Améliorer les méthodes de décomposition pour le dénombrement exact de solutions

Philippe Jégou **Hanan Kanso** **Cyril Terrioux**
Aix Marseille Univ, Université de Toulon, CNRS, ENSAM, LSIS, Marseille, France
{philippe.jegou, hanan.kanso, cyril.terrioux}@lsis.org

Résumé

Le problème du dénombrement de solutions d'une instance CSP, appelé #CSP, constitue un problème extrêmement difficile qui possède de multiples applications en Intelligence Artificielle. S'il est le plus souvent résolu par des méthodes approchées, ici, nous nous focalisons sur le dénombrement exact. Nous montrons comment il est possible d'améliorer les méthodes basées sur les décompositions structurelles en améliorant la recherche d'une nouvelle solution, qui est une étape essentielle, en particulier pour de telles méthodes. De plus, si les ressources en temps ou en espace sont insuffisantes, nous montrons comment notre approche est capable de fournir une borne inférieure du nombre de solutions. Des expérimentations sur des benchmarks CSP mettent en avant l'intérêt pratique de notre approche par rapport aux meilleures méthodes de la littérature.

Ce papier est un résumé de [6].

Abstract

The problem of counting solutions in CSP, called #CSP, is an extremely difficult problem that has many applications in Artificial Intelligence. This problem can be addressed by exact methods, but more classically it is solved by approximate methods. Here, we focus primarily on the exact counting. We show how it is possible to improve the methods based on structural decomposition by offering to enhance the search for a new solution which is a critical step for counting, particularly for such methods. Moreover, if the resources in time or in space are insufficient, we show that our approach is still able to provide a lower bound of the result. Experiments on CSP benchmarks show the practical advantage of our approach w.r.t. the best methods of the literature.

This is a summary of [6].

1 Contexte

Dénombrer les solutions d'une instance CSP (problème #CSP) ou les modèles d'une instance SAT (problème #SAT) constituent des problèmes extrêmement difficiles qui possèdent de multiples applications. Ces problèmes étant connus pour être #P-complet [10], leur résolution d'un point de vue pratique s'effectue souvent par le biais de méthodes approchées offrant généralement une borne inférieure du nombre de solutions. Cependant, en exploitant certaines propriétés de l'instance, il est envisageable de proposer des méthodes exactes qui soient efficaces en théorie comme en pratique [2, 3, 5, 4]. C'est à ce type d'approche que nous nous intéressons ici et plus particulièrement à la méthode #BTD [4] qui dénombre les solutions d'une instance CSP à l'aide d'une décomposition arborescente. Bien qu'ayant fait preuve de résultats pratiques intéressants, cette méthode peut être améliorée, comme nous le montrons dans la section suivante.

2 Améliorer #BTD

Une instance de CSP (pour Problème de Satisfaction de Contraintes) est définie par la donnée d'un triplet (X, D, C), où $X = \{x_1, \ldots, x_n\}$ est un ensemble de n variables, $D = \{d_{x_1}, \ldots, d_{x_n}\}$ est un ensemble de domaines finis de taille au plus d, et $C = \{c_1, \ldots, c_e\}$ est un ensemble de e contraintes. Chaque contrainte c_i est un couple $(S(c_i), R(c_i))$, où $S(c_i) = \{x_{i_1}, \ldots, x_{i_k}\} \subseteq X$ définit la *portée* de c_i, et $R(c_i) \subseteq d_{x_{i_1}} \times \cdots \times d_{x_{i_k}}$ est une *relation de compatibilité*. La structure d'une instance CSP est donnée par un hypergraphe, appelé *hypergraphe de contraintes*, dont les sommets correspondent aux variables et les arêtes aux portées des contraintes. Une affectation d'un sous-ensemble de X est dite *cohérente* si toutes les contraintes portant sur ce sous-ensemble sont satisfaites. Une *solution* est une affectation cohérente de toutes les variables.

Certaines méthodes, comme #BTD, exploitent la

notion de *décomposition arborescente de graphes* [7] pour identifier des sous-problèmes indépendants.

Définition 1 *Une* décomposition arborescente *d'un graphe $G = (X, C)$ est un couple (E, T) où $T = (I, F)$ est un arbre (I est un ensemble de nœuds et F un ensemble d'arêtes) et $E = \{E_i : i \in I\}$ une famille de sous-ensembles de X, telle que chaque sous-ensemble (appelé cluster) E_i est un nœud de T et vérifie : (i) $\cup_{i \in I} E_i = X$, (ii) pour chaque arête $\{x, y\} \in C$, il existe $i \in I$ avec $\{x, y\} \subseteq E_i$, et (iii) pour tout $i, j, k \in I$, si k est sur un chemin de i à j dans T, alors $E_i \cap E_j \subseteq E_k$. La largeur d'une décomposition est égale à $max_{i \in I}|E_i|-1$. La largeur arborescente dite* tree-width *w de G est la largeur minimale pour toutes les décompositions arborescentes de G.*

Dans #BTD, chaque sous-problème est défini vis-à-vis d'un cluster E_i (ce sous-problème porte sur toutes les variables présentes dans la descendance de E_i) et de l'affectation courante sur les variables de l'intersection de E_i avec son père. Pour chaque sous-problème, #BTD va procéder à une exploration systématique et dénombrer le nombre de solutions locales qu'il possède avant de les combiner pour établir le nombre de solutions du problème global. Ce faisant, #BTD obtient une complexité en temps en $O(n.e.d^{w^+ +1})$ (avec w^+ la largeur de la décomposition utilisée), qui d'un point de vue théorique, est une des meilleures possibles. D'un point de vue pratique, bien qu'ayant obtenu des résultats intéressants, cette méthode peut être améliorée. En effet, pour chaque sous-problème, #BTD s'attache à dénombrer toutes ses solutions. Pour autant, à ce stade-là, il ne possède aucune garantie que les solutions locales ainsi trouvées pourront s'étendre sur le reste du problème pour former des solutions du problème global. Si ces solutions locales ne participent pas à des solutions globales, #BTD aura effectué un calcul inutile des plus coûteux compte tenu de la complexité théorique du problème #CSP. Aussi, ici, afin d'éviter cela, nous proposons de modifier #BTD de sorte que, lorsque l'algorithme trouve une solution locale pour un sous-problème donné, celui-ci s'assure d'abord que cette solution participe à une solution globale avant de dénombrer toutes les solutions de ce sous-problème. Si cette idée peut paraître simple et naturelle, elle modifie significativement le comportement de l'algorithme initial et nécessite, au final, de définir un nouvel algorithme, appelé #EBTD (pour Enhanced BTD), et un nouveau cadre formel avec notamment la notion de *goods structurels partiels.*

Théorème 1 *#EBTD a une complexité en temps en $O(n.(re+ns).d^{w^+ +1})$ pour une complexité en espace en $O(n.s.d^s)$ avec s la taille de la plus grande intersection entre deux clusters et $r = max_{c \in C}|S(c)|$.*

Par ailleurs, si les ressources en temps ou en espace sont insuffisantes, par construction, #EBTD est en mesure de fournir une borne inférieure du nombre de solutions, qui, en pratique, s'avère souvent non nulle.

Nous avons comparé expérimentalement #EBTD à Toulbar2/#BTD [4] (c'est-à-dire #BTD), Cachet [8], c2d [2], relsat [1] et sharpSAT [9] qui constituent les méthodes de référence pour la résolution des problèmes #CSP ou #SAT. Il en résulte que #EBTD résout plus d'instances (à savoir 908 instances sur les 1 059 instances considérées) que sharpSAT (899 instances), Toulbar2/#BTD (877 instances), Cachet (608 instances), relsat (618 instances) ou c2d (382 instances) tout en se révélant plus rapide sur la plupart des instances.

3 Conclusion

Nous avons proposé un nouvel algorithme, appelé #EBTD, pour calculer le nombre exact de solutions d'une instance CSP. Si la complexité de cet algorithme est similaire à celle de #BTD, en pratique, il se révèle bien plus efficace et résout plus d'instances et plus rapidement que les méthodes de l'état de l'art.

Plusieurs extensions sont envisageables comme notamment l'exploitation de décompositions adaptées au problème #CSP, ou l'utilisation de cette approche pour calculer de meilleures approximations du nombre de solutions.

Références

[1] R. Bayardo and J. Pehoushek. Counting Models Using Connected Components. In *AAAI*, pages 157–162, 2000.

[2] A. Darwiche. New Advances in Compiling CNF into Decomposable Negation Normal Form. In *ECAI*, pages 328–332, 2004.

[3] R. Dechter and R. Mateescu. The Impact of AND/OR Search Spaces on Constraint Satisfaction and Counting. In *CP*, pages 731–736, 2004.

[4] A. Favier, S. de Givry, and P. Jégou. Exploiting Problem Structure for Solution Counting. In *CP*, pages 335–343, 2009.

[5] V. Gogate and R. Dechter. Approximate Solution Sampling (and Counting) on AND/OR search spaces. In *CP*, pages 534–538, 2008.

[6] P. Jégou, H. Kanso, and C. Terrioux. Improving Exact Solution Counting for Decomposition Methods. In *ICTAI*, pages 327–334, 2016.

[7] N. Robertson and P.D. Seymour. Graph minors II : Algorithmic aspects of treewidth. *Algorithms*, 7 :309–322, 1986.

[8] T. Sang, F. Bacchus, P. Beame, H. Kautz, and T. Pitassi. Combining Component Caching and Clause Learning for Effective Model Counting. In *SAT*, 2004.

[9] M. Thurley. sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP. In *SAT*, pages 424–429, 2006.

[10] L. Valiant. The Complexity of Computing the Permanent. *Theoretical Computer Science*, 8 :189–201, 1979.