Génération d'explications de contraintes globales par leur décompositions

A. Gontier[1], C. Prud'homme[2], C. Truchet[3]

[1] Univ. Rennes, IRISA, Rennes, France
[2] IMT-Atlantique Nantes, LS2N-CNRS, F-44307, Nantes, France
[3] Univ. Nantes, LS2N-CNRS, F-44307, Nantes, France
arthur.gontier@irisa.fr, charles.prudhomme@imt-atlantique.fr

Mots-clés : *CP, Explication, Décomposition, Contrainte globale*

Apprendre des échecs peut permettre d'améliorer la résolution de problèmes modélisés en Programmation par Contraintes. Depuis les adaptations de l'algorithme CDCL aux solveurs CP ([1],[2]), une clause expliquant un échec peut être construite dès lors que toutes les contraintes impliquées sont expliquées. L'explication d'une contrainte détermine tous les évènements responsables d'un filtrage de cette contrainte. Malheureusement, expliquer chaque contrainte est une tâche fastidieuse, souvent complexe pour les contraintes globales. Dans ce résumé, nous présentons une méthode basée sur la réécriture pour générer automatiquement des explications depuis un langage de décomposition simple et extensible. Un langage de décomposition permet de reformuler une contrainte avec un ensemble de contraintes plus simples. Les décompositions de contraintes reposent souvent sur la réification de contrainte, c'est à dire que le respect de chaque contrainte du langage est associé à une variable booléenne qui peut être utilisée dans les autres contraintes. Par exemple, un évènement de modification de borne supérieure, modélisé par la contrainte : $(x_i \leq t) \iff b_{it}$, est réifié à b_{it}. Cette variable peut ensuite être utilisée dans d'autres contraintes comme une conjonction ou une somme de booléens. La réification permet de limiter le nombre de contraintes du langage, par exemple la somme de booléens supérieure à t est obtenue en prenant la négation de la variable réifiée de la somme inférieure à t. Notre méthode a pour but de générer des explications de contraintes globales à partir de leur décomposition. Pour montrer sa validité, nous définissons le langage simple suivant :

Définition 1 (Langage de Décomposition) *Soit $i \in \mathbb{N}$, $t \in \mathbb{Z}$, x une variable entière et b une variable booléenne, potentiellement indexées. Une variable booléenne constante peut être remplacée par $\top$ ou $\bot$.*

1. $(x_i = t) \iff b_{it}$
2. $(x_i \leq t) \iff b_{it}$
3. $(\bigvee_i b_i) \iff b$
4. $(\bigwedge_i b_i) \iff b$
5. $(\sum_i b_i \leq t) \iff b$

Pour chacune des contraintes de ce langage, nous avons besoin d'explications sous la forme de règles de réécriture. Une règle de réécriture se note $\langle a \rangle \xrightarrow{R} \langle b \rangle$: le terme a est remplacé par le terme b avec la règle R. Dans notre cas, a représente un filtrage, b est la raison de ce filtrage et R est la contrainte qui a effectué ce filtrage. Le langage de décomposition peut être enrichi de nouvelles contraintes tant que leurs règles de réécriture sont connues. Pour présenter un exemple, nous définissons ci-dessous les règles correspondant aux contraintes 1. et 5. de notre langage.

Définition 2 (Règles de réécriture)

1. $\langle x_i = t \rangle \xrightarrow{R_=} \langle b_{it} \rangle$ $\quad \langle b_{it} \rangle \xrightarrow{R_=} \langle x_i = t \rangle$ $\quad \langle x_i \neq t \rangle \xrightarrow{R_\neq} \langle \neg b_{it} \rangle$ $\quad \langle \neg b_{it} \rangle \xrightarrow{R_\neq} \langle x_i \neq t \rangle$

5. $\langle b_i \rangle \xrightarrow{R^1_\Sigma} \langle \neg b \rangle \langle \neg b_j \rangle_{\forall j \neq i}$ $\quad \langle \neg b_i \rangle \xrightarrow{R^2_\Sigma} \langle b \rangle \langle b_j \rangle_{\forall j \neq i}$ $\quad \langle b \rangle \xrightarrow{R^3_\Sigma} \langle \neg b_i \rangle_{\forall i}$ $\quad \langle \neg b \rangle \xrightarrow{R^4_\Sigma} \langle b_i \rangle_{\forall i}$

Une fois ces règles définies, nous pouvons les utiliser dans un algorithme de réécriture pour générer l'explication de chacun des évènements d'une contrainte globale. Donnons un exemple de génération d'explications pour la contrainte $\textsc{AtMost}(t, X, v)$. Cette contrainte assure qu'il y a au plus t variables dans X qui prennent la valeur v. Nous la décomposons avec une contrainte d'égalité et une somme tel que :

$$\textsc{AtMost}(t, X, v) \iff \begin{cases} (x_i = v) \iff b_{iv},\ \forall i \in [1, |X|] \\ (\sum_{i \in [1,|X|]} b_{iv} \leq t) \iff \top \end{cases}$$

Expliquons l'évènement d'affectation de x_i à la valeur v. Seule la règle $R_=$ peut être utilisée ici donc l'algorithme réécrit $x_i = v$ en b_{iv}. Cette variable booléene n'existe pas dans la contrainte globale donc nous devons continuer la réécriture. Pour réécrire b_{iv}, nous interdisons de réutiliser la même règle pour ne pas avoir de boucles de réécriture. La seule règle utilisable est donc la règle R^1_Σ mais cette règle réécrit la négation de la variable réifiée et notre somme est réifiée à $\top$. Donc l'explication devient $\bot$. Cette absence d'explication est normale puisque la contrainte $\textsc{AtMost}(t, X, v)$ ne peut pas affecter une valeur à une variable. Expliquons à présent la suppression de la valeur v du domaine de x_i. L'algorithme utilise $R_\neq$ pour écrire $\neg b_{iv}$ et cette fois l'explication a la bonne variable réifiée donc l'algorithme utilise la règle R^2_Σ et l'explication devient $\top \wedge (\forall j \neq i \mid b_{jv})$. Cette explication n'est pas une explication de la contrainte $\textsc{AtMost}(t, X, v)$ car b_{jv} n'est pas une variable d'$\textsc{AtMost}(t, X, v)$. Il y a deux règles possibles pour expliquer b_{jv} : R^1_Σ et $R_=$. On sépare les deux possibilités avec une disjonction et l'explication deviens $\top \wedge ((\forall j \neq i \mid \bot) \vee (\forall j \neq i \mid x_j = v))$. On la simplifie en $(\forall j \neq i \mid x_j = v)$. L'explication signifie que l'évènement de la suppression de la valeur v d'un domaine est causée par les autres variables qui sont affectées à cette même valeur. Cette explication est bien une explication de la contrainte $\textsc{AtMost}(t, X, v)$. Nous avons implémenté un générateur capable d'utiliser ces règles et de générer ces explications. Nous l'avons utilisé pour expliquer une dizaine de contraintes globales. Pour tester leur qualité, nous avons implémenté deux d'entre elles dans le solveur expliqué Chuffed. Nous avons ensuite lancé des problèmes de la compétition MiniZinc avec ces contraintes [1]. Nous observons que les explications générés sont toujours meilleures que l'explication naïve qui consiste à expliquer par toutes les variables de la contrainte. Nous observons aussi que les explications générées ont des performances comparables à celles du solveur LCG qui décompose ces contraintes. Expliquer une contrainte globale par sa décomposition pose la question de la différence de qualité de filtrage. Quand la contrainte est algorithmiquement globale, sa décomposition permet le même filtrage et donc nos explication générées sont exactes. Dans le cas d'une contrainte opérationnellement globale, dont le filtrage est plus fort que celui de sa décomposition, nos explications peuvent être complétées par l'explication naïve sur les évènements qu'elles ne savent pas expliquer. Dans ce résumé nous présentons une méthode pour générer des explications de contraintes à partir d'un langage de décomposition. Notre générateur et les tests menés sur les problèmes de la compétition minizinc dans un solveur expliqué montre la validité de cette approche. Nous pensons que la génération d'explications est une approche prometteuse pour se munir d'un grand ensemble de contraintes expliquées.

Ces travaux sont réalisés dans le cadre du projet DeCrypt (ANR-18-CE39-0007).

Références

[1] Peter J. Stuckey. Lazy clause generation : Combining the power of SAT and CP (and mip ?) solving. In Andrea Lodi, Michela Milano, and Paolo Toth, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 7th International Conference, CPAIOR 2010, Bologna, Italy, June 14-18, 2010. Proceedings*, volume 6140 of *Lecture Notes in Computer Science*, pages 5–9. Springer, 2010.

[2] Michael Veksler and Ofer Strichman. A proof-producing CSP solver. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010.

1. `https://github.com/ArthurGontierPro/Explaining-Global-Constraints-from-Decompositions`