Suppression de symétries pour les méthodes rétro-prospectives

Guillaume Richaud, Hadrien Cambazard, Narendra Jussien

Suppression de symétries pour les méthodes rétro-prospectives

Guillaume Richaud **Hadrien Cambazard** **Narendra Jussien**

École des Mines de Nantes – LINA CNRS 2729 – 4 rue Alfred Kastler – BP 20722
F-44307 Nantes Cedex 3, France
Guillaume.Richaud@emn.fr Hadrien.Cambazard@emn.fr Narendra.Jussien@emn.fr

Résumé

Les techniques de suppression de symétries développées pour la programmation par contraintes ont pour objectif d'améliorer l'efficacité des méthodes de résolution en réduisant drastiquement la taille de l'espace de recherche. Parallèlement, les méthodes de résolution récentes, telles que les méthodes rétro-prospectives, permettent de s'attaquer à des problèmes de plus en plus grands, que ce soit par la taille des domaines ou le nombre de variables considérées, et dont l'espace de recherche augmente bien sûr lui aussi. Il paraît donc intéressant de faire profiter les méthodes rétro-prospectives des améliorations permises par les techniques de suppression de symétries qui ont cependant été développées pour un cadre plus classique d'exploration. Dans cet article, nous proposons donc un algorithme hybridant une méthode rétro-prospective (`decision-repair`) et une technique générique de suppression de symétries (SBDS). Nous présentons plusieurs approches pour exprimer et traiter efficacement dans ce cadre les contraintes liées à la suppression des symétries. De premiers résultats expérimentaux valident la démarche.

1 Introduction

Les problèmes de satisfaction de contraintes possèdent souvent des symétries qui ont pour effets de grossir artificiellement la taille de l'espace de recherche. Ainsi, une méthode de résolution, confrontée à un problème possédant de nombreuses symétries, peut perdre beaucoup de temps à visiter des solutions partielles équivalentes. L'objectif des méthodes de suppression de symétries est précisément de réduire la taille de l'espace de recherche en exploitant les relations d'équivalence entre affectations (totales ou partielles).

Une limite de ces méthodes est que la majorité d'entre elles repose sur le parcours arborescent (en profondeur d'abord) de l'espace de recherche. Or, des méthodes de résolution récentes [8], de type rétro-prospectif, utilisent les informations déduites par la propagation (les explications) afin de guider la phase de recherche. L'exploration de l'espace de recherche n'est plus arborescente et la façon dont interagissent les méthodes de suppression de symétries avec l'algorithme de choix n'est pas compatible avec le fonctionnement de ce type de méthodes d'exploration. Notre objectif est donc de mettre au point une méthode permettant de supprimer les configurations symétriques dans le cadre d'une recherche rétro-prospective.

1.1 Techniques de suppression de symétries

Un problème de satisfaction de contraintes est un triplet (X, D, C), où X est l'ensemble des variables $\{x_1, \ldots, x_n\}$ défini sur $D = \{d_1, \ldots, d_n\}$. Pour chaque variable x_i, l'ensemble $D(x_i)$ noté d_i représente l'ensemble fini des valeurs possibles de la variable x_i. $C = \{c_1, \ldots, c_p\}$ est l'ensemble des contraintes portant sur les variables X. Chaque contrainte $c_i \in C$ est définie par $Var(c_i)$ un sous-ensemble de X et par $R(c_i)$ un sous ensemble de D correspondant aux tuples autorisés.

Les symétries dans un CSP sont des bijections transformant des affectations en d'autres affectations équivalentes au sens de la consistance. Elles sont donc définies par :

Définition 1 *Pour un CSP L défini par ensemble de contraintes C, une symétrie de L est une fonction bijective f telle que pour toute affectation (complète ou partielle) A, A satisfait les contraintes C si et seulement si f(A) satisfait les contraintes de C.*

Ainsi, si (C, A) est un nogood alors $(C, f(A))$ l'est aussi. On peut donc déduire, grâce aux parties déjà visitées, qu'il est inutile d'explorer certaines parties de l'espace de recherche.

Les symétries d'un problème forment un groupe [5] et permettent de définir des classes d'équivalences entre les affectations symétriques. Ainsi les techniques de suppression de symétries permettent de limiter l'exploration à un seul représentant de chaque classe d'équivalence durant l'exploration de l'espace de recherche. Elles peuvent être mises en œuvre à différents moments de la résolution d'un problème, aussi bien durant la modélisation que durant la phase de recherche.

Les méthodes statiques de suppression des symétries interviennent durant la phase de modélisation du problème. La difficulté est de trouver les contraintes permettant de supprimer l'ensemble des configurations à l'exception d'un unique représentant par classe d'équivalence. L'inconvénient de ce genre de solution est qu'elle impose a priori ce représentant. Ainsi une configuration qui aurait du être visitée en premier peut être éliminée lors de la suppression des symétries. Ces méthodes ont le défaut de modifier la forme du problème initial ce qui interfère avec les choix de parcours dans l'arbre et peut ralentir la recherche des solutions.

Pour éviter ce genre de désagrément, les méthodes dynamiques ont été mises au point. Plutôt que de tenter de trouver une modélisation sans symétries, les méthodes dynamiques interviennent durant la phase de recherche en interdisant la visite de configurations équivalentes à celles déjà testées, que ce soit par l'ajout de contraintes (Symmetry Breaking During Search – SBDS) [6, 1] ou par la vérification de l'inclusion de la configuration courante par rapport à celles déjà explorées (Symmetry Breaking by Dominance Detection – SBDD) [3, 4]. La suppression dynamique nécessite donc de stocker les configurations déjà rencontrées. Les techniques d'enregistrement proposées reposent sur un parcours en profondeur d'abord afin de pouvoir compacter au mieux les informations. Ceci n'est pas compatible avec les méthodes rétro-prospectives qui ne s'appuient pas sur une recherche arborescente mais sur l'exploration plus libre de l'espace de recherche.

1.2 Méthodes rétro-prospectives

Les méthodes rétro-prospectives ont pour objectif d'exploiter les enseignements tirés de la propagation des contraintes (prospection) pour réparer les choix et orienter l'exploration de l'arbre de recherche (rétrospection). La réparation est rendue possible grâce à l'utilisation d'explications [14] qui indiquent quelles sont les décisions responsables du retrait d'une valeur d'un domaine, et par extension quels sont les choix qui ont mené à un domaine vide. Ainsi la méthode générique *decision-repair* [8] analyse les informations décrivant les affections responsables de l'échec d'une affectation afin de remettre en cause les variables réellement impliquées tout en permettant l'utilisation simultanée d'une méthode prospective. Plusieurs méthodes de ce type ont été proposées, mais afin de nous placer dans un cadre plus global, nous allons nous concentrer sur cette méthode générique.

```
decision-repair(C)
(1)  début
(2)    C_D ← Ensemble des contraintes de décisions initiales ;
(3)    répéter
(4)      si condition d'échec vérifiée alors
(5)          retourner chec ;
(6)      sinon
(7)          C' ← φ(C ∪ C_D) ;
(8)          S ← Inference(C') ;
(9)          si S = "pas de solution" alors
(10)             k ← conflit expliquant l'échec ;
(11)             C_D ← réparation(C_D, k, Γ) ;
(12)         finsi
(13)         si S = "solution" alors
(14)             retourner C' ;
(15)         sinon
(16)             C_D ← extension(C_D, Γ) ;
(17)         finsi
(18)     finsi
(19)   jusqu'à faux
(20) fin
```

FIG. 1 – `decision-repair`

L'idée (l'algorithme est présenté dans la figure 1) est d'appliquer, après chaque nouvelle affectation, l'algorithme de filtrage ϕ afin de détecter et de pouvoir corriger au plus tôt (de manière non nécessairement chronologique) une instanciation partielle qui ne pourra pas être complétée en une solution valide. Ainsi, à chaque itération la valeur d'une variable est fixée en ajoutant la contrainte de décision $(V_i = a)$, puis l'algorithme de filtrage est appliqué. Trois cas se présentent alors :

- Le filtrage détecte que les décisions prises ne permettent pas de trouver une solution. On calcule alors l'ensemble des contraintes qui sont entrées en conflit (le paramètre k) et en fonction des conflits déjà rencontrés (paramètre Γ) on remet en cause une des contraintes de décision. On répare ainsi les mauvaises décisions faites précédemment.
- Une solution a été trouvée, dans ce cas là on peut renvoyer l'ensemble des décisions.
- Des contraintes de décision doivent encore être

ajoutées car l'espace de recherche est encore trop grand. On continue donc l'exploration de l'espace de recherche.

Lorsque la propagation se fait grâce à un algorithme d'arc-consistance ces trois cas correspondent respectivement aux situations où un des domaines est vide, où le domaine de chaque variable est réduit à une valeur et où le domaine de chaque variable contient au moins une valeur et qu'il existe une variable dont le domaine contient au moins deux valeurs.

Le type de *réparation* (par l'intermédiaire de la fonction `réparation()`) détermine la structure de voisinage. Par exemple si on choisit de revenir sur la décision la plus récente, présente dans le conflit expliquant l'échec, on retrouve un algorithme de type Dynamic Backtracking.

L'intérêt des méthodes rétro-prospectives est que la partie rétrospective et la partie prospective coopèrent. La partie prospective raccourcit la longueur des branches de l'arbre de recherche en permettant de détecter plus tôt les inconsistances, tandis que la partie rétrospective exploite les échecs détectés afin de revenir sur des points de choix pertinents et limiter les phénomènes de thrashing. Ainsi, ces méthodes se distinguent des méthodes classiques par le fait que la recherche n'est pas vue comme l'exploration d'un arbre mais comme la pose et la remise en cause des contraintes de décision (affectations). La recherche ne repose donc plus sur le retour arrière mais sur la réparation des instanciations partielles inconsistantes. Cependant, le fait que les méthodes rétro-prospectives soient obligées de calculer des explications peut les pénaliser par rapport à des techniques plus simples.

1.3 Problématique

Même si au cours de ces dernières années plusieurs solutions ont été proposées pour supprimer les symétries, aucune ne semble vraiment adaptée aux méthodes rétro-prospectives. Même le cadre que Focacci et Milano [4] mettent en place et qui permet la suppression des configurations équivalentes à partir des nogoods (Global Cut Seed) découverts durant l'exploration de l'espace de recherche nécessite une recherche arborescente pour une mise à jour efficace des contraintes.

Cependant l'hybridation *méthode de suppression de symétries-méthodes rétro-prospectives* pourrait avoir plusieurs avantages. Le principal argument en sa faveur est que le calcul des explications rend l'exploration de l'espace de recherche plus coûteuse. Ainsi le gain apporté par le retrait des configurations équivalentes devrait rapidement dépasser le surcoût nécessaire à leur suppression. Il peut donc être intéressant d'utiliser, conjointement aux méthodes rétro-prospectives, des techniques de suppression de symétries. Nous avons décidé de nous concentrer sur SBDS car elle semble être la seule technique générique qui permette de vraiment tirer parti des explications et de la phase de propagation puisqu'elle repose sur l'ajout de contraintes. Nous allons donc présenter notre hybride SBDS-Decision-repair et les solutions choisies pour les contraintes de suppression des symétries.

2 Suppression des symétries par SBDS

SBDS a été décrite par Gent et Smith [6] avec la volonté de proposer une méthode de suppression des symétries qui respecte autant que possible les choix des heuristiques et qui puissent supprimer n'importe quelle forme de symétrie. Le principe est donc d'ajouter une nouvelle contrainte après chaque retour arrière afin de supprimer les branches équivalentes non encore visitées. Ainsi si l'ajout de l'affectation $(var = val)$ à l'affectation partielle en cours A mène à un échec, la contrainte conditionnelle $\sigma(A) \Rightarrow \sigma(var \neq val)$ est ajoutée à l'ensemble des contraintes du problème afin de supprimer la symétrie σ.

SBDS semble être la méthode la plus adaptée pour être utilisée avec une méthode rétro-prospective. En effet, elle est une des seules techniques, avec SBDD, à permettre la suppression de n'importe quel type de symétrie sans interférer avec le parcours de l'espace de recherche. De plus, SBDS se montre plus efficace que SBDD grâce à l'information ajoutée au problème par l'intermédiaire des contraintes de suppression de symétries. En effet lorsque seule une partie des variables est utilisée pour la recherche de solution, les contraintes supplémentaires permettent de propager les conséquences à l'ensemble des variables présentes dans les contraintes [12]. Or la propagation est un élément important des méthodes rétro-prospectives. Il semble donc plus intéressant d'utiliser une méthode de suppression de symétries qui puisse en tirer partie.

3 Méthodes rétro-prospective et suppression de symétries

Hybrider SBDS et `decision-repair` revient à poser des contraintes de suppression de symétries après chaque remise en cause de l'instanciation partielle. Cependant, le fait que les algorithmes rétro-prospectifs soient capables d'identifier les affectations responsables de l'échec d'une instanciation peut être exploité afin d'augmenter l'efficacité des contraintes posées. En effet dans les méthodes SBDS classiques, la

recherche et la suppression des branches symétriques portent sur la totalité de l'instanciation courante. Or, souvent seules quelques affectations de l'instanciation peuvent être responsables de l'échec (nogood).

La suppression des configurations équivalentes repose sur la suppression des configurations symétriques aux nogoods et chaque contrainte devrait donc permettre d'éliminer une plus grande portion de l'espace de recherche. On augmente ainsi leur capacité de filtrage tout en limitant la taille des configurations considérées [10].

3.1 Decision-repair et SBDS

Par rapport à l'algorithme `decision-repair` de base, les parties correspondant au filtrage et à l'exploitation des nogoods restent inchangées. Seule la partie utilisée pour la détection d'un échec est modifiée afin de permettre l'ajout des contraintes. La fonction `supprime-GE`(k, G) (voir l'algorithme de la figure 2) est chargée d'ajouter les contraintes empêchant d'explorer des affectations partielles équivalentes aux nogoods déjà testés. Ainsi, lorsqu'un conflit k est découvert, `supprime-GE` calcule l'ensemble de ses symétriques par le groupe des symétries du problème G et ajoute la contrainte les interdisant à l'ensemble des contraintes du problème. Il suffit de rajouter à ligne 11 de l'algorithme : $C \leftarrow C \cup$ `supprime-GE`(k, G) afin de rajouter la nouvelle contrainte à l'ensemble des contraintes du problème.

```
supprime-GE(k, G)
(1)  début
(2)      % Ajout de la nouvelle contrainte
(3)    C_GE ← CalculSymetrique(k, G) ;
(4)    Liste − GE ← Liste − GE ∪ {GE} ;
(5)    Ajoute la contrainte C_GE au problème ;
(6)      % Retrait des contraintes subsumées
(7)    pour tout C'_GE ∈ Liste − GE faire
(8)      si k et C'_GE sont inconsistants faire
(9)          Retire C_GE'
(10)     finsi
(11)   finpour
(12)   Filtre() ;
(13) fin
```

FIG. 2 – La méthode `supprime-GE`(k, G)

Notre algorithme nous permet de trouver une unique solution par classe d'équivalence.

Preuve : Supposons que l'on trouve deux solutions S_1 et S_2 appartenant à la même classe d'équivalence. S_1 est découverte en premier, on pose alors une contrainte interdisant l'ensemble des $\sigma(S_1)$ avec $\sigma \in G$. On trouve ensuite S_2, donc il n'existe aucun $\sigma \in G$ tel que $\sigma(S_1) = S_2$ or S_1 et S_2 sont dans la même classe d'équivalence. Contradiction.
Supposons que l'on ne trouve aucune solution pour une classe d'équivalence E. Soit S_1 une solution de E. Elle a donc été éliminée par une contrainte de suppression de symétrie, autrement dit $\exists B \in E', \exists \sigma \in G\ \sigma(S_2) = S_1$, donc $\sigma(S_2) \in E$ et puisque G est un groupe $\sigma^{-1} \in G$ d'où $\sigma^{-1}(\sigma(S_2)) = S_2 \in E$. On a donc trouvé une solution de E. Contradiction.

3.2 Générateur de contraintes de symétries.

Le choix porte sur la forme des contraintes et sur l'algorithme de filtrage associé. La solution de base consiste à poser une contrainte par tuple symétrique, on obtient alors beaucoup de contraintes simples et sans aucune relation entre elles. Cette méthode découle directement de la définition de SBDS [6] : $\sigma(A) \wedge (var \neq val) \rightarrow \sigma(var \neq val)$. L'avantage est qu'elle est très simple à mettre en œuvre et que les contraintes générées sont assez naturelles. Cependant on est confronté ici au problème lié au nombre de symétries. En effet le solveur peut très vite se retrouver ralenti par le nombre croissant de contraintes à considérer. De plus la mise à jour de contraintes (retrait lorsqu'une contrainte plus générale est découverte) est difficilement réalisable dans le cadre d'un parcours non-arborescent. En effet, lorsqu'on effectue un parcours en profondeur d'abord, chaque remise en cause de l'instanciation entraîne un retour en arrière. On remonte donc dans l'arborescence et toutes les contraintes qui ont été posées durant la visite du sous-arbre peuvent être retirés. Or avec les méthodes rétro-prospectives il n'y a plus de retour en arrière. La mise à jour de ces contraintes dans le cadre d'une recherche non arborescente demanderait de les comparer deux à deux pour déterminer les inclusions.

Nous avons donc décidé de poser une contrainte par classe d'équivalence de manière à limiter leur nombre, augmenter leur capacité de filtrage et faciliter la mise à jour. La contrainte de suppression C_{GE} est définie par :

- $Var(C_{GE})$ qui correspond à l'ensemble des variables symétriques aux variables sur lesquelles portent les choix responsables de l'échec.
- $R(C_{GE}) = \Delta(k)$ avec $\Delta(k)$ l'ensemble des tuples symétriques à k.

Par exemple pour un CSP avec V=$\{X_1, X_2, X_3\}$ et D=$\{1, 2\}$. Supposons que nous ayons déjà visité les af-

fectations $(1,1,1)$ et $(1,1,2)$ avec comme responsables des échecs $(X_1 = 1, X_3 = 1)$ et $(X_1 = 1, X_3 = 3)$. On a donc posé les contraintes C_{GE}(G,$(X_1 = 1, X_3 = 1)$) et C_{GE}(G,$(X_1 = 1, X_3 = 2)$). Au moment de remettre en cause la décision $(X_1 = 1)$ nous allons rajouter C_{GE}(G,$(X_1 = 1)$). On obtient alors deux ensembles interdisant le même sous-espace. Ce genre de redondance est inutile et pourrait même pénaliser la résolution puisqu'elle force à considérer des contraintes qui filtrent moins. De plus, lorsque nous sommes confrontés à des problèmes de taille importante, l'accumulation de contraintes peut saturer la mémoire.

Il faut donc vérifier avant chaque nouvel ajout si la contrainte n'inclut pas des contraintes posées précédemment afin de les supprimer. Pour vérifier l'inclusion il faut que d'une part la nouvelle contrainte soit générée à partir d'une affectation comportant moins de variables (condition nécessaire) et qu'au moins un tuple de la classe d'équivalence ayant servi à générer la nouvelle contrainte soit inclus dans un tuple de la classe d'équivalence ayant servi à générer la contrainte à supprimer (condition nécessaire et suffisante). Autrement dit, il suffit de vérifier si k est consistant pour chacune des contraintes de suppression de symétries déjà posée.

> **Preuve :** Soient t_1 et t_2 des tuples des classes d'équivalences E_1 et E_2. Les contraintes C_1 et C_2 générées à partir de E_1 et E_2 sont définies par $R(C_1) = \Delta(t_1)$ et $R(C_2) = \Delta(t_2)$ avec $\Delta(k) = \{t \in D / \exists \sigma \in G, \sigma(k) \subseteq t\}$.
> Si $t_1 \subseteq t_2$ alors $\Delta(t_2) = \{t \in D / \exists \sigma \in G, \sigma(t_1) \subseteq \sigma(t_2) \subseteq t\} \subseteq \Delta(t_1)$. Autrement dit la contrainte C_1 est plus générale que la contrainte C_2.
> Réciproquement, si $R(C_2) \subseteq R(C_1)$ alors pour chaque $t \in \Delta(t_2)$ il existe $\sigma \in G | \sigma(t_1) \subseteq t$. Or $t_2 \in \Delta(t_2)$, donc il existe $\sigma \in G | \sigma(t_1) \subseteq t_2$, de plus $\sigma(t_1) \in E_1$ (classe d'équivalence). Donc il existe un tuple de E_1 inclus dans un tuple de E_2.

3.3 Filtrage

On utilise un algorithme générique (GAC2001) [2, 15] qui permet de réaliser l'arc-consistance généralisée sur le sous-ensemble $\Delta(k)$ correspondant à l'ensemble des tuples interdits par la classe d'équivalence. GAC consiste à trouver un tuple support (composé de valeurs présentes dans le domaine de chacune des variables et admis par la contrainte) pour chaque valeur du domaine d'une des variables. Lorsqu'une valeur n'a pas de support on peut la retirer puisqu'elle ne pourra pas être prolongée en une affectation totale.

En outre, on a intérêt à regrouper plusieurs tuples dans une même contrainte afin d'augmenter sa capacité de filtrage. Par exemple, considérons le CSP (V,D,C) avec $V = \{v_1, v_2\}$, $D(v_1) = D(v_2) = \{1, 2\}$ et $C = \{c_1, c_2\}$ avec $Var(c_1) = Var(c_2) = V$, $R(c_1) = \{(1,1), (2,2)\}$ et $R(c_2) = \{(1,2), (2,1)\}$. Lorsque l'on va réaliser l'arc-consistance aucune valeur ne va être retirée puisqu'elles ont toutes un support alors que si on avait considéré une seule contrainte $R(c_{12}) = R(c_1) \cup R(c_2)$ on aurait pu conclure immédiatement que le CSP ne possède aucune solution.

4 Un autre encodage des contraintes

Comme nos expérimentations le confirment, le défaut de la solution *une contrainte par classe d'équivalence* est qu'elle nécessite de poser un nombre, malgré tout, important de contraintes. De plus, ces contraintes portent sur un nombre limité de tuples et ont une arité élevée dès que l'on considère des symétries de variables, leur maintien est donc coûteux. Une autre approche possible est de poser une contrainte servant à stocker l'ensemble des tuples interdits. En effet, nous avons tout intérêt à regrouper au maximum les tuples interdits au sein d'une même contrainte (dans le cas où cela n'a pas d'impacts sur l'arité de la contrainte). Mais la question de la mise à jour se pose de nouveau. Notre solution consiste à poser une contrainte dans laquelle l'ensemble des tuples interdits (c'est à dire l'ensemble des configurations équivalentes aux nogoods rencontrés) est stocké à l'aide d'un automate. On considère un automate en couches. Chaque couche correspond à une variable de la contrainte. Les transitions permettent d'accéder d'une couche à l'autre et l'alphabet correspond aux valeurs des domaines de définition des variables. Ainsi les tuples autorisés par la contrainte correspondent aux mots acceptés par l'automate.

Initialement, l'automate accepte l'ensemble des tuples autorisés et, au fur et à mesure que la méthode de résolution va découvrir de nouveaux nogoods, les tuples symétriques vont être retirés du langage de l'automate. On utilise ensuite l'automate pour le filtrage. L'avantage d'utiliser un automate pour le stockage est qu'il permet de faire disparaître le problème de la mise à jour. En effet, tous les tuples interdits subsumés par de nouveaux tuples disparaissent puisque le retrait d'un mot du langage reconnu par l'automate entraîne le retrait de l'ensemble des *sur-mots*. Cependant, si on veut pouvoir rester efficace dans la gestion des tuples par l'automate, il faut pouvoir être capable de retirer des tuples de manière dynamique et minimiser le nombre d'états de l'automate de manière incrémentale.

4.1 Gestion de l'automate

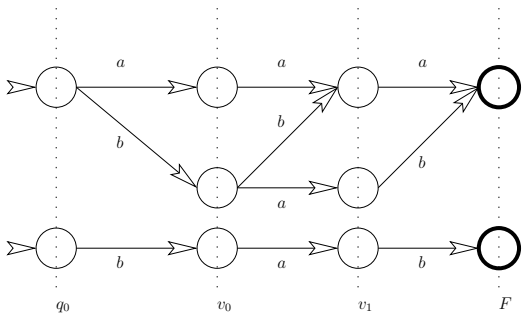

FIG. 3 – Automate acceptant les tuples {(a,a,a), (b,b,a), (b,a,b)} et automate représentant le tuple (b,a,b)

On considère l'automate $A = (Q, \Sigma, \delta, q_0, F)$, avec Q l'ensemble des états, Σ l'ensemble des valeurs possibles pour les variables considérées, $\delta : Q \times \Sigma \rightarrow Q$ la fonction de transition et $F \subseteq Q$ l'ensemble des états finaux. Le but est de maintenir de manière incrémentale l'automate des tuples valides. Autrement dit, nous devons être capables de rajouter des nogoods (retirer les mots correspondants du langage reconnu par l'automate) incrémentalement ainsi que de le réduire au fur et à mesure.

À tout instant, on dispose d'un automate A et d'un mot w, correspondant au nogood que l'on souhaite retirer (Figure 3). On procède alors en deux étapes : on retire le mot w de l'automate, puis on le minimise.

Retirer w, revient à calculer $R = A \cap \overline{w} = (\{q_0, \ldots, q_n\} \times \{p_0, \ldots, p_m\}, \Sigma, \delta, r_0, F')$ où p_i est l'état correspondant à la i^{eme} lettre de w. Donc 4 sortes d'états peuvent apparaître dans R ($\perp$ est l'état *poubelle*).

1. (q_i, p_j) qui est atteignable si et seulement si $\delta^*(q_0, w_0 w_1 \ldots w_j) = q_i$ [1].
2. $(\perp, p_j)$: l'état final n'est pas inatteignable depuis ce type d'état. Donc il est inutile de les créer car ils sont équivalents à l'état poubelle.
3. $(q_i, \perp)$ qui ont les mêmes transitions que l'état q_i de A. Pour les obtenir, il suffit donc de renommer tout les anciens états de A.
4. $(\perp, \perp)$ le nouvel état poubelle de R.

[1] δ^* étend δ tel que

$$\delta^*(q, w) = \begin{cases} \delta^*(\delta(q, x), y) & \text{si } w = xy \text{ avec } x \in \Sigma \text{ et } y \in \Sigma^* - \{\epsilon\} \\ \delta(q, w) & \text{si } w \in \Sigma \end{cases}$$

Ainsi nous n'avons pas besoin de construire l'intégralité du produit cartésien des états de A avec ceux de w parce que seuls les états du type 1 correspondent réellement à l'ajout d'un nouvel état par rapport à l'automate A. Les états de type 2, 3, 4 et de type 1 lorsque $\delta(q_{i-1}, w_m) = \perp$ n'ont pas besoin d'être considérés.

Construire (q_i, p_m) consiste à copier l'état q_i ainsi que ses transitions sortantes. On ajoute, couche par couche chaque (q_i, p_m) et ses transitions. Il y a au plus un état par couche car il y a au plus une nouvelle transition sortante par couche.

```
RemoveWord(A,w)
(1)  début
(2)     States ← initState() ;
(3)     pour i = 1 to |w| faire
(4)        NewStates ← ∅ ;
(5)        pour tout q_c ∈ States faire
(6)           q_n ← δ(q_c, w_i) ;
(7)           q ← clone(q_n) ;
(8)           δ(q_c, w_i) ← q ;
(9)           NewStates ← NewStates ∪ {q} ;
(10)       finpour
(11)       States ← NewStates ;
(12)    finpour
(13) fin
```

FIG. 4 – Retrait d'un mot du langage

Pour construire le nouvel automate, les états (q_i, p_m) sont ajoutés. C'est à dire, $|w|$ nouveaux états sont créés et pour chaque nouvel état q à la couche t, nous modifions une transition. On obtient l'algorithme de la figure 4 de complexité $O(|w|)$ pour réaliser le retrait.

Nous devons en outre, vérifier que les anciens états de A restent bien atteignables (arcs entrants) et que les nouveaux états permettent bien d'atteindre l'état final (arcs sortants).

Minimisation La minimisation passe par la détection des états équivalents. Deux états, p et q, sont dits équivalents, noté $Equiv(p, q)$, si $\forall w \in \Sigma^*, \delta^*(q, w) \in F \equiv \delta^*(p, w) \in F$. Donc $Equiv(p, q) \equiv \vec{\mathscr{L}}(p) = \vec{\mathscr{L}}(q)$ où $\vec{\mathscr{L}}(p)$ est le langage droit de l'état p, autrement dit : l'ensemble de tous les mots permettant d'atteindre l'état final à partir de l'état q.

Par conséquent, $Equiv(p, q) \equiv (p \in F \equiv q \in F) \wedge \Gamma_p = \Gamma_q \wedge (\forall a : a \in \Gamma_p \cap \Gamma_q : Equiv(\delta(p, a), \delta(q, a)))$ avec Γ_q le sous ensemble de Σ tel que $\forall x \in \Gamma_q, \delta(q, x) \neq \perp$. Ce qui signifie que deux états sont

équivalents si et seulement si, pour chaque valeur, leurs successeurs sont équivalents.

En s'appuyant sur le fait que l'automate est acyclique, on peut le minimiser de manière efficace en partant de l'état final. On utilise les propriétés suivantes pour accélérer la minimisation.

- Seuls les états appartenant à la même couche peuvent être équivalents. Donc il suffit de tester l'équivalence entre le nouvel état q_i et les états de la couche i.
- L'automate était minimal donc deux anciens états ne peuvent pas être équivalents.
- La couche n ne peut pas comporter d'états équivalents si la couche $n+1$ n'en a pas.
- Deux états équivalents sur la couche n ont l'état q_{n-1} comme prédécesseur.

L'algorithme recherche des états équivalents depuis l'état final jusqu'à l'état initial. Pour chaque nouvel état, nous construisons l'ensemble des états, *EquivStates*, qui ont les mêmes successeurs ($\Gamma_p = \Gamma_q$). Lorsque toutes les transitions sortantes ont été comparées, *EquivStates*, contient des états équivalents, et peuvent être fusionnés. Ainsi dans le pire des cas, nous avons à comparer $|w| \times d$ transitions et fusionner $|w|$ états.

On se limite, dans cet article, au cas où les nogoods considérés portent sur l'ensemble des variables de la contrainte. On se place en pratique dans un cadre plus général dans lequel les nogoods portent sur un sous-ensemble des variables de la contrainte. Si on considère un automate A, on ajoute alors, dans le pire des cas, $|A|$ états et la minimisation nécessite au plus $|A|$ fusions et $|A| \times d$ comparaisons de transitions.

4.2 Filtrage grâce à l'automate

L'algorithme de filtrage associé repose sur [11]. Il s'agit donc de construire un graphe support, c'est à dire un graphe acyclique dans lequel chaque couche correspond à une variable et chaque couple variable-valeur est représenté par les arcs allant d'une couche à l'autre. Les arcs permettent d'indiquer l'appartenance ou non de l'état à un tuple support (autrement dit, un mot du langage). Le filtrage consiste à parcourir deux fois le graphe acyclique (une fois en allant de l'état initial à l'état final et une fois en allant dans le sens inverse) afin d'éliminer les états qui n'appartiennent à aucun chemin de q_0 à F. En retirant des états on élimine des supports et lorsqu'une valeur n'a plus d'état support permettant d'aller de l'état initial à l'état final, on peut la retirer. Autrement dit on élimine ainsi les valeurs du domaine des variables.

Le problème de l'incrémentalité se pose. En effet, l'algorithme de filtrage de Pesant repose sur un automate statique alors que le nôtre évolue au cours de l'exploration. Nous devons donc le reconstruire à chaque ajout d'un nouveau nogood afin de maintenir un automate des supports valides. Cette construction se fait en $O(n \times d \times |A|)$ étapes.

5 Expérimentations

Afin d'évaluer les performances de notre algorithme nous avons choisi le problème des N-Reines ainsi que celui des Graphes Gracieux. Il s'agit d'observer le comportement de l'algorithme dans un cas simple (taille réduite et peu de symétries) afin de mettre en évidence ses principales caractéristiques. L'objectif est de voir si l'utilisation des conflits peut améliorer les performances de SBDS et valider nos hypothèses sur l'intérêt que représente la suppression de symétries pour les méthodes rétro-prospectives. Nos expérimentations sont réalisées dans le solveur de contraintes Choco/Palm (`choco-solver.net`) en hybridant notre algorithme avec la méthode rétro-prospective MAC-DBT [7] (cas particulier complet de *decision-repair*). Les expérimentations ont été réalisées sur un ordinateur à 2,4GHz et possédant 512Mo de mémoire vive.

5.1 Le problème des N-Reines

Ce problème consiste à placer N reines sur un échiquier sans qu'elles s'attaquent. Autrement dit, les reines doivent se situer sur des colonnes, lignes et diagonales distinctes. Il y a une reine par ligne de l'échiquier. On considère donc les variables $c_1, \ldots, c_n$ correspondant à la colonne sur laquelle sont placées, respectivement, les reines $1, \ldots, n$. Les variables c_n sont définies sur $\{1, \ldots, n\}$. On pose aussi les variables m_n et p_n, définies par $m_n = c_n - n$ et $p_n = c_n + n$ correspondant aux diagonales sur lesquelles est placée la reine n. Les contraintes sont `Different`(c_i, c_j), qui correspond au fait que les reines doivent être sur des colonnes différentes, ainsi que `Different`(m_i, m_j) et `Different`(p_i, p_j) qui traduisent le fait que deux reines ne peuvent pas être sur la même diagonale.

5.1.1 Symétries

Les symétries de ce problème sont des symétries géométriques. En effet l'échiquier reste inchangé par des rotations et des symétries axiales et centrales (il y en a 7, sans compter l'identité). Avec notre modélisation la symétrie axiale verticale (V) et la symétrie axiale horizontale (H) correspondent respectivement à une symétrie de valeur (puisqu'elle consiste

à permuter les valeurs) et une symétrie de variable (puisqu'elle consiste à permuter les variables). La rotation est (R) est une symétrie plus compliquée puisque le symétrique dépend de la valeur et de la variable.

5.1.2 Évaluation

Les évaluations portent sur la suppression des différentes symétries pour de méthodes prospectives (MAC) et des méthodes rétro-prospectives (MAC-DBT, MAC-CBJ) en posant une contrainte par classe d'équivalence ou avec l'utilisation d'une contrainte stockant les tuples dans un automate. Lors de nos expérimentations nous avons considéré les symétries H, V et R et les différentes combinaisons possibles [9]. Nous ne présentons ici que les résultats portant sur H et V et l'utilisation de MAC et MAC-DBT. Ils sont exprimés en ms et en nombre de retour-arrière.

Les différents tests font apparaître que pour que le parcours sans symétries soit plus rapide que le parcours normal, il faut que les calculs supplémentaires à effectuer pour supprimer les branches soient moins coûteux que le parcours lui-même. Or dans le cas de MAC le coût est très faible (le CSP est composé de contraintes binaires) il est donc difficile d'améliorer les performances (Tableau 1). De plus le surcoût en espace mémoire des contraintes de suppression de symétrie limite la taille des problèmes que l'on peut résoudre.

Si la suppression de symétries est aussi avantageuse dans le cas MAC-DBT (Tableau 2) c'est parce que le calcul des explications augmente aussi le coût de l'exploration. Ainsi, il est beaucoup plus simple de rentabiliser les contraintes d'autant plus que leur arité est limitée grâce à l'utilisation des explications de contradiction sans pour autant restreindre leur capacité de filtrage.

Cependant l'efficacité dépend des symétries considérées. Ainsi la suppression de la symétrie V est avantageuse car elle supprime la moitié des affectations à l'aide de contraintes de la même taille (arité) que l'affectation, alors que la suppression de la symétrie H, qui permet elle aussi de supprimer la moitié des affectations, utilise des contraintes d'arité double (dans le pire des cas). Ainsi les méthodes les plus efficaces, ne sont pas celles qui suppriment le plus mais celles qui permettent d'atteindre le meilleur compromis.

En effet l'intérêt de la suppression de symétries semble encore plus évident lorsqu'elle est utilisée conjointement à une méthode de recherche coûteuse. Cependant la contrainte utilisée pour supprimer les configurations équivalentes reste un élément critique. En effet le coût de la suppression des symétries dépend essentiellement du coût du maintien de ces contraintes. Les expérimentations confirment l'intérêt de l'utilisation de l'automate aussi bien en terme d'espace mémoire occupé, qu'en terme de capacité de filtrage ou d'efficacité. En effet on peut résoudre des instances plus grandes et pour les problèmes de même taille le nombre de retours-arrière ainsi que le temps sont meilleurs.

5.2 Le problème des Graphes Gracieux

Un graphe gracieux est un graphe pour lequel il existe une valuation gracieuse. Une valuation f des sommets d'un graphe à q arrêtes est gracieuse si chaque sommet a une valeur distincte dans $\{1, \ldots, q\}$ et que chaque arrête xy reliant deux sommets x et y a une valeur distincte et est égale à $|f(x) - f(y)|$. Un modèle consiste à considérer une variable v_x par sommet x et une variable d_{xy} par arête reliant deux sommets x et y. Ces variables ont pour valeur la valuation du sommet ou de l'arc correspondant, et sont donc définies sur $\{1, \ldots, q\}$. Il faut ensuite ajouter des contraintes pour s'assurer que les valeurs des arrêtes soient bien égales à la distance entre les sommets (1), et que chaque valeur ne soit affectée qu'une seule fois à l'ensemble des sommets (2) et à l'ensemble des arrêtes (3).

$$\forall d_{xy}, |v_x - v_y| = d_{xy} \quad (1)$$

$$AllDifferent(v_x, \ldots, v_n) \quad (2)$$

$$AllDifferen(d_{xy}, \ldots, d_{x'y'}) \quad (3)$$

5.2.1 Symétries

Ce modèle en lui même ne comporte pas de symétrie, hormis le fait que chaque solution peut être remplacée par son complémentaire (chaque v_x peut être remplacée par $(q - v_x)$ puisque $|(q - v_x) - (q - v_y)| = |v_x - v_y)|$). Les autres symétries dépendent du graphe choisi [13]. On considère des graphes de la forme $K_n \times P_m$, c'est à dire les graphes construits en reliant les sommets équivalents de m graphes identiques complets à n sommets. On introduit ainsi plusieurs symétries : les graphes complets sont identiques on peut donc permuter les affectations d'un graphe à l'autre. De plus les graphes sont complets, autrement dit, à l'intérieur d'un graphe tous les sommets jouent le même rôle, on peut donc les permuter. Les résultats présentés dans le tableau 4 sont ceux obtenus lorsqu'on ne considère qu'un sous ensemble du groupe des symétries.

5.2.2 Évaluation

Cette deuxième série d'expérimentations prouve encore une fois l'intérêt de la suppression de symétries

	MAC			**MAC-SBDS[VH]**		
n	Solutions	Temps	Retour-arrière	Solutions	Temps	Retour-arrière
2	0	0	1	0	16	1
3	0	0	2	0	15	1
4	2	16	6	1	31	3
5	10	47	19	3	47	8
6	4	78	50	2	93	18
7	40	141	149	12	156	47
8	92	266	520	-	-	-
9	352	485	2040	-	-	-

TAB. 1 – Résolution du problème des N-Reines avec MAC

	MAC-DBT			**MAC-DBT-SBDS[VH]**		
n	Solutions	Temps	Retour-arrière	Solutions	Temps	Retour-arrière
2	0	15	1	0	21	1
3	0	31	2	0	21	2
4	2	78	5	1	63	3
5	10	109	13	3	93	6
6	4	281	39	2	203	15
7	40	439	107	12	391	41
8	92	1704	383	24	1281	135
9	352	7462	1478	-	-	-

TAB. 2 – Résolution du problème des N-Reines avec MAC-DBT

dans le cas de l'utilisation de MAC-DBT pour la résolution. La variante MAC-DBT-SBDS (avec automate) donne de meilleurs résultats que MAC-DBT mais aussi de meilleurs résultats que MAC. Cependant, il apparaît que les meilleurs performances en terme de temps ne sont pas atteintes lorsqu'on enlève le plus de symétries. En effet, avec l'utilisation de l'automate la limitation ne porte plus sur la taille des tuples considérés mais sur le temps nécessaire à la génération de l'ensemble des tuples symétriques. Il s'agit donc de trouver un compromis entre temps de génération et nombres de symétries supprimées [9]. En fonction du but recherché on peut privilégier le temps ou le nombre de retours-arrière.

6 Conclusion et perspectives

Dans cet article, nous proposons un algorithme générique permettant d'utiliser une technique de suppression de symétries tout en utilisant une méthode de résolution rétro-prospective. En interdisant la visite des parties équivalentes de l'espace de recherche on diminue le temps nécessaire à la résolution du problème. Cependant les résultats indiquent qu'une gestion efficace des contraintes est nécessaire pour rentabiliser l'élimination des symétries. Le travail futur consistera donc à approfondir les tests, dont les premiers résultats sont encourageants, dans le cas où on utilise un automate pour stocker les configurations symétriques.

Références

[1] Rolf Backofen and Sebastian Will. Excluding symmetries in constraint-based search. In Joxan Jaffar, editor, *Principle and Practice of Constraint Programming (CP'99)*, volume 1713 of *Lecture Notes in Computer Science*, pages 73–87, Berlin, 1999. Springer–Verlag.

[2] C. Bessière, J.C. Régin, R.H.C. Yap, and Y. Zhang. An optimal coarse-grained arc consistency algorithm. In Elsevier, editor, *Artificial Intelligence*, volume 165(2), pages 165–185, 2005.

[3] Torsten Fahle, Stefan Schamberger, and Meinolf Sellmann. Symmetry breaking. *Lecture Notes in Computer Science*, 2239 :93–107, 2001.

[4] Focacci and Milano. Global cut framework for removing symmetries. In *Principle and Practice of Constraint Programming (CP'01)*, pages 77–91, 2001.

[5] Ian P. Gent, Warwick Harvey, and Tom Kelsey. Groups and constraints : Symmetry breaking du-

	MAC-DBT			**MAC-DBT-SBDS[VH] (avec automate)**		
n	Solutions	Temps	Retours-arrière	Solutions	Temps	Retours-arrière
2	0	15	1	0	78	1
3	0	31	2	0	15	1
4	2	78	5	1	62	3
5	10	109	13	3	62	6
6	4	281	39	2	125	15
7	40	439	107	12	203	35
8	92	1704	383	24	687	128
9	352	7462	1478	92	3016	452

TAB. 3 – Résolution du problème des N-Reines avec MAC-DBT

	MAC			**MAC-DBT**		**MAC-DBT-SBDS (avec automate)**		
n	Solutions	Temps	Retours-arrière	Temps	Retours-arrière	Solutions	Temps	Retours-arrière
$K_3 \times P_2$	96	1546	1153	3187	1078	8	781	106
$K_4 \times P_2$	1140	431625	123998	10666094	121917	90	149937	4486
$K_3 \times P_1$	12	32	38	234	11	2	78	3
$K_4 \times P_1$	48	172	132	266	99	6	156	18
$K_5 \times P_1$	0	1656	1122	3297	1073	0	844	148
$K_6 \times P_1$	0	53859	14238	98938	13901	0	20656	1425

TAB. 4 – Résolution du problème des Graphes Gracieux

ring search. Technical Report APES-48-2002, APES Research Group, May 2002.

[6] Ian P. Gent and Barbara M. Smith. Symmetry breaking in constraint programming. In *14th European Conference on Artificial Intelligence*, 2000.

[7] Narendra Jussien, Romuald Debruyne, and Patrice Boizumault. Maintaining arc-consistency within dynamic backtracking. In *Principles and Practice of Constraint Programming (CP 2000)*, number 1894 in Lecture Notes in Computer Science, pages 249–261, Singapore, September 2000. Springer-Verlag.

[8] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artificial Intelligence*, 139(1) :21–45, July 2002.

[9] Iain McDonald and Barbara Smith. Partial symmetry breaking. Technical Report APES-49-2002, APES Research Group, May 2002.

[10] Pedro Meseguer and Carme Torras. Exploiting symmetries within constraint satisfaction search. *Artif. Intell.*, 129(1-2) :133–163, 2001.

[11] Gilles Pesant. A regular language membership constraint for finite sequences of variables. In *Principle and Practice of Constraint Programming (CP'04)*, pages 482–495, 2004.

[12] Karen E. Petrie. Combining SBDS and SBDD. Technical Report APES-86-2004, APES Research Group, July 2004.

[13] Karen E. Petrie and Barbara M. Smith. Symmetry Breaking in Graceful Graphs. Technical Report APES-56a-2003, APES Research Group, June 2003.

[14] Thomas Schiex and Gérard Verfaillie. Nogood Recording for Static and Dynamic Constraint Satisfaction Problems. *International Journal of Artificial Intelligence Tools*, 3(2) :187–207, 1994.

[15] Zhang Yuanlin and Roland H. C. Yap. Making AC-3 an optimal algorithm. In *IJCAI*, pages 316–321, 2001.