Recherche adaptative et contraintes musicales

Charlotte Truchet — Carlos Agon — Gérard Assayag

IRCAM
1 place Igor Stravinsky
75 004 Paris

{truchet,agonc,assayag}@ircam.fr

RÉSUMÉ. *Cet article présente un environnement de résolution de contraintes (CSP) musicales, basé sur le langage visuel OpenMusic. Nous présentons notamment une implémentation d'un algorithme de recherche locale, appelé recherche adaptative, dans le domaine musical. La recherche adaptative reprend le principe de résolution par optimisation d'une fonction de coût, mais en affinant la notion de coût de manière à tenir compte du poids de chaque variable. Nous montrons qu'un algorithme incomplet est particulièrement bien adapté dans le cas des CSP musicaux. Les premiers résultats expérimentaux sur des problèmes musicaux réels sont satisfaisants, notamment en temps de calcul.*

ABSTRACT. *We propose an environment for musical constraint solving, in the visual programming language OpenMusic. We describe an implementation of a local search algorithm, called adaptive search, in the musical field. Adaptive search refines the concept of cost function by taking into account the weight of each variable. We show that an incomplete algorithm is well adapted for musical problems. The first experimental results on real musical problems are satisfying, including in terms of computation time.*

MOTS-CLÉS : *programmation par contraintes, CSP, recherche locale, recherche adaptative, contraintes musicales, composition assistée par ordinateur, séries tous intervalles*

KEYWORDS: *constraint programming, CSP, local search, adaptive search, musical constraints, computer assisted composition, all-interval series*

1. Introduction

Les méthodes incomplètes de résolution de contraintes ont montré leur efficacité pour résoudre des problèmes fortement combinatoires tels que le voyageur de commerce [WAL 99]. Parmi elles, la recherche locale fonctionne par améliorations successives à partir d'un état initial. Cela suppose d'avoir une manière d'évaluer la qualité de la configuration courante. Ces algorithmes peuvent grossièrement se résumer en : initialisation aléatoire, exploration du voisinage de la configuration courante et sélection du meilleur candidat.

Nous présentons une adaptation d'un de ces algorithmes, appelé recherche adaptative [Cod00], dans le domaine de l'informatique musicale. C'est un domaine où la programmation par contraintes a une place assez naturelle. Le cas de l'harmonie classique, déjà plusieurs fois traité, en donne un bon exemple. Les traités d'harmonisation donnent un ensemble de règles à respecter : mouvements contraires entre deux voix, pas de quintes parallèles, etc. Ils sont en quelque sorte entièrement déclaratifs. En musique contemporaine, domaine où l'informatique est déjà très utilisée, la notion de règle musicale reste, et quelques compositeurs utilisent déjà la PLC.

Notre but est de construire un système de programmation par contraintes à destination des compositeurs contemporains. Nous utilisons une méthode incomplète, la recherche adaptative. C'est un algorithme de recherche locale, de la famille de GSAT [SLM92].

2. Motivations

2.1. *Composition assistée par ordinateur (CAO)*

La composition assistée par ordinateur 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. L'un des exemples de logiciels de cette CAO moderne est OpenMusic (OM), langage visuel de programmation objet, développé par Gérard Assayag et Carlos Agon à l'IRCAM.

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 effectués, essentiellement dans le domaine de l'harmonisation automatique qui est un problème fortement combinatoire.

Le précurseur dans ce domaine est Kemal Ebcioglu qui propose un système de composition de chorals dans le style de Bach [EBC 97]. Musicalement, le problème traité est celui de l'harmonisation à quatre voix : à partir d'un chant donné, il faut produire des accords respectant les règles de l'harmonie. Le même problème est traité par Tsang [TsA91], mais de manière encore assez lente. Philippe Ballesta a réalisé un système basé sur Ilog-Solver. Lui aussi traite du problème de l'harmonisation à quatre voix [BAL 98]. De même, François Pachet et Pierre Roy (Sony CSL) ont traité le cas de l'harmonisation classique [PAR95]. Implémenté en Backtalk, leur solver comporte une application musicale pour l'harmonisation automatique, en utilisant les structures musicales propres à la musique tonale.

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. Lorsqu'ils proposent un framework plus général, il est lié à la musique tonale, très contrainte en matière de structure harmonique. Enfin, l'IRCAM propose déjà dans OM deux solvers de contraintes, Situation et PWConstraints, dont nous parlerons plus en détail ci-dessous. Ces deux solvers traitent de problèmes musicaux plus larges que la seule musique tonale.

Notre but est de construire un système de programmation par contraintes à l'usage des compositeurs contemporains, dans le logiciel OM. Cela suppose d'abord de trouver un système général de spécification de contraintes musicales, adapté à l'interface visuelle intuitive d'OM. Comme le système est destiné à la composition contemporaine, il ne doit pas être marqué stylistiquement : bien sûr, on ne se limitera pas à la musique tonale, mais on ne doit pas non plus favoriser certaines catégories musicales (mélodie, harmonie, rythme, contrepoint, etc).

2.2. *Expressivité*

Nous avons commencé par consulter plusieurs compositeurs liés à l'IRCAM, qui avaient parfois déjà utilisé OM pour résoudre un problème de contraintes (voire déjà résolu un problème de contraintes empiriquement à la main !). Il en est sorti deux constats : les problèmes posés ne sont pas triviaux, et ils sont assez variés, à la fois dans la structure des objets utilisés et dans les contraintes. Les objets à contraindre vont d'une suite de tempi (suite finie d'entiers), à une suite d'accords (suite finie de suites finies d'entiers), en passant par des motifs rythmiques ou mélodiques, des structures harmoniques, etc. Les contraintes sont parfois simplement arithmétiques, plus souvent elles comportent des alldiff, des $\exists$, $\forall$, $\in$. Les domaines sont toujours finis.

L'une des difficultés est donc de fournir une bibliothèque de contraintes raisonnablement expressive. On a en effet le choix entre deux extrêmes : écrire une librairie adaptée à un problème bien précis (contraintes harmoniques par exemple), facile d'utilisation. Mais alors elle risque de ne servir qu'à un compositeur. On peut aussi donner un solver qui accepte n'importe quelle contrainte, mais d'une part l'utilisateur devra faire une effort important pour écrire ses contraintes, d'autre part on risque de perdre en efficacité : on ne peut pas demander au compositeur un trop grand travail d'optimisation dans l'expression des contraintes (n'oublions pas qu'un compositeur n'a aucune raison de faire la différence entre poser une contrainte "strictement croissante" sur une suite, et poser une contrainte "alldiff", puis une contrainte "croissante", par exemple).

Pour éviter le premier écueil, nous avons commencé par la deuxième approche, de manière à avoir un langage de contraintes suffisamment expressif. Ainsi, nous pourrons ensuite proposer une bibliothèque de primitives plus élaborées.

2.3. *Progressivité*

En faisant appel à un solver de contraintes, le compositeur a bien sûr des attentes assez proches de celles de n'importe quel utilisateur (il veut une solution), mais il l'utilisera probablement un peu différemment. Nous avons constaté que le principe "attendre longtemps pour obtenir une solution exacte" n'était probablement pas le mieux adapté. En effet, il est probable qu'un compositeur, à qui on fournit une solution exacte, la retravaillera en fonctions de critères esthétiques (non formalisables). Par ailleurs, les solutions atypiques ou surprenantes ne sont pas à négliger, même si elles ne sont qu'approchées. Quant à l'attente, comme dans tout problème informatique, elle n'est pas souhaitable. Si on ne peut pas la réduire, on peut éventuellement la meubler, par exemple en affichant des solutions partielles, qui, si elles sont assez pertinentes, seront d'un réel intérêt pour l'utilisateur.

Ce sont là les raisons pour lesquelles nous nous sommes tournés vers un algorithme de recherche locale par améliorations successives du résultat. Cela permet d'abord de chercher une solution approchée. Par ailleurs, afficher les solutions intermédiaires fait sens. En effet, contrairement par exemple à un backtracking classique, les instanciations sont dirigées vers l'amélioration d'une fonction de coût. Chaque solution est meilleure que la précédente, jusqu'aux minima locaux. Le compositeur peut donc, s'il le souhaite, arrêter l'évaluation lorsqu'il considère "être assez près".

3. OpenMusic

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 décrirons ici que quelques aspects du langage, voir [AGO 98] et [ASS 99] 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. Sur le workspace sont notamment sauvés les patchs, l'un des objets centraux d'OM. Dans sa forme développée le patch constitue pour le langage visuel une expression correspondant à la notion commune de corps de programme, un exemple en est donné figure 1. Un programme en OM s'exprime comme un agencement d'icônes interconnectées représentant des fonctions, des méthodes, ou des objets. 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.

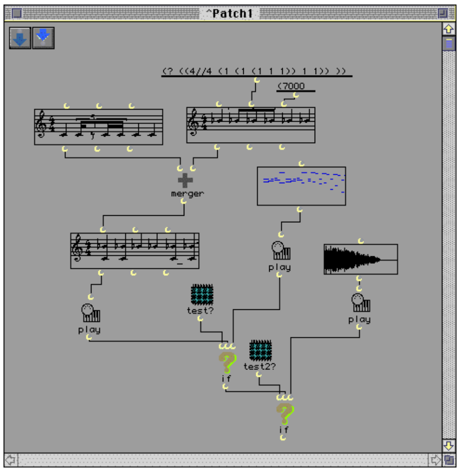

FIG. 1. *Exemple de patch OpenMusic.*

Le workspace contient également les packages, des dossiers spéciaux archivant les classes et fonctions génériques. 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 2 montre la hiérarchie des classes musicales. Un package *user* stocke les classes créées par l'utilisateur (qui peuvent hériter des autres classes OM). 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.

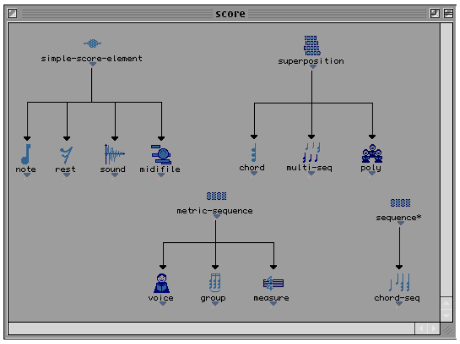

FIG. 2. *La hiérarchie des classes musicales dans OpenMusic.*

3.1. *Situation*

Situation a été conçu par Camilo Rueda en collaboration avec le compositeur Antoine Bonnet [RUV97]. C'est un moteur de contraintes par first-found forward checking, muni d'une interface graphique en OM. Le problème musical est représenté par des objets, eux mêmes constitués d'un certain nombre de points et de distances. Cela permet d'exprimer à la fois des rythmes et 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.

3.2. *PWConstraints*

PWConstraints (PWCS) a été conçu par Mikaël Laurson [Lau96]. 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.

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.

3.3. *Screamer*

L'un des premiers travaux a été d'importer dans OM un solver écrit en Common Lisp, Screamer [SMA93]. Screamer ajoute à Common Lisp une forme d'indéterminisme via deux constructions, $either$ et $fail$, qui introduisent le point de choix dans le langage. $(either\, e_1 ... e_n)$ évalue d'abord e_1, puis si l'évaluation de e_1 donne un $fail$, e_2, etc. Nous utilisons Screamer surtout pour cette possibilité, même s'il propose en outre un solver avec propagation. OM possède maintenant une librairie Screamer, qui introduit toutes les fonctions permettant le backtracking et l'appel au solver. On a ajouté au langage visuel une nouvelle classe de patchs pour les fonctions indéterministes. Pour ces patchs, la génération du code fait directement appel aux primitives Screamer.

Le premier intérêt de Screamer dans OM est qu'il permet de traiter les cas où l'on veut toutes les solutions. Bien que ce ne soit pas la requête la plus fréquente, il arrive qu'un utilisateur veuille générer l'ensemble d'un matériau musical respectant certaines contraintes, cas dans lequel un algorithme de recherche locale est inutilisable. En outre, nous avons pu tester ainsi l'écriture de contraintes dans le langage visuel. Contrairement à Situation et PWCS, l'accès aux primitives du solver est complètement visuel.

4. Recherche adaptative

4.1. *Description*

L'algorithme de recherche adaptative est proposé par Philippe Codognet (LIP6) [Cod00], qui l'a testé sur des problèmes classiques (N-reines et carrés magiques). Le problème est donné sous la forme d'un CSP, avec variables, domaines finis associés, et contraintes sur les variables. Il existe plusieurs solvers de contraintes dans le cas des domaines finis, comme ILOG Solver [Pug94], ou GNU-Prolog [CDi00]. La recherche adaptative fait partie des algorithmes de recherche locale, tel que GSAT [SLM92], qui tirent profit de la représentation du problème en CSP, même s'ils se distinguent des techniques classiques de résolution. De tels algorithmes ont largement prouvé leur efficacité sur des problèmes comme celui du voyageur de commerce, des N-reines, etc.

La représentation d'un problème est celle du format CSP, avec $V_1...V_n$ les variables, $Dom_1...Dom_n$, les domaines finis associés, et $C_1...C_p$ les contraintes portant sur les variables. Les contraintes sont écrites sous la forme d'une relation logique entre les variables. On notera V_i la variable et v_i une valeur de V_i.

Le principe en recherche locale est de guider la recherche de solution par une mesure de la qualité d'une configuration. On peut résumer grossièrement ce type d'algorithme par : intialisation aléatoire, puis itérativement exploration d'un voisinage, recherche d'une meilleure configuration, remplacement. Cela suppose d'avoir une mesure de la qualité de la configuration courante, ce qui est fait en représentant les contraintes par une fonction de coût, qui sert à la recherche d'une meilleure configuration.

L'algorithme de recherche adaptative fonctionne sur ce principe, mais en affinant la notion de coût. Il s'agit de tirer le maximum d'information à partir des contraintes, au niveau de chaque variable V_i et non plus de la configuration $V_1...V_n$. Cela permet de sélectionner à chaque pas la variable la plus mauvaise. Nous remplaçons l'étape "exploration du voisinage" par le calcul des coûts de chaque variable, la sélection de la plus chère, et l'exploration du domaine de cette variable pour trouver une meilleure valeur.

Les différentes fonctions utilisées sont représentées ci-dessous :

$$f_{v_i,C_j}(v_i\ ,\ (v_1...v_n)) \overset{somme\ en\ C_j}{\longrightarrow} f_{v_i}(v_i\ ,\ (v_1...v_n)) \overset{somme\ en\ v_i}{\longrightarrow} f_{total}((v_1...v_n))$$

$f_{v_i,C_j}(v_i\ ,\ (v_1...v_n))$ n'est pas directement utilisée dans l'algorithme, elle sert à la définition d'une grammaire de contraintes (voir ci-dessous). $f_{v_i}(v_i\ ,\ (v_1...v_n))$ représente le poids d'une variable dans la configuration courante. Avec par exemple trois variables V_1, V_2 et V_3, et deux contraintes $V_1 = V_2$ et $V_2 = V_3$, on peut choisir $|V_1 - V_2|$ pour V_1, $|V_1 - V_2| + |V_2 - V_3|$ et V_2 et $|V_2 - V_3|$ pour V_3. Cette fonction sert à sélectionner la variable la plus chère $V_{plus-chere}$ à chaque pas. La dernière fonction, $f_{total}((v_1...v_n))$ représente le poids de la configuration courante, dans notre exemple $2 * |V_1 - V_2| + 2 * |V_2 - V_3|$. Elle est utilisée pour déterminer une meilleure valeur sur le domaine de $V_{plus-chere}$.

Dans le cas oùla variable $V_{plus-chere}$ n'a pas de meilleure valeur (aucune substitution de $v_{plus-chere}$ par une autre valeur du domaine ne permet de diminuer le coût global), l'algorithme boucle (minimum local de f_total sur l'axe $V_{plus-chere}$). Pour éviter de boucler, nous utilisons une mémoire adaptative, à la manière du Tabu Search [AAR 97]. Si l'exploration du domaine $Dom_{plus-chere}$ ne donne pas de meilleure valeur, $V_{plus-chere}$ est marquée Tabu et ne pourra être modifiée pendant un nombre fixé d'itérations.

Il se peut aussi que l'on arrive dans un minimum local de f_{total}, sur tous les axes $V_1...V_n$. Dans ce cas, toutes les variables seront successivement marquées Tabu. Dans ce cas, nous choisissons de ré-initialiser aléatoirement toutes les variables comme suggéré dans[GSK00].

4.2. *Algorithme*

1. Initialisation aléatoire

Repeat

2. *Calcul* des coûts de toutes les variables, sauf sauf celles marquées Tabu. Sélection de la plus-chere, $V_{plus-chere}$.

3. *Test* du coût global en remplaçant $V_{plus-chere}$ par toutes les valeurs de son domaine. Sélection de v' la meilleure.

4. *Si* à la fin de l'exploration du domaine, aucune valeur n'améliore le coût global, *alors* $V_{plus-chere}$ est marquée tabu.

5. *Si* toutes les variables sont tabu *alors* réinitialisation aléatoire.

Until l'erreur globale est inférieure à ϵ.

Si le problème n'a pas de solution, l'algorithme ne termine pas. On peut éviter cela en fixant un nombre maximum d'itérations. Nous avons implémenté cet algorithme en Lisp, avec des structures de données qui permettent d'utiliser les résultats directement dans OM (listes et listes de listes).

4.3. *Avantages*

Dans le cas des problèmes musicaux, cet algorithme présente plusieurs avantages. D'abord, il répond à notre objectif de progressivité. Nous avons ajouté dans l'implémentation une variable $affichage$, nulle par défaut. Cette variable fonctionne comme un seuil : dès que l'erreur globale de la configuration est inférieure à $affichage$, les minima locaux et l'erreur courante sont affichés.

Il traite naturellement les solutions approchées. Dans notre implémentation, la condition d'arrêt est que la somme des coût pour toutes les variables soit inférieure à un nombre fixé, soit ϵ. De cette manière, on obtient les solutions exactes en prenant $\epsilon = 0$, et des solutions approchées pour $\epsilon > 0$.

La représentation des contraintes par des coûts apporte une souplesse supplémentaire au programme. En effet, on peut donner plus d'importance à certaines contraintes en pondérant leurs fonctions de coûts, et inversement laisser une tolérance sur d'autres. Ainsi, si l'on part d'un ensemble de contraintes $C_1 ... C_n$ de coûts $f_{C_1} ... f_{C_n}$, et que l'on souhaite une solution exacte sauf pour C_j, on prendra comme fonction de coût totale $f_{total} = M * (\sum_{k \neq j} f_{C_k}) + f_{C_j}$, et l'on arrêtera le calcul dès que le coût est strictement inférieur à M (en fixant $\epsilon = M$).

Enfin, le cas de contraintes globales portant sur un ensemble de variables se traite facilement, ce qui représente un progrès par rapport à Situation et PWCS. Une contrainte globale a pour seule particularité d'avoir une fonction de coût constante sur les V_i. Cela ne gène en rien la résolution.

4.4. *Grammaire*

Il est évidemment hors de question de demander à l'utilisateur de trouver lui-même la fonction de coût associée à chaque contrainte. Nous avons donc écrit une fonction G de traduction, qui passe de la contrainte à sa fonction de coût (ici, $f_{v_i\,C_j}$, dont on déduit les autres). Les contraintes sont écrites en Lisp.

Pour pouvoir générer les coûts nous devons distinguer les termes t des contraintes C. Le langage des contraintes contient les égalités, inégalités, appartenance, *et* et *ou* logiques, $alldiff$, et les quantificateurs existentiels et universels. Plus formellement, la grammaire s' écrit :

$C ::= (t = t) \,|\, (t \leq t) \,|\, (t < t) \,|\, (t \in t) \,|\, (C \wedge C) \,|\, (C \cup C) \,|\, (alldiff t)$ $|(\exists t \in t,\, C(t))|(\forall t \in t,\, C(t))$

$t ::= n \,|\, (f\, t...t)$

avec n un entier et f une fonction de Lisp.

4.5. *Génération des fonctions de coût*

Chaque contrainte a bien sûr une classe de fonctions de coût (pour la relation d'équivalence "avoir les mêmes zéros"). On peut choisir n'importe quel représentant, jusqu'à la fonction caractéristique de $\{V_1...V_n, C(V_1...V_n)\}$, qui n'a pourtant pas grand sens (elle n'exprime pas réellement ce qu'il coûte de s'éloigner d'une bonne instanciation). On cherche donc à avoir des coûts vraiment significatifs, ce qui est facile pour $=$, $>$, $\wedge$, etc. Le seul problème vient évidemment du $alldiff$.

Dans un but d'optimisation non encore implémentée, on choisit de préférence une fonction raisonnablement continue (sur les réels) dans chaque classe, et même affine par morceaux, ce que l'on obtient facilement par construction de G (hors $alldiff$). La génération d'une fonction de coût se fait selon les règles suivantes. Les termes ne sont pas modifiés : $G(t) = t$. Pour une contrainte C :

$G(t_1 = t_2) = |t_1 - t_2|$

$G(t_1 \leq t_2) = max(0,\, t_1 - t_2)$

$G(t_1 < t_2) = max(0, 1 + t_1 - t_2)$

$G(t_1 \in t_2) = min_{t \in t_2}(|t_1 - t)|)$

$G(C_1 \wedge C_2) = max(G(C_1), G(C_2))$

$G(C_1 \vee C_2) = min(G(C_1), G(C_2))$

$G(alldiff(t_1...t_n)) = Card\{t_i = t_j, i < j \leq n\} - 1$

$G(\forall t_1 \in t_2\, C(t_1)) = max_{t_1 \in t_2} G(C(t_1))$

$G(\exists t_1 \in t_2\, C(t_1) = min_{t_1 \in t_2} G(C(t_1))$

4.6. *Problème du alldiff*

Le $alldiff$ est difficile à représenter par une fonction de coût. La méthode grossière qui consiste à choisir pour $f_{alldiff}(V_i)$ la fonction caractéristique de $\{V_j = V_i \; j \neq i\}$ peut être un peu améliorée en prenant pour $f_{alldiff}(V_i)$ le nombre de j tels que $V_i = V_j$, mais ce n'est pas très satisfaisant. Evidemment, n'importe quelle fonction de coût pour un $alldiff$ aura cette forme.

Dans le cas où les domaines des $V_1 ... V_n$ (variables sur lesquelles on pose le $alldiff$) sont égaux, et de cardinal t, une solution est de changer le domaine. Cette technique est utilisée par P. Codognet dans [Cod00], pour le problème des carrés magiques. Au lieu de modifier les V_i n'importe comment dans Dom_i, puis de calculer le $alldiff$, on travaille sur les transpositions qui échangent V_i avec une autre variable. De cette manière, si à l'initialisation le $alldiff$ est respecté, il le sera à chaque pas également et on l'obtient gratuitement en temps. Cela vaut par exemple pour les problèmes portant sur des séries (au sens musical : permutation des douze notes de la gamme). C'est ce principe que nous avons utilisé pour le problème des all-intervals series.

Nous envisageons d'étendre cette idée au cas où l'on a $alldiff$ sur m variables, sur le même domaine de taille t (avec évidemment $m \leq t$). L'idée est de retirer du domaine les valeurs instanciées. Cela peut se faire facilement sans modifier le solver, par exemple en ajoutant $t - m$ variables fantômes, dont les coûts seront toujours nuls, qui ne pourront donc être choisies pour une modification.

5. Expériences

Nous avons effectué une série d'expériences dans OM pour comparer le temps de réponse du système de backtracking de Screamer et d'un prototype de solver par recherche adaptative. L'ordinateur est un Mac G4. Le temps de calcul donné ici ne comprend pas le garbage collecting.

Pour la recherche adaptative, le nombre d'itérations désigne le nombre d'appels à la fonction principale. Nous donnons ici les résultats moyens pour dix résolutions. Signalons que l'écart-type est assez élevé. Pour le backtracking, le nombre d'itérations désigne le nombre de backtracks.

Nous avons pris comme tests trois problèmes musicaux. Le premier est un problème musical classique. Le deuxième a été posé par Fabien Lévy, compositeur. Le troisième vient de Mauro Lanza, compositeur. Ces trois problèmes sont plus à prendre comme un panorama de ce que l'on doit pouvoir traiter que comme des buts ultimes.

5.1. *All-intervals series*

Il s'agit de trouver une permutation σ des n premiers entiers, telle que les $|\sigma_{i+1} - \sigma_i|$ soient une permutation des $n - 1$ premiers entiers. C'est un problème musical classique, décrit notamment dans [MOR 74]. Avec $n = 12$, on obtient une série au sens musical (motif dans lequel toutes les notes de la gamme chromatique apparaissent exactement une fois). La contrainte sur les différences successives impose que chaque intervalle soit aussi entendu exactement une fois 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. Le problème a une solution triviale avec $1\; n\; 2\; (n-1)\; 3$ etc. On cherche bien sûr les autres solutions. Les all-intervals series ont été utilisé comme test pour Ant-P-Solver, de C. Solnon [Sol00], dont nous indiquons les résultats dans ce tableau. Pour la recherche adaptative, on utilise le $alldiff$ par transpositions comme défini ci-dessus.

Nombre d'entiers	Backtracking	Adaptative	Ant-P-Solver
8	6 min	<0.01 s	
10	>1 h	0.02 s	0.0 s
12	>1 h	0.1 s	0.1 s
14	>1 h	0.2 s	0.5 s
16	>1 h	2 s	2 s
18	>1 h	5 s	3.7 s
20	>1 h	18 s	10.4 s

Ces résultats peuvent aussi être comparés à ceux d'Ilog Solver [Sol00]. Pour 20 entiers, Ilog Solver met plus d'une heure.

5.2. *Suite d'accords avec note commune*

On considère une suite d'accords (chaque accord étant une suite d'un même nombre de notes, représentées uniquement par leur hauteur donc par un entier). On souhaite avoir une ou plusieurs notes communes entre deux accords successifs. Les accords ont en plus une structure particulière : dans un même accord toutes les notes doivent être équidistantes en fréquence. On utilisera donc évidemment une représentation en fréquence.

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 :

$$\forall i \leq I,\, \forall j_1, j_2 \leq J-1,\, \alpha_i^{j_1+1} - \alpha_i^{j_1} = \alpha_i^{j_2+1} - \alpha_i^{j_2}$$

$$\forall i \leq I-1\; \alpha_i \cap \alpha_{i+1} \neq \emptyset$$

La première contrainte se réduit évidemment 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 int_i entre deux notes, et J le nombre de notes par accord (avec dans ce cas $\alpha_i^j = f_i + int_i * j$ pour $j \leq J$).

Les tests ont été faits pour des suites de 20, 30 et 40 accords, avec à chaque fois des accords 4 notes et de 8 notes. Les contraintes sont d'une part que tous les accords soient différents, d'autre part qu'il y ait une et une seule note commune entre deux accords successifs. Les variables ont un domaine de taille 100.

		Backtracking		Adaptative	
Accords	Notes	Itérations	Temps	Itérations	Temps
20	8	218	4 s	46	1,8 s
20	4	355	1,8 s	42	1,3 s
30	8	460	17 s	63	1,9 s
30	4	569	6 s	53	2 s
40	8	780	52 s	56	1,7 s
40	4	948	17 s	53	2 s

5.3. *Rythmes sans simultanéité*

Il s'agit de composer un quasi-canon rythmique. On a n voix jouant simultanément. La i-ième voix répète un motif rythmique de période T_i, formé d'un ensemble d'onsets $o^i_j \leq T_i$ (on appelle onset le temps, plus exactement la date, où un rythme est frappé). L'unité de temps est donnée, de sorte que les onsets sont représentés par des entiers. Il s'agit de trouver les motifs pour que jamais deux voix n'aient deux onsets simultanés, et ce pendant une durée D fixée, $D \leq ppcm(T_1...T_n)$. Les T_i sont en général choisis premiers entre eux, pour avoir des séquences suffisamment longues. De ce fait, le nombre de variables est généralement élevé. La figure 3 montre une solution approchée, avec des T_i de 8, 14 et 12 unités de temps, ici la double croche. La figure 4 est une solution exacte.

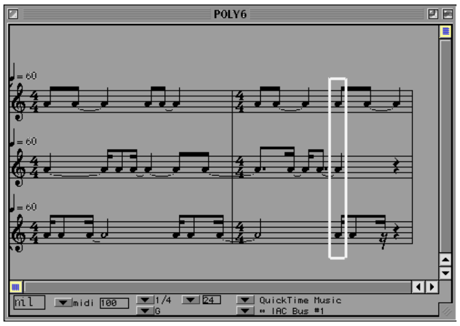

FIG. 3. *Une solution approchée. L'erreur est encadrée en gris, les voix inférieures et supérieures frappant un onset sur ce temps.*

o^i_j représente le j-ième onset de la i-ième voix, et on note n_i le nombre d'onset dans le i-ième motif. Les contraintes sont alors décrites par les formules suivantes :

$$\forall i \leq n \; o^i_{n_i} = L_i$$

$$\forall i_1,\, i_2 \leq n \, \forall j \neq k \leq D \; o^{i_1}_j \neq o^{i_2}_k$$

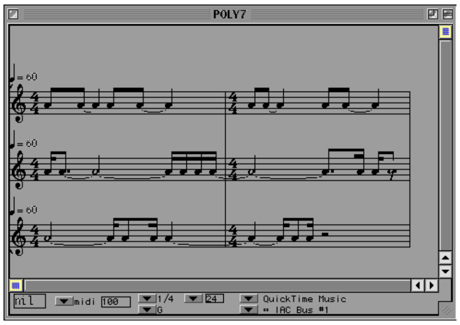

FIG. 4. *Une solution exacte du même problème.*

En réalité, le compositeur souhaite éventuellement avoir 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 (de manière évidente, dès que les nombres d'onsets deviennent trop grands). Il accepte donc des solutions approchées.

Pour la résolution, les paramètres à choisir par l'utilisateur sont le nombre de voix, la durée D, et la densité d'onsets, qui représente le rapport entre le nombre d'onsets joués sur la longueur totale jouée (par exemple, pour une densité de deux, on entendra en moyenne un onset toutes les deux unités de temps). Les tests ont été effectués avec une densité de 2. La durée choisie est 128 pulsations. Les longueurs des voix sont dans l'ordre 19, 23, 29, 31, 37 et 43.

	Backtracking	Adaptative	
Nombre de voix	Temps	Itérations	Temps
3	>1h	350	12 s
4	>1h	463	22 s
5	>1h	923	58 s
6	>1h	1208	108 s

6. Conclusion

Les premiers résultats de recherche adaptative en musique sont encourageants. Bien sûr, le temps de calcul est raisonnable. Mais les particularités de la recherche adaptative (incomplétude, progressivité) en font un algorithme particulièrement bien adapté aux problèmes musicaux.

Il reste cependant à améliorer l'algorithme et son implémentation. Nous envisageons notamment de tester le programme avec d'autres stratégies que la réinitialisation aléatoire en cas de minium local, d'implémenter un $alldiff$ plus efficace comme décrit ci-dessus, d'améliorer la partie tabu de l'algorithme et enfin d'utiliser le caractère affine des fonctions de coût pour optimiser la recherche du meilleur candidat.

Viendra ensuite le moment d'écrire des primitives plus élaborées pour le langage de contraintes, et de réfléchir à une intégration visuelle satisfaisante de la recherche adaptative dans OM.

7. Bibliographie

[AAR 97] AARTS E. H. L., LENSTRA J. K., *Local search in combinatorial optimization*, John Wiley and Sons, 1997.

[AGO 98] AGON A., An environment for computer assisted composition , Thèse de doctorat, IRCAM-UniversitÈ de Paris VI, 1998.

[ASS 99] ASSAYAG G., ANS MIKAEL LAURSON C. R., AGON C., DELERUE O., Computer Assisted Composition at Ircam : PatchWork & OpenMusic , *Computer Music Journal*, , 1999.

[BAL 98] BALLESTA P., *Contraintes et objets, clefs de voûte d'un outil d'aide à la composition*, Editions Hermès, 1998.

[CDi00] *The implementation of GNU Prolog*, Como, Italy, 2000, ACM Press.

[Cod00] *Adaptive Search, Preliminary Results*, Venise, 2000.

[EBC 97] EBCIOGLU K., *An expert System for Harmonizing Chorales in the style of J.-C. Bach*, AAAI Press, 1997.

[GSK00] *Boosting Combinatorial Search Through Randomization*, Madison, 2000.

[Lau96] *Patchwork : a visual programming language and some Musical applications*, Helsinky, 1996, Sibelius Academy.

[MOR 74] MORRIS R., STARR D., The Structure of the All-Interval series , *Journal of Music Theory*, vol. 13, n° 2, 1974.

[PAR95] *Integrating Constraint Satisfaction Techniques with Complex Object Structures*, Cambridge, Décembre 1995.

[Pug94] *A C++ Implementation of CLP*, Singapore, 1994.

[RUV97] *Improving Forward Checking with delayed evaluation*, Santiago, Chile, 1997.

[SLM92] *A new method for solving Hard Satisfiability Problems*, San Jose, 1992.

[SMA93] *Nondeterministic Lisp as a Substrate for Constraint Logic Programming*, 1993.

[Sol00] *Ant-P-Solver : un solver de contraintes à base de fourmis artificielles*, Editions Hermès, 2000.

[TsA91] *Harmonizing Music as a Discipline of Constraint Logic Programming*, 1991.

[WAL 99] WALSER J. P., *Integer Optimization by Local Search : a Domain-independant Approach*, Springer Verlag, 1999.