondes. L'ordinateur utilisé est un Macintosh G4 gris sous Mac OS 9.

Ces résultats peuvent être comparés à ceux de [42], repris dans le tableau 27.10. Il s'agit de moyennes de dix tests sur une machine Pentium 733. Si la comparaison des temps de calcul ne vaut qu'en ordre de grandeur, la comparaison du nombre d'itérations est favorable à l'implémentation de Daniel Diaz. On peut avancer les explications suivantes : OMClouds génère automatiquement des fonctions de coût moins bonnes que les fonctions de coûts pondérées par

|    | OMClouds |       | Diaz Codognet |       |
|----|----------|-------|---------------|-------|
|    | Iter     | Temps | Iter          | Temps |
| 10 | 276      | 0, 19 | 14            | 0     |
| 12 | 1514     | 1,8   | 46            | 0     |
| 14 | 7014     | 13    | 85            | 0     |
| 16 | 54317    | 149   | 191           | 0,01  |
| 18 | 593429   | 2337  | 694           | 0,04  |

FIG. 27.10 – Comparaison entre OMClouds et l'implémentation de Daniel Diaz en C.

| Taille $n$ | tabu = $n$ |        | tabu = $n/2$ |       | tabu = $n/3$ |       |
|------------|------------|--------|--------------|-------|--------------|-------|
|            | Iter       | Temps  | Iter         | Temps | Iter         | Temps |
| 10         | 3,8        | 295    | 2            | 219   | 1,9          | 276   |
| 12         | 25         | 1879,5 | 14           | 1059  | 18           | 1514  |
| 14         | 169        | 899,1  | 163          | 8291  | 130          | 7015  |
| 16         | 3320       | 116635 | 1828         | 64697 | 1493         | 54317 |

FIG. 27.11 – Résultats sur le problème des  $n$ -reines dans OMClouds. Les temps sont en secondes. Moyenne sur dix résolutions.

la taille de l'intervalle. Les langages utilisés, Common Lisp et C, ne sont probablement pas comparables. Les paramètres ont été moins finement réglés dans notre implémentation (tabu =  $n/2$ , toutes les variables ré-initialisées) que dans celle de [42] (tabu =  $n/10$ , 10 % des variables ré-initialisées). Et surtout, l'implémentation d'OMClouds, qui n'est pas axée sur la vitesse de résolution, empêche un grand nombre d'optimisation. En particulier, le fait de laisser les fonctions de coût en paramètre et générées fait perdre probablement beaucoup de temps dans la résolution par rapport à une implémentation *ad hoc*. Le temps moyen pour une itération pour  $n = 18$  est ainsi de 3,9 ms pour OMClouds contre 0,05 ms pour [42].

### 27.6.2 Reines

Nous avons testé le problème des  $n$ -reines dans OMClouds avec la même machine et les mêmes options que ci-dessus. Les résultats présentés tableau 27.11 confirment qu'une petite longueur tabu est préférable, si l'on recherche avant tout l'efficacité. De la même manière que ci-dessus, OMClouds est beaucoup plus lent que [42] qui résoud des instances jusqu'à une taille 100000 en moins d'une heure. Les raisons sont probablement les mêmes que détaillées ci-dessus.

Nous avons retrouvé sur la plupart des problèmes muaicaux admettant des solutions des résultats similaires : l'algorithme est plus efficace avec des longueurs tabu de l'ordre du nombre de variables, divisé par 2 ou 3.

### 27.6.3 Conclusion

La remarque principale est que, dans le cas où il y a des solutions, l'algorithme termine plus vite avec une petite longueur tabu. Cela suggère de donner par défaut dans OMClouds une longueur tabu de l'ordre du nombre de variables, divisé par 2 ou 3 par exemple. Le choix par défaut est crucial car il est peu probable que l'utilisateur vienne modifier ces paramètres.

Cependant, ceci ne vaut que pour les problèmes ayant une solution. Pour les problèmes surcontraints, la qualité des solutions approchées est plus grande avec une plus grande longueur tabu (rappelons que un minimum local avec une longueur tabu de  $k$  est garanti un minimum local sur  $k$  des  $n$  dimensions du problème). Or, parmi les problèmes musicaux traités, nombreux sont surcontraints. Il nous a semblé plus important de donner de bonnes solutions approchées que de gagner un facteur dans la résolution exacte, de sorte que la longueur tabu fixée par défaut dans OMClouds est égale au nombre de variables.

On constate également que l'implémentation d'OMClouds est beaucoup moins rapide que le code C de Daniel Diaz et Philippe Codognet. Ceci a été expliqué ci-dessus. OMClouds reste raisonnablement efficace pour notre application.

## 27.7 Conclusions

OMClouds est intégrée à la version 4.6.5 d'OpenMusic. Il est prévu de lui ajouter des fonctionnalités : d'abord, de permettre à l'utilisateur de ralentir ou stopper la résolution de manière à voir plus facilement les résultats intermédiaires. Ensuite, nous envisageons d'ajouter des couleurs dans les éditeurs musicaux pour distinguer les variables dont l'erreur est nulle des autres. Il y a cependant un problème pour savoir à quelles variables correspondent quels objets musicaux. Enfin, nous prévoyons d'utiliser la liste tabu pour réduire le problème en cours de résolution. Par exemple, si l'utilisateur est satisfait par la première moitié d'un résultat intermédiaire, mais pas par la suite, il serait utile de pouvoir bloquer la première moitié de l'instanciation et de continuer à résoudre sur la deuxième. D'un point de vue CSP, il faudrait redéfinir et résoudre le sous-CSP correspondant. On peut très facilement le faire en recherche adaptative puisque ce sont les variables qui sont marquées tabu : il suffit de forcer les premières variables dans la liste tabu, de manière à ce qu'elles ne soient jamais sélectionnées. L'avantage est de ne pas avoir à ré-instancier le CSP.

Concernant la composition assistée par ordinateur, l'apport le plus important de OMClouds est certainement l'édition des résultats intermédiaires, et le contrôle donné à l'utilisateur sur la résolution. Au contraire d'une boîte opaque qui renvoie des résultats au bout d'un temps indéfini, résultats pouvant être "il n'y a pas de solutions", OMClouds fournit des propositions. Nous pensons que cette manière très souple de voir la programmation par contraintes est beaucoup mieux adaptée à la composition, et correspond à la philosophie de calcul musical interactif défendue par OpenMusic



# Chapitre 28

## Conclusions

### 28.1 Conclusion (français)

Nous avons présenté un travail sur l'utilisation de certaines techniques de programmation par contraintes à l'informatique musicale, ainsi qu'un catalogue de quatorze Constraint Satisfaction Problems musicaux. Nous avons utilisé notamment un algorithme de recherche local implémenté sous la forme d'une librairie dans le logiciel OpenMusic. La musique offre probablement à la programmation par contraintes des problèmes originaux et une application où l'utilisation d'un solveur doit rester très souple.

L'utilisation de la recherche adaptative a soulevé un certain nombre de questions. Le constat général est que plus grande est la partie aléatoire, plus l'AS est efficace. D'un point de vue formel, il serait intéressant et probablement possible d'étudier la recherche adaptative ou la recherche Tabu de manière probabiliste, de manière par exemple à donner une probabilité sur le comportement à l'infini en fonction des distributions aléatoires choisies au départ (ou même pendant la recherche).

D'un point de vue musical, nous avons contribué (de loin) à huit créations. D'une manière générale, la programmation par contraintes semble avoir confirmé avoir une place en musique contemporaine. L'application aux contraintes instrumentales, avec les doigtés de guitare, semblent prometteuses et pourraient être étendues à d'autres instruments (mais cela suppose avoir accès à un expert musical). Par ailleurs, la recherche locale, dans laquelle on peut conserver une "meilleure instanciation calculée jusqu'à maintenant", pourrait servir à des applications *anytime*.

### 28.2 Conclusion (english)

In this document, works about the use of some constraint programming techniques have been presented, and a catalogue of fourteen musical Constraint Satisfaction Problems. A local search method has mainly been used and implemented as a library of the software OpenMusic. Music offers to Constraint Programming original problems, and an application where the use of a solver must be seen in a very flexible way.

The use of adaptive search rose some questions. In practice, the main conclusion is that the bigger is the place of random choices in the algorithm, the best it works. Formally, it would be interesting and probably possible to study such algorithms as Adaptive or Tabu Search with probability tools, in order to give a probability distribution of the algorithm's behavior to

infinity, from the random initial distribution.

Concerning music, we have contributed (from far) to eight creations. Generally, constraint programming has confirmed to apply well to contemporary music. The instrumental application to guitar fingerings seems promising and could be extended to other instruments (but in order to to it, a musical expert is needed). Furthermore, local search, where a 'best instantiation computed so far" can be stored, could be used in anytime applications.

# Bibliographie

- [1] E. H. L. Aarts and J. K. Lenstra. *Local search in combinatorial optimization*. John Wiley and Sons, 1997.
- [2] Abderrahmane Aggoun and Nicolas Beldiceanu. Overview of the chip compiler system. *Proceedings of ICLP91*, pages 775–789, 1991.
- [3] Carlos Agon. *An environment for computer assisted composition*. Thèse de doctorat, IRCAM-Université de Paris VI, 1998.
- [4] F. Ajili and M. Wallace. Hybrid problem solving in eclipse. *Chapter 6 in Constraint and Integer Programming : Toward a Unified Methodology*, pages 169–201, 2003.
- [5] E. Amiot, G. Assayag, C. Malherbe, and A. Riotte. Duration structure generation and recognition in music writing. *Proceedings of ICMC'86*, 1986.
- [6] Moreno Andreatta. *La théorie mathématique de la musique de Guerino Mazzola et les canons rythmiques*. Rapport de dea, ATIAM - Université Paris IV, 1999.
- [7] Alexis Anglada, Philippe Codognet, and Laurent Zimmer. Nscsp, définition and resolution by transformation. *Proceedings of the 5th International Workshop on Soft Constraints in CP03*, 2003.
- [8] G. Assayag, M. Castellengo, and C. Malherbe. Functional integration of complex instrumental sounds in music writing. *Proceedings ICMC'85*, 1985.
- [9] Gérard Assayag. Applications on contemporary music creation, esthetic and technical aspects. *1st Symposium on Music and Computers*, 1998.
- [10] Gérard Assayag, Camilo Rueda Mikael Laurson, Carlos Agon, and Olivier Delerue. Computer assisted composition at ircam : Patchwork & openmusic. *Computer Music Journal*, 1999.
- [11] H. Kautz B. Selman and B. Cohen. Noise strategies for improving local search. *Proc. AAAI'94*, 1994.
- [12] Fahiem Bacchus, Xinguang Chen, Peter van Beek, and Toby Walsh. Binary vs non-binary constraints. *Artificial Intelligence*, 2002.
- [13] M. Balaban, K. Ebcioglu, and O. Laske. Understanding music with ai : Perspectives on music cognition. *MIT Press*, 1992.
- [14] Philippe Ballesta. *Contraintes et objets, clefs de voûte d'un outil d'aide à la composition*. Editions Hermès, 1998.
- [15] V. Barichard, H. Deleau, J.K. Hao, and F. Saubion. A hybrid evolutionary algorithm for constraint satisfaction problems. *Lecture Notes in Computer Science*, 2003.
- [16] Romain Barták. Foundations of constraint satisfaction. *IJCAI 2003*, 2003.

- [17] Marc Battier and Gilbert Nouno. L'électronique dans l'opéra de kaija saariaho, l'amour de loin. *Musurgia*, 2003.
- [18] Frédéric Benhamou. Interval constraints. *Encyclopedia of Optimization*, 1999.
- [19] Christian Bessières and Jean-Charles Régin. Refining the basic constraint propagation algorithm. *Proceedings of IJCAI'01 : 17th International Joint Conference on Artificial Intelligence*, pages 309–315, 2001.
- [20] Anthony Beurivé and Myriam Desainte-Catherine. Representing musical hierarchies with constraints. *Proceedings of Musicons workshop at CP'01*, 2001.
- [21] S. Bistarelli, P. Codognet, Y. Georget, and F. Rossi. Approximation of local consistency for soft constraints. *Proc. PADL'2000 (Practical Aspects of Declarative Languages)*, 2000.
- [22] S. Bistarelli, Spencer K. L. Fung, Jimmy H. M. Lee, and Ho-Fung Leung. A local search framework for semiring-based constraint satisfaction problems. *Proceedings of the 5th International Workshop on Soft Constraints in CP03*, 2003.
- [23] S. Bistarelli, R. Gennari, and F. Rossi. General properties and termination conditions for soft constraint propagation. *Constraints*, 8, 2003.
- [24] Shi-Kuo Chang. Visual languages : A tutorial and survey. *IEEE Software Magazine*, 4 :29–39, 1987.
- [25] Shi-Kuo Chang, G. Polese, S. Orefice, and M. Tucci. A methodology and interactive environment for iconic language design. 1997.
- [26] Marc Chemillier. *La musique de la harpe, Une esthétique perdue*. Editions E. de Dampierre, Presses de l'Ecole Normale Supérieure, 1995.
- [27] Marc Chemillier. György ligeti et la logique des textures. *Analyse Musicale*, 38 :167 – 177, 1999.
- [28] Marc Chemillier. Générateurs musicaux et singularités. *JIM'99*, pages 75 – 85, 2001.
- [29] Marc Chemillier and Charlotte Truchet. Computation of words satisfying the rhythmic oddity property (after simha arom's works). *Information Processing Letters*, 2003.
- [30] Marc Chemillier, Charlotte Truchet, and Louis-Martin Rousseau. Analyse musicale et contraintes. *JIM'02*, 2002.
- [31] B.M.W. Cheng, J.H.M. Lee, and J.C.K. Wu. Speeding up constraint propagation by redundant modeling. *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming*, pages 91–103, 1996.
- [32] John Chowning. The simulation of moving sound sources. *Journal of the Audio Engineering Society*, 1971.
- [33] J.-P. Clendinning. The pattern-meccanico compositions of györgy ligeti. *Perspectives of New Music*, pages 192–234, 1993.
- [34] Philippe Codognet and Daniel Diaz. Compiling constraints in clp(fd). *Journal of Logic Programming (JLP)*, 27, 1996.
- [35] Philippe Codognet and Daniel Diaz. Yet another local search method for constraint solving. *LNCS 2246, SAGA 2001, first Symposium on Stochastic Algorithms : Foundations and Applications*, 2001.
- [36] Philippe Codognet and Daniel Diaz. An efficient library for solving csp with local search. *Proceedings of MIC03, Fifth Metaheuristics International Conference*, 2003.

- [37] Philippe Codognet, Daniel Diaz, and Charlotte Truchet. The adaptive search method for constraint solving and its application to musical csps. *Proceedings of the First International Workshop on Heuristics*, 2002.
- [38] Alain Colmerauer. An introduction to prolog iii. *Communications of the ACM*, 33, 1990.
- [39] Francis Courtot. Carla : Knowledge acquisition and induction for computer. *Rapport de recherche, Ircam*, 1990.
- [40] A. Davenport, E. Tsang, Z. Kangmin, and C. Wang. Genet : a connectionist architecture for solving constraint satisfaction problems by iterative improvement. *Proc. AAAI 94*, 1994.
- [41] P. Dellacherie. Modélisation et optimisation d'un harmoniseur automatique de chorals en plc(fd). *Mémoire de DEA*, 1998.
- [42] Daniel Diaz and Philippe Codognet. Design and implementation of the gnu prolog system. *Journal of Functional and Logic Programming*, 6, 2001.
- [43] R. Dorne and J.K. Hao. A new genetic local search algorithm for graph coloring. *Lecture Notes in Computer Science 1498*, pages 745–754, 1998.
- [44] Kemal Ebcioglu. *An expert system for harmonizing chorals in the style of J.-C. Bach*. AAAI Press, 1987.
- [45] E.Goncharov and Yu.Kochetov. Probabilistic tabu search algorithm for the multi-stage uncapacitated facility location problem. *E.Goncharov and Yu.Kochetov*, 2000.
- [46] Xavier Rodet et Pierre Cointe. Formes composition et ordonnancement de processus. *Rapport de recherche numéro 36, IRCAM*, 1985.
- [47] H. Fang, Y. Kilani, J. H. M. Lee, and P. J. Stuckey. Reducing search space in local search for constraint satisfaction. *Proceedings of the Eighteenth national conference on Artificial intelligence*, pages 28 – 33, 2002.
- [48] Serge Fenet and Christine Solnon. Searching for maximum cliques with ant colony optimization. *Lecture Notes in Computer Science N°2611, Applications of evolutionary computing (EvoCOP 2003)*, pages 236–245, 2003.
- [49] Ian P Gent, Holger H Hoos, Andrew G D Rowley, and Kevin Smyth. Using stochastic local search to solve quantified boolean formulae. *Proceedings of CP03*, 2003.
- [50] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.
- [51] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. *AAAI'98*, 2000.
- [52] Volker Haarslev. Formal semantics of visual languages using spatial reasoning. *Proceedings of the 11th Symposium on Visual Languages*, 1995.
- [53] Volker Haarslev. A fully theory for describing visual notations. *Proceedings of the International Workshop on the Theory of Visual Languages*, 1996.
- [54] Jin-Kao Hao, Philippe Galinier, and Michel Habib. Métaheuristiques pour l'optimisation combinatoire et l'affectation sous contraintes. *Journal of Heuristics*, 1998.
- [55] Warwick Harvey. The fully social golfer problem. *Proceedings of SymCon'03 : Third International Workshop on Symmetry in Constraint Satisfaction Problems, a workshop of CP 2003*, pages 75–85, 2003.

- [56] Martin Henz, Stefan Lauer, and Detlev Zimmermann. Compoze - intention music composition through constraint programming. *Proceedings of the 8th International Conference on Tools with Artificial Intelligence*, 1996.
- [57] Edward A. Hirsch and Arist Kojevnikov. Solving boolean satisfiability using local search guided by unit clause elimination. *Proceedings of CP01, 7th International Conference on Principles and Practice of Constraint Programming*, 2001.
- [58] J. H. Holland. *Adaptation in natural and artificial systems : An introductory analysis with applications in biology, control, and artificial intelligence*. MIT Press/Bradford Books, 1992.
- [59] Jean-Marc Jot. Real-time spatial processing for music, multimedia and interactive human-computer interfaces. *Multimedia Systems*, 7 :55 – 69, 1999.
- [60] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1) :21–45, 2002.
- [61] François Laburthe and Yves Caseau. Salsa : A language for search algorithms. *constraints*, 7 :255–288, 2002.
- [62] Mikaël Laurson. Patchwork : a visual programming language and some musical applications. *Sibelius Academy*, 1996.
- [63] Mikaël Laurson and Mika Kuuskankare. A constraint based approach to musical textures and instrumental writing. *Proceedings of the CP01 Workshop on Musical Constraints*, 2001.
- [64] S. Lin. Computer solutions of the traveling salesman problem. *Bell System Technical Journal*, 44 :2245–2269, 1965.
- [65] A.K. Mackworth and E.C. Freuder. The complexity of some polynomial network consistency algorithms for constraint satisfaction problems [ac1-3]. *Artificial Intelligence*, 25 :65–74, 1985.
- [66] Kim Marriott and Bernd Meyer. Formal classification of visual languages. *Proceedings of the International Workshop on Theory of Visual Languages*, 1996.
- [67] Kim Marriott and Bernd Meyer. Towards a hierarchy of visual languages. *Proceedings of the 12th International IEEE Workshop on Visual Languages*, 1996.
- [68] Kim Marriott and Peter J. Stuckey. *Programming with Constraints : An Introduction*. MIT Press, 1998.
- [69] Marcel Mesnage. Sur la modélisation des partitions musicales. *Analyse Musicale*, 1991.
- [70] Zbigniew Michalewicz and David B. Fogel. *How to Solve It : Modern Heuristics*. Springer, 2000.
- [71] L. Michel and P. Van Hentenryck. Localizer : A modeling language for local search. *INFORMS Journal of Computing*, 11(1) :1–14, 1999.
- [72] L. Michel and P. Van Hentenryck. Localizer. *Constraints*, 5 :41–82, 2000.
- [73] L. Michel and P. Van Hentenryck. Localizer++ : An open library for local search. *Technical Report, CS-01-03, Brown University*, 2001.
- [74] S. Minton, M. Johnston, A. Philips, and P. Laird. Minimizing conflicts : a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58 :161–205, 1992.

- [75] Robert Morris and Daniel Starr. The structure of the all-interval series. *Journal of Music Theory*, 13(2), 1974.
- [76] François Nicolas. Le feuilleté du tempo (essai sur les modulations métriques). *Revue Entretemps*, 9, 1990.
- [77] François Pachet. The muses system : an environment for experimenting with knowledge representation techniques in tonal harmony. *First Brazilian Symposium on Computer Music*, 1994.
- [78] François Pachet, Olivier Delerue, and Peter Hanappe. Dynamic audio mixing. *Proceedings of ICMC*, 2000.
- [79] François Pachet and Pierre Roy. Musical harmonization with constraints : a survey. *Constraints*, 2001.
- [80] François Pachet and Pierre Roy. Integrating constraint satisfaction techniques with complex object structures. *15th Annual Conference of the British Computer Society Specialist Group on Expert Systems*, pages 11–22, Décembre 1995.
- [81] François Pachet and Aymeric Zils. Musical mosaicing. *Proceedings of DAFX'01*, 2001.
- [82] Andreas Paepcke. Object-oriented programming - the clos perspective. *MIT Press*, 1993.
- [83] G. Pesant and M. Gendreau. A constraint programming framework for local search methods. *Journal of Heuristics*, 1999.
- [84] Stephen Pope. Introduction to mode : The musical object development. *MIT press*, 1991.
- [85] Y. Potard, P-F. Baisnée, and J-B.Barrière. Experimenting with models of resonance produced by a new technique for the analysis of impulsive sounds. *Proceedings of ICMC*, 1986.
- [86] V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith. Modern heuristic search methods. *Wiley*, 1996.
- [87] Monasson Remi, Zecchina Riccardo, Kirkpatrick Scott, Selman Bart, and Troyansky Lidror. 2+p-sat : Relation of typical-case complexity to the nature of the phase transition. *Random Structures*, pages 414–435, 1999.
- [88] André Riotte and Marcel Mesnage. Analyse musicale et systèmes formels : un modèle informatique de la 1ère pièce pour quatuor à cordes de stravinsky. *Analyse Musicale*, 1988.
- [89] Camilo Rueda, Gloria Alvarez, Luis O. Quesada, Gabriel Tamura, Frank D. Valencia, and Gérard Assayag. Integrating constraints and concurrent objects in musical applications : a calculus and its visual language. *Constraints*, 1999.
- [90] Camilo Rueda, Mikael Laurson, Georges Bloch, and Gérard Assayag. Integrating constraint programming in visual musical composition languages. *Proceedings of ECAI'98*, 1998.
- [91] Camilo Rueda and Franck Valencia. Formalizing timed musical processes with a temporal concurrent constraint programming calculus. *Proceedings of MusiCons Workshop at CP'01*, 2001.
- [92] C. Schulte, G. Smolka, and J. Würtz. Encapsulated search and constraint programming in oz. *Lecture Notes in Computer Science, Springer Verlag*, 1000 :324–343, 1994.
- [93] Meinolf Sellmann. Approximated consistency for knapsack constraints. *Proceedings of CP03*, 2003.

- [94] Meinolf Sellmann and Warwick Harvey. Heuristic constraint propagation using local search for incomplete pruning and domain filtering of redundant constraints for the social golfer problem. *Proceedings of CP-AI-OR*, 2002.
- [95] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. *AAAI'92*, pages 440–446, 1992.
- [96] Jeffrey Mark Siskind and David Allen McAllester. Nondeterministic lisp as a substrate for constraint logic programming. *AAAI' 93*, pages 133–138, 1993.
- [97] Christine Solnon. Ant-p-solver : un solver de contraintes à base de fourmis artificielles. *Journées Francophones de la Programmation Logique par Contraintes*, pages 189–204, 2000.
- [98] Christine Solnon. Ants can solve constraint satisfaction problems. *IEEE Transactions on Evolutionary Computation*, 6(4) :347–357, 2002.
- [99] Guy L. Steele. *Common LISP The language, second edition*. Digital Press, 1998.
- [100] H. Taube. Common music : A music composition language in common lisp and clos. *Computer Music Journal*, 15, 1991.
- [101] Godfried Toussaint. Classification and phylogenetic analysis of african ternary rhythm timelines. *Proceedings of BRIDGES : Mathematical Connections in Art, Music and Science*, pages 25–36, 2003.
- [102] C. P. Tsang and M. Aitken. Harmonizing music as a discipline of constraint logic programming. *ICMC*, pages 61–64, 1991.
- [103] E. P. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, 1993.
- [104] R. Jagadeesan V. Saraswat and V. Gupta. Foundations of timed concurrent constraint programming. *Proc. of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 71–80, 1994.
- [105] Michel Vasquez. Arc-consistency and tabu search for the frequency assignment problem. *Proc. CP-AI-OR 02*, pages 359–371, 2002.
- [106] Michel Vasquez, Djamel Habet, and Audrey Dupont. Neighborhood design by consistency checking. *Proc. MIC 03*, pages 19–27, 2003.
- [107] Dan Tudor Vuza. Supplementary sets - theory and algorithms. *Muzica*, pages 75–99, 1995.
- [108] Joachim P. Walser. *Integer Optimization by Local Search : a Domain-independant Approach*. Springer Verlag, 1999.
- [109] Ryan Williams, Carla Gomes, and Bart Selman. *Backdoors To Typical Case Complexity*. Proc. IJCAI-03, 2003.

# Glossaire

Ci-après sont définies quelques notions de solfège ou de musique utilisées. Attention, il ne s'agit pas de donner des définitions musicologiques précises de chaque entrée, mais simplement d'éclairer le lecteur peu familier avec ces termes.

**Accord** Un accord est un ensemble de notes jouées simultanément. La notion fait essentiellement référence à l'harmonie. Il arrive peut qu'on appelle accord soit un ensemble de notes jouées à des instants différents, soit un chiffage dans une tonalité donnée (un accord de quinte).

**Agrégat** Accord quelconque. Le terme agrégat sert à différencier celui-ci d'un accord, dont la structure est souvent codée.

**Altération** Signe modifiant la hauteur d'une note. Il peut être placé devant celle-ci pour modifier seulement les occurrences de cette note dans mesure courante, ou en tête de la partition et valoir pour toute la partition (voir tonalité). Un dièse  $\sharp$  monte la note d'un demi-ton, un double dièse d'un ton, un bémol  $\flat$  la descend d'un demi-ton, un double bémol d'un ton, un bécarré  $\natural$  replace la note à sa position normale.

**Ambitus** Qu'il s'agisse d'une mélodie, d'un accord ou d'une voix, l'ambitus désigne l'amplitude jouée ou possiblement jouée. Pour un accord ou une mélodie, c'est l'intervalle dans lequel sont les notes. Pour une voix, c'est l'étendue de la voix (ou tessiture).

**Barré** A la guitare, faire un barré signifie jouer un accord en maintenant le premier doigt posé à plat sur plusieurs cordes à la fois, le plus souvent sur toutes les cordes, parfois seulement sur des cordes les plus proches de la main.

**Basse chiffrée** Mode musicale consistant à noter sur la partition la voix de basse et le chiffage des accords, dont la réalisation est laissée au musicien. Les exercices d'harmonisation consistent à réaliser les accords à partir de ces deux informations, sans faire de fautes d'harmonie.

**Canon** Un canon à  $n$  voix est une partition sur laquelle la première voix joue une mélodie, reprise avec un décalage temporel par la deuxième voix, reprise avec un décalage temporel par la troisième voix, etc (plusieurs chansons populaires peuvent se chanter en canon, comme "Frère Jacques" ou "Vent frais", avec un décalage d'une phrase). Un canon est très difficile à construire pour des problèmes d'harmonie.

**Chiffage d'un accord** En musique classique, les accords sont codés par des chiffres qui représentent les intervalles joués dans l'accord, à quelques elisions près parfois. Le chiffage d'un accord classique donne théoriquement les intervalles entendus dans l'accord, à partir de la basse. La notation est conditionnée par l'usage : ainsi, mi - sol - do sera chiffré 6 (sous entendu,  $\frac{6}{3}$  mais la tierce n'est pas précisée d'habitude), alors que sol - do - mi sera chiffré  $\frac{6}{4}$  (on précise ici les deux intervalles entendus, la sixte et la quarte). Le chiffage d'un accord donne des indications sur l'ordre vertical des notes (voir renversement), sur la nature de l'accord (quinte, septième, etc), mais pas sur le degré sur lequel il est joué ni la tonalité.

**Contrepoint** Art d'écrire la musique sous la forme de plusieurs voix mélodiques, qui, jouées ensemble, sonnent harmonieusement. C'est une vision d'abord horizontale de la musique. Par exemple, le canon ou la fugue relèvent du contrepoint.

**Crescendo, decrescendo** Indication sur la partition signifiant à l'interprète qu'il doit jouer les notes de plus en plus fort. Le crescendo est noté comme un < étiré dont longueur désigne la durée de validité du crescendo. Respectivement, decrescendo, >.

**Degré** Dans une gamme fixée, les notes sont parfois désignée comme des degrés, parfois aussi par des noms : en sol majeur par exemple, le sol est la tonique ou le premier degré, le la est la sus-tonique ou le second degré, puis les noms sont médiate, sous-dominante, dominante, sus-dominante, sensible.

**Dodécaphonisme** Principe de composition dû à Arnold Schönberg dans les années 20. Il s'agit de fonder la composition sur une série dans laquelle chaque note de la gamme chromatique apparaît exactement une fois, une note ne pouvant pas réapparaître avant que les 11 autres aient été jouées.

**Echelle** Une échelle est une gamme généralisée. L'échelle constitue un ensemble de notes autorisées dans la partition. On peut utiliser une échelle à partir de n'importe quelle tonique, et donc une échelle est un ensemble d'intervalles.

**Fondamentale** La fondamentale est la note la plus basse d'un accord.

**Fondamentale virtuelle** La fondamentale virtuelle est utilisée en musique contemporaine. Pour un accord quelconque, c'est la note la plus haute telle que les notes effectivement présentes dans l'accord soient des harmoniques de cette note. Elle n'apparaît pas dans l'accord.

**Frêt** Les instruments à corde permettent de jouer toutes les notes possibles, y compris hors tempérament, car le doigt peut être posé n'importe où sur le manche. Exception, la guitare limite les notes jouées grâce à des petites barres collées sur le manche, sous la corde, ce qui crée une grille composée de petites cases où l'instrumentiste place les doigts. Ces barres s'appellent les frêts. Ils sont placés de telle sorte que chaque case corresponde à une note de la gamme chromatique.

| Nom                | Demis-tons | Notation |
|--------------------|------------|----------|
| Unisson            | 0          |          |
| Seconde mineure    | 1          | 2m       |
| Seconde majeure    | 2          | 2M       |
| Tierce mineure     | 3          | 3m       |
| Tierce majeure     | 4          | 3M       |
| Quarte             | 5          | 4        |
| Quarte augmentée   | 6          | 4+       |
| Quinte diminuée    | 6          | 5 barré  |
| Quinte (juste)     | 7          | 5        |
| Quinte augmentée   | 8          | 5+       |
| Sixte mineure      | 8          | 6m       |
| Sixte majeure      | 9          | 6        |
| Septième           | 10         | 7        |
| Septième augmentée | 11         | 7+       |

FIG. 28.1 – Notations des intervalles

**Gamme** Suite de notes à intervalles déterminés, dont la somme vaut un octave. La gamme chromatique par exemple est une suite de demi-tons. En musique classique occidentale, il existe trois gammes diatoniques : majeure, mineure mélodique, mineure harmonique qui se divise elle-même en deux, ascendante et descendante. Une gamme peut être utilisée sur n'importe quelle tonique : ainsi la gamme de do majeur est do ré mi fa sol la si do, ré majeur ré mi fa $\sharp$ sol la si do $\sharp$ , etc. Une gamme majeure a pour intervalles (en nombre de tons) 1 1 1/2 1 1 1 1/2. Une gamme mineure mélodique a pour intervalles 1 1/2 1 1 1/2 3/2 1/2. Enfin, la gamme par ton interdit l'emploi du demi-ton, ce qui donne sur do : do ré mi fa $\sharp$ sol $\sharp$ la $\sharp$ .

**Harmonie** Art d'écrire la musique sous la forme d'une succession d'accords (au contraire du contrepoint). C'est une vision d'abord verticale de la musique. Par exemple, les exercices de basse chiffrée relèvent de l'harmonie. De nombreux traités fixent des règles à respecter pour écrire une harmonie correcte. Les règles sont nombreuses, parfois complexes, très diverses. En effet, elles peuvent porter aussi bien sur la structure des accords que sur le mouvement des voix, le mouvement des intervalles, des notes particulières, etc. Par exemple : on n'a pas le droit d'écrire deux quintes ou deux octaves qui se suivent, la sensible doit se résoudre sur la tonique, la basse doit être en mouvement contraire de la soprano, etc.

**Hauteur** La hauteur d'une note est sa position verticale dans la partition. On désignera presque toujours les hauteurs par leurs valeurs MIDI, où le do4 est codé par 60 avec un décalage de 1 par demi-ton, ou en midicents, qui sont les hauteurs MIDI multipliées par 100, ce qui permet de représenter les micro-intervalles. Les hauteurs en OpenMusic sont codées en midicents.

**Intervalle** On note souvent les intervalles avec des chiffres comme dans le tableau 28.1

**Mesure** Division de la partition en sous-ensembles de durée égale. Les mesures sont séparées par des barres de mesure. La taille d'une mesure est donnée par la signature.

**Micro-intervalle** Intervalle de moins d'un demi-ton utilisé dans les musiques autres qu'occidentale classique. Le plus couramment utilisé est le quart de ton, codé par 50 en midicents.

**MIDI** (Musical Instrument Digital Interface) MIDI est un format de communication pour la musique entre ordinateurs, synthétiseurs, et instruments électroniques. MIDI spécifie 16 canaux différents, ce qui signifie qu'avec un câble on peut contrôler jusqu'à 16 instruments différents sur un synthétiseur par exemple. Pour chaque canal, le son n'est représenté qu'à travers un minimum d'informations, comme par exemple note-on et note-off. Ces instructions sont envoyées pour un canal donné, le 0 par exemple. note-on est alors ignorée par les canaux 1 à 15, et pousse l'instrument (quel qu'il soit : synthétiseur, carte-son) choisi dans le canal 0 à jouer une note. C'est ce qui se passe quand on appuie sur une touche d'un synthétiseur. D'autres instructions contrôlent le volume, la vélocité, etc. Une bonne analogie pour une partition MIDI est celle du rouleau de papier envoyé à un piano mécanique, où sont notées les dates de début et de fin des notes et leur hauteur, mais sans informations sur la nature du son.

**Modal** La musique modale respecte les mêmes modes que la musique tonale, en remplaçant la gamme tonale par une échelle. Le compositeur Olivier Messiaen utilise notamment des modes dits "à transposition limitée", qui ont pour propriété de pouvoir être transposés en eux-mêmes. Par exemple, la gamme par ton, do ré mi fa#sol#la#, dont tous les intervalles successifs sont de un ton.

**Mute** Muet (fréquemment utilisé pour les signaux audio)

**Onset** Le terme onset fait référence au rythme. Il désigne un rythme joué, donné par sa position temporelle.

**Ostinato** Procédé d'écriture dans lequel une voix répète un même motif, en accompagnement d'autres voix qui elles varient.

**Quatre voix** La musique classique est souvent composée de quatre voix, différenciées bien sûr par leur rôles, et aussi notamment par leurs positions et leurs tessitures. La tessiture d'une voix est l'intervalle des notes dans laquelle elle joue.

soprano Voix la plus aigüe. Sa tessiture s'étend de ré4 à do6, en valeurs MIDI de 61 à 84

alto Deuxième voix la plus aigüe. Sa tessiture s'étend de do4 à la5, en valeurs MIDI de 60 à 81

tenor Deuxième voix la plus grave. Sa tessiture s'étend de sol3 à ré5, en valeurs MIDI de 55 à 74

basse Voix la plus grave. Sa tessiture s'étend de fa2 à ré4, soit en hauteurs MIDI de 41 à 62.

**Position** Pour les instruments à corde, la position désigne la hauteur de la main sur le manche de l'instrument. Les positions sont notées à partir de 0 (corde à vide, main en bas du manche). Suit la première position pour le premier doigt qui joue la première note de la corde, etc. Pour la guitare, les positions sont notées sans ambiguïtés en nombre de frêt (pour d'autres instruments à corde, en notes de la gamme).

**Pulsation** Découpage du temps en valeurs entières sur lesquelles sont définis des rythmes symboliques.

**Renversement** En musique classique, un accord est donné soit sous sa forme fondamentale, soit renversé, c'est à dire qu'on fait passer la note la plus basse une octave au dessus. Si un renversement ne modifie pas vraiment les notes présentes dans l'accord, il en modifie les intervalles entendus donc son chiffrage. Par exemple, les renversements de l'accord de quinte sont 5, 6,  $\frac{6}{4}$ .

**Rythme** On a en durée :  $\circ = 2 * \text{♩} = 4 * \text{♪} = 8 * \text{♫}$ , qui se lisent dans l'ordre ronde, blanche, noire, croche. Lorsque l'une de ces unités est divisée en un nombre  $n$  qui n'est pas une puissance de deux, on le note par une liaison au dessus du temps concerné. Certaines de ces divisions portent un nom. Par exemple pour  $n = 3$ , on aura  $\overset{3}{\text{♩}}$  division en trois parties de la blanche, qui se lit triolet de noires.

**Signature** La signature d'une partition est l'indication, en tête de partition, de deux chiffres notés verticalement. Le chiffre du bas donne l'unité de temps comme division de la ronde : 2 pour la blanche, 4 pour la noire, 8 pour la croche. Le signe du haut donne le nombre d'unités de temps par mesure. Une signature à  $\frac{4}{4}$  désigne une mesure contenant quatre fois quatre noires.

**Scordatura** Pour les instruments à cordes, en particulier la guitare, la scordature désigne une modification de la manière habituelle d'accorder l'instrument. Une guitare est accordée le plus souvent sur les notes mi la ré sol si mi (du plus grave au plus aigu), c'est l'accord que l'on entend en jouant toutes les cordes à vide. Mais certains morceaux utilisent une scordature différente qui est alors indiquée en début de morceau.

**Spatialisation** La spatialisation consiste à déplacer artificiellement une source sonore dans l'espace, voir [32]. Le signal audio envoyé *via* des baffles ou des écouteurs à un auditeur est modifié de manière à contrôler la position des sources sonores. Ce contrôle est basé sur un déphasage du signal. Plusieurs systèmes existent parmi lesquels le SPAT développés à l'IRCAM [59].

**Tempérament** Il serait vain de donner en quelques mots une leçon de tempérament, qui désigne l'art de diviser l'octave en parties égales. On se limitera ici à quelques précisions. Dans la musique classique, l'octave est divisée en 12 demi-tons, le demi-ton étant le plus petit intervalles autorisé. Ce principe est parfois respecté dans la musique contemporaine. Mais très souvent, la notion de plus petit intervalles autorisé n'a plus de sens, en particulier pour la musique spectrale, où l'on s'autorise jusqu'à désigner une note par sa fréquence. Une des options les plus fréquentes est le tempérament en quarts de ton, où l'octave est divisée en 24.

**Tempo** Le tempo est la vitesse à laquelle est jouée la partition. Il est parfois donné sous forme d'indications (Largo, Andante, etc), ou sous forme de chiffres. Dans ce cas, le chiffre désigne le

nombre de pulsations par minute. Ainsi un tempo de 60 signifie une pulsation par seconde, et si la signature de la partition est la noire, on note  $\text{♩} = 60$ . Un changement de tempo se note soit directement sous la forme ci-dessus, soit par un rapport, plus simple à réaliser pour le musicien. Ainsi  $\text{♩} = \text{♪}$  signifie que la nouvelle noire vaut l'ancienne croche, soit un nouveau tempo de 120 dans notre exemple.

**Tonalité** En musique classique, la tonalité est donnée par une note, et un mode : majeur (noté M) ou mineur (noté m). Elle désigne l'ensemble des notes autorisées dans la partition. Ainsi, un morceau écrit en sol majeur pourra comporter seulement les notes de la gamme de sol M : sol, la, si, do, ré, mi, fa $\sharp$ . On appelle musique tonale la musique qui respecte ce principe de ne pas faire jouer de notes en dehors de la tonalité (ou alors rarement, ces notes étant qualifiées d'accidents).

**Transposition** A ne pas confondre avec une transposition mathématique, une transposition en musique est un décalage vertical. Par exemple, do transposé à la quinte donne sol.

# Contraintes, recherche locale et composition assistée par ordinateur

La Programmation par Contraintes (CP) permet de modéliser et de résoudre des problèmes fortement combinatoires par la spécification d'une certaine information partielle sur des variables, inconnues du problème. Dans cette thèse, nous avons étudié des problèmes de contraintes musicaux, posés par des compositeurs contemporains, d'analyse musicale et d'instrumentation, en collaboration avec l'IRCAM. Quatorze problèmes ont été modélisés et résolus au cas par cas, ce qui a permis d'en dresser une typologie détaillée. Ceci a servi à la conception et l'implémentation d'un algorithme de recherche locale dans l'environnement de Composition Assistée par Ordinateur OpenMusic, sous la forme d'une librairie dont l'architecture permet notamment de définir visuellement un problème de contraintes, de le résoudre, et le cas échéant d'éditer des résultats partiels ou approchés pendant le processus de résolution.