Problème de satisfaction de contraintes n-aires : une étude expérimentale

Mihaela Butaru, Zineb Habbas

Problème de satisfaction de contraintes n-aires Une étude expérimentale

Mihaela Butaru **Zineb Habbas**

LITA, Université de Metz
Ile du Saulcy, F-57045 METZ Cedex
butaru@sciences.univ-metz.fr zineb@iut.univ-metz.fr

Résumé

Depuis quelques années, la communauté I.A. affiche un intérêt croissant pour les Problèmes de Satisfaction de Contraintes (CSPs), cadre de formalisation puissant pour de nombreux problèmes du monde réel et généralement NP-complets. Les méthodes de recherche de solutions pour les CSPs de type Forward Checking (FC) ont été intensivement étudiées dans le cas binaire. Ces derniéres années ont vu arriver de nombreux résultats relatifs aux CSPs n-aires. Notre objectif général est de développer une bibliothèque d'algorithmes de recherche arborescente distribués capables de s'attaquer à des CSPs généraux suite à l'analyse préalable des algorithmes n-aires d'arc-consistance et de résolution, qui permettrait d'en sélectionner ceux qui sont les plus efficaces. Plutôt que d'utiliser des solveurs existants, nous avons développé notre propre solveur pour mener des études expérimentales sur des CSPs n-aires.

1 Introduction

Les problèmes de satisfaction de contraintes (CSP pour Constraint Satisfaction Problem) sont au cœur de nombreuses applications en intelligence artificielle et en recherche opérationnelle. Dans le contexte des CSPs sur des domaines finis, les contraintes n-aires peuvent souvent être formulés exclusivement à l'aide de contraintes binaires, grâce à l'équivalence bien connue [12] entre les problèmes n-aires et binaires qui résout le problème de recherche d'algorithmes n-aires mais, en pratique, la complexité en temps rajoutée est importante. Mais certains problèmes restent inhéremment n-aires dans leur formulation (le Golomb ruler). Dans cette perspective, nous nous intéressons à la définition d'un algorithme de recherche complet par consistance en avant, permettant de résoudre un CSP n-aire directement sans transformation préalable. Nous renvoyons à [10] pour une comparaison des définitions de consistance n-aire utilisées et à [1] où les auteurs proposent une généralisation formelle du FC binaire en FC n-aire. Dans notre travail de recherche, nous avons développé un cadre pour mener des études expérimentales des algorithmes d'arc-consistance n-aires et d'observer les comportements des algorithmes de résolution n-aires. Concrètement, nous proposons un solveur n-aire composé de trois variantes d'algorithmes d'arc-consistance n-aire et de cinq versions de FC n-aire. Dans cet article, nous présentons les résultats séquentiels et des premiers résultats de résolution parallèle avec OpenMP obtenus avec notre solveur pour deux problèmes académiques, Golomb ruler et lemme de Schur.

2 Notions de base

Définition 1 (due à Montanari [11]) *Un* **Problème de Satisfaction de Contraintes (CSP)** $\mathcal{P}$ *est un quadruplet* $\mathcal{P} = (X, D, C, R)$*, où :*

- X *est un ensemble fini de* n variables $\{X_1, ..., X_n\}$.
- D *est un ensemble de* n domaines $\{D_1, ..., D_n\}$. *Chaque domaine* D_i *est l'ensemble fini de* valeurs *pour la variable* X_i.
- C *est un ensemble fini de* m contraintes $\{C_1, ..., C_m\}$. *Chaque contrainte* C_p *est définie par l'ensemble de variables* $Vars(C_p) = \{X_{p_1}, ..., X_{p_{n_p}}\} \subseteq X$.
- R *est un ensemble de* m relations $\{R_1, ..., R_m\}$. *Chaque relation* R_p *est un sous ensemble du produit cartésien* $D_{p_1} \times ... \times D_{p_{n_p}}$.

L'**arité** d'une contrainte est le nombre de variables sur lesquelles porte la contrainte. Un CSP n-aire (ou

non-binaire) est un CSP qui comporte des contraintes d'arité supérieure à 2. Une **solution** est l'affectation d'une valeur à l'ensemble des variables du problème de telle sorte que chaque contrainte soit satisfaite. Un CSP est dit **consistant** (ou satisfiable) ssi il admet une solution. Un tuple t de la contrainte C_p est dit **autorisé** ssi $t \in R_p$ et dit **support** pour une valeur $v \in D_i$, $X_i \in Vars(C_p)$, ssi t est autorisé et t contient v dans la position correspondante à X_i dans la contrainte. Vérifier si un tuple donné est autorisé par une contrainte s'appelle **test de consistance**. Les combinaisons possibles des instanciations des variables dans un CSP créent un espace de recherche qui peut être vu comme un arbre, appelé **l'arbre de recherche**.

3 Arc-consistance et Forward Checking n-aires

L'*arc-consistance* reste l'une des propriétés fondamentales des CSPs. Elle garantit que toute valeur du domaine d'une variable possède au minimum un support dans toute contrainte. Cette propriété peut être établie comme étape de prétraitement ou pendant la recherche. Mackworth [9] décrit les algorithmes de base AC1, AC2 et AC3. AC3 a connu beaucoup de successeurs, mais il reste l'algorithme d'arc-consistance le plus simple connu jusqu'ici. Afin de préserver la simplicité de AC3 tout en améliorant son efficacité, dans [2] les auteurs ont proposé AC2000 et AC2001 et affirmé que ces nouvelles idées peuvent être facilement généralisées aux versions n-aires. AC3 exige la gestion d'un ensemble Q, mémorisant les révisions restantes à effectuer, qui correspond en théorie à un ensemble d'arcs (dans [9]). Cependant, il est également possible de le considérer comme un ensemble de variables (dans [2]) ou un ensemble de contraintes. Récemment, dans [8], les auteurs ont présenté trois variantes de AC3, à savoir orienté-arc, orienté-variable et orienté-contrainte. Nous avons étendu les algorithmes d'arc-consistance aux CSP non-binaires et développé trois algorithmes n-aires : nAC3, nAC2000 et nAC2001, basés sur le principe de AC3, AC2000 et AC2001, respectivement, chacun d'eux avec les variantes orientées arc, variable et contrainte.

L'algorithme *Forward Checking* (FC) est un des algorithmes les plus répandus de recherche en avant. Cet algorithme, présenté dans [6], effectue des vérifications de contraintes de type avant (*look-ahead*). Lors de l'instanciation d'une variable, il retire des domaines courants des variables *futures* toutes les valeurs incompatibles avec cette instanciation. Quand une nouvelle variable est considérée, on peut alors être sûr que toutes les valeurs de son domaine courant sont consistantes avec les variables *passées*. L'algorithme nFC0 introduit dans [14] est une première généralisation de l'algorithme populaire FC. Plus tard, dans [1], les auteurs ont effectué une étude plus générale sur FC. Ce travail fournit differentes versions de nFC1, nFC2, nFC3, nFC4, nFC5 qui diffèrent entre elles par la façon dont le filtrage en avant (*look-ahead*) est effectué à chaque nœud de l'arbre de recherche. Cependant, l'intention des auteurs est principalement conceptuelle, aucun détail de mise en œuvre n'a été donné. Nous avons implémenté la famille d'algorithmes FC n-aire, à savoir nFC0, nFC2, nFC3, nFC4, nFC5 afin d'entreprendre une étude comparative et expérimentale sur divers problèmes.

4 Expérimentations

Notre solveur n-aire a été développé en langage C++ en utilisant le compilateur CC sous Unix. Les résultats d'exécution présentés dans cet article sont obtenus sur une machine de CINES, de la gamme SGI3800 de 768 processeurs R1400 cadencés à 500 Mhz, dotée de 384 Go de mémoire centrale. Dans notre solveur les contraintes du problème sont stockées de façon à ce qu'elles soient indépendantes des algorithmes de recherche. Les contraintes peuvent être d'arité non-uniforme (l'arité n'est pas fixe), ce qui rend possible de formaliser de vraies applications à l'avenir. Dans nos expérimentations, nous avons mesuré : le temps CPU (cpu), le nombre de nœuds visités (#nœuds) et le nombre de tests de consistance (#ccks).

Dans nos expériences, nous avons étudié plusieurs problèmes de CSPLib ; dans cet article nous présentons les résultats obtenus sur deux problèmes académiques : le Golomb ruler et le lemme de Schur. Le *Golomb ruler* a été proposé par Peter Van Beek comme *prob006*, disponible à l'adresse `http ://csplib.cs.strach.ac.uk`. Il y a trois codages possibles [13] de représenter ce problème : un modèle uniforme (contraintes 4-aires) et deux modèles non-uniforme (contraintes 3-aires et binaires, contraintes 3-aires et $alldiff$). Le *lemme de Schur* proposé par Toby Walsh comme *prob015* disponible à l'adresse `http ://4c.ucc.ie./~tw/csplib`, peut être codé comme un CSP de deux manières : avec des contraintes 3-aires ou contraintes 3-aires modifiées (en ajoutant la contrainte $alldiff$) et avec des contraintes 3-aires et binaires.

Résultats en séquentiel

Premièrement, nous considérons l'application de l'arc-consistance comme phase de prétraitement (sans recherche de solutions). Les expérimentations ont

Alg.	Type	GR-8-34		GR-9-34	
		cpu	#ccks	cpu	#ccks
AC3	arc	12.424	50.6M	19.791	78.9M
	var	15.928	64.9M	25.586	102.4M
	cnt	12.393	50.6M	19.758	79M
AC2001	arc	5.807	24M	9.115	37M
	var	6.051	24.7M	9.463	37.8M
	cnt	5.818	24M	9.077	37M

TAB. 1 – Arc-consistance n-aire : Golomb ruler uniforme

Alg.	GR-8-34			GR-9-34		
	cpu	#nœuds	#ccks	cpu	#nœuds	#ccks
nFC0	0.125	905	0.17M	21.784	102727	25M
nFC2	0.674	665	0.87M	66.667	44139	64M
nFC3	0.459	566	0.8M	64.782	35361	55M
nFC4	1.854	541	2.8M	196.553	32531	181M
nFC5	1.733	386	2.1M	181.22	26315	169M
Alg.	GRM-8-34			GRM-9-34		
	cpu	#nœuds	#ccks	cpu	#nœuds	#ccks
nFC0	6.64	20464	5M	514.487	2277728	1094.9M
nFC2	1.388	3766	1.1M	109.633	569432	251.1M
nFC3	2.154	3108	1.8M	166.831	446552	304M
nFC4	3.574	2796	4.6M	276.647	374852	839.7M
nFC5	5.214	1764	5.5M	394.215	295862	1255.5M

TAB. 2 – Forward Checking n-aire : Golomb ruler

concerné plusieurs instances des problèmes. Pour illustrer les résultats, nous présentons ici deux instances du problème Golomb qui correspond à la règle de longueur 34 sur laquelle il faut placer 8 et respectivement 9 marques (les instances notées *GR-8-34* et *GR-9-34*, pour le modèle uniforme). Pour ces instances, nous avons appliqué les algorithmes nAC3 et nAC2001 énoncés dans la Section 3, avec leurs trois versions. Les résultats sont présentés dans le tableau 1. Clairement, nAC2001 est meilleur en terme de temps CPU et nombre de vérifications de consistance. En fait, il améliore nAC3 avec une efficacité proche de 47%.

Nous allons présenter maintenant les résultats expérimentaux pour les algorithmes de la famille nFC. Nous considérons bien évidemment comme algorithme de filtrage le meilleur algorithme retenu dans les tests précédents. Nous illustrons dans le tableau 2 les tests sur les instances *GR-8-34*, *GR-9-34* pour le modèle uniforme et *GRM-8-34*, *GRM-9-34* pour le modèle mixte de Golomb. Le nombre maximal de marques possibles sur une règle de longueur 34 est 8, donc *GR-9-34* et *GRM-9-34* n'ont pas de solution. Pour le modèle uniforme, l'algorithme nFC0 est le meilleur en terme de temps CPU et nombre de vérifications de consistance. En revanche, les autres algorithmes explorent moins de nœuds. En analysant les résultats du modèle mixte, nous pouvons observer un comportement différent ; dans ce cas là, l'algorithme nFC2 est le meilleur. Cela se justifie par la présence des contraintes binaires qui font que le côut du filtrage soit moins onéreux. Nous pouvons conclure que nFC0 est meilleur sur des CSPs uniformes et moins performant quand il s'agit des contraintes d'arité différente.

Résultats en parallèle avec OpenMP

OpenMP est un standard définissant un ensemble de directives et de fonctions permettant de gérer le partage du calcul entre plusieurs processeurs. L'interface OpenMP utilise la notion de *threads*. Un *thread* est un processus léger qui partage des données. Du point de vue du programmeur, celui-ci dispose de n *threads*, cela équivaut virtuellement à n processeurs qui peuvent effectuer du calcul en parallèle.

La résolution parallèle des CSPs peut être abordée de trois façons différentes : la *décomposition structurelle* du CSP telle que le *Tree Clustering* [4], la *décomposition des domaines* [7, 3] et la *division de l'arbre de recherche*. L'arbre de recherche est divisé à des profondeurs différentes en fonction de la granularité de parallélisme visée, l'idée étant de générer un nombre de tâches indépendantes, supérieur au nombre de processeurs. Dans le cas de la résolution des CSPs, il suffit de donner une valeur à un certain nombre de variables. Le choix de ces variables initiales fixées et leur nombre peut faire appel à différentes stratégies (voir [5]). Pour le nombre de ces variables, il suffit qu'il y ait suffisamment de problèmes à distribuer aux processeurs. Nous avons choisi d'instancier en premier les variables les plus contraintes.

Le tableau 3 montre respectivement les résultats obtenus pour différentes instances du lemme de Schur avec 4 boîtes et 5 boîtes, pour trouver toutes les solutions, avec l'algorithme nFC0, en faisant varier d'une part le niveau de division de l'arbre de recherche de 2 à 4 (la colonne D) et d'autre part le nombre de *threads* de 4 à 64 (les colonnes E_n pour l'efficacité) ; la colonne T_{seq} indique le temps séquentiel nécessaire pour la résolution et la colonne $\#tsk$, le nombre des tâches générées. On constate que pour les instances avec 4 boîtes, c'est la profondeur 2 qui donne les meilleurs résultats, tandis que pour 5 boîtes la profondeur 3 est la meilleure. Nous pouvons en déduire raisonnablement que plus le problème est difficile, plus il est intéressant d'augmenter la granularité du parallélisme, et donc la profondeur de division. Concernant les efficacités, nous observons des accélérations super linéaires en tous les cas.

5 Conclusion

Dans ce papier, nous avons mené une étude expérimentale sur plusieurs algorithmes n-aires d'arc-consistance de la famille AC et de résolution de type FC, en utilisant notre propre solveur qui inclut une bibliothèque d'algorithmes pour les CSPs n-aires. La maquette obtenue nous a permis de mettre en évidence que les algorithmes d'arc-consistance tels que

4box	D	T_{seq}	#tsk	E_4	E_8	E_{16}	E_{32}	E_{64}
23	2	1881	16	0.95	0.89	0.78	-	-
	3		64	0.91	0.82	0.81	0.77	0.72
	4		256	0.88	0.82	0.77	0.74	0.70
24	2	2736	16	0.94	0.88	0.78	-	-
	3		64	0.90	0.82	0.81	0.77	0.73
	4		256	0.88	0.81	0.77	0.74	0.69
5box	D	T_{seq}	#tsk	E_4	E_8	E_{16}	E_{32}	E_{64}
16	2	7221	25	0.92	0.84	0.75	-	-
	3		125	0.96	0.93	0.91	0.87	0.83
	4		625	0.91	0.90	0.88	0.85	0.80
17	2	26542	25	0.93	0.83	0.77	-	-
	3		125	0.96	0.92	0.90	0.86	0.82
	4		625	0.90	0.89	0.88	0.85	0.81

TAB. 3 – Problème de Schur avec 4 et 5 boîtes en paralléle

nAC2001 donnent des meilleurs résultats lors de l'application en phase de prétraitement, ainsi que pendant la recherche des solutions. En ce qui concerne les algorithmes de résolution de type nFC, leurs performances dépendent de plusieurs éléments : le coût du filtrage, l'arité des contraintes, l'uniformité des problèmes... L'algorithme nFC0 s'est avéré comme étant le meilleur sur les CSPs uniformes, tandis que l'algorithme nFC2 semble être le meilleur sur les CSPs non-uniformes. Certes, ces algorithmes méritent d'être confrontés avec des problèmes variés et, surtout, avec des problèmes réels. Ensuite, afin d'améliorer les performances, nous avons utilisé la technique de parallélisation par "division de l'arbre de recherche" basée sur OpenMP.

Les résultats présentés ici laissent entrevoir un certain nombre de perspectives pour des recherches futures. Les CSPs en extension étant très gourmands en mémoire, il serait intéressant de les représenter de façon optimale, en passant à la représentation en intention. Il peut s'avérer nécessaire de définir des nouvelles heuristiques plus orientées n-aire qui soient à la fois performantes et suffisament diverses pour améliorer l'efficacité des algorithmes de résolution et le choix des variables à instancier lors de la division de l'arbre de recherche pour l'exécution en parallèle. Nous projetons de nous orienter prochainement vers un autre modèle de parallélisme tel que MPI.

Remerciements

Ce travail est supporté par le Centre Informatique National de l'Enseignement Supérieur (CINES) de Montpellier (`http ://www.cines.fr`).

Références

[1] C. Bessière, P. Meseguer, C. Freuder, and J. Larossa. On forward checking for non binary constraint satisfaction. *Artificial Intelligence*, 141 :205–224, 2002.

[2] C.Bessière and J. C Régin. Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI'01*, pages 309–315, Seattle WA, 2001.

[3] A. Chmeiss. *Réseaux de contraintes : algorithmes de propagation et de décomposition.* PhD thesis, Université des Sciences et Techniques du Languedoc, Aix-Marseille I, 1996.

[4] R. Dechter and J. Pearl. Tree-clustering schemes for constraint-processing. In *Proceedings of the sixth National Conference on Artificial Intelligence (AAAI-88)*, pages 150–154, Saint Paul, MN, 1988.

[5] Z. Habbas, M. Krajecki, and D. Singer. Shared memory implementation of constraint satisfaction problem resolution. In *Proceedings of HLPP*, Orléans, France, 2001.

[6] R. M. Haralick and G. L. Elliot. Increasing the search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14 :263–313, 1980.

[7] Ph. Jégou. On the consistency of general constraint satisfaction problems. In *Proceedings of the eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 114–119, 1993.

[8] C. Lecoutre, F. Boussemart, and F. Hemery. Au coeur de la consistance d'arc. In *Proceding des 9ièmes Journées Nationales de la résolution pratique des problèmes NP Complets*, pages –, Amiens, 2003.

[9] A. K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8 :99–118, 1977.

[10] P.-P. Mérel, Z. Habbas, F. Herrmann, and D. Singer. N-ary consistencies and constraint-based backtracking. In *Proceedings of the second International Conference on Principles and Practice of Constraint Programming, CP'96*, Cambridge, MA, 1996. extended abstract.

[11] U. Montanari. Networks of constraints : Fundamental properties and applications to pictures processing. *Information Sciences*, 7 :95–132, 1974.

[12] F. Rossi, C. Petrie, and V. Dhar. On the equivalence of constraint satisfaction problems. In *Proceedings of the seventh European Conference on Artificial Intelligence*, pages 550–556, Stockholm, 1990.

[13] B. Smith, K. Stergiou, and T. Walsh. Modelling the golomb ruler problem. In *Proceedings of the IJCAI-99 Workshop on Non-Binary Constraints*, Stockholm, Sweeden, 1999.

[14] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, Cambridge, 1989.