Etude et réalisation d'un système de satisfaction de contraintes avec application à l'harmonisation musicale.

Pierre Roy

Résumé

Ce rapport est une mise à jour du rapport de stage de DEA IRO de Pierre Roy, effectué au LAFORIA de juin à septembre 1994. Il concerne l'étude de l'état de l'art en matière de techniques de propagation de contraintes, en particulier les techniques dites d' *arc-cohérence*, et une implémentation en Smalltalk-80 des algorithmes principaux. En outre, ce travail comporte une application de ces techniques à un problème réputé difficile: l'harmonisation automatique de chorals à 4 voix. Les résultats de ce système sont comparés aux approches existantes. Ils montrent que l'utilisation systématique d'objets, au sens de la PPO (structure *et* méthodes) permet de réduire considérablement la complexité du système, en limitant le nombre de variables contraintes.

Abstract

This is the final version of Pierre Roy's report for his research project (DEA-IRO) spent at LAFORIA from June to September 1994. It concerns the study of the state of the art in constraint propagation techniques, especially *arc consistency* techniques, and an implementation in Smalltalk of the main algorithms. Moreover, we applied these techniques to a problem known to be hard: automatic harmonization of four-voice chorals. The performances of the resulting system are compared to previous approaches. They show that the systematic use of objects (structures and methods) allows to reduce drastically the system's complexity by limiting the number of constrained variables.

Table des matières

1. Introduction.

L'objectif de ce stage était d'étudier l'intégration des mécanismes de la programmation par contrainte avec les objets. A cette fin nous avons développé un système de satisfaction de contraintes intégré à Smalltalk-80, et l'avons appliqué à la réalisation d'un système d'harmonisation musicale automatique.

Un problème de satisfaction de contraintes est un problème dans lequel on cherche à instancier des variables, chacune ayant un domaine fini de valeurs possibles, tout en respectant un certain nombre de prédicats portant sur ces variables. Ces prédicats sont les contraintes du problème.

Le problème d'harmonisation musicale que l'on se propose d'étudier est celui de l'harmonisation à quatre voix avec des accords de trois sons. Dans cet exercice on donne un matériau musical restreint, par exemple la voix la plus aiguë (soprano), et le système doit le compléter de manière à obtenir un morceau achevé, et ce, dans le respect des règles de l'harmonie classique. Le système peut également travailler à partir d'autres données que la mélodie, comme un morceau à 4 voix partiellement achevé.

Nous nous sommes donc attachés à trouver des mécanismes de satisfaction de contraintes suffisamment généraux pour pouvoir traiter un tel problème, et suffisamment efficaces pour le faire dans des temps raisonnables.

Enfin l'objectif global de ce stage était d'étudier ce que les mécanismes de la programmation par objets, et en particulier Smalltalk-80, peuvent apporter à la résolution de problèmes de contraintes, notamment en matière d'intégration et de réutilisabilité de structures complexes étrangères au système.

2. Problèmes de contraintes

Dans ce chapitre nous allons définir ce qu'est un problème de contraintes en l'illustrant d'un exemple. De plus nous présenterons les méthodes de résolution les plus courantes. Dans toute la suite on emploiera le terme CSP (Constraint Satisfaction Problem) pour désigner un problème de satisfaction de contraintes. Une excellente introduction du sujet est faite par Mackworth [11]. Pour des développements plus vastes on se reportera à Berlandier [3]. Deux ouvrages traitent le sujet en profondeur. D'un point de vue théorique, celui d'Edward Tsang [21] et plus appliqué, celui d'Annick Fron [10].

2.1. Définitions

Un CSP est défini par un ensemble fini de *variables* prenant des valeurs dans des domaines finis, et par un ensemble de *contraintes* entre ces variables.

Une *solution* d'un CSP est une instanciation des variables satisfaisant toutes les contraintes du problème.

Résoudre un CSP signifie trouver une ou toutes les solutions qu'il admet.

Le nombre de variables qu'implique une contrainte est son *arité*. Le nombre de contraintes impliquant une variable est son *degré*.

Les contraintes peuvent être définies de plusieurs manières différentes. On peut par exemple définir une contrainte par l'ensemble des instanciations des variables qui la vérifient (en extension). Plus généralement une contrainte sera définie par un prédicat, par une équation ou encore une inéquation (en intension).

2.2. Exemple

Considérons le problème d'arithmétique suivant :

> Trouver une solution de l'équation x + y = z avec x, y et z appartenant à {1,2,3} et x > y.

Exemple 1

Mettons ce problème sous forme de CSP. Les variables sont x, y et z. Leur domaine est {1,2,3}. Les contraintes du problème sont :

en intension : x + y = z et x > y,

en extension :

{(1,1,2),(1,2,3),(2,1,3)} et
{(2,1,1),(2,1,2),(2,1,3),(3,1,1),(3,1,2),(3,1,3),(3,2,1),(3,2,2),(3,2,3)}.

La seule solution est ici : (2,1,3).

2.3. Représentations

On peut représenter un CSP sous plusieurs formes différentes, notamment en un hypergraphe ou un graphe bipartie. Pour approfondir ces questions on pourra se reporter à Edward Tsang [21].

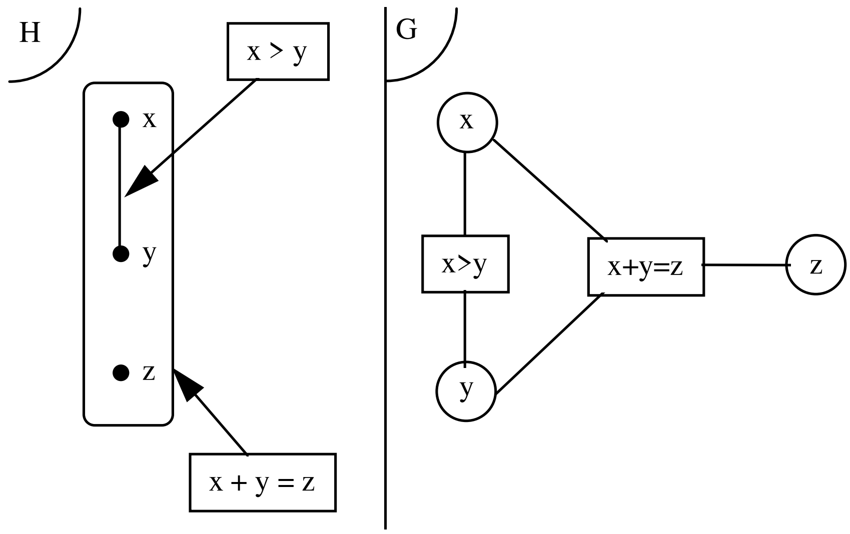


Figure 1

On peut représenter un CSP par un hypergraphe H de la façon suivante. Les sommets sont les variables du CSP. Pour chaque contrainte, on crée une arête sur les variables impliquées par cette contrainte. La Figure 1 représente l'Exemple 1 sous forme d'hypergraphe (à gauche).

On peut aussi représenter ce CSP sous forme d'un graphe biparti G de la manière suivante. Les contraintes et les variables sont représentées par les sommets. Pour chaque contrainte, on crée une arête entre la contrainte et toutes les variables qu'elle implique. Sur la Figure 1 à droite on a représenté l'Exemple 1 en distinguant les variables (encerclées) des contraintes (encadrées).

2.4. Résolution des CSP

Résoudre un CSP signifie trouver une ou toutes ses solutions, s'il en existe. Pour ce faire, on dispose de plusieurs techniques. On peut par exemple chercher les solutions de manière purement analytique. Cette voie est efficace sur certains problèmes comme la recherche de solutions pour une équation du second degré. Cependant, lorsque les domaines des variables sont finis, on se heurte à des problèmes plus complexes. On emploie alors essentiellement des techniques énumératives. Ces problèmes étant par nature très combinatoires, la plupart des travaux ont porté sur la recherche d'algorithmes permettant de faire une énumération *efficace* des solutions.

2.4.1. Génération-test

La première méthode pour résoudre un CSP est l'algorithme de *génération-test*. Celui-ci consiste à énumérer toutes les combinaisons de valeurs possibles et, pour chacune d'entre elles, à vérifier que c'est une solution. Si on ne cherche qu'une solution on s'arrête dès qu'on en a trouvé une. Si on les cherche toutes on procède à une énumération complète.

2.4.2. Retour arrière chronologique

Cette méthode est une amélioration de la précédente dans laquelle on va éviter de parcourir tout l'espace de recherche. L'algorithme consiste à instancier les variables une par une, et à intercaler une vérification des contraintes dont tous les attributs sont déjà instanciés. On coupe ainsi des branches entières de l'espace de recherche. En effet si dès l'instanciation de la seconde variable une contrainte se trouve violée il devient inutile de prolonger la construction de cette combinaison. Cette méthode est exposée dans l'Algorithme 1.

Voici le détail des procédures mises en jeu :

- RAC est la procédure principale qui commence l'énumération.
- Avancer est celle qui instancie les variables une par une. Si une solution complète a été calculée elle l'ajoute à la liste des solutions. Sinon elle continue l'instanciation. A chaque pas elle vérifie que les contraintes dont tous les attributs sont déjà instanciés ne sont pas violées. Si tel n'est pas le cas elle abandonne la solution en cours, c'est à dire qu'elle passe à la valeur suivante pour la variable courante si possible, sinon elle revient à la variable précédente. Si aucune contrainte n'est violée elle continue cette solution potentielle. Cette procédure est récursive.
- Tester vérifie que toutes les contraintes reçues en deuxième argument sont vérifiées par la solution partielle reçue en premier argument.

Le retour arrière chronologique est une méthode affectée de plusieurs défauts majeurs dont les deux suivants :

- Contraintes unaires. Supposons qu'une contrainte unaire (d'arité 1) porte sur une variable v, et qu'elle n'admette pas toutes les valeurs du domaine de v. Dans ce cas ces valeurs indésirables vont "distraire" l'algorithme de résolution. En effet elles seront prise en considération à de nombreuses reprises, alors qu'un simple prétraitement du CSP suffit à les supprimer. On réglera ce problème au 2.4.3.1.
- Une autre situation piège le RAC. Supposons que C soit une contrainte entre les variables V et W. Soit P le prédicat binaire définissant cette contrainte. Supposons qu'il existe une valeur v pour V telle que : $\forall w \in$ Domaine(W); non(P(v,w)). Dans cette situation la valeur v pour V ne peut faire partie d'aucune solution complète du CSP. Il serait alors convenable de l'enlever. On réglera ce problème au 2.4.3.2.

Pour résoudre ces problèmes plusieurs solutions ont été imaginées. Notamment des techniques de *prétraitement* du problème avant énumération visant à réduire la combinatoire du problème sans en changer les solutions, ainsi que des techniques mixtes dans lesquelles on réduit le problème pendant l'énumération. Les paragraphes suivants présentent certaines de ces idées.

```
RAC(P) "P est un CSP, ici une liste de contraintes"
     Φ := ∅; "liste des solutions"
     avancer(variables(P),[],Φ);
     retourner Φ;
fin RAC.

Avancer(V,σ,Φ) "σ est la solution partielle en construction"
               "c'est un dictionnaire variable/valeur"
     si V = ∅ "il ne reste plus de variable à instancier"
          Φ := Φ ∪ {σ}; "on la solution à la liste"
     sinon
          v := choisirVariable(V);
          "En général on choisit la variable ayant le plus"
          "petit domaine"
          E := {C ∈ C(v) | (Variables(C)\{v}) ∩ V = ∅};
          "E est la liste des contraintes adjacentes à v "
          "et dont tous les attributs sont instanciés"
          ∀x ∈ D(v) faire
               σ[v] := x;
               si tester(σ,E) = true
                    "on avance ssi les valeurs satisfont à"
                    "toutes les contraintes de E"
                    avancer(V\{v},σ,Φ);
               fin si;
          fin faire;
     fin si;
fin avancer.

tester(σ,E)
     tant que E ≠ ∅ faire
          si evaluer(C,σ) = true
               E := E\{C};
          sinon
               retourner nil;
          fin si;
     fin tant que;
     retourner true
fin tester.
```

Algorithme 1

2.4.3. Prétraitements du problème

Le but de ces algorithmes est de réduire la combinatoire du problème en supprimant des domaines des variables certaines valeurs ne pouvant faire partie d'une solution. On accélère ainsi l'énumération des solutions.

2.4.3.1. La cohérence aux noeuds d'un CSP

On dit d'une variable qu'elle vérifie la propriété de *cohérence des noeuds* si chacune des valeurs de son domaine vérifie toute contrainte unaire portant sur cette variable. Un CSP vérifie la cohérence des noeuds si et seulement si toutes ses variables le sont.

On peut considérer que chaque variable est impliquée au plus par une contrainte unaire.

L'algorithme qui réalise la cohérence des noeuds parcourt l'ensemble des variables du CSP. Pour chacune d'entre elles il parcourt son domaine. Pour chaque valeur du domaine il évalue la valeur de la contrainte unaire correspondante. Si cette valeur est *vrai* la valeur reste dans le domaine, sinon elle est retirée.

2.4.3.2. L'arc cohérence d'un CSP

Le domaine D_i de la variable V_i est *arc cohérent* si, et seulement si : pour tout $x \in D_i$, pour tout C portant sur V_i, il existe une instanciation des variables impliquées par C, respectant C, et dans laquelle $V_i = x$. Intuitivement, un domaine est arc cohérent si, et seulement si toutes ses valeurs sont compatibles avec chacune des contraintes du CSP, **prises individuellement**.

La terminologie d'arc cohérence vient de la représentation par un graphe que l'on peut faire des CSP binaires (i.e. dont les contraintes sont unaires ou binaires).

Un CSP est arc cohérent si, et seulement si les domaines de toutes ses variables le sont. Par exemple le CSP suivant :

$x = y$ avec $x \in \{1,2,3\}$ et $y \in \{1,2\}$

n'est pas arc cohérent. En effet la valeur 3 pour x n'est pas compatible avec la contrainte x=y. Plusieurs transformations du problème sont envisageables. On peut par exemple ajouter 3 au domaine de y. Cela rend le CSP arc cohérent mais modifie l'ensemble de ses solutions. On peut encore enlever la contrainte x=y, ce qui modifie également l'ensemble des solutions. On peut retirer 3 du domaine de x. Cette méthode laisse invariant l'ensemble des solutions du CSP.

Il faut remarquer que :

- Un CSP peut être arc cohérent sans avoir de solution. En effet l'arc cohérence peut se voir comme la meilleure propriété qu'on puisse obtenir en considérant les contraintes individuellement. Mais il se peut toujours qu'en les considérant toutes il n'y ait pas de solution. Par exemple si x et y ont pour domaine {1,2} et qu'on exige à la fois $x = y$ et $x \neq y$. Ce CSP est arc cohérent sans avoir aucune solution.
- La propriété d'arc cohérence peut toujours s'obtenir en réduisant les domaines des variables, et ce sans changer l'ensemble des solutions du problème. C'est ce qui va permettre d'accélérer l'énumération des solutions.

L'action d'éliminer les valeurs qui mettent en défaut une contrainte s'appelle le *filtrage*.

2.4.3.3. Les algorithmes d'arc cohérence

On trouve une littérature abondante sur le sujet. Waltz [23] le premier, en 1972, pose les bases des techniques de cohérence avec son algorithme de filtrage. Mackworth [11] implémente les techniques de Waltz et les améliore sensiblement (algorithmes AC-2 puis AC-3), et présente les algorithmes de path-consistency (PCn) en 1977. Mohr et Henderson [12] en 1986 obtiendront AC-4, un algorithme optimal. Cependant il s'avère moins performant que AC-3 car il traite des structures de données très complexes [22]. Deville et Hentenryck [8] ont proposé AC-5 un algorithme plus performant sur certains types de CSP. Nous avons beaucoup utilisé Berlandier [3] qui offre une synthèse détaillée de toute cette littérature et de nouveaux algorithmes à la fois généraux et efficaces.

L'algorithme que nous présentons ici est AC-3 de Mackworth [11]. Nous l'avons choisi car tous les autres en découlent et qu'il est le plus intuitif. Il est présenté dans sa version pour contraintes binaires pour des raisons de lisibilité.

```
AC-3(P) "P est la liste des contraintes du problème"
     i=1 to i=n do NC(i);  "on réalise la cohérence des"
                           "noeuds de P"
     Q := {(i,j) | (i,j) ∈ P et i≠j};
     "Q contient la liste des contraintes à filtrer"
     tant que Q ≠ ∅ faire
          choisir et effacer une contrainte (k,m) de Q;
          "on choisira par exemple la contrainte portant"
          "sur la variable de plus petit domaine"
          si REVISE((k,m)) "on note (i,j) une contrainte"
                           "entre i et j"
               Q := Q∪{(i,k) ∈ P| i≠k et i≠m};
          fin si;
     fin tant que;
fin AC-3.

REVISE((i,j))
"retourne vrai si elle a réduit un domaine, ceci pour que"
"les contraintes adjacentes à ce domaine soient toutes"
"remises dans la liste Q dans AC-3"
     delete := false;
     ∀x∈Di faire
          si (∀y∈Dj (x,y)∉ R(i,j))
               Di := Di \ {x};
               delete := true;
          fin si;
     fin faire;
     retourner delete;
fin REVISE.
```

Algorithme 2

Dans cet algorithme la procédure principale est AC-3. Elle travaille avec la liste Q qui contient en permanence la liste des contraintes qui doivent être *filtrées*. Quand une contrainte C vient d'être filtrée on la retire de la liste. Si le filtrage d'une autre contrainte réduit le domaine d'une des variables de C, alors celle-ci revient dans la liste pour être de nouveau filtrée. La raison de cette stratégie est facile à comprendre si on revient à la définition de l'arc cohérence. L'algorithme s'arrête quand la liste Q est vide. Il est évident de montrer qu'il s'arrête toujours.

La procédure REVISE est celle qui filtre la contrainte passée en argument. Elle retourne vrai si elle a effectué une réduction de domaine. Ceci pour que les contraintes adjacentes au domaine filtré soient remises dans la liste Q par AC-3.

La procédure NC réalise la cohérence du noeud passé en argument.

L'Algorithme 3 est une généralisation de AC-3 aux contraintes d'arité quelconque. Il est donné par Berlandier [3]. Les méthodes de filtrage (Algorithme 4) employées sont optimisées. C'est celui que nous utilisons dans notre système.

```
AC(P,Δ)
"P est la liste des contraintes du problème; Δ un "
"dictionnaire variable/domaine qui tient à jour les "
"domaines des variables du problème"
     origine := []; "dictionnaire contrainte/variable qui"
                    "contient la variable origine d'une "
                    "contrainte"
     tampon := []; "dictionnaire variable/valeur qui"
                   "les valeurs supprimées par l'algorithme"
     tant que P ≠ ∅ faire
          C := choisir_contrainte(P);
          "on choisira par exemple la contrainte de plus"
          "petite arité"
          V := filtrer(C,origine[C],Δ,tampon);
          "V contient la liste des variables dont le domaine"
          "a été réduit par filtrage"
          si V = nil
               retourner nil; "pas de solution"
          sinon
               P := P \ {C}; "on retire la dernière "
                             "contrainte filtrée"
               ∀v ∈ V faire
                    "maintenant on ajoute à la liste les"
                    "contraintes devant être refiltrées"
                    "du fait des réductions de domaines"
                    "précédentes"
                    ∀K ∈ C(v) faire
                         si K ∈ P alors
                              si origine[K] ≠ v
                                   origine[K] := nil;
                              fin si;
                         sinon
                              si K ≠ C
                                   origin[K] := v;
                                   P := P ∪ {K};
                              fin si;
                         fin si;
     fin tant que;
     retourner true;
fin AC.
```

Algorithme 3

La méthode FILTRER est l'équivalent de REVISE. L'implémentation est plus complexe du fait du traitement des contraintes d'arité quelconque.

```
FILTRER(C,f,Δ,tampon)
"cette procédure retourne la liste des variables dont le "
"domaine a été réduit par le filtrage"
     deja_vus := []; "dictionnaire variable/valeurs"
                     "mémorisant les valeurs déjà justifiées"
                     "pour la contrainte C"
     reduits := ∅;
     ∀v∈A(C) tel que v≠f faire
          σ := FILTRER_PASSIF(v,C,Δ,deja_vus);
          si σ≠∅
               D := DΔ(v)\σ; "où DΔ(v)=contenu de Δ[v]"
               si D=∅
                    retourner nil; "un domaine est vide"
               sinon
                    tampon[v] := tampon[v]∪σ;
                    Δ[v] := D;
                    reduits := reduits∪{v};
               fin si;
          fin si;
     fin faire;
     retourner reduits;
fin FILTRER.

FILTRER_PASSIF(v,C,Δ,deja_vus)
"elle commande la justification des valeurs qui ne sont pas"
"encore dans deja_vus"
     exclus := ∅;
     ∀x∈DΔ(v) tel que x∉deja_vus[v] faire
          si justifier([(v,x]),A(C)\{v},Δ,deja_vus)=nil
               exclus := exclus∪{x};
          fin si;
     fin faire;
     retourner exclus;
fin FILTRER_PASSIF.

JUSTIFIER(c,V,Δ,deja_vus)
"tient à jour la liste deja_vus"
     ∀σ∈{[vi,xi]|vi∈V;(x1,...,xn)∈DΔ(v1)×...× DΔ(vn)} faire
          si evaluer(C,σ∪c)
               ∀v∈objets(σ) faire
                    deja_vus[v] := deja_vus[v]∪σ[v];
               fin faire;
               retourner true;
          fin si;
     fin faire;
     retourner nil;
fin JUSTIFIER.
```

Algorithme 4

La méthode JUSTIFIER renseigne la liste `deja_vus`. Cette liste est construite de la manière suivante : quand une solution partielle a été évaluée comme correcte cela justifie toutes les valeurs de cette solution, et pas seulement celle de la variable actuellement concernée. A cet effet on conserve une trace de ces justifications pour éviter de les refaire. Cette table de valeurs correctes est remise à zéro à chaque nouvelle contrainte.

On remarque qu'un filtrage requiert un calcul du produit cartésien des domaines des attributs de la contrainte filtrée. Ceci explique l'aversion du système pour les contraintes d'arité élevée. Il sera souvent judicieux de scinder certaines contraintes en plusieurs contraintes d'arité inférieure.

2.4.4. Les algorithmes d'énumération avec filtrage de contraintes

Les algorithmes d'arc cohérence ne font que réduire la combinatoire du problème en vue de sa résolution. Les algorithmes que nous présentons dans ce paragraphes sont bâtis sur le RAC et comme lui sont destinés à l'énumération de solutions. Cependant ils tirent avantage des méthodes d'arc cohérence en les employant de manière plus ou moins systématique. Pour une présentation complète et originale des méthodes d'énumération avec filtrage on se reportera à Bernard Nadel [14].

2.4.4.1. Le forward checking

La première technique que nous présentons (Algorithme 5) est le *forward checking* (contrôle avant). Le principe est de faire un RAC mais à chaque nouvelle instanciation on ne se contente plus de vérifier les contraintes déjà totalement instanciées. On se propose de filtrer toutes les contraintes *adjacentes* à la variable courante. Ceci a pour conséquence de réduire encore plus les domaines du problème. Bien sûr cette plus grande rapidité de réduction se paye par un plus grand nombre de tests. Cependant la balance penche très fortement en faveur du forward checking sur la plupart des CSP (i.e. sur ceux de densité moyenne).

Cet algorithme est peu différent du RAC. Il travaille avec une variable Δ qui contient en permanence le domaine de chaque variable. Les différences principales avec le RAC sont écrites en caractères gras dans l'énoncé de l'algorithme.

Il existe d'autres techniques similaires comme les méthodes *look ahead* [15]. On en distingue plusieurs : le *partial look ahead*, le *full look ahead* et le *real full look ahead*. Elles se distinguent par l'ampleur du filtrage des contraintes. Dans le *real full look ahead* le niveau de cohérence établie est le plus fort. A chaque instanciation on filtre *toutes* les contraintes non encore totalement instanciées, c'est à dire celles dont tous les attributs ne sont pas encore instanciés.

Avec une telle stratégie le nombre de branches de l'arbre de recherche coupées est encore plus important qu'avec le forward checking. Cependant il est évident que, si on gagne du temps en recherche arborescente, on en perd en effectuant un filtrage plus important. Dans notre cas nous avons de meilleurs résultats avec un algorithme de type forward checking. L'équilibre entre les différentes méthodes est dépendant du type de CSP à résoudre.

2.4.4.2. Ordre de recherche

Un autre paramètre important dans la phase d'énumération est l'ordre d'instanciation des variables. Dans notre système, comme dans beaucoup d'autres, le critère choisi est celui du plus petit domaine d'abord. Un RAC classique ne filtre pas les domaines des variables restant à être instanciés. Ainsi, on peut choisir a priori l'ordre des variables. Par contre, le forward checking modifie les domaines des variables durant l'énumération. Il est donc nécessaire de recalculer la taille des domaines pour choisir à tout moment la variable ayant le plus petit.

```
RAC-CA(P)
     Φ := ∅;
     Δ := [];
     pour toute v∈variables(P) faire
          Δ[v] := D(v);
     fin faire;
     avancer(variables(P),[],Δ,Φ);
     retourner Φ;
fin RAC.

avancer(V,σ,Δ,Φ)
     si V = ∅
          Φ := Φ ∪ {σ};
     sinon
          v := choisirVariable(V);
          E := {C ∈ C(v) | (A(C)\{v}) ∩ V ≠ ∅};
          "on ne s'intéresse plus aux contraintes déjà "
          "instanciées, mais à toutes celles qui sont"
          "adjacentes à v".
          D := DΔ[v];
          ∀x ∈ D faire;
               Δ[v] := {x};
               σ[v] := x;
               tampon := [];
               "filtrage des contraintes adjacentes"
               si tester(v,E,Δ,tampon) = true
                    "correct => on continue cette branche"
                    avancer(V\{v},σ,Δ,Φ);
               fin si;
               "on restaure les valeurs supprimées à"
               "l'instanciation précédente pour dépiler"
               ∀v∈objets(tampon) faire
                    Δ[v] := Δ[v] ∪ tampon[v];
               fin faire;
          fin faire;
          Δ[v] := D;
     fin si;
fin avancer.

Tester(v,E,Δ,tampon)
"contrairement au RAC cette méthode filtre les contraintes"
"passées en paramètre au lieu de les évaluer"
     tant que E ≠ ∅ faire
          C := choisir-contrainte(E);
          si filtrer(C,v,Δ,tampon) = true
               E := E\{C};
          sinon
               retourner false;
          fin si;
     fin tant que;
     retourner true
fin tester.
```

Algorithme 5

Pour l'ordre de filtrage des contraintes on choisit la contrainte de plus petite arité d'abord. La raison de ce choix est qu'elle se filtre plus vite. Ainsi on peut espérer avoir fait des réductions faciles de domaines avant d'attaquer les "grosses" contraintes.

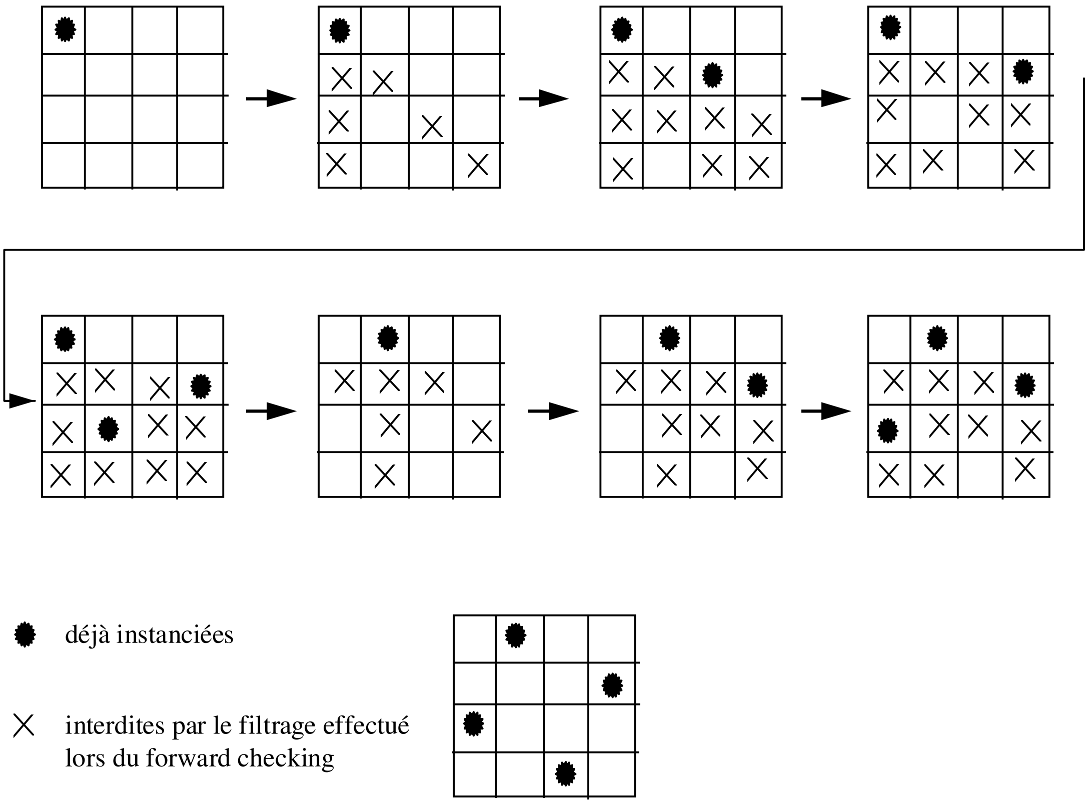
Il faudrait raffiner cette politique en tenant compte de la spécificité de certaines contraintes. Par exemple les contraintes du type $x \neq y$ n'ont pas à être filtrées car sauf si les domaines de x et de y sont le même singleton, il n'y aura pas de réduction de domaines.

Pour ces questions on consultera Dechter et Pearl [7].

2.4.4.3. *Exemple de résolution avec le* **forward checking**

Nous allons ici résoudre le problème des n reines sur l'échiquier (ici n = 4) pour illustrer le fonctionnement du forward checking. Les disques noirs symbolisent les reines. Une case avec une croix est une case interdite.

Les cases interdites le sont par le filtrage des domaines effectué par le forward checking.



3. L'harmonisation musicale.

3.1. Le problème

Le problème qu'on veut résoudre est celui des exercices d'harmonie classique. On veut à partir d'une mélodie donnée construire un morceau musical polyphonique à 4 voix (soprano, alto, ténor et basse). La voix donnée constitue par exemple le soprano.

Le morceau final doit respecter un certain nombre de contraintes qui sont les règles de l'harmonie classique.

Pour une présentation approfondie de ce qu'est l'harmonie on se reportera n'importe quel traité d'harmonie, par exemple Bitsch [4]. R. Mouton [13] donne en quelques pages les règles essentielles, ce sont celles que nous avons retenues pour notre système.

La grande complexité de ce problème vient du nombre de notes possibles et du fait que les règles d'harmonie ne contraignent pas très fortement les variables. Plutôt que de déterminer avec précision une connaissance, la plupart de ces règles ne font qu'interdire certaines propriétés, ce qui ne réduit que peu la combinatoire du problème. On le constate sur les exemples suivants :

> Deux accords successifs ne doivent pas avoir le même degré.
> Deux notes successives dans la même voix ne doivent pas être en rapport de septième.
> La note sensible sur un accord de dominante doit monter à la tonique.

De plus les lois de l'harmonie ne sont pas absolues, pour s'en convaincre :

> Les voix extrêmes seront <u>autant que possible</u> en mouvement contraire.
> L'accord de sixte et de quarte doit être manié <u>avec précaution</u>, il est <u>principalement</u> employé sur la dominante.

On méditera encore sur cet extrait du traité d'harmonie de Bitsch [4] :

« les règles ont une valeur relative et non absolue. Elles tiennent compte de l'habituel et non de l'exceptionnel. On peut affirmer que dans neuf cas sur dix tel enchaînement sera disposé de telle façon, mais rien de plus : la règle n'a qu'une valeur statistique. (...) c'est dire aussi qu'il faut s'attendre à de nombreuses exceptions ».

La résolution directe de ce problème par les contraintes ne pourra donc pas avoir pour ambition d'obtenir une grande qualité musicale. Il faut plutôt voir là un champ d'expérimentation, intéressant et difficile, pour les mécanismes de programmation par contrainte.

Une voie intéressante serait de traiter des hiérarchies de contraintes [5] pour mettre en valeur l'importance primordiale de certaines règles, et secondaire d'autres. Un véritable outil d'aide à la composition devra proposer ce type d'outil au musicien.

3.2. Les règles

Pour une présentation plus complète des règles et leur implémentation informatique sous forme de contraintes on se reportera à la thèse de Ballesta [2]. Nous nous sommes largement inspirés des travaux en cours de R. Mouton [13], qui offrent une synthèse des règles d'harmonie et de contrepoint.

3.2.1. Les tessitures

Chaque note appartient à une unique voix. A chacune des voix correspond un intervalle : sa tessiture. Pour chaque voix la tessiture mesure environ 2 octaves, ce qui autorise une douzaine de notes possibles par note.

Cette règle s'exprime comme une contrainte unaire.

3.2.2. Les dispositions relatives entre voix

Les voix ne doivent pas se croiser. Cette règle s'exprime par des contraintes binaires reliant chaque note à) chacune des trois autres jouées en même temps.

De plus les écarts constatés entre deux voix consécutives doivent être au plus d'une octave. Ceci s'exprime encore par un ensemble de contraintes binaires.

3.2.3. Les mouvements mélodiques

Deux notes consécutives doivent être distantes d'au plus une octave, et ne doivent pas faire une septième entre elles. Cette règle est une contrainte binaire.

Si une septième est franchie en deux sauts l'un des deux doit être une seconde. Cette règle est ternaire.

En deux sauts on ne doit pas franchir plus d'une octave (binaire).

3.2.4. Mouvements directs

Deux accords successifs ne doivent pas comporter d'octaves ou de quintes parallèles (binaire). Cette règle comporte de nombreuses exceptions que nous n'avons pas toutes considérées.

3.2.5. Mouvements obligés

La note sensible (septième degré de la gamme) doit monter à la tonique (premier degré). Cette règle comporte de nombreuses exceptions que nous avons négligées.

3.2.6. Les doublures

Les règles concernant la doublure d'un accord sont nombreuses. Nous n'en avons programmé qu'une partie. Ces règles tiennent compte de la structure de l'accord ainsi que du mode, mineur ou majeur, de l'exercice.

3.3. Représentation par des contraintes

Les règles qu'on vient de présenter peuvent aisément se représenter sous forme de contraintes entre objets. En effet, ces règles sont en fait des relations que certains objets doivent vérifier entre eux. Ceci est en parfaite concordance avec la nature relationnelle des contraintes.

De plus les règles mises en jeu dans un exercice d'harmonie sont locales. Elles ne portent que sur un petit nombre d'objets, souvent deux. Ceci justifie l'emploi de mécanismes de contraintes qui sont parfaitement adaptés à ce contexte.

3.4. Les études précédentes

3.4.1. CHORAL

CHORAL de K. Ebcioglu [9] est un système expert, disposant d'environ 350 règles, destiné à la composition de chorals dans le style de Jean-Sébastien Bach. Ce système repose sur un mécanisme de retour arrière "intelligent" avec une gestion passive des contraintes.

3.4.2. CARLA

CARLA de Courtot [6] est un outil d'aide à la composition reposant sur prolog. Il offre une interface graphique qui permet au non informaticien de l'utiliser. Le mécanisme de résolution repose sur l'emploi de prédicats activés dès l'instanciation des variables sur lesquelles ils portent. Les contraintes sont actives dès la déclaration du problème achevée et réduisent ainsi l'espace de recherche avant toute instanciation.

3.4.3. C.P.Tsang et M.Aikten

Tsang et Aitken [20] utilisent la programmation logique avec contraintes. Ils s'attaquent au même problème que nous (mélodie donnée). Ils disposent d'une vingtaine de règles. Leur structure principale est appelée *pulse*. Un pulse représente un accord (degré, inversion, notes, durée etc.). Les termes de degré et d'inversion ne sont pas instanciés au début et le seront au cours de la résolution.

Pour calculer la première solution avec 11 notes données au soprano, leur programme exige 5 minutes sur SUN SPARC 1, avec une occupation mémoire de 70 méga-octets! Ces chiffres rendent le système quasiment inutilisable de manière courante, à moins de disposer d'une machine extrêmement puissante. Un de nos objectifs est de parvenir au même résultat avec des temps d'exécution et une occupation en mémoire raisonnables.

3.4.4. Les travaux de Ovans

Ovans [16] [17] a été le premier à proposer des techniques de satisfaction de contraintes et en particulier d'arc cohérence pour la résolution de problèmes de contrepoint automatique. Grâce à ces techniques, Ovans obtient un système très efficace, validant ainsi son approche par satisfaction de contraintes. La critique principale de ces travaux est d'utiliser des contraintes portant sur des valeurs terminales (nombres), et donc de ne pas proposer de structuration des objets musicaux. Une critique plus précise de ces travaux peut être trouvée dans [1].

3.4.5. Les travaux de Ballesta

Ballesta utilise le langage de programmation par contrainte Ilog-Solver (nouveau nom de PECOS [19], [10]).

Son l'objectif est de mélanger la richesse de la représentation par objets des structures musicales à l'efficacité des contraintes. Cependant sa représentation par objet inclut elle-même des contraintes, par exemple : *« une note est un objet constitué de 5 attributs mis en relation au moyen de contraintes »*. Pour chaque note deux contraintes sont créées. Le résultat est une grande quantité de contraintes à filtrer lors de la résolution.

Le problème abordé est différent du nôtre, il s'agit d'harmonisation avec basse et chiffrages donnés. Les degrés des accords sont donc connus car la tonalité est déterminée par le système.

Pour 12 notes, données à la basse, le temps de calcul de la première solution est de 3 minutes sur SUN SPARC 10, et une occupation mémoire d'environ 7 méga-octets.

3.5. La méthode "humaine"

Confronté à un exercice d'harmonie, un musicien ne va pas raisonner uniquement sur les notes. A partir des notes de la mélodie proposée il va décider de quels accords il va se servir. Il sait quels types d'accords il peut utiliser sous chaque note de la mélodie. Ensuite, il s'intéresse aux transitions possibles, et souhaitables, entre ces différents accords.

La structure sur laquelle s'appuie le compositeur est l'accord, et non pas la note.

Pour le musicien, résoudre un exercice d'harmonie, c'est trouver une succession d'accords harmoniquement acceptable. Ce n'est pas remplacer les silences des voix inférieures par des notes jusqu'à obtenir un ensemble convenable.

3.6. Notre système

3.6.1. Représentation du problème

Au paragraphe 3.5 nous avons dit que le musicien raisonne principalement sur les accords. Au paragraphe 3.2 nous avons énoncé certaines règles d'harmonie. On remarque que certaines d'entre elles portent sur les accords. Cela fait deux raisons d'utiliser la structure d'accord :

- il est *naturel* d'exprimer toute règle sur les structures musicales sur lesquelles elle porte. Il est difficile d'exprimer toute règle comme une relation entre notes. Par exemple la règle exigeant de conclure par une cadence parfaite s'exprime laborieusement sur le groupe des huit notes concernées. Il est donc important de faire cohabiter dans la représentation du problème plusieurs catégories d'objets : les accords et les notes.

- Si les musiciens raisonnent sur les accords sous-jacents à la mélodie c'est que c'est la méthode la plus *efficace*. L'accord n'est pas seulement un terme *commode* pour désigner un groupe de quatre notes. L'accord est une structure complexe, construite suivant des règles bien définies et disposant de caractéristiques fondamentales en harmonie. La différence entre un accord et un groupe de quatre notes est que l'accord vérifie un certain nombre de règles structurelles, ce qui lui confère des propriétés particulières.

Le musicien est efficace pour deux raisons. La première est qu'il utilise l'accord comme structure principale de son modèle car il est *capable* de raisonner sur les accords. La seconde est que confronté à une mélodie il lui est possible de *déterminer* rapidement les accords utilisables pour son harmonisation.

Pour rendre notre système efficace nous avons choisi de le doter de capacités semblables à celles du musicien :

- capacité à raisonner sur des objets complexes comme les accords.
- Capacité à déterminer rapidement les accords qu'il peut employer dans un contexte (mélodie et tonalité) donné.

Nous avons choisi d'utiliser l'outil MusES de Pachet [18] pour représenter le problème. MusES fournit toutes les structures musicales nécessaires à cet exercice : les notes, les accords, les gammes etc. Ces structures sont fournies sous forme de classes Smalltalk avec les méthodes permettant de les manipuler.

3.6.2. Déclarer des contraintes sur des objets qui n'existent pas encore

A la création du problème il n'existe pas encore d'accords. Les seules données dont le système dispose sont les notes de la mélodie et la tonalité. Avant de pouvoir mettre en jeu les règles concernant les accords il faut donc déterminer ces objets. Comme le musicien le système doit "calculer" les accords qu'il peut utiliser dans ce contexte avant d'entreprendre la recherche de solutions.

La méthode que nous proposons pour résoudre un exercice d'harmonies est la suivante :

- détermination des accords du problème par le système.
- Construction du CSP ayant pour variables les notes *et* les accords de l'exercice et comme contraintes les règles d'harmonie portant sur tous ces objets.
- Application de méthodes de la programmation par contraintes à ce système.

3.6.3. Les objets musicaux

Il y a une différence fondamentale entre la représentation des structures musicales par Pachet [18] ou par Ballesta [2]. Dans MusES les objets sont des instances de classes Smalltalk. Chez Ballesta les objets sont des constructions constituées d'attributs liés entre eux par des contraintes structurelles.

Les deux systèmes ont des aptitudes différentes. MusES représente les objets musicaux usuels et permet de leur appliquer les traitements utiles au musicien. Par exemple étant données deux notes on peut comparer leur fréquence ou savoir l'intervalle qui les sépare. Le système MusES offre à l'utilisateur tout ce pourquoi il a été prévu.

Par contre, le système de Ballesta est beaucoup plus général. Les structures étant représentées au moyen de contraintes elles jouissent d'une grande souplesse. Par exemple, on peut poser entre deux structures note une contrainte spécifiant que ces deux notes doivent être séparées par une tierce. Alors si la première des deux notes prend la valeur Mi 3, la contrainte va être activée et donner la valeur Sol 3 à la seconde note. A chaque fois qu'une des deux notes est modifiée, le changement est répercuté sur la seconde. C'est l'aspect relationnel des contraintes qui permet ce comportement dynamique du système.

Dans la représentation des accords Ballesta dispose de six attributs symbolisant les intervalles entre les notes constituantes. Si un accord a trois de ses notes déjà instanciées et que l'attribut d'un des intervalles impliquant la quatrième est connu alors la cette dernière sera automatiquement instanciée.

Dans MusES étant donné un accord on peut connaître les notes ou les intervalles le constituant. Mais le genre de traitement qu'on vient de voir n'est pas envisageable directement.

Cependant une telle généralité dans la représentation ne nous est pas utile. En effet MusES nous fournit toutes les structures et toutes les opérations sur ces structures dont on a besoin. Pour résoudre un exercice d'harmonie, activité parfaitement codifiée, la représentation de Ballesta est bien trop riche.

3.6.4. Intérêt de cette représentation

Les travaux de Tsang [20] ou Ballesta [1] montrent qu'il est possible de représenter un exercice d'harmonie en utilisant "seulement" les contraintes. Nous préférons employer les classes de MusES qui offrent tout le nécessaire pour représenter le problème sans encombrer le système avec des contraintes structurelles qu'il faudra filtrer pendant la résolution.

Le Tableau 1 montre les temps de calculs comparés de différents systèmes sur des problèmes comparables. La lenteur des systèmes de Ballesta et de Tsang est due au nombre trop important de contraintes gérées.

	11 notes	12 notes	16 notes
Tsang & Aitken	5 minutes (*)		
Ballesta		3 minutes	4 minutes
Notre système	20 secondes	20 secondes	40 secondes

Tableau 1

Temps de calcul de la première solution sur SUN SPARC 10 sauf (*) sur SUN SPARC 1.

De plus Ballesta traite un problème différent du nôtre puisqu'il donne les degrés des accords ce qui diminue sensiblement la complexité du problème.

3.6.5. Notre stratégie de résolution en détail

La Figure 2 montre notre manière de résoudre le problème. Il y a deux phases principales. La première consiste à transformer le problème initial en un problème d'accords. La seconde est la résolution proprement dite du problème.

Nous allons détailler chacune des étapes de la Figure 2.

1) *Construction du réseau des notes.* A partir de la mélodie on crée un CSP de la façon suivante : pour chaque note de la mélodie on crée quatre variables. Le domaine d'une de ces variables est la note de la mélodie. Chacune des trois autres variables correspond à une voix du morceau. Ainsi, leur domaine est la tessiture de la voix associée. Les contraintes sont toutes les règles d'harmonie concernant les notes : règles mélodiques, harmoniques, règles sur les croisements entre voix.
2) *Arc cohérence.* On traite le CSP précèdent avec un algorithme d'arc cohérence pour réduire les domaines des notes de façon à accélérer les traitements qui vont suivre.
3) *Calcul des accords admissibles.* On ajoute au CSP précédent autant de variables que la mélodie compte de notes. Chacune de ces nouvelles variables est associée à une note de la mélodie. Leur domaine est ensuite calculé. Il contient tous les accords parfaits ayant la note correspondante dans la mélodie pour soprano, et étant autorisés dans la tonalité du morceau.
4) *Ajout des contraintes impliquant les accords.* On ajoute au CSP les règles d'harmonie concernant les accords entre eux, et concernant les accords et les notes. Par exemple une contrainte lie un accord aux quatre notes le constituant. Une autre relie un accord aux accords voisins en exigeant qu'ils n'aient pas même degré.
5) *Arc cohérence.* Pour accélérer l'énumération qui va suivre on traite le CSP avec un algorithme d'arc cohérence. Cette phase est amplement justifiée par une réduction des domaines très importante.
6) *Enumération des solutions.* C'est la dernière phase, on emploie un algorithme d'énumération pour trouver une solution au CSP. L'algorithme implémenté dans le système est un RAC avec *forward checking*. Cette solution est ensuite proposée à l'usager.

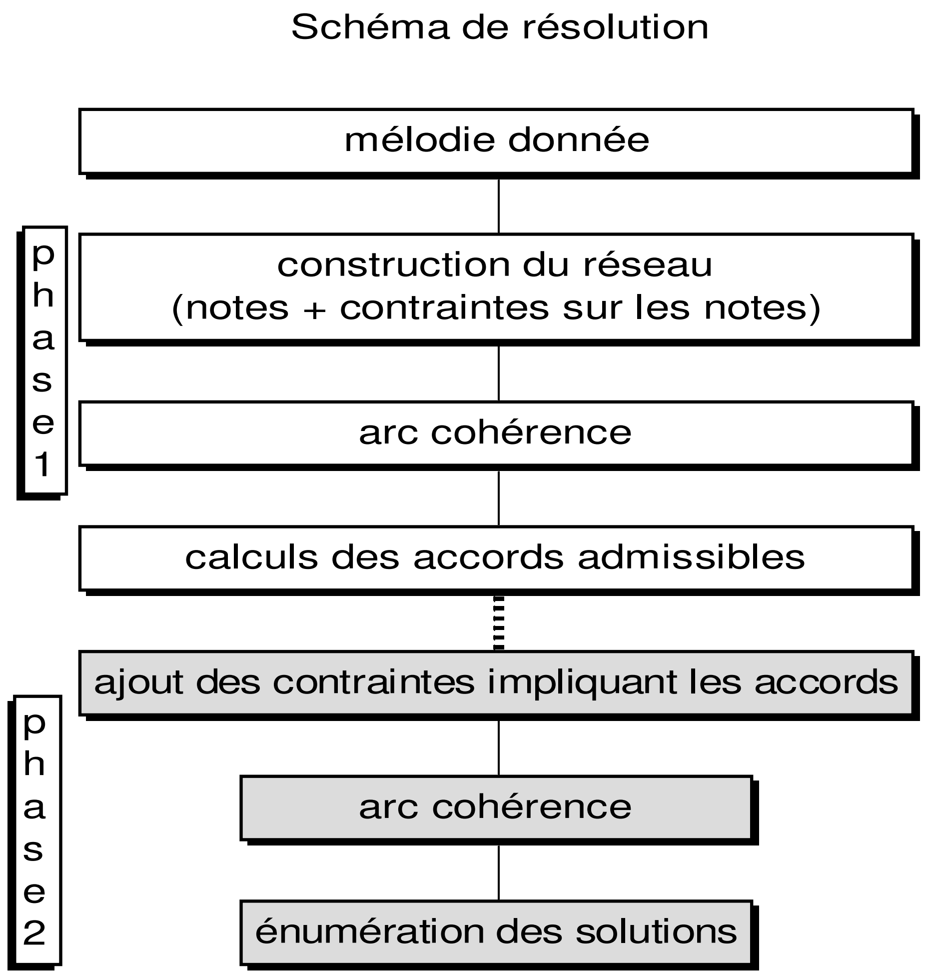


Figure 2

L'opération d'arc cohérence qui précède le calcul des accords permet une première réduction des domaines des notes, ce qui permet de ne pas trop calculer d'accords, chaque calcul étant assez long.

Le fait de calculer les accords de manière séparée permet de restreindre le nombre de contraintes. En effet le respect de la structure d'accord par les notes constituante est assuré ici du seul fait qu'on a "calculé" les accords. Lors de cette phase de calcul on a exclu toute note non convenable. Il n'est ainsi plus nécessaire de laisser des contraintes pour veiller dynamiquement au respect de cette structure.

La prise en charge de la structure d'accord revient à des objets et non à des contraintes. On perd ainsi l'aspect dynamique mais on gagne en efficacité.

4. Le système plus en détail

4.1. Les possibilités offertes

Le système consiste en un certain nombre de classes et leurs méthodes. Les classes principales sont CSP, Variable et Constraint. A partir de ces classes nous en avons défini d'autres comme PolyphonicCSP qui représente un exercice d'harmonie avec ses notes, ses accords et les règles musicales. Une FVChordVariable désigne une variable accord. La classe NoteVariable une variable note.

Nous avons défini des sous-classes de Constraint qui sont soit des sous-classes abstraites désignant un sous-groupe de contraintes, soit des contraintes. Les classes BinaryConstraint, TernaryConstraint et UnaryConstraint sont des sous-classes abstraites permettant de définir des contraintes binaires, ternaires ou unaires. D'autres classes comme BinaryEgalityConstraint sont des contraintes prédéfinies.

4.1.1. Les CSP

Un objet CSP est la représentation d'un problème de contraintes avec ses variables et ses contraintes.

La hiérarchie de la classe CSP est la suivante :

```
Object ()

    CSP ('constraints' 'loops' 'variables')
        PolyphonicCSP ('scale' 'name').
```

La classe principale dispose de trois variables d'instance qui pointent sur ses contraintes, ses variables et ses boucles (contraintes unaires).

Un CSP dispose de méthodes d'arc cohérence, de RAC, de détermination de domaines, d'interface etc.

La sous-classe PolyphonicCSP dispose en plus d'une instance scale qui est la gamme dans laquelle on travaille.

Cette classe dispose de plus de méthodes pour créer des instances à partir d'une mélodie donnée.

4.1.2. Les variables

Un objet Variable représente une variable avec son domaine, son nom et sa valeur éventuelle. Une variable connaît les contraintes l'impliquant et le CSP auquel elle appartient.

La classe Variable est définie par :

```
Object subclass: #Variable
	instanceVariableNames: 'constraints domain val label owner '
	classVariableNames: ''
	poolDictionaries: ''
	category: 'Variables-Abstract'.
```

Constraints pointe sur la liste des contraintes donc la variable est attribut. Val désigne la valeur de la variable (défini seulement quand elle est instanciée).

La hiérarchie des variables est la suivante :

```
Object ()

	Variable ('constraints' 'domain' 'val' 'label' 'owner')
		FVChordVariable ('s' 'a' 't' 'b' 'scale' 'startTime' 'duration')
		NoteVariable ('startTime' 'duration')
		NumberVariable ('x' 'y').
```

La sous-classe FVChordVariable dispose de sa propre méthode de détermination de domaines qui est environ 10 fois plus rapide que la méthode générale définie dans la classe CSP (Algorithme 6).

4.1.3. Les contraintes prédéfinies

Dans notre système nous avons défini des classes de contraintes. Les instances de ces classes sont les objets contraintes. La hiérarchie est la suivante :

```
Object ()
	Constraint ('attributes')
		AdditiveConstraint ('orderedAttributes')
		BinaryConstraint ('left' 'right')
			BinaryDifferentConstraint ()
			BinaryEgalityConstraint ()
			BinarySuperiorOrEqualConstraint ()
			...
			NotDirectFifthConstraint ()
			NotDirectOctaveConstraint ()
			NotSameDegreeConstraint ()
		BlockConstraint ('relationBlock' 'orderedAttributes')
		ChordConstraint ('chord' 's' 'a' 't' 'b')
		TernaryConstraint ('first' 'second' 'third')
			MelodicsJumpConstraint ()
			TernaryAdditiveConstraint ()
			TernaryEgalityConstraint ()
		UnaryConstraint ()
			ChordValidityConstraint ()
```

Détaillons un peu cette hiérarchie. Au sommet se trouve la classe abstraite `Constraint` responsable des méthodes de filtrage générales. Les classes `BinaryConstraint`, `TernaryConstraint`, `UnaryConstraint` ainsi que `AdditiveConstraint`, `BlockConstraint` et `ChordConstraint` en héritent.

Les trois premières sont des classes elle mêmes abstraites. Citons les particularités de deux de ces classes.

BinaryConstraint.

Cette classe représente les contraintes binaires.

Cette classe dispose de deux variables d'instance left et right, qui sont ses attributs. Ceci évite de systématiquement se déplacer dans la liste des attributs de la contrainte. Cela apporte un gain en facilité d'usage et en efficacité.

Mais le plus important est que cette classe dispose de sa propre méthode filtrage. En effet la méthode générale de filtrage n'est pas aussi efficace que la méthode REVISE de AC-3. Ainsi nous avons implanté cette dernière pour la classe BinaryConstraint.

On s'assure ainsi un gain substantiel en efficacité.

On déclare une contrainte binaire entre deux objets `x` et `y` par le message `BinaryConstraint newWith: x with: y.`

ChordConstraint.

Cette contrainte est celle qui lie une variable accord à quatre variables notes. Elle exige que l'accord ait pour notes constituantes les valeurs de ces variables.

Cette classe dispose de variables d'instances qui représentent ses attributs avec des noms plus "parlants".

Mais, là encore, le plus important se situe dans la méthode filtrage.

Celle-ci est extrêmement rapide comparée à la méthode générale de filtrage qui commence, rappelons-le, par calculer le produit cartésien des domaines de ses attributs. En ce qui concerne les complexités, on passe de $|c| \times |s| \times |a| \times |t| \times |b|$ pour le filtrage général à $|c| \times (|s| + |a| + |t| + |b|)$, avec |c| la taille du domaine de la variable chord, |s| la taille du domaine de la variable s etc.

Voici cette méthode implémentée en Smalltalk.

```
generateVariables: vars origin: v withDomains: delta
	| return domSop domAlt domTen domBas voiceName |
	domSop := delta at: s.
	domAlt := delta at: a.
	domTen := delta at: t.
	domBas := delta at: b.
	return := OrderedCollection new.
	v == s ifTrue: [voiceName := #soprano].
	v == a ifTrue: [voiceName := #alto].
	v == t ifTrue: [voiceName := #tenor].
	v == b ifTrue: [voiceName := #base].
	v == chord ifTrue: [^(delta at: chord)
			select: [:c | (domSop includes: c soprano)
				and: [(domAlt includes: c alto)
					and: [(domTen includes: c tenor)
						and: [domBas includes: c base]]]]].
	(delta at: chord)
		do: [:c | ((domSop includes: c soprano)
				and: [(domAlt includes: c alto)
					and: [(domTen includes: c tenor)
						and: [domBas includes: c base]]])
				ifTrue: [(return includes: (c perform: voiceName))
					ifFalse: [return add: (c perform: voiceName)]]].
	^return
```

Contraintes arithmétiques.

Dans la hiérarchie se trouvent quelques contraintes arithmétiques. Une contrainte d'égalité `BinaryEgalityConstraint` qui, posée entre deux variables, exige leur égalité. On peut déclarer cette contrainte entre deux variables *x* et *y* de deux façons différentes. On peut soit le faire par `BinaryEgalityConstraint newWith: x with: y` soit par l'envoi de message `x equals: y`.

On a aussi défini une contrainte `BinarySuperiorConstraint` qui peut se déclarer de deux manières dont `x > y.`

La contrainte d'inégalité `BinaryDifferentConstraint` exige que ses deux attributs soient de valeurs différentes.

Contraintes générales.

Une classe de contraintes permet de définir une expression Smalltalk quelconque entre deux variables. C'est la classe `BlockConstraint`. La déclaration d'une telle contrainte se fait en définissant un *block* dont l'évaluation est soit vrai soit faux. Les arguments du bloc seront remplacés par les valeurs des variables pour l'évaluation de la contrainte. Prenons un exemple. On veut que la variable numérique `x` ait une valeur plus grande que dix fois celle de la variable numérique `y`. On va définir le block Smalltalk suivant : `[:a :b | a > (10 * b)]`. Puis on va poser une contrainte de type entre x et y de la façon suivante :

```
BlockConstraint
  newWithAll: (OrderedCollection with: x with: y)
  block: [:a :b | a > (10 * b)].
```

Cette classe permet de définir toute contrainte exprimable par une expression Smalltalk.

4.1.4. Déterminations de domaines indéfinis

Nous avons dit que le système devait être capable de déterminer les accords du problème avant la résolution. Berlandier [3] expose un mécanisme général de détermination de domaines indéfinis que nous avons implémenté (cf. Algorithme 6). Cet algorithme permet de déterminer toutes les valeurs *possibles* pour une variable contrainte. Ces valeurs possibles sont celles qui conviennent à la première contrainte adjacente filtrée par l'algorithme. Cet algorithme ne fait pas beaucoup de calculs. Il ne filtre que peu de contraintes. Mais le but est de calculer très rapidement les domaines indéfinis du problème afin de pourvoir le traiter comme un CSP classique. Un calcul plus précis limiterait la taille de ces domaines mais serait plus long. Nous avions implémenté un algorithme de détermination des domaines utilisant le forward checking, il s'est avéré pénalisant. La technique du filtrage minimal est donc plus judicieuse.

4.1.5. Les algorithmes

Le système dispose d'un algorithme d'arc cohérence. Il s'agit de l'Algorithme 3 [3] exposé au 2.4.3.3. Il dispose aussi d'un algorithme d'énumération. C'est l'Algorithme 5 présenté au 2.4.4.1 qui utilise un mécanisme de retour arrière chronologique avec contrôle avant (forward checking).

4.1.6. Un outil orienté vers le musicien

Le système MusES [18] dispose d'un éditeur de partitions permettant de travailler avec la souris de façon tout à fait intuitive. Cette interface permet à un utilisateur non informaticien de créer ses propres mélodies et de demander leur harmonisation. Le résultat lui est proposé par ce même éditeur de partition sous la forme d'une mélodie à quatre voix.

Si la proposition du système ne convient pas à l'utilisateur il peut la modifier sans restriction. Il est même possible de modifier la mélodie. Le morceau partiellement complet qui en résulte peut être proposé au système pour une nouvelle harmonisation.

L'utilisateur a donc la possibilité d'interagir avec le système jusqu'à l'obtention d'une solution pleinement satisfaisante.

Malheureusement dans son état actuel le système n'offre pas d'interface intuitive pour la création et la modification des contraintes. Pour modifier les règles employées par le système il est obligatoire de définir des classes en Smalltalk.

Le paragraphe 4.2.1 montre un exemple d'interaction entre l'utilisateur et le système.

```
DETERMINER(P,Δ)
"P est la liste des contraintes du problème"
"Δ est la table qui mémorise les domaines des variables"
     geles := ∅;
     "geles est la liste des contraintes non encore"
     "activables"
     agenda := {C∈P | ∃v ∈ A(C), |D(v)| = ∞};
     "agenda est la liste des contraintes adjacentes à au"
     "moins une variable dont le domaine est indéfini"
     tant que agenda ≠ ∅ faire
          soit C ∈ agenda;
          infinis := {v ∈ A(C) | |D(v)| = ∞};
          agenda := agenda\{C};
          si infinis ≠ ∅
               si infinis = {e} et deductible(e,C)
               "si C n'est adjacente qu'a une seule variable"
               "de domaine indéfini et qu'elle peut déduire"
               "la valeur de cette variable connaissant"
               "toutes les autres"
                    Δ[e] := generer(e,C,Δ);
                    "on calcule toutes les valeurs possibles"
                    "pour la variable e selon C"
                    ∀K∈C(e)∩geles faire
                         "puis on ajoute les contraintes"
                         "gelées adjacentes à e. Ici pas"
                         "assez de tests pour aller vite (*)"
                         geles := geles\{K};
                         agenda := agenda ∪ {K};
                    fin faire
               sinon
               "C devrait être dans geles, on l'y remet"
               "Explication : (*)"
                    geles := geles ∪ {C};
               fin si;
          fin si;
     fin tant que;
     "on retourne geles qui est vide si ça a marché, sinon"
     "c'est qu'on ne peut pas résoudre le CSP par"
     "énumération"
     retourner geles;
fin DETERMINER.

Où deductible(e,C) retourne true ssi la contrainte C peut
déduire immédiatement la valeur de la variable e (contrainte
fonctionnelle activable).

Generer(e,C,Δ) revoie la liste des valeurs possibles pour e
qui satisfont la contrainte C.
```

Algorithme 6

4.2. Utilisation du système

4.2.1. Session de travail avec le système

Le système MusES [18] nous fournit une interface pour définir et éditer des mélodies, monophoniques ou polyphoniques, de manière intuitive. On ajoute et déplace les notes avec la souris, des menus permettent de changer tous les paramètres visuels de la vue, ou de changer les caractéristiques d'une note etc.

Les figures suivantes donnent un aperçu de cette interface.

Ballesta déplore lui-même la non-accessibilité de son système au compositeur non familier de programmation. L'interface de MusES a le mérite de permettre à un utilisateur qui ne connaît pas Smalltalk d'utiliser le système. De construire graphiquement sa mélodie, de demander une harmonisation, de modifier la proposition du système à sa guise et de redemander une harmonisation, etc.

Voici un exemple de session de travail : ci -dessous la mélodie, créee avec l'éditeur d'équations de MusES, proposée par l'utilisateur.

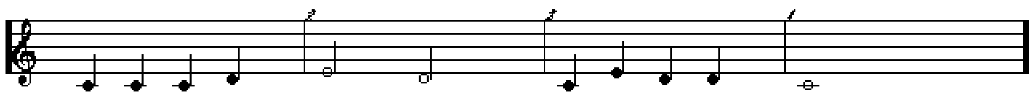


Figure 3

Ci-dessous la première solution proposée par le système.

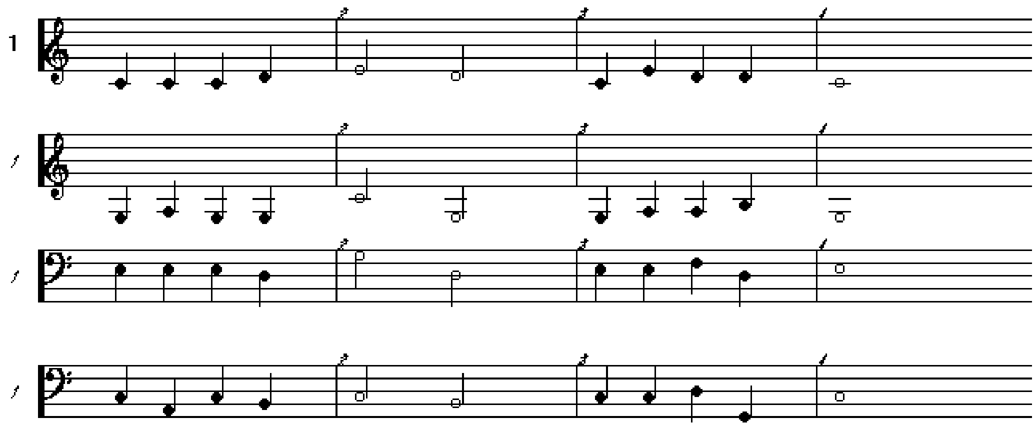


Figure 4

Cette solution ne convient pas forcément à l'utilisateur. Il peut alors la modifier à sa guise et demander au système d'harmoniser ce qui en résulte.

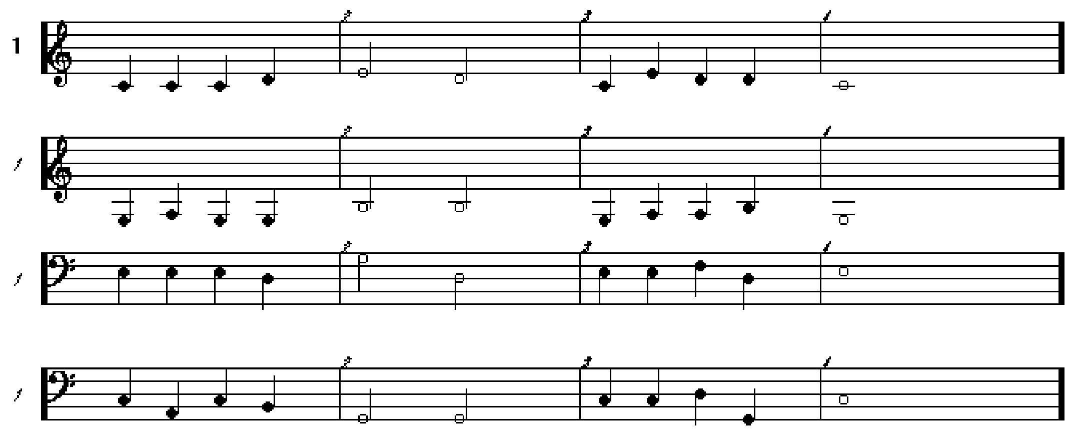


Figure 5

le système propose une solution qui tient compte des modifications demandées par l'utilisateur à laFigure 5.

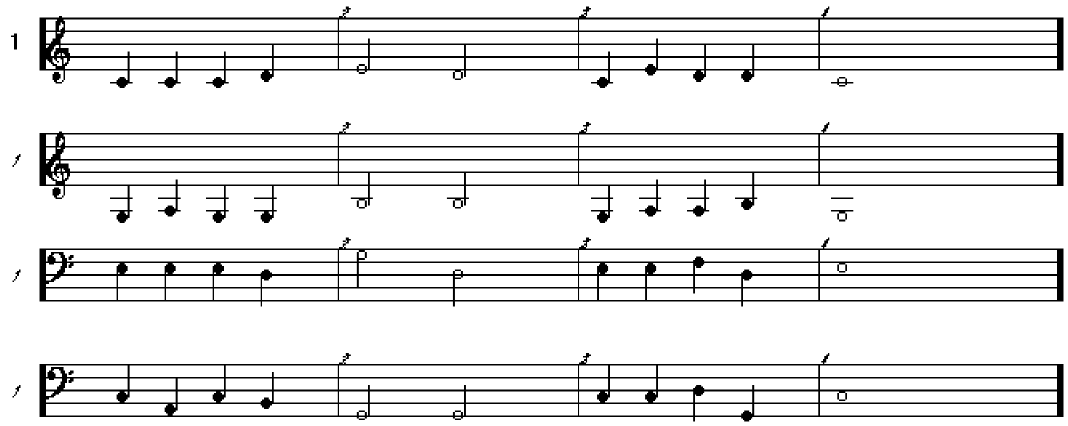


Figure 6

Bien entendu, au cours de l'interaction entre le système et l'usager il se peut que le système ne trouve pas de solution, il ouvre alors une mélodie vide. La signification de cette absence de solution est que les notes définies par l'utilisateur ne peuvent pas faire partie d'un morceau respectant scrupuleusement les règles fixées.

Deux solutions sont pour l'instant offertes :

1) Revenir en arrière et faire des modifications jusqu'à ce que le système trouve des solutions.
2) Se servir de l'interface d'édition des CSP pour enlever les contraintes qui ne doivent pas être vérifiées au goût de l'usager (cf. §4.2.2).
3) Il serait intéressant de disposer d'une possibilité de déclaration hiérarchique des contraintes [5]. Le système s'adapterait de lui-même au plus juste pour trouver des solutions respectant le plus de contraintes possibles (avec hiérarchie). Cette possibilité devra être implantée pour un système d'aide à la composition véritablement ambitieux.

4.2.2. Les CSPInspector

La classe CSPInspector est une interface sur la classe CSP. Un exemple est donné par laFigure 7. on y voit la liste des contraintes du CSP observé, la liste de ses variables, et le domaine de la variable sélectionnée. On dispose de certaines facilités comme la possibilité de trier les variables selon plusieurs critères. On peut également inspecter tout objet affiché, ou encore effacer des variables, des contraintes ou modifier les domaines. Cette interface offre quelques possibilités pour l'utilisateur de modifier les contraintes.

Bien sûr une interface graphique permettant au compositeur de définir ses contraintes à la souris directement sur une partition serait nécessaire pour un outil d'aide à la composition réellement opérationnel.

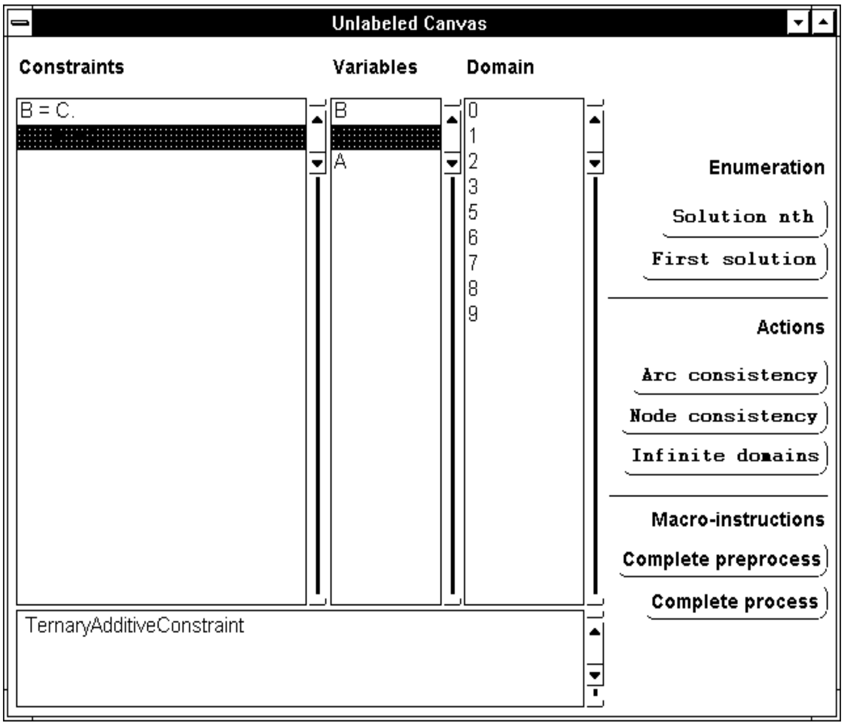
4.2.3. Définition de contraintes par l'utilisateur

L'utilisateur peut définir ses propres contraintes en écrivant des classes Smalltalk correspondantes, ou bien en utilisant la classe `BlockConstraint`. Pour définir ses propres classes il faut les faire hériter de la classe abstraite qui convient. Exemple pour définir une contrainte exigeant qu'un entier soit pair on va créer une sous classe de `UnaryConstraint`.

Les méthodes à redéfinir pour rendre ces contraintes opérationnelles sont les suivantes :

- `valueWithTable:` c'est la méthode principale. Elle prend en argument une instanciation de ses attributs et retourne vrai si elle est vérifiée par cette instanciation. Elle retourne faux sinon.
- `canDeduceVariable:` cette méthode prend en argument une variable `v` et retourne vrai si la contrainte peut déduire les valeurs de la `v` connaissant les valeurs de ses autres attributs.
- `deduceVariable: v inSubstitution: s` cette méthode prend en argument une variable v et une solution partielle s. Elle retourne la valeur de v pour que s vérifie la contrainte.

Les deux dernières méthodes sont utilisées pour la détermination des domaines infinis. En effet pour pouvoir calculer le domaine d'une variable il faut qu'au moins une des contraintes portant sur cette variable puisse déduire sa valeur connaissant celle de ses autres attributs. Par exemple dans le CSP suivant : $x = y$ avec $x \in \{1,2,3\}$ et y entier naturel, il est évident qu'on peut réduire le domaine de y à $\{1,2,3\}$ en utilisant le pouvoir déductif de l'égalité. Par contre le CSP suivant : $x < y$ avec $x \in \{1,2,3\}$ et y entier naturel n'est pas soluble par énumération. En effet la contrainte $x > y$ ne permet pas de réduire le domaine de y à un ensemble fini.



A gauche, on inspecte un CSP avec trois variables (A, B et C). C étant sélectionnée on voit son domaine à droite. Les 2 contraintes sont affichées à gauche (B=C et A=B+C). On peut calculer les solutions ou encore rendre le CSP arc cohérent depuis l'interface.

Figure 7

Ces deux méthodes permettent aussi de traiter les contraintes *fonctionnelles* (i.e. pouvant déduire un de leurs attributs connaissant tous les autres) de façon plus efficace. En effet la méthode `deduceVariable:inSubstitution:` permet de déterminer directement la valeur de la variable passée en paramètre au lieu de faire des tests. Par exemple la contrainte x=y dans le cas où x vaut 1 peut déduire automatiquement que y vaut 1 aussi. Le traitement classique essaye toutes les valeurs de y jusqu'à trouver la bonne.

C'est la méthode de filtrage qui envoie à la contrainte le message `canDeduceVariable:` et qui, en cas de réponse positive, appelle la méthode de filtrage actif au lieu de la méthode filtrage classique (passif). Le principe de cette méthode de filtrage actif est d'utiliser `deduceVariable:inSubstitution:` plutôt que de faire des tests jusqu'à trouver la bonne valeur. L'utilisateur aura intérêt à définir cette méthode pour ses propres contraintes s'il veut bénéficier d'un filtrage plus performant et s'il veut pouvoir utiliser les mécanismes de détermination de domaines.

Voyons l'exemple de définition de l'égalité binaire. La classe `BinaryConstraint` comporte deux variables `left` et `right` qui sont ses deux attributs. Les trois méthodes précédentes s'écrivent dans ce cas :

```
valueWithTable: s
	^(s at: left)		"la valeur du premier attribut dans la table s"
		= (s at: right)	"la valeur du second"
canDeduceVariable: v
	^true			"peu importe l'attribut proposé (left ou right) la déduction est possible"
				"cette contrainte n'est pas directionnelle"
deduceVariable: v inSubstitution: s
	v == left		"si on cherche la valeur de left"
		ifTrue: [^s at: right]. "c'est celle de right qu'il faut retourner"
	^s at: left		"sinon c'est le contraire"
```

5. Quelques exemples.

5.1. Send + more = money.

Rappelons cet exercice de crypto-arithmétique :

```
On cherche à instancier les variables S, E, N, D, M, O, R et Y
dans {0,1,2,3,4,5,6,7,8,9} pour satisfaire "l'équation" :

                  S E N D
                + M O R E
                ---------
                M O N E Y

On exige de plus que les variables soient deux à deux distinctes,
et que S et M soient non nulles.
```

La mise sous forme de contrainte de ce problème est faite par exemple dans [10].

Remarquons simplement que la contrainte exigeant que les variables soient deux à deux distinctes peut être exprimée de deux façons différentes :

1) comme une contrainte d'arité 8
2) comme 28 contraintes d'arité 2

la deuxième manière est préférable du point de vue complexité.

La solution unique de ce système est : 9567 + 1085 = 10652.

Comme illustration de l'intérêt de l'obtention de l'arc cohérence du problème avant l'énumération voici quelques résultats :

ARC COHERENCE + ENUMERATION	8 secondes (5 sec. + 3 sec.)
ENUMERATION DIRECTE	15 secondes.

(Sur PC Pentium).

5.2. Les 8 reines.

Ce problème consiste à placer sur un échiquier huit reines de façons à ce que deux quelconques d'entre elles ne soient pas en prise.

Sur cet exemple le CSP est déjà arc cohérent. On procède donc ici directement à l'énumération. La première solution est trouvée en moins d'une seconde par le système *(SUN SPARC 10).*

Le système est capable de traiter le même problème avec 100 reines sur un échiquier 100×100, et ce en moins d'une minute.

5.3. Harmonisation

5.3.1. La marseillaise

La mélodie est donnée Figure 8. Elle comporte 18 notes. Le système trouve une solution en environ 35 secondes.

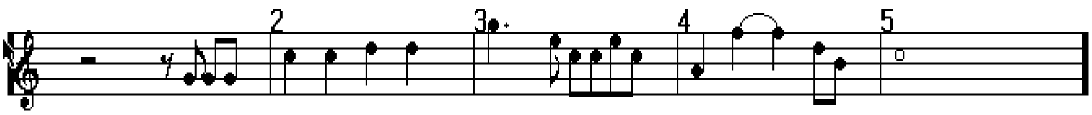


Figure 8

Sur la Figure 9 est représentée la première solution proposée par le système.

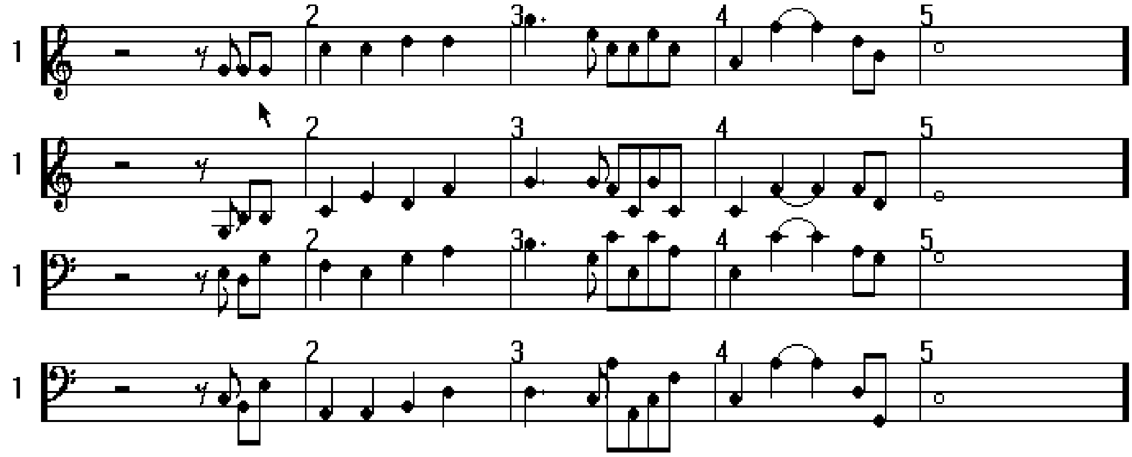


Figure 9

5.3.2. Au clair de la lune

La mélodie comporte 11 notes (c'est la première phrase). La première solution est trouvée par le système en 20 secondes sur SUN SPARC 10. Ce problème est illustré au paragraphe 4.2.1.

La Figure 10 donne un exemple d'harmonisation de Au clair de lune en sol majeur (bien sûr la mélodie a été donnée en sol).

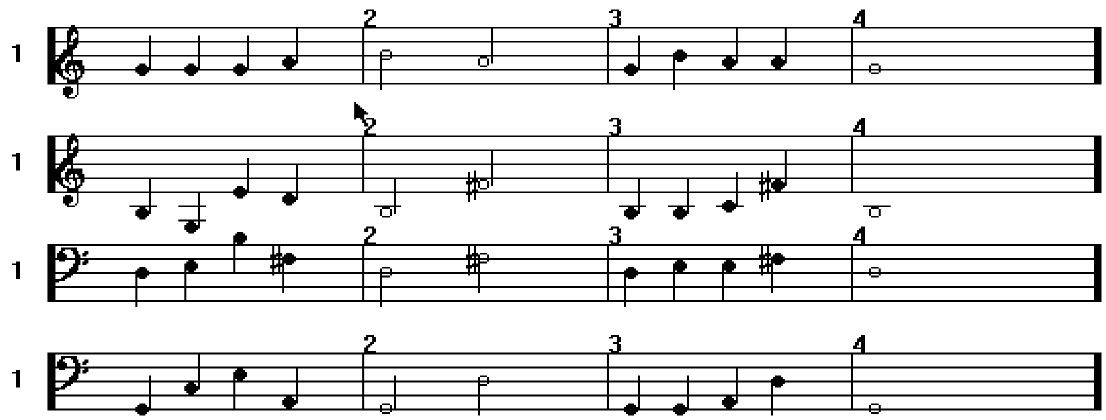


Figure 10

5.3.3. Exemple donné par Ballesta

La Figure 11 montre une mélodie de 16 notes donnée comme exemple par Ballesta [1]. La Figure 12 montre la première solution proposée par le système.

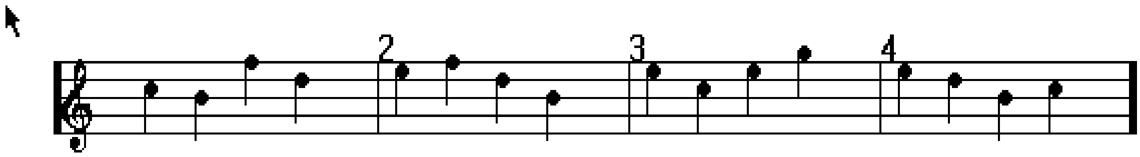


Figure 11

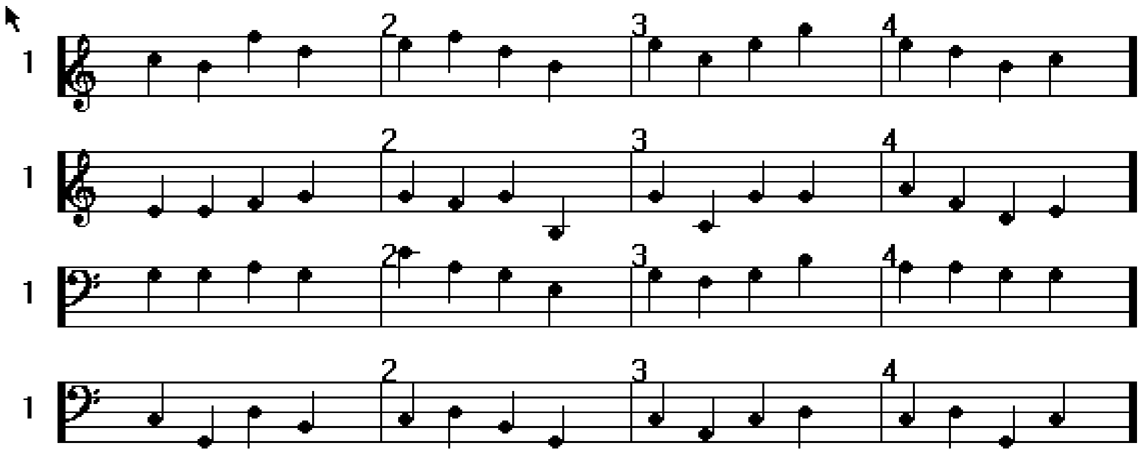


Figure 12

6. Conclusion

Dans ce rapport nous avons montré que la construction d'un système de satisfaction de contraintes dans un langage objet comme Smalltalk présente bien des intérêts.

- Dans le système les contraintes et les variables sont organisées en une hiérarchie. On peut ainsi spécialiser les algorithmes de filtrage ou d'évaluation de contraintes, ce qui améliore grandement l'efficacité du système.

- Il est facile de faire fonctionner le système sur n'importe quelles structures Smalltalk car on peut définir des contraintes portant sur tout objet du langage. C'est ce qui nous a permis de construire un outil de composition automatique en réutilisant la bibliothèque de classes MusES.

- Enfin la programmation par objets est une garantie d'extensibilité. On peut facilement ajouter à ce système de nouveaux algorithmes, de nouvelles contraintes ou variables. Il suffit pour cela de définir de nouvelles classes dans la hiérarchie déjà définie, et de redéfinir certaines méthodes dans ces classes.

Le système d'harmonisation automatique que nous avons présenté dans ce rapport ne peut pas, dans son état actuel, prétendre au statut de véritable outil d'aide à la composition. En effet, même si, contrairement à certains systèmes précédents, il est doté d'une interface orientée vers le musicien, il est pour l'instant trop rigide. On déplore, par exemple, que la définition des règles musicales et leur place dans la partition ne soient pas modifiables pour un non-programmeur. Il y a donc un effort à faire dans ce sens, par exemple en le dotant d'une interface graphique pour la définition des contraintes.

De plus il traite toutes les contraintes sur un même niveau ce qui le rend trop sensible aux données initiales. Il arrive qu'il ne trouve pas de solutions alors qu'en ôtant une contrainte qu'un musicien jugera mineure on en trouverait. Un remède sera de le rendre capable de gérer des hiérarchies de contraintes [5].

Le bilan est par contre très encourageant en ce qui concerne l'efficacité. Les temps de calculs sont très inférieurs à ceux des auteurs cités plus haut ce qui le rend exploitable de manière intensive sur un matériel informatique de taille raisonnable.

La raison de cette efficacité est que le système résout le problème en se plaçant au niveau d'abstraction adéquat, celui de l'accord, et en adoptant une stratégie de résolution en deux phases distinctes. Cette manière de travailler permet de profiter pleinement de la richesse expressive des accords. Le gain en efficacité de cette approche est particulièrement important, car elle permet d'avoir des temps d'exécution très satisfaisants, alors que le solveur proprement dit n'est pas particulièrement rapide. Nous pensons qu'une telle stratégie de séparation de la résolution en plusieurs phases est généralisable à beaucoup de problèmes de contraintes comportant des objets de catégories ontologiques distinctes.

7. Références

[1] Ballesta, Philippe. Contraintes et objets, clefs de voûte d'un outil d'aide à la composition? *Journées d'informatique musicale, Bordeaux, 1994.*

[2] Ballesta, Philippe. *Thèse de doctorat. 1994.*

[3] Berlandier, Pierre. Etude et réalisation d'un ensemble de primitives pour la satisfaction de contraintes en domaines finis. *Rapport INRIA, 1994.*

[4] Bitsch, Marcel. Précis d'harmonie tonale, *Editions musicales Alphonse Leduc, 1957.*

[5] Borning, Alan. Constraints hierarchies. *OOPSLA Proceedings, pp. 48-60, 1987.*

[6] Courtot, Francis. Logical representation and induction for computer assisted composition. *Understanding music with AI : perspectives on music cognition. AAAI press, Menlo Park, California, pp.156-181, 1992.*

[7] Dechter, Rina. Pearl, Judea. Network-based heuristics for constraint-satisfaction problems. *Artificial Intelligence, vol 34, 1-38, 1988.*

[8] Deville, Yves. Van Hentenryck, Pascal. An efficient arc consistency algorithm for a class of CSP problems. *Proceedings of IJCAI, pp.325-330, 1991.*

[9] Ebcioglu, Kemal. An expert system for harmonizing four parts chorales in the style of J.S.Bach. *Understanding music with AI : perspectives on music cognition. AAAI press, Menlo Park, California, pp.294-333, 1992.*

[10] Fron, Annick. Programmation par contraintes, *Editions Addison-Wesley, France, 1994.*

[11] Mackworth, Alan K. Consistency on networks of relations, *AI, N°8, pp.99-118, 1977.*

[12] Mohr, R. Henderson, T.C. Arc and path consistency revisited. *AI, vol. 28, N°2, pp.225-233, 1986.*

[13] Mouton, R. Condensé de traité d'harmonie. *Notes personnelles.*

[14] Nadel, Bernard. Tree search and arc consistency in constraint satisfaction algorithms. *Search in Artificial Intelligence, Springer-Verlag, pp.287-340. 1988.*

[15] Nudel, Bernard. Consisten-labeling problems and their algorithms : expected complexity and theory-based heuristics. *AI vol 21 n°1 et 2, pp.135-178. 1983.*

[16] Ovans, R. Efficient music composition via consistency techniques, *Center for Systems Science, Simon Fraser University, Burnaby, Canada, 1992.*

[17] Ovans, R. An interactive constraint-based expert assistant for music composition, *Proceedings of the ninth canadian conference on artificial intelligence, University of British Columbia, Vancouver, 1992.*

[18] Pachet, François. An object-oriented représentation of pitch-classes, intervals, scales and chords. *Journées d'informatique musicale, Bordeaux, 1994.*

[19] Puget, J-F. Intensional and cardinality constraints. *Rapport interne, ILOG, 1992.*

[20] Tsang, C.P. Aitken, M. Harmonizing music as a discipline of constraint logic programming. *Proceedings of the International Computer Music Conference, Montreal, Canada, 1991.*

[21] Tsang, Edward. Foundations of constraint satisfaction. *Computation in cognitive science, Academic Press. 1993.*

[22] Wallace, R.J. Why AC-3 is almost alwais better than AC-4 for establishing arc consistency in CSP. *Proceedings of the IJCAI93, Chambéry, France, 1993.*

[23] Waltz, D. Generating semantic descriptions from drawings of scenes with shadows. *Technical report,AI271, MIT, MA, 1972.*