Implantation de SBDS en GNU Prolog

Guillaume ARNAUD*

* *INRIA Rocquencourt – BP 105 – F-78 153 Le Chesnay Cedex*
Guillaume.Arnaud@inria.fr

RÉSUMÉ. *Nous présentons une implantation en GNU Prolog d'une méthode de "labeling" qui élimine dynamiquement des symétries données par l'utilisateur. Cette technique repose sur l'algorithme SBDS (Symmetry Breaking During Search) de plus en plus connu par la communauté CSP. Cet article a pour objectif de formaliser cet algorithme, de l'implanter en GNU Prolog sous la forme d'une nouvelle fonction de labeling afin qu'il soit facile d'utilisation et de discuter d'optimisations possibles de ses performances.*

ABSTRACT. *An implementation, in GNU Prolog, of a labeling function which breaks dynamically symmetries given by the user is introduced. This technique rests on the SBDS (Symmetry Breaking During Search) algorithm increasingly known by CSP community. This paper aims to formalize this algorithm, to implement it in GNU Prolog so that it is easy to use and to prospect potential optimisations of its performances.*

MOTS-CLÉS : *Programmation avec Contraintes, Symétries, Groupes de permutation, SBDS*

KEYWORDS: *Constraint Programming, Symmetries, Permutation groups, SBDS*

1. Introduction

La programmation par contraintes, qui permet de résoudre des problèmes de grande complexité combinatoire, nécessite souvent le parcours d'arbres de recherche imposants. Si les capacités purement calculatoires des solveurs sont de plus en plus grandes, tout comme la puissance des nouveaux processeurs, nous continuons à bloquer sur des problèmes d'une telle complexité qu'il semble impossible de les résoudre en temps raisonnable et en parcourant la totalité de l'arbre de recherche. C'est dans cette optique que des travaux ont été menés sur le problème des symétries en programmation par contraintes.

Afin de cerner le sujet prenons, par exemple, le problème en apparence assez simple du golfeur social (N joueurs formant g groupes de s golfeurs jouent pendant w semaines sans jamais rencontrer plus d'une fois un même joueur), le nombre de configurations symétriques est tel que pour certaines instances on ne peut espérer développer tout leur arbre de recherche. Dans ce problème on considère en général les symétries entre chaque joueur à l'intérieur d'un groupe, chaque groupe à l'intérieur d'une semaine, chaque semaine à l'intérieur d'une solution et enfin la symétrie entre chaque joueur. En combinant ces quatre symétries pour chaque configuration on obtient $(s!)^{gw}(g!)^w w!(gs)!$ autres configurations. Par exemple pour le cas $g = 5$, $s = 3$ et $w = 7$, on aura environ $4 \cdot 10^{57}$ configurations symétriques à une même configuration. Un certain nombre de ces symétries peuvent être cassées statiquement en changeant la modélisation, en imposant une affectation partielle ou en ajoutant certaines contraintes. Cependant on a pu remarquer que cela n'était pas toujours suffisant, pour essentiellement deux raisons : certaines symétries ne sont pas facilement représentables sous la forme de contraintes statiques et les combinaisons entre symétries ne sont pas traitées.

Nous verrons dans la section 2 qu'il est parfois beaucoup plus intéressant de casser dynamiquement les symétries. Nous tenterons également de donner quelques définitions sur les symétries qu'on désire étudier et on s'arrêtera sur un des algorithmes d'élimination de symétrie (SBDS : Symmetry Breaking During Search), défini par Backofen et Will dans [BAC 98], développé entre autre par Gent et Smith dans [GEN 00] et amélioré plusieurs fois depuis. Dans la section 3 nous expliquerons comment nous avons implanté [1] cet algorithme en GNU Prolog et comment on modélise les symétries. Dans la section 4 on discutera de quelques optimisations dont, entre autres, celle consistant à diminuer la quantité de symétries qu'il est nécessaire d'expliciter lorsqu'on a une symétrie totale soit de variables soit de valeurs. Enfin dans la section 5 nous ferons quelques tests de performance.

1. Ces travaux sont partiellement financés par le projet RNTL OAdymPPaC [OAD 01].

2. Notions sur les symétries et l'algorithme SBDS

2.1. *Définitions des symétries*

Si la notion de symétrie est intuitivement comprise de tous, en donner une définition précise et sans ambiguïté est plus compliqué. Nous tenterons dans cette section de donner des définitions à la fois rigoureuses et utilisables dans la pratique pour justifier par la suite les différentes techniques d'élimination de symétries. Nous nous sommes inspirés, entre autres, des travaux de [PEA 02].

Définition 2.1 (CSP) *Un problème de satisfaction de contrainte est défini par le triplet :*

$$\mathcal{P} = (V, D, C)$$

où C représente l'ensemble des contraintes sur les variables $V = \{v_1, \ldots, v_n\}$ et $D = \bigcup_{i=1\ldots n} D_i$, où les D_i représentent les domaines de chaque variable.

Notation : On notera $Aff(V, D)$ l'ensemble des fonctions de type $V \rightarrow D$. Chacune de ces fonctions représente une valuation des variables dans leur domaine respectif : $\forall f \in Aff(V, D), \forall v \in V, \exists d \in D,\ f(v) = d$.

Définition 2.2 *L'ensemble $Bij(Aff(V, D))$ représente l'ensemble de toutes les bijections sur l'ensemble $Aff(V, D)$. Cet ensemble de fonctions forme un groupe par rapport à la composition.*

Exemple : Soit $\mathcal{P} = (V = \{v_1, v_2\}, D = \{1, 2\} \cup \{3, 4\}, C)$, alors $Aff(V, D)$ contient quatre éléments f_1, f_2, f_3 et f_4 tels que :

$$\begin{aligned} f_1(v_1) &= 1, \quad f_1(v_2) = 3; \\ f_2(v_1) &= 1, \quad f_2(v_2) = 4; \\ f_3(v_1) &= 2, \quad f_3(v_2) = 3; \\ f_4(v_1) &= 2, \quad f_4(v_2) = 4. \end{aligned}$$

$Bij(Aff(V, D)) = \{\sigma : \{f_1, f_2, f_3, f_4\} \rightarrow \{f_1, f_2, f_3, f_4\} | \sigma\ bijective\}$

Ceci nous permet de donner une définition générale d'une symétrie :

Définition 2.3 (fonction symétrique d'un CSP) *Une fonction symétrique d'un CSP $\mathcal{P} = (V, D, C)$ est une fonction $\sigma \in Bij(Aff(V, D))$, telle que pour toute solution f de $\mathcal{P}$, $\sigma(f)$ soit aussi une solution de $\mathcal{P}$. On dit que $\sigma(f)$ est la symétrie de f.*

Cette définition permet de considérer toutes les fonctions qui préservent les solutions et donc, si on est capable de les expliciter, on peut se contenter de ne chercher qu'une des deux solutions symétriques pour chaque fonction (nous verrons plus tard que la façon dont on fait ce choix est très importante lorsqu'on cherche à casser plusieurs symétries). Cependant si on se contente de cette définition, le nombre de symétries à étudier peut-être immense, car elle concerne toutes les permutations des solutions, donc si on a s solutions à notre problème on aura $s! - 1$ symétries possibles en dehors de l'identité. De plus la plupart de ces symétries ne peuvent être explicitées qu'une fois les solutions connues. Dès lors il semble évident que l'on a besoin de se restreindre aux symétries facilement exprimables à partir des données initiales de $\mathcal{P}$. Parmi elles les symétries de variables, de valeurs, et la composition des deux offrent un grand nombre de possibilités d'études.

Définition 2.4 (symétrie de variable) *On a une symétrie de variable σ_π dans V s'il existe une fonction symétrique de variable dans $Bij(Aff(V, D))$ telle que : $\sigma_\pi(f) = f \circ \pi$, où π est une permutation dans V.*

Exemple : Soit $\mathcal{P} = (V = \{v_1, v_2, v_3\}, D = \{1, 2, 3\} \cup \{2, 3, 4\} \cup \{3, 4, 5\}, C)$, soit σ_π une fonction symétrique de variable de $\mathcal{P}$, avec $\pi = \{v_1 \to v_2, v_2 \to v_1, v_3 \to v_3\}$, supposons que la fonction $f \in Aff(V, D)$ soit solution de $\mathcal{P}$, définie telle que :

$$f(v_1) = 2;\ f(v_2) = 3;\ f(v_3) = 5,$$

alors la fonction $g \in Aff(V, D)$, telle que $g = \sigma_\pi(f) = f \circ \pi$, sera définie telle que :

$$g(v_1) = f(v_2) = 3;\ g(v_2) = f(v_1) = 2;\ g(v_3) = f(v_3) = 5,$$

et sera solution de $\mathcal{P}$.

Définition 2.5 (symétrie de valeur) *On a une symétrie de valeur σ_π dans D s'il existe une fonction symétrique de valeur dans $Bij(Aff(V, D))$ telle que : $\sigma_\pi(f) = \pi \circ f$, où π est une permutation sur D.*

Exemple : Soit $\mathcal{P} = (V = \{v_1, v_2, v_3\}, D = \{1, 2, 3\} \cup \{2, 3, 4\} \cup \{3, 4, 5\} = \{d_1, d_2, d_3\} \cup \{d_2, d_3, d_4\} \cup \{d_3, d_4, d_5\}, C)$, soit σ_π une fonction symétrique de valeur de $\mathcal{P}$, avec $\pi = \{d_1 \to d_1, d_2 \to d_3, d_3 \to d_2, d_4 \to d_5, d_5 \to d_4\}$, supposons que la fonction $f \in Aff(V, D)$ soit solution de $\mathcal{P}$, définie telle que :

$$f(v_1) = 2;\ f(v_2) = 3;\ f(v_3) = 5,$$

alors la fonction $g \in Aff(V, D)$, telle que $g = \sigma_\pi(f) = \pi \circ f$, sera définie telle que :

$$g(v_1) = \pi(f(v_1)) = 3;\ g(v_2) = \pi(f(v_2)) = 2;\ g(v_3) = \pi(f(v_3)) = 4,$$

et sera solution de $\mathcal{P}$.

Parmi ces symétries de variables et de valeurs on distingue un certain type de symétries qui portent sur un ensemble de variables ou/et de valeurs. Elles permettent, dans la pratique, de réduire le nombre de symétries à exprimer, par exemple si $\{v_1, v_2, v_3\}$ sont trois variables symétriques deux à deux, on ne sera pas obligé d'expliciter les six fonctions symétriques résultantes (voir 3).

Notation : Soit V' un sous-ensemble des variables V, on appellera Π l'ensemble des permutations sur V et $\Pi_{V'}$ l'ensemble des permutations de V qui laisse invariant l'ensemble $V_{|V'}$.

Définition 2.6 (symétrie totale de variable) *Si $\forall \pi_{V'} \in \Pi_{V'}$, la fonction $\sigma_{\pi_{V'}} \in Bij(Aff(V,D))$, de la forme $\sigma_{\pi_{V'}}(f) = f \circ \pi_{V'}, \forall f \in Aff(V,D)$, est une fonction symétrique, alors on a une symétrie totale par rapport à V'.*

Exemple : La contrainte de cardinalité, $card(n, v, V)$, contraint n variables dans V à prendre la valeur v. On a une symétrie totale par rapport à V si tous les domaines initiaux sont égaux.

Notation : Soit D' un sous-ensemble des valeurs de D ($D' = \bigcup_{i=1...n} D'_i$, où $D'_i \subset D_i, \forall i$), on appellera $\Pi^{D'}$ l'ensemble des permutations sur D qui laisse invariant l'ensemble $D_{|D'}$.

Définition 2.7 (symétrie totale de valeur) *Soit D' un sous-ensemble de D, si $\forall \pi_{D'} \in \Pi^{D'}$, la fonction $\sigma_{\pi_{D'}} \in Bij(Aff(V,D))$, de la forme $\sigma_{\pi_{D'}}(f) = \pi_{D'} \circ f$, $\forall f \in Aff(V,D)$, est une fonction symétrique, alors on a une symétrie totale par rapport à D' sur l'ensemble V des variables.*

Exemple : (**Schur**) Il faut placer n jetons numérotés $1, 2, \cdots, n$ dans 3 boîtes de telles sortes que 2 jetons numérotés i et 2*i ne soit pas dans la même boîte et que 3 jetons numérotées i, j et i+j non plus. Si on formalise les jetons par les variables dont leur domaine $D = \{1, 2, 3\}$ représente les 3 boîtes alors on a une symétrie totale de valeur par rapport à cet ensemble.

Définition 2.8 (symétrie variable/valeur) *On a une symétrie variable/valeur sur le couple (V, D) s'il existe une fonction symétrique variable/valeur $\sigma_{\pi_v \pi_d}$ dans l'ensemble $Bij(Aff(V,D))$ de la forme : $\sigma_{\pi_v \pi_d}(f) = \pi_d \circ f \circ \pi_v$, où π_d est une permutation sur D et π_v sur V.*

Notation : Soit f une solution de $\mathcal{P}$, soit $v \in V$ et $f(v) = d$, on notera $\sigma(v = d)$ la contrainte symétrique par σ de la contrainte $v = d$. De même $\sigma(v \neq d)$ représentera la contrainte symétrique de $v \neq d$.

Exemple : Soit σ_π une symétrie de variable, soit f une solution, alors $\forall v \in V$, la contrainte $\sigma_\pi(v = d)$ est équivalente à la contrainte $\pi(v) = d$.

Dans le cas général la spécification de la symétrie d'une contrainte d'affectation n'est pas facile car elle peut dépendre de la solution, i.e pour une affectation $v \to d$ possible dans $\mathcal{P}$, $\sigma(v = d)$ devrait dépendre de la solution à laquelle appartient cette affectation. En effet on pourrait avoir deux solutions f et g telles $f(v) = g(v) = d$ et $\sigma(f)(v) \neq \sigma(g)(v)$, on aurait alors $\sigma_f(v = d) \neq \sigma_g(v = d)$. L'algorithme d'élimination de symétries que nous allons présenter ne considère que les symétries qui ne donnent qu'une unique image d'une affectation de variable quelque soit la solution à laquelle cette affectation appartient. On remarquera que les symétries de variables et de valeurs vérifient une telle propriété et l'expérience prouve qu'elle recouvre une classe importante des symétries.

Définition 2.9 (Casser une symétrie) *On dit qu'une symétrie σ d'un problème $\mathcal{P}$ est cassée si $\forall f$ solution de $\mathcal{P}$, $\nexists g$ telle que : g solution de $\mathcal{P}$ et $\sigma(f) = g$.*

2.2. *Symmetry Breaking During Search (SBDS)*

Cette méthode a été proposée par R. Backofen et S. Will [BAC 98], puis développée par Ian Gent et Barbara Smith [GEN 00]. L'idée directrice est de rajouter des contraintes dynamiquement, pendant la résolution, afin que les symétries des solutions issues du point de choix courant soient supprimées. Elle vérifie les propriétés suivantes :

1) toutes les symétries données par l'utilisateur sont cassées ;

2) l'heuristique du parcours de l'arbre de choix est préservée.

Pour la suite nous avons besoin d'introduire des nouvelles notations afin de formaliser l'arbre de recherche. Soit $\mathcal{P}$ un problème CSP, C un ensemble de contraintes, on notera $\mathcal{P}' = \mathcal{P} \cup C$ un problème CSP, équivalent au problème $\mathcal{P}$ auquel on a ajouté les contraintes de C. On modélisera chaque nœud de l'arbre de recherche par le triplet $N = (\mathcal{P}, \Sigma, C_{aff})$, où Σ est l'ensemble des symétries de $\mathcal{P}$ données dans le problème, et C_{aff} l'ensemble des contraintes d'affectation posées depuis la racine. Ainsi le premier nœud de notre arbre de recherche sera de la forme $N_0 = (\mathcal{P}, \Sigma, \{\})$, Σ contenant l'ensemble des symétries données par l'utilisateur. L'algorithme pour développer l'arbre de recherche est donné figure 1.

L'arbre de recherche est donc un arbre binaire (voir figure 2), la première branche pose $v = d$ et cherche les symétries restantes parmi les symétries initiales et la deuxième branche pose $v \neq d$ et un ensemble de contraintes conditionnelles C_b. Ces dernières contraintes permettent de ne pas retomber sur un ensemble d'affectations symétriques de la première branche. Dans l'étape 4 on doit tester toutes les symétries de Σ afin de savoir si elles restent des symétries pour $\mathcal{P}^a$. Donc pour chaque symétrie $\sigma \in \Sigma$ on calcule l'ensemble C^a des affectations symétriques de C^a_{aff}, *1) Initialisation : Nœud_courant=* N_0 *;*

2) Nœud_courant= $(\mathcal{P}, \Sigma, C_{aff})$ *;*

3) on choisit une affectation possible dans $\mathcal{P}$ *:* $v \to d$ *;*

4) $\mathcal{P}^a = \mathcal{P} \cup \{v = d\}$ *;* $\Sigma^a \subset \Sigma$*, tel que* $\forall \sigma \in \Sigma^a$*,* σ *symétrie de* $\mathcal{P}^a$ *;* $C^a_{aff} = C_{aff} \cup \{v = d\}$ *;* $N^a = (\mathcal{P}^a, \Sigma_a, C^a_{aff})$ *;*

5) Nœud_courant= N^a*, on continue en* $2)$ *avec ce nouveau nœud ;*

6) une fois ce sous-arbre complètement développé on revient en arrière : Nœud_courant= $(\mathcal{P}, \Sigma, C_{aff})$ *;*

7) $C^b = \{\bigwedge_{aff \in C_{aff}} \sigma(aff) \Rightarrow \sigma(v \neq d)\}_{\sigma \in \Sigma}$ *;* $\mathcal{P}^b = \mathcal{P} \cup \{v \neq d\} \cup C^b$ *;* $N^b = (\mathcal{P}^b, \Sigma, C_{aff})$ *;*

8) Nœud_courant= N^b*, on continue en* $2)$ *avec ce nouveau nœud.*

Figure 1. *Algorithme SBDS*

$C^a = \{\sigma(aff)\}_{aff \in C^a_{aff}}$, si $\mathcal{P}^a \cup C^a$ conduit par propagation à une insatisfaisabilité, on retire σ, sinon on la garde.

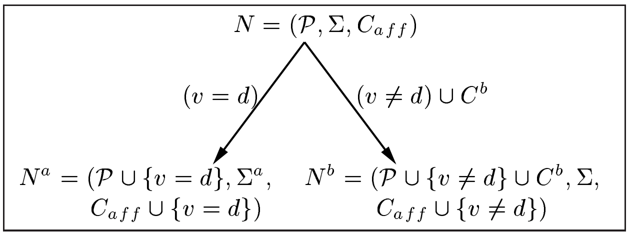

Figure 2. *Un nœud de l'arbre de recherche lorsqu'on applique la SBDS*

On peut voir assez facilement que toutes les symétries exprimées et données au solveur seront bien éliminées. Mais on peut parfois atteindre assez vite une saturation du nombre de contraintes à poser. Pour remédier à cela on peut ne s'intéresser qu'à un sous-ensemble des symétries [MCD 02], ou utiliser les propriétés de groupe des symétries [GEN 02], permettant de ne pas avoir à expliciter toutes les symétries pour ensuite les traiter une à une, ce qui est coûteux en temps de calcul. Certaines techniques d'élimination dynamique de symétries, au lieu de poser des contraintes, font des comparaisons entre le point de choix courant et ceux déjà explorés [FAH 01], les limitations viennent alors de la capacité et de la gestion de la mémoire.

3. Implantation en GNU Prolog

Comme nous l'avons dit plus haut la SBDS a l'avantage de ne pas être dépendante de la méthode de parcours de l'arbre de recherche (first_fail, most_constrained, random, ...), elle n'intervient qu'à chaque point de choix. Il nous a donc semblé intéressant de construire une fonction d'élimination de symétrie, dans le solveur de contraintes GNU Prolog [DIA 01], la plus simple et la plus transparente possible pour l'utilisateur. Nous aborderons dans un premier temps le problème de la spécification des symétries que doit fournir l'utilisateur et ensuite l'implantation de l'algorithme décrite à la section 2.2.

3.1. *La spécification des symétries*

Pour spécifier une symétrie on a besoin d'associer à chaque affectation possible (un couple (Var, Val)) une affectation symétrique. Le prédicat *calcule_sym/3* prenant en entrée le nom du prédicat de la symétrie, défini par l'utilisateur, et le couple (Var, Val) fournira en sortie le couple $(VarSym, ValSym)$ symétrique.

Exemple : Prenons l'exemple des 8 reines, $V = \{v_1 \ldots v_8\}$ représente les colonnes du damier, $D = [1 \ldots 8]$. Supposons que l'utilisateur a spécifié la symétrie verticale *vert*, échangeant la colonne 1 avec la 8, la 2 avec la 7, etc..., et on veut connaître l'affectation symétrique de $v_1 = 1$. On fait alors un appel à la fonction implantée `calcule_sym(vert,(1,1),(VarSym,ValSym))` qui donnera le résultat $(VarSym, ValSym) = (8, 1)$.

L'écriture du prédicat de symétrie peut soit être faite directement par l'utilisateur soit, dans certains cas, peut être générée par un prédicat *generateur_sym/2*.

Exemple : Reprenons le cas des 8 reines, le prédicat de la symétrie verticale s'écrira :

```
vert((Var,Val),(VarSym,Val)):-
        VarSym is 9-Var.
```

On a également une symétrie de rotation de $+90°$, *rot90* ($\forall i$, $v_i = j$ a pour image, par $rot90$, $v_j = 9 - i$) qui s'écrira :

```
rot90((Var,Val),(Val,ValSym)):-
        ValSym is 9-Var.
```

On notera que Var représente l'indice de la variable, i.e sa position dans la liste donnée par l'utilisateur, il est donc crucial que cette liste corresponde bien au modèle. De plus toutes les symétries doivent être de la forme *nom_sym((Var,Val),(VarSym, ValSym)*. Par contre il n'est pas nécessaire de traîter le cas où un couple reste invariant.

Exemple : Soit $V = \{v_1 \ldots v_n\}$ un ensemble de variables, supposons qu'il existe une symétrie de variable entre v_1 et v_2, on posera alors seulement ces deux prédicats :

```
sym((1,Val),(2,Val)).
sym((2,Val),(1,Val)).
```

Certaines symétries peuvent être laborieuses à écrire ou dans certains cas on peut éprouver le besoin de les paramétrer (par exemple en fonction du nombre de variables). C'est dans ce but qu'une fonction génératrice de symétries a été implantée.

Exemple : Soit $V = \{v_1 \dots v_n\}$, on veut construire une symétrie totale de variable sur l'ensemble $\{v_1, v_2, v_3\}$ on écrira generateur_sym([sym_var([1,2,3])],Liste_Sym) qui générera tous les prédicats spécifiant cette symétrie, par exemple *sym_var_1_2/2*, symétrie entre la variable 1 et 2.

Le prédicat *sym_var(L)* indique au générateur de symétries que l'on a une symétrie de variable sur L, *Liste_Sym* contient l'ensemble des noms des prédicats des symétries générées (plusieurs symétries sont générées car on a une symétrie de variable entre v_1 et v_2, v_1 et v_3, etc...). L'utilisateur peut créer lui-même un tel prédicat, qui générerait par exemple toutes les symétries pour le problème des *n-reines*. A terme il serait également utile de créer un prédicat générant le groupe entier des symétries à partir de celles fournies par l'utilisateur.

3.2. *La nouvelle fonction de labeling*

Une fois que l'utilisateur a spécifié tous ces prédicats de symétrie il n'a plus qu'à rassembler tous leurs noms dans une liste utilisée comme argument de la fonction de labeling `fd_labeling_sbds(ListeVariables,ListeSymetries,ListeOptions)`.

Dans la liste des options on peut spécifier, comme dans un labeling normal de GNU Prolog, l'ordre d'instanciation des variables et des valeurs. Cette fonction va ensuite appeler la fonction principale d'énumération dont on donne une version très simplifiée (sans certaines optimisations entre autres) :

```
sbds([],_,_,_):-!.
sbds(Lvar,Lsym,Laff,LOptions):-
    % On choisit une affectation selon les méthodes
    % passées dans LOptions
      choix_affectation(Lvar,Var,Val,LOptions),
      !, % On veut contrôler nous même le retour arrière
      (
      % Affectation
        Var#=Val,
      % On retire Var de la liste des variables
        retirer_liste(Lvar,Var,Lvar_nv),
      % On ajoute la nouvelle affectation dans la liste
        ajouter_liste(Laff,(Var,Val),Laff_nv),
      % Réduction des symétries superflues après affectation
        reduction_ensemble_symetries(Laff_nv,Lsym,Lsym_nv),
      % On descend dans l'arbre de recherche
        sbds(Lvar_nv,Lsym_nv,Laff_nv,LOptions)
      ; % Retour arrière après l'échec d'une affectation
        Var##Val,
      % On pose les contraintes pour casser les symétries
        casser_symetries((Var,Val),Lsym,Laff),
      % On essaye une nouvelle affectation
        sbds(Lvar,Lsym,Laff,LOptions)
      ).
```

La disjonction ' ;' permet de créer un point de choix dans l'arbre de recherche afin de rajouter dans la deuxième branche les contraintes d'éliminations. La coupure après *choix_affectation* interdit tout retour arrière sur ce prédicat, dans le cas contraire on perdrait les contraintes rajoutées dynamiquement.

Le prédicat *reduction_ensemble_symetries* calcule pour chaque symétrie l'affectation symétrique de l'affectation courante et teste si elle est consistante avec l'affectation courante.

Dans la deuxième partie de la disjonction on pose les contraintes nécessaires avant d'essayer une nouvelle affectation de la variable courante. Comme nous l'avons vu dans le section 2.2, on a besoin de poser les contraintes conditionnelles $sym(Laff) \Rightarrow (VarSym\#\#ValSym)$.

```
casser_symetries(_,[],_).
casser_symetries((Var,Val),[Sym|LSym],Laff):-
    % calcule l'affectation symétrique de (Var,Val) par Sym
      calcule_sym(Sym,(Var,Val),(VarSym,ValSym)),
    % calcule l'affectation symétrique de Laff
    % LCtes est l'équivalent de LaffSym sous forme de contraintes
      affectation_sym(Sym,Laff,LaffSym,LCtes),
    % Nombre d'affectation dans l'état courant
      length(Laff,Nb_aff),
    % La nouvelle variable contraintes Nb_Ctes_true aura pour valeur
    % le nombre de contraintes de LCtes vraies
      fd_cardinality(LCtes,Nb_Ctes_true),
    % Si toutes les contraintes symétriques de Laff sont vraies
    % alors on peut retirer la valeur ValSym à VarSym
      (Nb_Ctes_true#=Nb_aff)#==>(VarSym##ValSym),
      casser_symetries((Var,Val),LSym,Laff).
```

LCtes est de la forme *[VarSym1#=ValSym1,...]*, on utilise ensuite une variable intermédiaire *Nb_Ctes_true* car la contrainte d'implication *#==>* ne travaille que sur des booléens. La contrainte *fd_cardinality* est une contrainte globale GNU Prolog, c'est une des rares contraintes non ISO et la conversion pour un autre solveur de contraintes reste assez simple.

4. Discussion sur quelques optimisations possibles

L'un des problèmes principal de la SBDS vient de la quantité de contraintes à rajouter et du nombre de symétries à considérer. Comme nous l'avons déjà dit à la section 2.2 plusieurs solutions ont été étudiées. Par exemple on peut se contenter de n'analyser qu'un sous-groupe de symétries comme il est décrit dans [MCD 02] où on constate qu'il y a, en fonction du problème et de la technique de sélection des symétries, un nombre optimal de symétries qui rend le temps de résolution minimal. Il est cependant difficile de connaître ce nombre et les symétries ne sont que partiellement cassées. Une deuxième technique proposée dans [GEN 02] utilise les propriétés de groupe des permutations en se servant du programme *GAP (Groups, Algorithms* *and Programming)* [Gap 02] et qui donne des résultats très intéressants. En effet elle permet, entre autres, de ne pas expliciter toutes les symétries, le groupe complet étant généré à partir de quelques unes de ces fonctions, ce qui permet d'en manipuler beaucoup sans dépasser les capacités de stockage. De plus la réduction de l'ensemble des symétries lorsqu'on descend dans l'arbre de recherche se fait algébriquement, donc de façon beaucoup moins coûteuse en temps. Comme il l'est évoqué par Justin Pearl dans [PEA 02] on peut s'appuyer sur les propriétés de l'algorithme SBDS pour trouver des heuristiques réduisant le nombre de symétries qu'il est nécessaire de spécifier. C'est dans cette optique que nous allons démontrer que l'on peut réduire le nombre de symétries à spécifier pour casser une symétrie totale de variable ou de valeur.

Les symétries totales sont composées de plusieurs fonctions de symétries, par exemple pour une symétrie totale sur 3 variables $\{v_1, v_2, v_3\}$ nous avons 5 permutations possibles en plus de l'identité : $p_1 = \{v_1 \to v_2, v_2 \to v_1\}$, $p_2 = \{v_1 \to v_3, v_3 \to v_1\}$, $p_3 = \{v_1 \to v_2, v_2 \to v_1\}$, $c_1 = \{v_1 \to v_2, v_2 \to v_3, v_3 \to v_1\}$ et $c_2 = \{v_1 \to v_3, v_3 \to v_2, v_2 \to v_1\}$. Pour un ensemble de n variables il existe $n! - 1$ permutations possibles, l'algorithme exige a priori de toutes les traiter. Pourtant on remarque dans la pratique que casser uniquement les permutations deux à deux (notées p_i dans l'exemple), suffit à casser toutes les symétries, i.e il n'est pas nécessaire de spécifier les c_i.

Démonstration : Soit un problème $\mathcal{P} = \{V, C, D\}$, ayant une symétrie totale de variable par rapport au sous-ensemble $V' \subset V$, avec $|V'| > 2$, Σ l'ensemble des symétries de variable sur V' et $\Sigma_2 \subset \Sigma$ l'ensemble des symétries de variable ne permutant que deux variables. Il faut démontrer que les symétries contenues dans Σ_2 suffisent à casser toutes les symétries sur Σ, donc que les contraintes conditionnelles posées par les symétries n'appartenant pas à Σ_2 sont redondantes. Soit un nœud $N = \{\mathcal{P}_n, \Sigma_n, C_{aff}\}$ de l'arbre de recherche, soit $c \in \Sigma$ telle que $c \notin \Sigma_2$, soit π_c la permutation de variable correspondante ($\forall f\ solution,\ c(f) = f \circ \pi_c$), soient $v \in V'$, $d \in D$, on considère l'affectation $v \to d$. On se place après le retour arrière sur la contrainte $v = d$, on pose alors la contrainte $v \neq d$ et la contrainte conditionnelle : $C_c = (\{\bigwedge_{aff \in C_{aff}} c(aff)\} \Rightarrow c(v \neq d))$. Supposons que $\pi_c(v) = w$, $w \in V'$, alors $c(v \neq d) \equiv w \neq d$. Deux cas sont possibles :

1) w n'a pas encore été affectée. La symétrie $p_{vw} \in \Sigma_2$, permutant les variables v et w, appartient à Σ_n car elle laisse invariant C_{aff} et reste une symétrie de $\mathcal{P}_n$. De plus C_{aff} restant invariant par cette fonction, la condition $\{\bigwedge_{aff \in C_{aff}} p_{vw}(aff)\} = \{\bigwedge_{aff \in C_{aff}} aff\} = C_{aff}$ est vraie. Donc la contrainte $p_{vw}(v \neq d)$ est posée, i.e $w \neq d$. L'existence de la symétrie p_{vw} rend donc la contrainte conditionnelle C_c inutile ;

2) w a déjà été affectée. Si $w \neq d$ alors C_c est évidement inutile. Supposons à présent que $w = d$ (donc $\{w = d\} \in C_{aff}$), soit $x \in V'$ telle que $c^2(v) = c(w) = x$. A nouveau deux cas se présentent :

a/ x n'est pas encore affectée alors, à l'instar du cas 1), la permutation p_{vx} impliquera que l'on pose la contrainte $x \neq d$, donc la contrainte $c(w = d)$ est fausse et la condition de la contrainte C_c est fausse aussi, C_c est inutile encore une fois ;

b/ De même si x est affectée alors soit $x \neq d$ et C_c est toujours inutile, soit $x = d$. On continue ainsi de suite par itération ($c^3(v)\ldots$), mais comme c est une permutation sur un ensemble fini de variable il existe un entier i tel que $c^i = id$ et donc $c^i(v) = v$, et lorsqu'il s'agira d'étudier, comme dans les premières itérations, le cas $c^i(v) = d$ (i.e $v = d$), on sera en contradiction avec la contrainte posée lors du retour arrière : $v \neq d$.

$\square$

Ce résultat permet de limiter le nombre de fonctions de symétries. En effet pour casser une symétrie totale de variable sur l'ensemble V de cardinalité n, nous n'avons plus besoin que de C_n^2 fonctions, i.e le nombre de permutations deux à deux dans V. Pour $n = 5$ par exemple on passe de $n! - 1 = 119$ fonctions à $C_n^2 = 10$ fonctions. On a un résultat similaire pour les symétries totales de valeurs, la démonstration est plus immédiate que pour le cas des variables et ne sera pas développé ici par faute de place. On peut généraliser ces résultats en disant qu'il n'est pas nécessaire de traiter une symétrie si, pour tous les points de choix où elle apparaît, la contrainte conditionnelle associée est soit fausse soit déjà vérifiée par d'autres symétries. Ce résultat peut sembler trivial mais il nous a permis de justifier lors de nos expérimentations, des cas où des symétries étaient cassées sans qu'on ait eu besoin de les spécifier. De plus nous avons pu constater que cette condition était fortement dépendante de l'ordre dans lequel on affectait les variables, il sera intéressant dans le futur de mieux cerner ces cas de figures afin de minimiser la taille de l'ensemble des symétries exprimées.

5. Expérimentations

Nous avons commencé une série de tests sur plusieurs exemples connus en programmation par contraintes. Ils ont tous été exécutés sur un Pentium IV, 2Ghz, 256Mo de Ram. On utilise la version 1.2.16 de GNU Prolog. Dans les différents tableaux suivants la méthode *normale* signifie que l'on a utilisé la méthode de labeling de GNU Prolog *fd_labeling* (lorsqu'on utilise notre méthode SBDS implantée mais que l'on ne définit aucune symétrie, le programme se servira de *fd_labeling*).

Le problème des *n-reines* a été évoqué plus haut, il existe 7 symétries géométriques (verticale, horizontale, les deux diagonales et les trois rotations à +90°, +180° et +270°) liées à la configuration du damier. Dans le tableau 1 on peut comparer les deux labeling lorsqu'on cherche toutes les solutions. Nous pouvons observer une réelle réduction de l'arbre de recherche avec environ 5 fois moins d'échecs pour 8 fois moins de solutions trouvées en 3 fois moins de temps dans le cas de la méthode SBDS par rapport à un labeling normal. Ceci peut laisser supposer qu'il reste un certain nombre d'optimisation à faire, entre autre sur le temps d'éxécution. Pour l'instant nous sommes à une implantation méta-programmée permettant un portage plus facile vers d'autres types de solveurs. Cependant pour amiélorer ces performances on envisage d'utiliser les interfaces en C proposées par GNU Prolog, entre autres pour le choix des valuations et le calcul des permutations.

Méthode	n	Nb Sol	Echecs	Temps Cpu (s)
Normal	8	92	416	< 0,001
	12	14 200	146 102	0,7
	14	365 596	4 198 220	23
	16	147 722 512	150 027 440	964
SBDS (sym. géo)	8	12	166	< 0,001
	12	1 787	29 992	0,38
	14	45 752	791 492	8,7
	16	1 846 955	28 171 408	350

Tableau 1. *Les n-reines*

Le carré magique consiste à remplir un tableau nxn avec les n^2 premiers entiers de sorte que chaque colonne, chaque ligne et chaque diagonale aient la même somme. Outre les 7 symétries géométriques il existe des symétries liées aux équations arithmétiques, i.e, en posant une variable par case, il existe des permutations de variable laissant invariantes certaines équations. On voit dans le tableau 2 que le nouveau labeling élague beaucoup l'arbre de recherche. Dans la dernière ligne l'ensemble des symétries spécifiées représente un sous-ensemble des symétries géométriques et arithmétiques (2 géométriques et 3 arithmétiques) et la stratégie du labeling s'appuie sur une heuristique ad hoc, portant sur l'ordre dans lequel les variables sont affectées et qui dans le cas présent élimine les 7 symétries géométriques. Ce type de stratégie rejoint celle évoquée dans la section 4 mais reste encore à formaliser dans des cas plus généraux. Son idée directrice est que si une affectation a la même image par deux symétries différentes et si ces deux symétries sont toujours présentes lorsque l'affectation a lieu alors casser l'une revient à casser l'autre.

Méthode	n	Nb Sol	Echecs	Temps Cpu (s)
Normal	4	7040	609 766	10,7
SBDS (sym. géo)	4	880	99 465	1,7
SBDS (sym. partielles)	4	440	42 838	0,8

Tableau 2. *Carré magique*

Nous avons également testé le problème du golfeur sociable. Notre modélisation supprime le symétrie à l'intérieur d'un groupe et entre les groupes dans une semaine. Il reste alors deux types de symétries, celles entre les joueurs (symétries de variables) et celles entre les semaines (symétries de valeurs) qui sont faciles à supprimer avec notre fonction de labeling. Par contre la composition entre ces deux symétries n'a pas été traité car la gestion de groupe reste encore à implanter. C'est pourquoi il nous reste encore un certain nombre de solutions symétriques. Cependant on arrive à obtenir des résultats satisfaisants pour certaines instances difficiles. $7-3-9$ par exemple a été trouvé en moins d'une heure et $5-3-7$ s'obtient très rapidement mais contient encore beaucoup de symétries.

6. Conclusion

Nous avons présenté une nouvelle fonction de labeling pour GNU Prolog qui implante la méthode SBDS. Les premières performances ne sont pas encore très satisfaisantes mais cela est du essentiellement au fait que la fonction de labeling est métaprogrammée. On a vu que des améliorations étaient possibles aussi bien au niveau de l'algorithme que de la programmation. Cependant on est en possession d'un outil aisément adaptable et qui s'interface facilement avec des outils de débogages. L'implantation proposée est libre de droit, tout comme GNU Prolog, et accessible sur le site `http://contraintes.inria.fr/~arnaud/sbds/index.html`. Par ailleurs se pose la question de la détection de symétries. Ce problème est loin d'être évident car elles ne sont pas forcément visibles juste en "regardant" la modélisation. Il n'existe pas beaucoup de travaux portant sur cette question mais on peut imaginer dans certain cas une détection automatique des symétries en regardant par exemple la stabilité de l'ensemble des contraintes lorsqu'on permute les couples (variable,valeur). Une autre approche, développée dans le cadre du projet OADymPPaC[OAD 01], consiste à développer des outils visuels pouvant guider l'utilisateur à la fois sur la détection de symétries et sur les choix à prendre sur diverses heuristiques (nombre de symétries à considérer, lesquelles utiliser, l'ordre des variables et de valeurs, etc…).

7. Bibliographie

[BAC 98] BACKOFEN R., WILL S., « Excluding Symmetries in Concurrent Constraint Programming », *Workshop on Modeling and Computing with Concurrent Constraint Programming*, , 1998, http ://citeseer.nj.nec.com/backofen98excluding.html.

[DIA 01] DIAZ D., CODOGNET P., « Design and Implementation of the GNU Prolog System », *Journal of Functional and Logic Programming*, vol. 6-10, 2001.

[FAH 01] FAHLE T., SCHAMBERGER S., SELLMANN M., « Symmetry Breaking », WALSH T., Ed., *Contraint Programming, CP'01*, Springer-Verlag, 2001, p. 93–107, LNCS 2239.

[Gap 02] GAP GROUP, « GAP - Groups Algorithms and Programming, Version 4r3 », , 2002, http ://www.gap-system.org.

[GEN 00] GENT I. P., SMITH B. M., « Symmetry Breaking in Constraint Programming », HORN W., Ed., *14th European Conference on Artificial Intelligence*, Berlin, Allemagne, 24 et 25 Juillet 2000, ECCAI and German infortics society, IOS press, p. 599–603.

[GEN 02] GENT I. P., HARVEY W., KELSEY T., « Groups and Constraints : Symmetry breaking during search », *Constraint Programming, CP'02*, Ithaca, NY, USA, 2002, Springer, p. 415–431, LNCS 2470.

[MCD 02] MCDONNALD I., SMITH B., « Partial Symmetry Breaking », *Constraint Programming, CP'02*, Ithaca, NY, USA, 2002, Springer, LNCS 2470.

[OAD 01] OADYMPPAC, « Tools for Dynamic Analysis and Debugging of Constraint Programs », 2001, RNTL project. http ://contraintes.inria.fr/OADymPPaC.

[PEA 02] PEARSON J., « Reducing the number of constraints in symmetry breaking during search », Draft, 2002.