CAO et contraintes

Ch. Truchet * C. Agon * G. Assayag * P. Codognet †

March 12, 2001

Mots Clefs Programmation par contraintes, OpenMusic, CAO, backtracking, Screamer, recherche locale, recherche adaptative.

Résumé Nous présentons deux méthodes complémentaires pour la programmation par contraintes en CAO. La première méthode est complète. C'est un solver par backtracking et propagation, qui a été importé dans le logiciel OpenMusic. La seconde, une méthode incomplète, est un algorithme de recherche locale dont nous montrons qu'il est particulièrement bien adaptée à la CAO. Les tests déjà réalisés sont très satisfaisants.

1 Introduction

Nous présentons deux méthodes de résolution de contraintes adaptées à la Composition Assistée par Ordinateur. L'une est complète et assez classique, l'autre, appelée recherche adaptative [Cod00] est incomplète. C'est un algorithme de recherche locale.

Dans le domaine de l'informatique musicale, 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 programmation par contraintes.

Le précurseur dans ce domaine est Kemal Ebcioglu qui propose un système de composition de chorals dans le style de Bach [Ebc97]. Le même problème est traité par Tsang [TsA91], de manière encore assez lente, puis par Philippe Ballesta dont le système est basé Ilog-Solver [Bal98]. 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.

[1]IRCAM, 1 place Igor Stravinsky, 75004 Paris {truchet, agonc, assayag}@ircam.fr

[2]LIP6, 8 rue du Capitaine Scott, 75015 Paris Philippe.Codognet@lip6.fr

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 pour la musique contemporaine, Situation et PWConstraints.

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. Situation est efficace mais présente plusieurs inconvénients. 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 (PWCS) a été conçu par Mikaël Laurson [Lau96]. Le moteur de résolution utilise les algorithmes classiques de forward checking et backjumping. 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 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 utilisé par PW Constraints permet de décrire beaucoup de problèmes musicaux, mais pas tous. Enfin, PWCS traite mal les contraintes globales et l'aspect visuel est inexistant.

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 Cahier des charges

2.1 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 domaines sont en général entiers, mais pas toujours (fonctions, motifs rythmiques par exemple). Ils sont toujours finis. Les contraintes sont parfois simplement arithmétiques, plus souvent elles comportent des alldiff, des $\exists$, $\forall$, $\in$.

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.

2.2 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.

Nous avons décidé de distinguer les problèmes de génération d'un matériau musical à l'aide de contraintes, cas où une méthode complète est inévitable, des problèmes de résolution. Les problèmes de résolution seront traités avec une méthode incomplète, un algorithme de recherche locale par améliorations successives du résultat, plus adéquat aux besoins exposés ci-dessus.

3 Méthode complète

L'un des premiers travaux a été d'importer dans OpenMusic, développé par Gérard Assayag et Carlos Agon ([Ago98] et [ARL99]) 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

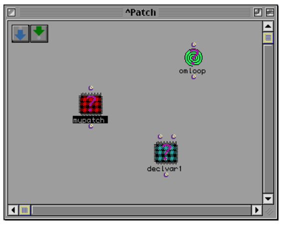

Figure 1: Exemples de patchs non-déterministes

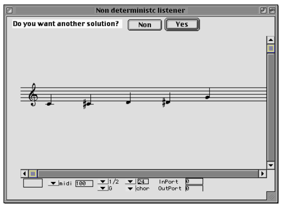

Figure 2: L'énumération des solutions a lieu dans le Nondeterministic Listener.

de e_1 donne un $fail$, e_2, etc. Screamer 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. Pour l'instant, seules les fonctions faisant appel au backtracking 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 patchs, qui correspondent aux fonctions déclarées indéterministes par Screamer. Ils sont notés avec un point d'interrogation, voir figure 1.

Lors d'une évaluation à la sortie d'un patch indéterministe, OM rajoute automatiquement l'une des trois fonctions 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 Nondeterminitic Listener, voir figure 2, 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 entre passer à une autre solution et garder la solution affichée.

Pour la déclaration de variables, l'utilisateur peut utiliser directement les primitives Screamer a-member-of et an-integer-between qui font ce que leur nom indique. Nous avons ajouté list-of-members-of et list-of-integers-between, qui font également ce que leur nom indique sauf quand on utilise l'entrée optionnelle : celle-ci permet de poser des contraintes globales à vérifier pour chaque nouvelle variable, ceci pour gagner du temps de calcul. Ces quatre fonctions

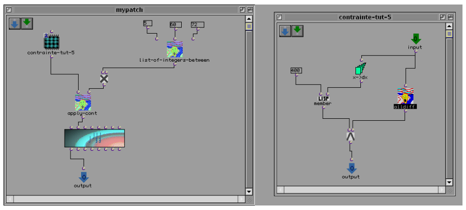

Figure 3: Un exemple de patch utilisant Screamer pour calculer des valeurs midi pour un chord-seq. A droite, la contrainte utilisée, exprimée comme un préicat OM. Elle impose que toutes les notes soient différentes, et que l'accord contienne une tierce majeure.

sont identifiées par une icône ad-hoc, voir figure 3. Dès qu'une de ces quatres fonctions est appelée dans un patch, le patch est classé non-déterministe. Les contraintes sont écrites comme des prédicats OM. Elles sont passées au solver via une fonction apply-cont, voir figure 3.

4 Méthode incomplète

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. 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 [AaL97]. 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.

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. Les solutions approchées sont traitées de la même manière. 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 des contraintes et génération des coûts

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.

La grammaire des contraintes s' écrit :

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

$t ::= n \mid (f t ... t)$

avec n un entier et f une fonction de Lisp.

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 :

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

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

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

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

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

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

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

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

4.5 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 donnet un panorama de ce que l'on doit pouvoir traiter.

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 [MOS74]. Avec $n = 12$, on obtient une série au sens musical. La contrainte sur les différences successives impose que chaque intervalle soit aussi entendu exactement une fois lorsque l'on joue la série. 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. Ces résultats peuvent aussi être comparés à ceux d'Ilog Solver. Pour 20 entiers, Ilog Solver met plus d'une heure.

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

5.2 Suite d'accords avec note commune

On considère une suite d'accords représentés par leurs valeurs midi. 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 avec la fondamentale, l'intervalle entre deux notes, et le nombre de notes par accord.

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. 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 4 montre une solution approchée, avec des T_i de 8, 14 et 12 unités de temps, ici la double croche. La figure 5 est une solution exacte.

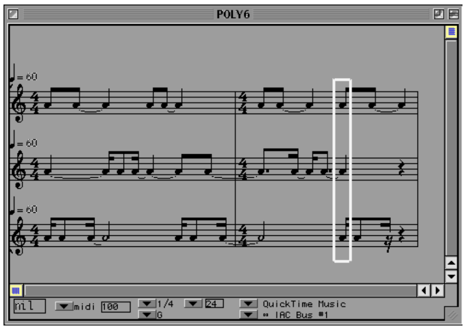

Figure 4: Une solution approchée. L'erreur est encadrée en gris, les voix inférieures et supérieures frappant un onset sur ce temps.

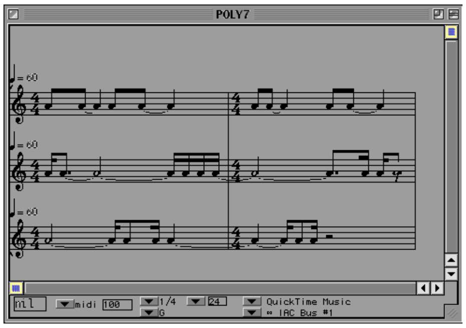

Figure 5: Une solution exacte du même problème.

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. Nous pensons élargir la gamme des problèmes tests.

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.

References

[AaL97] E. H. L. Aarts and J. K. Lenstra. *Local search in combinatorial optimization.* John Wiley and Sons, 1997.

[Ago98] Augusto Agon. *An environment for computer assisted composition.* Thèse de doctorat, IRCAM-UniversitÈ de Paris VI, 1998.

[ARL99] Gérard Assayag, Camilo Rueda ans Mikael Laurson, Carlos Agon, and Olivier Delerue. Computer assisted composition at ircam : Patchwork & openmusic. *Computer Music Journal*, 1999.

[Bal98] Philippe Ballesta. *Contraintes et objets, clefs de voûte d'un outil d'aide à la composition.* Editions Hermès, 1998.

[Cod00] Philippe Codognet. Adaptive search, preliminary results. 2000.

[Ebc97] Kemal Ebcioglu. *An expert System for Harmonizing Chorales in the style of J.-C. Bach.* AAAI Press, 1997.

[Lau96] Mikal Laurson. Patchwork : a visual programming language and some musical applications. 1996.

[MOS74] Robert Morris and Daniel Starr. The structure of the all-interval series. *Journal of Music Theory*, 13(2), 1974.

[PAR95] Franois Pachet and Pierre Roy. Integrating constraint satisfaction techniques with complex object structures. pages 11–22, Décembre 1995.

[RUV97] Camilo Rueda and Franck Valencia. Improving forward checking with delayed evaluation. 1997.

[SLM92] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. pages 440–446, 1992.

[SMA93] Jeffrey Mark Siskind and David Allen McAllester. Nondeterministic lisp as a substrate for constraint logic programming. pages 133–138, 1993.

[Sol00] Christine Solnon. Ant-p-solver : un solver de contraintes à base de fourmis artificielles. pages 189–204, 2000.

[TsA91] C. P. Tsang and M. Aitken. Harmonizing music as a discipline of constraint logic programming. pages 61–64, 1991.