Les réseaux distribués et valués de contraintes et le backtracking

Distributed Valued Constraint Networks and Backtracking

Mohamed Azouazi

Université Hassan II Mohammedia, Faculté des Sciences Ben M'sik, B.P 7955, Casablanca, Maroc

azouazii@gmail.com

Mustapha Belaïssaoui

Université Hassan 1er, ENCG, Km 3, Route de Casa, B.P 658, Settat, Maroc m.belaissaoui@encg-settat.ma

Khalid Moussaid

Université Hassan II Aïn Chock, Faculté des Sciences, B.P 5366, Casablanca, Maroc khmoussaid@fsac.ac.ma

Résumé

Nous proposons une nouvelle méthode pour résoudre des Problèmes de Satisfaction de Contraintes Valués et Distribués (DisVCSP). Cette méthode utilise à la fois certaines des bonnes propriétés de la version centralisée de dynamic backtracking valué (Dago, 1997), pour la résolution des problèmes de satisfaction de contraintes valués, et la méthode dynamic backtracking distribué (Belaissaoui, 2001) pour la résolution des problèmes de satisfaction de contraintes distribués. Elle vise à bénéficier des avantages des deux approches : la gestion des nogoods valués, la complétude de la recherche et un haut niveau d'asynchronisme entre les agents. Les expérimentations mettent en œuvre DisDB (Belaissaoui, 2001) et notre méthode dans un environnement distribué utilisant une granularité p<n (p agents et n variables).

Abstract

We propose a framework for solving Distributed Valued Constraint Satisfaction Problems (DisVCSP). This approach permits us to define a new framework for the enumeration, which we expect that it will benefit from the advantages of two approaches: recording and exploiting the valued nogoods in Valued Dynamic Backtracking (Dago, 1997); the completness of search and the asynchronisme between agents in Distibuted Dynamic Backtracking (Belaissaoui, 2001).

Mots-clés

CSP (problèmes de satisfaction de contraintes), DisVCSP (problèmes de satisfaction de contraintes valués et distribués), IA distribué, backtracking, multi-agents, nogoods

Keywords

CSP (constraint satisfaction problems), DisVCSP (distributed valued constraint satisfaction problems), distributed AI, backtracking, nogoods

1. Introduction

De nombreux problèmes réels peuvent être représentés sous la forme d'un problème de satisfaction de contraintes (CSP). En particulier, le formalisme CSP permet d'exprimer des problèmes d'ordonnancement, de vision, de conception, de configuration, etc. Il vise à représenter sous forme de contraintes les propriétés et les relations qui existent entre les objets manipulés. Elles traduisent l'autorisation ou l'interdiction d'une combinaison de valeurs. Dans le formalisme CSP, la recherche d'une solution requiert de satisfaire toutes les contraintes. Dans la mesure où certaines contraintes doivent être obligatoirement satisfaites, on les qualifie généralement de contraintes "dures". Cependant, pour certains problèmes réels, certaines contraintes dites "molles" ne traduisent, dans la réalité, qu'une préférence, une possibilité, etc. Leur satisfaction n'est donc pas forcément nécessaire. Représenter ces contraintes par des contraintes dures rend souvent les CSP correspondants inconsistants. Afin de pouvoir exprimer de telles contraintes, plusieurs extensions des CSP ont été proposées parmi lesquelles les CSP valués (VCSP) (Schiex, 1995, 1997) (Jegou, 2003). Le formalisme VCSP augmente le pouvoir d'expression du formalisme CSP en introduisant une graduation dans la violation des contraintes. Une valeur (appelée valuation*)* est associée à chaque contrainte. La valuation d'une contrainte traduit l'importance de la violation de cette contrainte. Autrement dit, le cadre VCSP autorise la violation de certaines contraintes, les contraintes étant soit dures, soit molles. L'objectif est alors de trouver une affectation de toutes les variables qui optimise un critère donné et portant sur la satisfaction des contraintes. En d'autres termes, une solution du problème est une affectation qui peut éventuellement violer certaines contraintes et dont l'importance des violations est minimale suivant un critère et un ordre donnés. La méthode de base pour la résolution de VCSP est l'algorithme Branch and Bound (séparation et évaluation). Bien sûr, de nombreuses améliorations ont été proposées, notamment à partir du cadre CSP. Néanmoins, à l'heure actuelle, les meilleurs résultats semblent être fournis par des méthodes comme la méthode des poupées russes (notée RDS pour Russian Doll Search (Verfaillie, 1996) ou par des méthodes issues de la programmation dynamique (Koster, 1999). Il s'agit de méthodes qui divisent le problème en plusieurs sous-problèmes et qui exploitent les informations qu'elles produisent durant la résolution de chacun de ces sous-problèmes.

Les DisCSP consistent à représenter les systèmes à contraintes en agents, entités indépendantes et communicantes capables d'agir sur elles-mêmes et sur l'environnement (Yokoo, 1990). Plusieurs travaux envisagent la distribution du problème de satisfaction de contraintes. Ils sont motivés par l'existence de problèmes naturellement distribués, pour lesquels il est impossible ou peu souhaitable de rassembler toutes les données du problème sur un site, pour le résoudre par un algorithme centralisé. Les raisons les plus immédiates sont le temps de communication requis ou le sur-coût de traduction de tous les sous-problèmes dans un format commun. Mais donner à un agent unique toutes les données du problème peut aussi être exclu pour des raisons de sécurité ou de confidentialité. Les algorithmes complets pour la résolution de CSP distribués doivent beaucoup à Yokoo et à ses collaborateurs, pionniers du domaine avec asynchronous backtracking (ABT) et asynchronous weak-commitment search (WCS) (Yokoo, 1992, 1998). Ces algorithmes imposent un ordre total entre les agents, ordre statique pour ABT, dynamique pour WCS. Distributed Intelligent Backtracking (DIBT) (Hamadi, 1999) propose une approche différente, sans échange de nogoods. Malheureusement, DIBT n'est pas complet (Bessière, 2001). Plus récemment, Optimal Distributed Intelligent

Backtracking (ODIBT) est présenté dans (Belaissaoui, 2002). ODIBT est une version complète et optimale en envois de messages de DIBT.
Dans cet article, nous proposons une nouvelle méthode pour le modèle DisVCSP. Notre but en l'écrivant était de garder le meilleur de *Distributed Dynamic Backtracking* (Bessière, 2001), pour obtenir un algorithme aussi asynchrone que possible, qui minimise l'envoi de messages entre agents indépendants tout en restant complet. Notre algorithme a aussi une parenté évidente avec *Valued Dynamic Backtracking* (Dago, 1997). Nous avons donc gardé les nogoods de ce dernier.

2. Définitions préliminaires

Un CSP est défini par la donnée (X, D, C). X est un ensemble $\{x_1, ... x_n\}$ de n variables, chaque variable x_i prenant ses valeurs dans un domaine fini D_{xj} issu de D. Ces variables sont soumises à des contraintes issues de C. Chaque contrainte c est définie comme un ensemble $\{x_{c1}, ..., x_{ck}\}$ de variables telle que c représente l'ensemble des tuples autorisés sur $d_{xcl} \times ... \times d_{xck}$. Etant donné $Y \subseteq X$ tel que $Y=\{x_1,....x_k\}$, une instanciation des variables de Y est un tuple $B= (v_1, ... v_k)$ de $d_{x1} \times ... \times d_{xk}$. Nous appelons une affectation partielle un sous-ensemble de variables affectées. Lorsque toutes les variables sont affectées, l'affectation est dite complète. Deux affectations sont compatibles si les variables communes sont affectées aux mêmes valeurs. A la difference du cadre CSP, les contraintes dans le cadre VCSP peuvent être soit dures soit molles. Résoudre un problème VCSP revient alors à rechercher une affectation qui optimise une fonction portant sur la satisfaction des contraintes. Autrement dit, une solution du problème est une affectation qui peut éventuellement violer certaines contraintes et dont l'importance des violations est minimale suivant un critère et un ordre donnés. Pour quantifier l'importance d'une violation, une valeur (appelée *valuation*) est associée à chaque contrainte. Lorsque plusieurs contraintes sont violées simultanément, les *valuations* correspondantes doivent être agrégées pour déterminer l'importance de la violation de l'ensemble de ces contraintes. Dans ce but, on se dote d'une structure de *valuation* :
Définition 1 (Schiex, 1995) : Une structure de valuation est un triplet $(E, \leq, \oplus)$ avec un ensemble E de valuations totalement ordonné par $\leq$, muni d'un élément minimum (noté $\perp$), d'un élément maximum (noté T) et d'une loi de composition interne (notée $\oplus$) commutative, associative, monotone et telle que $\perp$ soit un élément neutre pour $\oplus$ et T un élément absorbant.
Les valuations permettent d'exprimer différents niveaux de violation. Par exemple, $\perp$ caractérise une absence de violation (i.e. la satisfaction d'une contrainte) et T une insatisfaction totale. Quant à la loi $\oplus$, elle permet de calculer la valuation correspondant à la violation simultanée de plusieurs contraintes. Notons que, dans certains cas, elle peut posséder d'autres propriétés comme l'idempotence ou la stricte monotonie. A partir de cette structure de valuation, on peut définir formellement la notion de CSP valué :
Définition 2 (Schiex, 1995) Un VCSP est un CSP classique P = (X, D, C) doté d'une structure de valuation $S=(E, \leq, \oplus)$ et d'une application Φ de C dans E, qui associe une valuation à chaque contrainte du CSP. On le note (X, D, C, S, Φ).
Il est dit binaire si chaque contrainte de C implique au plus deux variables. La valuation d'une instanciation complète B des variables de X correspond à la combinaison des valuations des contraintes violées par B :

Définition 3 (Schiex, 1995) Soient un VCSP P = (X, D, C, S, Φ) et une instanciation B sur X. La **valuation** de B dans P se définit par :

$$V_p(B) = \bigoplus_{c \in C | B \text{ viole } c} \phi(c)$$

Etant donnée une instance P, le problème VCSP consiste donc à trouver une affectation de toutes les variables de P qui soit de valuation minimale au sens de ≤. Cette valuation optimale est appelée valuation du VCSP. Par la suite, nous la noterons α. Déterminer la valuation d'un VCSP est un problème NP-difficile.

Exemple 1 Malek, Badr et Ali se rencontrent inopinément dans un couloir. Malek interpelle les deux autres et leur rappelle leur promesse de présentation de leurs travaux respectifs. Badr et Ali doivent présenter leurs travaux à Malek qui lui même doit leur présenter ses travaux. Il y a 4 exposés : de Badr à Malek, d'Ali à Malek, de Malek à Badr et enfin de Malek à Ali. Ils conviennent de bloquer 4 demi-journées pour faire ces exposés. Chaque exposé prend une demi-journée. Dès le début, Malek signale qu'il lui paraît important d'avoir écouté Badr et Ali avant de faire ses présentations. Puis, Malek indique qu'il aimerait présenter ses travaux à Badr lors de la deuxième demi-journée. Enfin, Malek signale qu'il ne veut pas présenter ses travaux à Badr et Ali en même temps.

Modélisation du problème : Ce problème se modélise sous forme d'un CSP ayant 4 varibles $\{M_b, M_a, B_m, A_m\}$, où chaque variable représente un exposé. Par exemple, M_b désigne l'exposé que doit faire Malek à Badr… Les exposés devant avoir lieu dans 4 demi-jounées, les domaines des variables sont identiques: {1,2,3,4}. Les contraintes sont de deux sortes : les contraintes d'integrité et les contraintes de préférence exprimées par Malek.

Contraintes d'intégrité.

- Un orateur ne peut être auditeur dans la même demi-journée : $C_1 : M_a \neq A_m$, $C_2: M_b \neq B_m$, $C_3: M_a \neq B_m$, $C_4 : M_b \neq A_m$.
- Une personne ne peut assister à deux présentations en même temps : $C_5: A_m \neq B_m$.

Contraintes de préférence.

- Malek signale qu'il lui paraît important d'avoir écouté Badr et Ali avant de faire ses présentations : $C_6 : M_a > A_m$, $C_7 : M_a > B_m$, $C_8 : M_b > B_m$, $C_9 : M_b > A_m$.
- Malek aimerait présenter ses travaux à Badr lors de la deuxième demi-journée et il ne veut pas présenter ses travaux à Badr et Ali en même temps : $C_{10} : M_b = 2$, $C_{11}: M_a \neq M_b$.

Nous employons S=(N, <, +) avec l'élément minimum est 0 et l'élément maximum est +∞. Nous définissons trois niveaux de violations : pour chaque contrainte d'intégrité, la valuation associée est 3 (contrainte *"dure")*. Les violations des contraintes d'antériorité ont un poids de 2 et les contraintes de simultanéité (C_{10} et C_{11}) sont associées à une valuation de 1. Pour ce VCSP, nous obtenons α=1 (il suffit de violer C_{10} pour avoir une solution).

Définition 4 (Schiex, 1995) Soient un VCSP P = (X, D, C, S, Φ) et une instanciation B sur Y⊆ X. La valuation locale de B dans P se définit par

$$v(B) = \bigoplus_{c \in C | c \subseteq Y \, B \text{ viole } c} \phi(c)$$

Nous notons $v_C(B)$ la valuation d'une affectation B sur le problème restreint au sous-ensemble de contraintes $C \subseteq$ C.

La méthode de base pour résoudre les CSP valués est l'algorithme branch and bound (séparation et évaluation, noté BB). Cette méthode énumérative utilise la valuation locale de l'affectation courante comme minorant et la valuation de la meilleure solution connue comme majorant. Si le minorant ne dépasse pas le majorant, alors on étend l'instanciation courante en affectant une nouvelle variable. Sinon, on revient en arrière sur la dernière variable instanciée et on l'affecte avec une nouvelle valeur. Si toutes ses valeurs ont été essayées, on revient en arrière une nouvelle fois, et ainsi de suite. Plusieurs améliorations de cet algorithme ont été proposées. La plupart d'entre elles proviennent du cadre CSP et ont conduit à des méthodes comme les algorithmes Forward-Checking valué (Schiex, 1995), Nogood Recording (Dago, 1996), la recherche arborescente bornée pour la résolution de CSP valué (Jegou, 2003), etc.

Définition 5 Un DisVCSP est un VCSP où les variables, valeurs, contraintes et valuations sont distribuées parmi les agents A_i (i.e pour chaque agent A_i, on trouve un VCSP (X_i, D_i, C_i, S, Φ)). Donc, un problème distribué de satisfaction de contraintes valuées (DisVCSP) est un P=(X, D, C, S, Φ, A) où X, D, *C,* S et Φ sont définis comme précédemment. A est un ensemble fini d'agents $\{A_1,.., A_p\}$. Chaque agent possède un sous ensemble des variables du problème. A constitue une partition de X.

Chaque agent possède l'ensemble des données du problème relatives à son sous-ensemble des variables (domaines, contraintes et valuations). Chaque variable appartient à un agent. La distribution des variables forme une bi-partition de C en $C_{intra}=\{c_{ij}$ / x_i et x_j appartiennent au même agent} et $C_{inter}=\{$ c_{ij} / x_i et x_j appartiennent à des agents différents }, qu'on nomme ensemble des contraintes intra-agents et inter-agents, respectivement.

Comme dans le cadre centralisé, une solution est une affectation de valeurs aux variables qui optimise les valuations portant sur la satisfaction des contraintes. Les DisVCSP sont résolus par l'action collective et coordonnée des agents de A, chacun exécutant un processus de satisfaction de contraintes. Les agents communiquent par envois de messages, avec les postulats suivants (Yokoo, 1998): Un agent ne peut envoyer de message que s'il connaît l'adresse du destinataire. Le délai de réception d'un message est fini mais aléatoire. Pour une paire d'agents donnée, les messages sont reçus dans l'ordre dans lequel ils ont été envoyés.

Pour des raisons de simplicité, et pour mettre l'accent sur les aspects relatifs à la distribution, dans la suite de cet article nous supposons que chaque agent a une seule variable. Nous identifions l'indice de l'agent à celui de sa variable.

3. Valued Dynamic Backtracking distribué (DisVDB)

3.1 Nogoods

Le Backtrack Dynamique valué (Dago, 1997) mémorise les zones de l'espace de recherche explorées et reconnues comme sans solution sous la forme d'affectations partielles qu'il est interdit de reproduire : les nogoods. Dans le cadre d'une optimisation, l'exploration constate que les valuations des solutions dans une certaine zone sont supérieures à une valeur donnée α. Si cette valuation est suffisamment élevée, l'interdiction d'explorer à nouveau devient effective. Afin de s'adapter à ce contexte, nous rappelons le concept de nogood valué ainsi que les propriétés qui permettent sa mise en oeuvre. Ces proprités simples ne sont pas démontrées ici ; on en trouvera une preuve formelle dans (Dago, 1996).

Définition 6 (Nogood valué) Un nogood valué est un triplet (B, v, C) , où B est une affectation partielle, v une valuation et C un ensemble de contraintes (appelé justification) tel que, pour toute affectation B' extension complète de B, $v \leq v_C(B')$.
Un Nogood valué est, en quelque sorte, une prévision par défaut de la valuation des violations que l'on obtiendrait à terme si l'extension complète d'une affectation était effectuée. Un nogood (B, v, C) donne un minorant des valuations globales de toutes les affectations partielles contenant B.

Propriété 1 ((Dago, 1996) construction**)** Soit B une affectation partielle, et C un ensemble de contraintes qu'elle viole, alors $(B, v_C(B), C)$ est un nogood.
Cette propriété permet la production des nogoods. Il en découle en particulier que si C est l'ensemble de toutes les contraintes violées, (B, v(B), C) est un nogood. Nous noterons n(B) ce nogood particulier. La propriété suivante permet d'agréger en un nouveau nogood plusieurs nogoods qui impliquent toutes les valeurs d'une variable. Intuitivement, puisque le choix d'une valeur pour cette variable est nécessaire, on peut prévoir (indépendamment de l'affectation de la variable) une valuation au moins égale à la plus optimiste des prévisions fournies par les nogoods.

Propriété 2 ((Dago, 1996) union**)** Soit B une affectation partielle, et x^1 ,. . . ,x^m les valeurs d'une variable x n'appartenant pas à B. Soient $B_1,\ldots,B_m$ des sous ensembles de B tels que $(B_1 \cup \{x^1\}, v_1, C_1),\ldots, (B_m \cup \{x^m\}, v_m, C_m)$ sont des nogoods.

$(\bigcup B, \min(v), \bigcup C)$ est un nogood.

Propriété 3 ((Dago, 1996) projection**)** Soit (B, v, C) un nogood, et B′ l'ensemble des valeurs de B absentes des n-uplets interdits par les contraintes de C.

(B-B', v, C) est un nogood.

Propriété 4 ((Dago, 1996) augmentation**)** Soit (B, v, C) un nogood, B' une affectation compatible avec B et c une contrainte violée par B' .

Si $c \notin C$, $(B \cup B', v \oplus \phi(c), C \cup \{c\})$ est un nogood.

Propriété 5 ((Dago, 1996) réduction**)** Soit (B, v, C) un nogood, et c une contrainte de C.

Soit $v' = \min(\varepsilon \in E \,/\, \varepsilon \oplus \Phi(c) \geq v)$, alors $(B, v', C - \{c\})$ est un nogood.

Soit B une affectation et n un nogood d'affectation compatible, nous noterons $\uparrow^B(n)$ le résultat des augmentations successives de *n* avec toutes les contraintes violées par *B*, et $\uparrow^B(n)$ le résultat des réductions successives du nogood *n* avec toutes les contraintes violées par l'affectation *B*. Ici intervient la précision d'un nogood : plus il est précis, plus il touche un grand nombre d'affectations (après réduction) et plus il fournira une borne inférieure élevée de leur valuation globale (après augmentation). La propriété qui suit prouve que la réduction n'entraîne aucune dégradation des prévisions.

Propriété 6 (**(Dago, 1997) équivalence)*** Soit *n = (B, v, C)* un nogood, et *c* une contrainte de *C* violée par une affectation *B*' compatible. Le nogood *n*' obtenu par réduction puis augmentation de *n* avec *c* possède une valuation supérieure ou égale à celle de *n*.

Corrolaire (Dago, 1997) la valuation d'un nogood n est inférieure ou égale à celle de $\uparrow^B(\downarrow^B(n))$.

Définition 7 ((Dago, 1997) Réduction partielle) Soit B une affectation partielle et n un nogood portant sur B. Soit $C_{B\cap n}$ l'ensemble des contraintes violées par B et présentes dans n, et C_{B-n} les autres contraintes violées par B. Nous appelons réduction partielle d'un nogood par B et n l'augmentation (propriété 4) avec les contraintes de $C_{B\cap n}$ suivie de la réduction (propriété 5) avec les contraintes de C_{B-n} par B.
Nous noterons $\downarrow\downarrow_{(B,\ n)}$ cet opérateur de réduction. Il a pour effet de projeter un nogood n′ de manière à ce que le résultat n’’ ait une valuation supérieure à son prédécesseur n.

3.2 Valued Dynamic Backtracking distribué

3.2.1 Dynamic Backtracking Valué

Dynamic Backtracking Valué (VDB) (Dago, 1997) est une procédure de recherche arborescente qui mémorise pour chaque valeur xi, ième valeur de Dx, un nogood N_xi donnant un minorant de la valuation globale de l’extension d’une affectation B avec la valeur xi. Ce nogood est une mémoire des anciennes tentatives infructueuses d’extension avec xi. VDB choisit une variable non instanciée, et tente de lui affecter une valeur. Si cette valeur et les variables déjà instanciées, donnent une valuation supérieure à α, elle est supprimée et le nogood correspondant est mémorisé. Lorsque toutes les valeurs de la variable courante x sont éliminées, les justifications de retrait sont utilisées pour générer un nouveau nogood comme suit. Soit B l’ensemble des variables déjà instanciées (x n’appartient pas à B). Le nouveau nogood n est l’union des nogoods de valuations maximales de B $\cup$ {xi}, ∀xi∈ Dx. Pour avoir un nogood de valuation maximale, il suffit d’utiliser la propriété d’augmentations successives de N_yj avec toutes les contraintes violées par

B∪{xi}, ∀xi∈ Dx et ∀yj∈B∪{xi}

$$n = Union(\uparrow^{B\cup\{x^i\}}(N_{y^j}), \forall x^i \in D_x, \forall y^j \in B \cup \{x^i\})$$

L’algorithme impose des conditions d’ordre entre les variables de l’affectation du nogood et la variable sur laquelle se porte le backtrack. Soit y^j appartenant à B, une valeur qui ne soit inférieure à aucune autre valeur de n selon l’ordre partiel. Les nogoods N_zk, dont l’affectation contient y^j, sont initialisés par le nogood $n_\varnothing$=(Ø, ⊥, Ø). N_yj est initialisé par ↓↓ (N, N_yj) (n) y^j est supprimée de B et l’algorithme cherche à réinstancier y. Il a été prouvé que cet algorithme est correct et complet (Dago, 1997).

3.2.2 Valued Dynamic Backtracking Distribué (DisVDB)

Valued Dynamic Backtracking Distribué (voir algorithme ci-dessous) est un algorithme calqué sur le modèle du Backtrack dynamique distribué présenté dans (Bessière, 2001). Il réalise des sauts dynamiques sur l'ensemble des agents en conflit. La recherche arborescente d'une solution nécessite la mise en place d'un ordre total entre les agents destinés au calcul. Le calcul de cet ordre revient à affecter des priorités entre agents. L'objectif commun des approches distribuées de types arborescents est d'offrir des méthodes distribuées calculant un ordonnancement des agents. Ces méthodes sont exécutées avant la méthode de résolution proprement dite. Dans ce sens, DisVDB utilise l'algorithme d'ordonnancement EDisAO (Extension Distributed Agents Ordering)

(Belaissaoui, 2002). EDisAO prend en entrée Γ, f et op. Γ est l'ensemble des agents du système avec lesquels self, l'agent générique, partage au moins une contrainte inter-variable. f et op sont respectivement la fonction heuristique et l'opérateur de comparaison. Il construit une hiérarchie d'agent en utilisant des critères heuristiques : Γ^-(self) désigne l'ensemble des agents partageant une contrainte avec self et classés plus haut dans la hiérarchie. Réciproquement, Γ^+(self) est l'ensemble des voisins de self classés plus bas dan la hiérarchie.
Chacun des agents exécute l'algorithme DisVDB et mémorise un contexte et un ensemble de nogoods. Le contexte de self est l'ensemble des valeurs qu'il croît affectées aux agents le précédant dans l'ordre. Il est toujours cohérent avec l'ensemble des nogoods mémorisés localement. Les agents échangent des affectations et des nogoods. Les affectations sont toujours acceptées, et le contexte mis à jour en conséquence. Un nogood est accepté s'il est cohérent avec le contexte de l'agent et sa propre instanciation, sinon il est obsolète. S'il est accepté, il est mémorisé comme justification du retrait de la valeur courante. Lorsque toutes les valeurs d'un agent sont éliminées, il génère un nouveau nogood n de la même manière que dans VDB, et l'envoie à l'agent qui ne soit inférieure à aucun autre agent selon l'ordre partiel (entre les agents responsables des variables de l'affectation de n). Self désinstancie cette variable dans le contexte, ainsi que toutes celles qui apparaissent dans le nogood n, mais pas dans Γ^-(self), et on actualise les nogoods en conséquence. Le processus s'achève lorsque les agents se stabilisent : une solution a été trouvée, ou le nogood vide est généré, ce qui signifie que le problème est insoluble. Les messages échangés sont de trois types : Stop(system), il n'y a pas de solution et le destinataire arrête la recherche. Ce message implique un agent supplémentaire appelé *system*, qui est responsable de l'arrêt du réseau. Info(fils,v, N_v), informe l'agent fils, qui est dans Γ^+(self), que self a pris la valeur v et que le nogood généré pour cette valeur est N_v, c'est pour que l'agent fils peut calculer la valuation de son nogood généré, s'il n'arrive pas à instancier sa variable. Back(nogood, A), message de retour arrière. Il contient un nogood et est adressé à l'agent A. A est le dernier agent qui a envoyé un message de type info, par exemple. Par ce que, self croit que ce dernier est la cause du retrait. Les messages sont échangés via les primitives getMsg et sendMsg.
DisVDB (voir algorithme ci-dessous) est la procédure principale. C'est une boucle de réception qui commute l'exécution suivant le type de message reçu. Après réception d'un message Stop de l'agent *System*, la procédure s'achève. Une borne supérieure α fournie par la dernière solution trouvée et fixée a priori à la valuation maximale. A chaque valeur i de la variable de self est associé un nogood N_i donnant un minorant de la valuation globale de l'extension d'un contexte avec la valeur i. Ce nogood est une mémoire des anciennes tentatives infructueuses d'extension avec i. L'ensemble des N_i est initialisé par des nogoods nuls : $n_{\emptyset}=(\emptyset, \bot, \emptyset)$.
GoAhead est exécutée lors de la réception d'un message de type Info. Elle tente détendre le contexte reçu par self. Après une mise à jour du contexte de self, désigné comme myContext (ligne 1), si la valeur courante myValue (ligne 2) est supprimée par la fonction Evalue, qui fournit sous forme de nogoods une estimation des valuations globales, GoAhead fait appel à la procédure ChooseValue pour trouver une autre valeur qui respecte la borne supérieure α avec myContext (ligne 3). Si c'est possible, self informe Γ^+(self) de sa nouvelle valeur (ligne 4). Sinon, self appelle la procédure ResolveNogoods (ligne 5). Si l'argument de GoAhead est vide, elle essaye de trouver une valeur de self respectant avec myContext la borne α (ligne 2 de DisVDB et ligne 10 de ResolveNogoods).

Algorithme : Distributed valued Dynamic Backtracking

```
Procedure DisVDB()
1 EDisAO(Γ, f, op) ;
2 GoAhead(null) ;
3 end ← false ;
4 while not(end) do
5       msg ← getMsg() ;
6       switch(msg.type)
7               Stop    : end ← true ;
8               Info    : GoAhead(msg) ;
9               Back    : ResolveConflict(msg) ;

Procedure GoAhead(msg)
1 if (msg) then Update (myContext, msg. Context) ;
2 if (Not(msg) or Valuation(Evalue(myContext ∪ {myValue})) > α) then
3       myValue ← ChooseValue() ;
4       if (myValue) then for each child in Γ+ (self) do sendMsg:Info(child, myValue, N_myValue );
5       else ResolveNogoods() ;

Procedure ResolveConflict(msg)
1 if Not(obsolete (msg.Context on Γ- (self) ∪ {self }, myContext)) then
2       Update (myContext,msg. Context) ;
3       N_myValue← ↓↓(msg.context,NmyValue)(n) ;
4       myNogoods ← myNogoods ∪ {N_myValue};
5       myValue ← ChooseValue() ;
6       if (myValue) then for each child appurtenant à Γ+ (self) do sendMsg:Info(child, myValue,
N_myValue );
7       else ResolveNogoods();
8 else if Not(obsolete (msg.Context on self, myValue)) then
9       SendMsg:Info(msg.sender, myValue, N_myValue);

Procedure ResolveNogoods()
1 n ←Union_i (Evalue(myContext ∪ {i})) ; /* i appartient à D_self */
2 if Affectation(n)= Ø then
3       end ← true ;
4       sendMsg:Stop(system) ;
5 else
6       Let A^i appurtenant à Value(myContext ∩last(msg.type=info))
7       sendMsg:Back(n, A);
8       Update(myContext, myContext\{A^i});
9       Update(myContext, mycontext\ ∪_j{x^j appartenant à Affectation(n)\myContext(Γ-(self))};
10      GoAhead(null);

Function ChooseValue()
1 for each v appartenant à D(self) not eliminated by myNogoods do
2       if Valuation(Evalue(myContext ∪ {v})) < α then return (v) ;
3       else N_v← Evalue(myContext ∪ {v});
4       myNogoods ← myNogoods ∪ {N_v};
5 return (Ø) ;
```

Procedure Update (myContext, new Context)
1 revise (myContext, newContext) ;
2 for each n appartenant myNogoods do
3 if obsolete(Affectation(n), myContext) then myNogoods ←myNogoods\ {n};

Function Evalue(Context)
Return(Nogood(maximal(valuation in { $\uparrow^{Context}(N_{x^i}), \forall x^i \in Context)$ })))

Un nogood, dans un message de type Back, est obsolète (ligne 1 de ResolveConflict), si son contexte ne respecte pas les valeurs des agents de Γ^-(self) et la valeur de self existants dans myContext. S'il n'est pas obsolète, ResolveConflict actualise myContext au vu des nouvelles informations (ligne 2). Elle construit un nouveau nogood, par la définition de la réduction partielle (définition 7), et elle le mémorise (lignes 3 et 4). Ensuite, elle tente de trouver une autre valeur. Si c'est possible, self informe Γ^+(self) (ligne 6). Sinon, ResolveConflict fait appel à la procédure ResolveNogoods (ligne 7). Si le message de type back respecte seulement la valeur de self, self renvoie cette valeur à l'expéditeur (lignes 8 et 9).
La procédure ResolveNogoods utilise la propriété de l'union (propriété 2) sur les nogoods générés par la fonction Evalue appliquée sur chaque valeur de D_{self} et myContext. Elle génère un nouveau nogood n. Si l'affectation de ce nogood est vide, le problème est insoluble et un message de type Stop est envoyé (ligne 4). Sinon, ResolveNogoods localise l'agent A qu'a envoyé le dernier message de type Info. Elle envoie à A un message de type Back contenant n (ligne 7). Self oublie l'instanciation de A ainsi que toutes les affectations de n n'appartenants pas aux agents de Γ^-(self). Il oublie aussi les nogoods locaux qui sont obsolètes.
La fonction ChooseValue choisit une valeur respectant la borne supérieure α avec myContext. Si une valeur est supprimée par la fonction Evalue, qui fournit sous forme de nogoods une estimation des valuations globales (propriétés 2 et 4), un nouveau nogood est généré par cette dernière. L'ensemble des nogoods (myNogoods) est actualisé. Si toutes les valeurs de D_{self} sont supprimées, la fonction renvoie la valeur vide.

4. Validité et complétude

Nous vérifions ici que l'arrêt qui n'est pas dû à l'obtention d'une meilleure solution (*ResolveNogoods* ligne 2) correspond à l'absence de solutions de valuation inférieure à la borne α. *Evalue(myContext* ∪ *{i})* renvoie au moins la valuation locale *v(myContext* ∪ *{i})*. Le test ligne 2 dans *ChooseValue* assure que toute affectation complète produite possède une valuation strictement inférieure à la borne maximale α. Il faut constater que le nogood *n* produit dans *ResolveNogoods* (ligne 1), comme tous les nogoods dont il est issu, possède une valuation supérieure ou égale à α. En cas d'arrêt, on dispose d'un nogood *n* d'affectation vide et de valuation supérieure ou égale à α. Ce nogood est la preuve que toute solution possède une valuation inférieure à α.
La terminaison est la propriété qui manque à la complétude de cet algorithme. Elle réside dans le choix de la valeur sélectionnée comme cible du backtrack (*ResolveNogoods*, ligne 5). Les nogoods produits par union ont toujours une valuation supérieure ou égale à α. Ils correspondent donc à des nogoods classiques interdisant une affectation. Pour assurer la

terminaison, il suffit d'appliquer la méthode EDisAO (Belaissaoui, 2002) pour établir un ordre partiel entre les agents responsables des variables contenues dans le nogood *n*. Si le choix du point de backtrack se fait de telle manière que l'ordre partiel établi par EDisAO est respecté, alors le processus termine (Belaissaoui, 2002)(Bessière, 2001).

5. Expérimentations

Ces expérimentations mettent en œuvre DisDB (Bessière, 2001) et notre méthode DisVDB dans un environnement distribué utilisant une granularité $p < n$ (p agents et n variables).
Une approche plus théorique et couramment utilisée consiste à tester les algorithmes sur un échantillon de problèmes tirés aléatoirement (voir algorithme ci-dessous) ; on prend habituellement pour mesure la moyenne des performances réalisées sur l'échantillon testé. Ceci implique : la description du modèle des DisCSP générés (paramètres et algorithme de génération) ; les tests pour chaque valeur du ou des paramètres de génération. Il reste enfin à définir les unités de mesure utilisées. Puisque nous cherchons l'optimalité de la recherche d'une solution, nous avons choisi l'unité couramment utilisée : la moyenne des temps d'exécution.
Notre générateur de problèmes distribués aléatoires est l'extension du générateur de Van Beek (VanBeek)] (la bibliothèque de programmes concernant les CSP de Van Beek) au cadre de DisCSP. Il nécessite cinq paramètres : le nombre de variables n, le nombre de valeurs par domaine d, la proportion (probabilité) de contraintes dans le réseau, p_1, la proportion de paires interdites dans une contrainte, p_2, et le nombre d'agents $p < n$. Nous avons généré deux types de problèmes : des problèmes faiblement contraints ou peu denses pour les petites valeurs de p_1 et des problèmes ou graphes complets pour $p_1=1$.
Après la création du CSP global (voir algorithme ci-dessous), nous l'avons distribué comme suit. Dans un premier temps, nous avons initialisé chaque sous-problème CSP_j (CSP_j est le sous-problème de l'agent j, $1\leq j \leq p$) par la variable i appartient à {1,.., n} telle que $i = j$ ainsi que son domaine D_i et l'ensemble des relations R_{ik} telle que $1\leq k \leq n$ et $k \neq i$. L'extension de ces sous-problèmes CSP_j est faite suivant la valeur d'une variable x tirée aléatoirement entre 0 et 1. Pour DisDB, les valuations de contraintes sont générées, aussi, d'une façon aléatoire. Ainsi que la valeur de α.
Pour chacune des méthodes que nous avons testé, nous avons utilisé la démarche suivante : chaque agent est activé à tour de rôle par un processus système, pour un cycle durant lequel l'agent peut lire tous les messages qui lui ont été adressés au cycle précédent, effectuer localement les calculs qui s'y rapportent et envoyer les messages adéquats. Lorsqu'il y a stabilisation de tout les processus, le processus système détecte la terminaison. Puisque tout les processus utilisent un seul processeur, nous avons mesuré le temps processeur permettant la résolution de chaque problème.
Nous avons généré deux types de problèmes : des problèmes denses $<p, n, d, p_1, p_2> = <5,15, 5, 1, p_2>$ et des problèmes peu denses $<p, n, d, p_1, p_2> = <5,15, 5, 0.3, p_2>$, pour différentes valeurs de p_2. Sur chaque instanciation de p_2, nous avons comparé les performances de DisDB et de notre méthode DisVDB, en utilisant l'heuristique MaxDeg pour ordonner les agents.
La figure 1 montre nos résultats sur un ensemble de problèmes denses en faisant varier p_2 par pas de 0.1. DisVDB s'est révélé nettement meilleur que DisDB pour p_2 dans [0.3,.., 0.5]. Ceci s'explique par le fait que le degré de connectivité des variables est de 100%

(p_1=1) et si p_2 appartient à l'intervalle [0.3,.., 0.5], le nombre d'envois de message augmente parce que le nombre de contextes de retour en arrière s'élève en balayant l'ensemble de valeurs avec presque 50% de valeurs interdites. Puisque DisVDB procède par des valuations de contraintes, ce dernier décide, soit de poursuivre la recherche de la solution, soit d'arrêter le processus de la recherche en trouvant une solution, s'elle existe, avec un coût inférieur à α. Ceci explique l'écart en temps de calcul dans cet intervalle.
La figure 2 montre aussi nos résultats sur des problèmes peu denses avec une variation de p_2 par pas de 0.1. Dans l'intervalle [0.5,.., 0.8] de p_2, DisVDB se comporte encore mieux que DisDB. Cet intervalle est supérieur à celui des problèmes denses parce que le degré de connectivité est inférieur à celui de ces derniers.

Algorithme : Générateur aléatoire de DisCSP

Début
  Pour chaque $1 \leq i < j \leq n$ faire
    X ← valeur aléatoire entre 0 et 1 ;
    Si $x < p_1$ alors
      {la contrainte C_{ij} est créée, initialement vide}
      $R_{ij} \leftarrow \emptyset$;
      Pour chaque (a,b)appartenant à $D_i \times D_j$ faire
        Y ← valeur aléatoire entre 0 et 1
        Si $y<p_2$ alors $R_{ij} \leftarrow R_{ij} \cup \{(a,b)\}$
      Finp
    Finsi
  Finp
 Pour chaque agent $1 \leq j \leq p$ faire Insérer (j, CSP_j) ;
   {insère la variable j, D_j et les R_{ij} dans CSP_j}
 Pour chaque agent $1 \leq j \leq p-1$ faire
   Pour chaque variable $p+1 \leq i \leq n$ faire
     X ← valeur aléatoire entre 0 et 1 ;
     Si $x < p/n$ alors Insère (i, CSP_j)
   Finp
 Finp ;
 Ajouter le reste des variables à CSP_p
Fin.

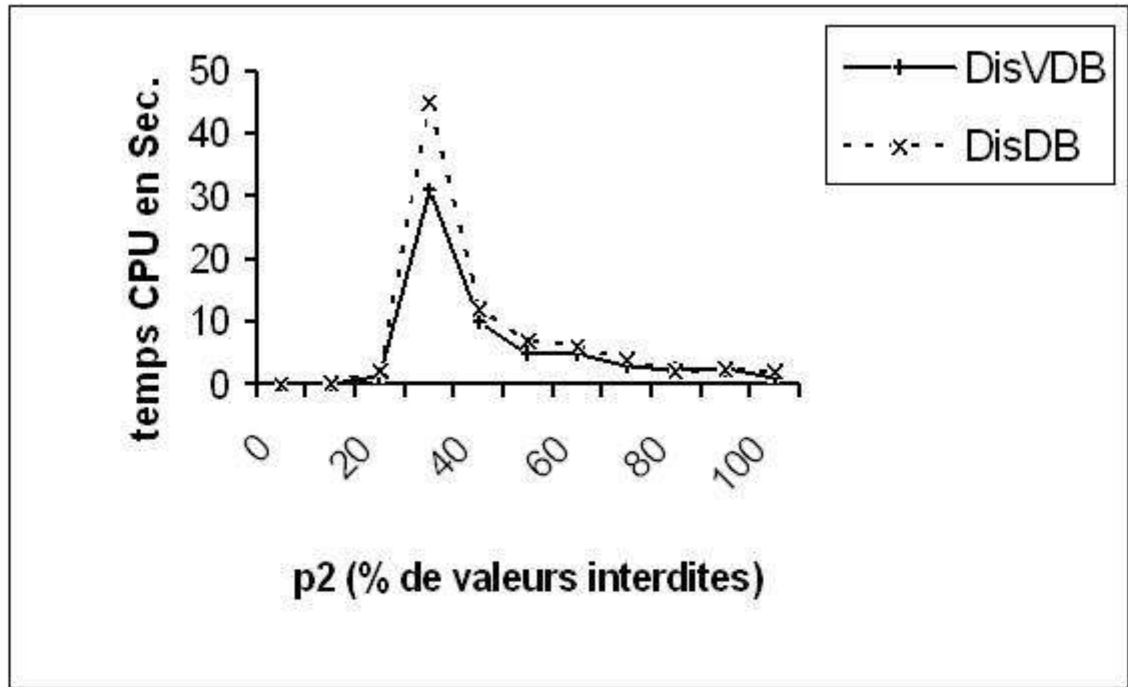

Figure 1. Comparaison entre DisVDB et DisDB sur des problèmes denses (p_1=100%).

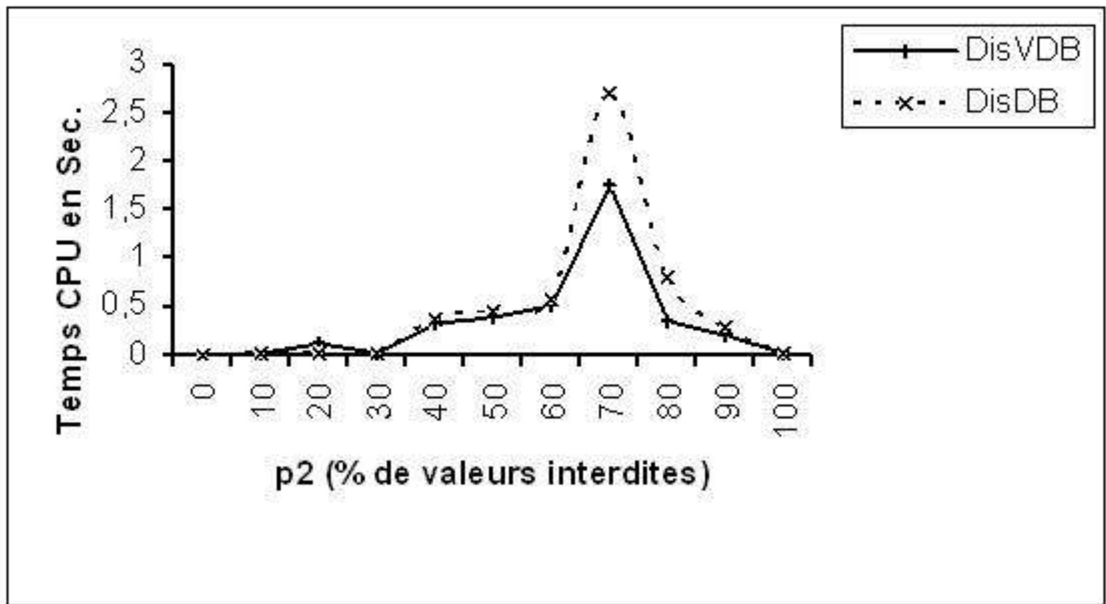

Figure 2. Comparaison entre DisVDB et DisDB sur des problèmes peu denses (p1=0.3)

6. Conclusion

Dans le domaine des Problèmes de Satisfaction de Contraintes Valués et Distribués (DisVCSP), nous avons consacré notre étude à la distribution de l'algorithme Valued Dynamic Backtracking VDB (Dago, 1997) et la valuation de la méthode Distributed Dynamic Backtracking (Bessière, 2001). C'est à dire, nous avons développé une méthode permettant de résoudre les DisVCSP : Distributed Valued Dynamic Backtracking

(DisVDB). Nous avons d'abord défini comme algorithme d'ordonnancement des agents EDisAO (Extension Distributed Agents Ordering) (Belaissaoui, 2002). EDisAO est utilisée pour accroître les performances des algorithmes de résolution des DisCSP. DisVDB a visée à bénéficier des avantages des deux approches : la gestion des nogoods valués, la complétude de la recherche et un haut niveau d'asynchronisme entre les agents. Il a été prouvé que cet algorithme est correct et complet. Les expérimentations mettent en œuvre DisDB (Bessière, 2001) et notre méthode dans un environnement distribué utilisant une granularité p<n (p agents et n variables).

Références

Belaissaoui, M., Bouyakhf, EH. (2001). Les problèmes de satisfaction de contraintes distribués et le backtracking, EJNDP, Volume 12, sept.-01, (rerir.univ-pau.fr/rerir2.html).

Belaissaoui, M. (2002). Contribution à l'étude de Raisonnement Temporel et au Traitement des Problèmes de Satisfaction de Contraintes Distribués, thèse de doctorat, université Mohammed V Agdal, Faculté des Sciences Rabat, 2002.

Bessière, C. Maestre, A. et Meseguer, P (2001). Distributed dynamic backtracking. In M. Silaghi, editor, Proceedings of the IJCAI'01 workshop on distributed constraint reasoning, Seattle, pages 9-16, 2001.

Dago, P. et Verfaillie, G. (1996). Nogood Recording for Valued Constraint Satisfaction Problems. Proceedings of ICTAI'96, pages 132-139, 1996.

Dago, P. (1997). Backtrack Dynamique valué, 6ème journées francophones de programmation par contraintes (JFPLC), 26-28 Mai, 1997, Orléans, France.

Verfaillie, G., Lemaîître, M. et T. Schiex(1996). Russian Doll Search for Solving Constraint Optimization Problems. Dans Proceedings of AAAI-96, pages 181-187,1996.

Hamadi, Y. (1999). Traitement des problèmes de satisfaction de contraintes distribués, thèse de doctorat, université Montpellier II, 99.

Koster, A. (1999). Frequency Assignment - Models and Algorithms. PhD thesis, University of Maastricht, Novembre 1999.

Jégou, P. et Terrioux, C.(2003). Recherche arborescente bornée pour la résolution de VCSP, JNPC, 2003, Amiens, France.

Van Beek, P. C sources for CSP programming. (http://web.cs.ualberta.ca:80/vanbeek/)

Schiex, T., Fargier, H. et Verfaillie, G.(1995). Valued Constraint Satisfaction Problems : hard and easy problems. Dans *Proceedings of IJCAI-95,* pages 631-637, 1995.

Schiex, T., Fargier, H. et Verfaillie, G.(1997). Problèmes de Satisfaction de Contraintes Valués, Revue d'IA, Volume 11, Numéro 3, 1997.

Yokoo, M., Ishida, T., et Kubawara, K. (1990). Distributed constraint satisfaction for DAI problems. In Huhns, M. N., editor, Proc. of *the 10th International Workshop on Distributed Artificial Intelligence*, chp 9.

Yokoo, M., Durfee, E., Ishida, T., et Kuwabara, K. (1992). Distributed constraint satisfaction for formalizing distributed problem solving. In *12th Int. Conf. on Distributed Computing Systems*, 614-624.

Yokoo, M. et Hirayama, K. (1998). Distributed constraint satisfaction algorithm for complex local problems. In *ICMAS*, 372-379.