La famille ABT

Christian Bessière
LIRMM-CNRS
161 rue Ada
34392 Montpellier
France
bessiere@lirmm.fr

Arnold Maestre
LIRMM-CNRS
161 rue Ada
34392 Montpellier
France
maestre@lirmm.fr

Pedro Meseguer
IIIA-CSIC
Campus UAB
08193 Bellaterra
Spain
pedro@iiia.csic.es

Résumé

Depuis quelques années, la communauté IA affiche un intérêt croissant pour la résolution de problèmes distribués. Dans le domaine du raisonnement par contraintes distribuées, plusieurs procédures de recherche arborescente ont été proposées pour trouver une solution dans un réseau de contraintes. Elles se distinguent par la manière dont elles mémorisent les combinaisons de valeurs infructueuses (nogoods) et par les méthodes qu'elles mettent en oeuvre pour détecter l'obsolescence potentielle des données mémorisées. Dans cet article, nous proposons un cadre unificateur pour ces algorithmes de recherche asynchrones. Nous étudions les choix qui peuvent être faits pour obtenir un algorithme correct et complet. Notre cadre permet de décrire et de comprendre les éléments de base de ces procédures, et de mettre en relief leurs similarités et leurs différences.

1 Introduction

La résolution de problèmes distribués et le paradigme agents suscitent un certain engouement dans le domaine de l'IA. Divers agents, au comportement autonome, détiennent chacun une partie du problème à résoudre, et doivent collaborer pour construire une solution globale. Internet recèle de nombreux problèmes réels qui peuvent être traités suivant cette approche.

Plusieurs travaux ont déjà été réalisés sur la satisfaction de contraintes dans sa forme distribuée (voir [13] pour une introduction). Ces travaux sont motivés par l'existence de problèmes naturellement distribués, pour lesquels il est impossible ou peu souhaitable de réunir toutes les données du problème sur un seul site, afin de le résoudre en utilisant un algorithme centralisé. Les raisons les plus immédiates sont le temps des communications et le coût de traduction de chaque sous-problème dans un format commun. Mais fournir à un agent unique toutes les données du problème peut aussi être exclu pour des raisons de sécurité ou de confidentialité. Si par exemple les agents appartiennent à différentes entreprises dans le cadre d'une collaboration industrielle ou commerciale, chaque partie aura à coeur de trouver une solution au problème sans pour autant révèler l'ensemble de ses données internes à un tiers.

Les algorithmes complets pour la résolution de CSP distribués doivent beaucoup à Yokoo et à ses collaborateurs, pionniers du domaine avec *asynchronous backtracking* (ABT) [11, 9, 12]. Cet algorithmes impose un ordre total entre les agents. Lors d'un échec, il peut être nécessaire d'ajouter des liens de communication entre des agents auparavant indépendants. Ensuite, des nogoods sont échangés entre les agents, et mémorisés. ABT a été utilisé comme base par plusieurs travaux, et plusieurs extensions ont été proposées, comme le réordonnancement dynamique des agents [8] ou le maintien de la cohérence [7].

L'algorithme *Distributed Backtracking* (DiBT) propose une approche différente, sans échange de *nogoods*. Malheureusement, tel qu'il est présenté dans [4, 3] DiBT n'est pas complet [10]. Plus récemment, un nouvel algorithme, appelé *Asynchronous Aggregation Search* (AAS) est présenté dans [6]. Il est basé sur l'échange d'ensembles de solutions partielles dans un modèle distribué orienté contraintes, alors que les algorithmes précédents se placent dans un modèle distribué orienté variables.

Saisir avec précision le comportement d'un algorithme est bien plus complexe dans un environnement distribué que dans le cadre classique, centralisé. Dans le cas d'une procédure de recherche, il est assez difficile de déterminer les conditions nécessaires ou suffisantes pour assurer la complétude d'une procédure. Dans cet article, nous proposons un cadre unificateur pour la recherche arborescente asynchrone dans un réseau de contraintes. Nous présentons en premier lieu une procédure simple qui comporte les principales caractéristiques des algorithmes de recherche pour la satisfaction de contraintes distribuées. Nous montrons pourquoi une telle procédure n'est pas adéquate. Puis, nous analysons les améliorations à introduire pour obtenir un algorithme adéquat et complet. Suivant les modifications apportées à notre procédure de base, nous obtenons des algorithmes déjà connus, ou d'autres, inédits. Nous pensons que cette démarche peut aider à saisir le comportement des méthodes de recherche asynchrones, et mettre l'accent sur les similitudes et les différences entre les différentes versions.

L'article est organisé comme suit. La Section 2 fournit quelques définitions essentielles pour la satisfaction de contraintes distribuée. La Section 3 présente la procédure élémentaire dont nous dériverons tous les autres algorithmes. Ces extensions sont décrites en Section 4. Quelques comparaisons expérimentales préliminaires constituent la Section 5. La Section 6 conclut.

2 Définitions préliminaires

Un réseau de contraintes est un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$, où $\mathcal{X} = \{x_1, \ldots, x_n\}$ est un ensemble de n variables, $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$ est l'ensemble de leurs domaines respectifs et $\mathcal{C}$ est un ensemble de contraintes sur les combinaisons de valeurs possibles pour les variables. Le problème de satisfaction de contraintes (CSP) revient à trouver pour les variables du problème des valeurs qui ne violent aucune contrainte. Dans cet article, nous ne considérerons que des contraintes entre deux variables (contraintes *binaires*). On notera c_{ij} une contrainte entre x_i et x_j. Un CSP distribué (DisCSP) est un CSP dont les variables, domaines et contraintes sont distribués sur un ensemble d'agents autonomes. Formellement, un réseau de contraintes distribué orienté variables est un quintuplet $(\mathcal{X}, \mathcal{D}, \mathcal{C}, \mathcal{A}, \phi)$, où $\mathcal{X}$, $\mathcal{D}$ et $\mathcal{C}$ sont définis comme précédemment. $\mathcal{A} = \{1, \ldots, p\}$ est un ensemble de p agents, et $\phi : \mathcal{X} \rightarrow \mathcal{A}$ est la fonction qui associe chaque variable à un agent.

La distribution des variables forme une bipartition de $\mathcal{C}$ en $\mathcal{C}_{intra} = \{c_{ij} | \phi(x_i) = \phi(x_j)\}$ et $\mathcal{C}_{inter} = \{c_{ij} | \phi(x_i) \neq \phi(x_j)\}$, qu'on nomme ensemble des contraintes *intra-agent* et *inter-agents*, respectivement. Une contrainte intra-agent c_{ij} n'est connue que de l'agent détenteur de x_i et x_j. On considère généralement qu'une contrainte inter-agents c_{ij} est connue des agents $\phi(x_i)$ et $\phi(x_j)$ [12, 4].

Comme dans le cadre centralisé, une solution est une affectation de valeurs aux variables qui ne viole aucune contrainte (bien que la littérature des CSP distribués se concentre sur la satisfaction des contraintes inter-agents). Les CSP distribués sont résolus par l'action collective et coordonnée des agents de $\mathcal{A}$, chacun exécutant un processus de satisfaction de contraintes. Les agents communiquent par envois de messages, avec les postulats suivants [12],

1. Un agent ne peut envoyer de message que s'il connaît l'adresse du destinataire
2. 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.

La plupart des algorithmes de recherche pour les DisCSP sont basés sur l'utilisation de nogoods. La suppression d'une valeur x_k est justifiée par un nogood orienté de la forme : $x_i = a \wedge x_j = b \wedge \ldots \Rightarrow x_k \neq c$. On définit la partie gauche et la partie droite de cette expression à partir de la position du $\Rightarrow$. Quand toutes les valeurs d'une variable x_k sont éliminées de la sorte, l'ensemble N_k de ces nogoods est résolu par rapport à l'une des variables incriminées, x_j, et un nouveau nogood est produit, avec en partie gauche la conjonction des parties gauches de tous les nogoods de N_k, en omettant x_j. La partie droite est $x_j \neq b$ si b était la valeur de x_j dans N_k. 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 ($\forall x_i \in \mathcal{X}, \phi(x_i) = i$). Toutes les contraintes sont donc des contraintes inter-agents : $\mathcal{C} = \mathcal{C}_{inter}$ et $\mathcal{C}_{intra} = \emptyset$.

Rappelons que cette définition des CSP distribués correspond à celle utilisée dans ABT et DIBT, mais pas celle évoquée dans AAS, où il n'y a pas de contraintes inter-agents. Une variable partagée par deux contraintes appartenant à des agents différents est dupliquée sur chacun des agents. Le protocole de communication garantit la cohérence des valeurs affectées à cette variable sur chaque agent (simulant une contrainte d'égalité entre les deux copies de la variable).

3 Le cadre unificateur

Nous allons décrire un cadre générique pour la résolutions de CSPs distribués, que nous appellerons ABT_{kernel}. On suppose que les agents sont ordonnés statiquement suivant un certain critère. Par la suite, on dira qu'un agent est plus haut dans l'ordre, ou a un niveau supérieur, s'il aune priorité plus forte que celui auquel il est comparé. Si l'on considère un agent générique $self$, $\Gamma^-(self)$ est l'ensemble des agents partageant une contrainte avec $self$ et placés plus haut dans l'ordre. Inversement, $\Gamma^+(self)$ est l'ensemble des agents contraints par $self$ et apparaissant plus bas dans l'ordre. On désignera couramment les agents de $\Gamma^-(self)$ comme les parents de $self$, ceux de $\Gamma^+(self)$ comme ses enfants. Les contraintes sont orientées des agents de forte priorité vers les faibles, ce qui produit un graphe de contraintes acyclique.

Chaque agent mémorise localement une certaine quantité d'informations sur l'état global de la recherche, à savoir un contexte et un ensemble de nogoods. Le contexte est l'ensemble des valeurs affectées (d'après les connaissances de $self$) aux agents de plus haut niveau. Il est nécessaire que ce contexte soit cohérent avec l'ensemble des nogoods de $self$. Dés lors, les agents échangent des affectations et des nogoods, et exécutent la recherche en répétant une boucle très simple jusqu'à ce qu'ils trouvent une solution ou détectent l'incohérence.

La boucle de réception principale est composée de quatre étapes élémentaires. Tout d'abord, l'agent lit un message reçu, et s'arrête si celui-ci annonce l'échec de la recherche. Autrement, la deuxième étape consiste, si le message est pertinent, à l'utiliser pour mettre à jour le contexte. Ensuite, une solution locale doit être sélectionnée. Enfin, un nouveau message est généré pour informer les enfants de $self$ de sa nouvelle valeur, ou pour stopper la procédure de recherche globale en cas d'échec.

L'ensemble du processus est illustré en Figure 1. Le filtrage par pertinence appliqué aux données reçues ($CheckAgentView()$, ligne 1) accepte tout les messages comportant une nouvelle affectation (message Info), mais défausse tout nogood (message Back) qui n'est pas cohérent avec le contexte local. Tous les messages acceptés sont utilisés pour mettre à jour la base de connaissances locale (ligne 2), soit en changeant une valeur dans le contexte et en assurant la cohérence de l'ensemble des nogoods avec cette nouvelle valeur, soit en mémorisant le nogood fraîchement reçu, ce qui a pour effet de supprimer la valeur courante de $self$. Si la valeur courante est valide, la boucle s'achève, $self$ est prêt à recevoir le message suivant. Sinon, l'agent doit à présent choisir une nouvelle valeur, qui soit cohérente avec les données mises à jour (ligne 4). Dans ce but, il choisit une valeur autorisée par tous ses nogoods et teste sa cohérence (fonction $ChooseValue()$). Un échec entraîne la suppression de cette valeur par un nouveau nogood ($ChooseValue()$, lignes 3-4). On remarquera que les nogoods ne font référence qu'à des variables de $\Gamma^{-}(self)$ lorsqu'ils sont générés. Dés qu'un valeur s'avère compatible, $self$ envoie un message pour informer tous ses enfants. ($CheckAgentView()$, lignes 5-7). Si le domaine se vide, $self$ devra résoudre ses nogoods et lui signifier par un message l'échec de la configuration courante, puis mettre à jour les données locales et poursuivre la recherche d'une valeur compatible (procédure $Backtrack()$). En cas d'échec sans possibilité de retour arrière, l'incohérence est prouvée, et la recherche peut se terminer ($Backtrack()$, ligne 4). Le message d'arrêt implique un agent supplémentaire, appelé $Systme$, qui doit à son tour arrêter l'ensemble du système. Tout agent peut assumer ce rôle, et une procédure classique de diffusion peut être utilisée.

La procédure de retour arrière mérite une attention particulière, car elle traite précisément les problèmes liés à la distribution. Quand un domaine se vide, les nogoods pour la variable correspondant sont résolus sur l'agent x_l de plus basse priorité dans le conflit. Le nouveau nogood est alors envoyé à x_l pour éliminer sa valeur courante. L'expéditeur peut oublier la valeur courante de x_l qui est destinée à changer, et défausser les nogoods correspondants. En outre, puisque x_l est le plus petit agent impliqué dans le conflit, le nogood qu'il reçoit ne fait référence qu'à des agents de plus haute priorité.

Finalement, la recherche peut se stabiliser dans un état où une valeur est affectée à chaque agent et aucune contrainte n'est violée. Ce type d'état peut être détecté par des algorithmes spécialisés (par exemple, [2]) lorsque plus aucun message ne circule dans le système. La recherche a abouti a une solution globale. Ce noyau élémentaire est adéquat, puisque lorsqu'une solution est annoncée, tous les agents sont dans un état stable. Hors si

```
procedure ABT_kernel()
compute Γ+(self), Γ−(self) ;
CheckAgentView(null) ;
end ← false ;
while (¬end) do
  msg ← getMsg() ;
  switch(msg.type)
    Stop     : end ← true ;
    else     : CheckAgentView(msg) ;

procedure CheckAgentView(msg)
if msg is relevant
  update local context ;
  if consistent(myValue, myContext) then return ;
  myValue ← ChooseValue() ;
  if (myValue) then
    for each child ∈ Γ+(self) do
      sendMsg :Info(child, myValue) ;
  else Backtrack ;

procedure Backtrack()
newNogood ← solve(myNogoods) ;
if (newNogood = empty)
  end ← true ;
  sendMsg :Stop(system) ;
else
  sendMsg :Back(newNogood) ;
  update local context ;
  CheckAgentView(null) ;

function ChooseValue()
for each v ∈ D(self) not eliminated by myNogoods do
  if consistent(v, myContext) then return (v) ;
  else /* v inconsistent with x_j's current value */
    add(X = val_X ⇒ ¬v, myNogoods) ;
return (empty) ;
```

FIG. 1 – Un algorithme générique pour la recherche asynchrone

une contrainte était violée, au moins un des agents (celui qui détecte le conflit) enverrait un message. De plus, ABT_{kernel} ne peut pas conclure à l'incohérence si une solution existe. En effet, tout nogood généré initialement comme résultat d'un message Info est une contrainte redondante dans le CSP à résoudre. Comme tous les nogoods supplémentaires sont le résultat d'une inférence logique, un nogood vide ne peut pas être généré lorsqu'une solution existe.

Si l'on fait correspondre les notions de variables *passées* et *futures* avec celles de haute ou basse priorité, la procédure peut être vue comme une version distribuée de l'algorithme *conflict directed backjumping* (CBJ), exécutant des sauts parmi l'ensemble des agents potentiellement responsables des conflits en remontant dans l'ordre des priorités, de la même manière que son homologue centralisé remonte dans l'ordre d'instantiation.

Malheureusement, la terminaison d'ABT_{kernel} n'est pas garantie. Le principal obstacle à cet objectif est la pertinence des informations mémorisées, et plus particulièrement des nogoods. On a vu que la manière dont ceux-ci sont générés et transmis garantit que toute variable apparaissant en partie gauche précède $self$. Mais ceci ne garantit pas que toutes ces variables sont dans$\Gamma^-(self)$.

Si c'était le cas, un nogood n_k pourrait être défaussé dés que $self$ reçoit une nouvelle affectation pour l'une des variables impliquées. Mais comme n_k peut contenir des informations à propos d'un agent x_u, inconnu et plus prioritaire, on ne peut pas tester sa pertinence localement, puisque x_u n'informe pas $self$ de ses changements de valeur. Ainsi, un agent peut être amené à stocker une information qui ne décrit plus l'état global du système, mais qui justifie la suppression d'une valeur dans son domaine. Si cette valeur fait partie d'une solution, celle-ci sera ignorée. On a vu que l'algorithme ne peut pas déduire l'insolubilité si une solution existe. Il peut donc ne pas se terminer si toutes les solutions sont ignorées de la sorte.

Pour atteindre la complétude, un algorithme basé sur ce noyau doit être capable de défausser les nogoods non pertinents, quelles que soient les variables impliquées. En fait, on doit s'assurer qu'une information obsolète ne peut pas persister indéfiniment dans le réseau. Nous allons à présent décrire de telles stratégies.

4 Stratégies pour atteindre la complétude

Puisque notre principal problème est le manque d'information en provenance des agents de heut niveau, une solution envisageable est d'ajouter des liens de communication dans le réseau formé par les agents, pour apporter au détenteur d'un nogood donné une vue plus précise de l'état global du système, et lui permettre de déterminer si ce nogood est valide.

Un lien de communication peut être vu comme une contrainte tautologique, ou encore une extension triviale de l'ensemble des voisins de l'agent de heut niveau. Bien que l'ajout de tels liens ne change pas l'espace des solutions pour le problème considéré, il permet au destinataire de défausser systématiquement les informations périmées.

4.1 Ajout de liens avant la recherche

L'opération peut être effectuée comme un pré-calcul sur le graphe de communication. Une approche brutale consisterait à relier chaque agent à tous ses homologues de priorité plus faible, mais c'est plus qu'il n'est nécessaire : deux agents qui ne partagent aucun descendant ne peuvent pas faire partie d'un même conflit. En revanche, pour chaque paire d'agent partageant un descendant, le plus prioritaire, x_i, doit ajouter l'autre, x_j, à son ensemble Γ^+ pour que x_j puisse traiter les nogoods se référant à x_i.

Nous appellerons cette procédure ABT_0, parce que qu'elle est assez proche de l'algorithme de Yokoo, tout en étant un peu moins subtile dans son application : tous les liens sont ajoutés à l'avance plutôt que pendant la recherche, ce qui signifie qu'il n'est pas nécessaire d'élaborer de messages particuliers ou de procédures spécialisées pour gérer la modification dynamique du réseau de communications.

Il est possible de modifier ABT_0 pour que les agents cessent de mémoriser des nogoods. Il suffit pour cela d'ajouter les liens symétriques durant la phase de pré-calcul. De cette manière, tout agent plus prioritaire pouvant entrer en conflit avec $self$ apparaît dans $\Gamma^-(self)$. Concaténer le contexte avec le nogood qui a supprimé la dernière valeur est dés lors suffisant pour s'assurer que la recherche s'effectue sans accrocs. Une forme partielle de cet algorithme est publiée dans [3] sous le nom de $DiBT$.

Malheureusement, cette modification n'est pas très bénéfique. En fait, bien qu'il fasse l'économie de l'espace de stockage des nogoods (qui reste polynomial dans le cas d'ABT_0), cet algorithme affaiblit le comportement du système en le rendant comparable à l'algorithme centralisé *graph based backjumping*. De ce fait, certains messages Back sont adressés à des agents qui ne font pas partie du conflit, et ne peuvent donc pas aider à le résoudre. En outre, l'insolubilité ne peut être détectée que par un agent qui n'a pas de parents, ce qui peut entraîner l'exploration de sous-arbres de recherches très importants, en vain.

On peut continuer à dégrader ABT_0, en ajoutant tous les agents de plus forte priorité dans $\Gamma^-(self)$, et tout ceux de plus faible priorité dans $\Gamma^+(self)$. De cette manière, il n'est pas non plus nécessaire de stocker de nogoods, car les retours arrières sont dirigés vers le premier agent de $\Gamma^-(self)$, qui est en fait le prédecesseur direct de $self$ dans l'ordre. Le contexte de $self$ est tout de même attaché au message pour permettre au destinataire de juger de son éventuelle obsolescence. On voit bien que cette procédure se comporte comme une version distribuée et asynchrone du backtrack chronologique, l'ordre des agents remplaçant l'ordre d'instantiation.

4.2 Ajout de liens pendant la recherche

Relier toutes les sources de conflit avant la recherche a quelques inconvénients, comme temps nécessaire au pré-calcul, l'encombrement du graphe de communication et le surcoût associé en terme d'envoi de messages. Au lieu de cela, il est possible d'attendre qu'un conflit soit effectivement détecté pendant la recherche, et de créer dynamiquement un lien à ce moment-là. C'est exactement l'algorithme ABT de Yokoo, qui sera désigné comme ABT_1 dans la suite de cet article par souci d'homogénéité.

ABT_1 est adéquat et complet. Il utilise un quatrième type de message, *AddLink*, pour réclamer l'ajout d'un nouveau lien de communication. Dés qu'un agent x_j reçoit des informations à propos d'un agent x_i qu'il ne connaissait pas, il envoie une requête *Addlink*. A sa réception, x_i étend son ensemble Γ^+ pour y inclure x_j, et envoie sa valeur courante à ce nouveau voisin. De cette manière, tout agent qui détient un nogood à la garantie d'être informé en un temps fini si l'une des variables impliquées dans ce nogood change.

4.3 Ajout de liens temporaires

Etant donné que les liens utilisés dans ABT_1 ne servent qu'à informer un agent que certains de ses nogoods ne sont plus pertinents, on peut préférer les ajouter temporairement. En effet, dés que la nouvelle valeur de l'agent lié est connue, tous les nogoods correspondants sont défaussés, et les informations en provenance de cet agent perdent leur intérêt. Il est alors possible de se débarasser du lien supplémentaire pour économiser quelques envois de message. Ce comportement peut être obtenu par une modification mineure du comportement des agents qui reçoivent un message *AddLink*.

Chaque fois qu'un agent x_j demande l'ajout d'un lien, il génère un jeton pour matérialiser le lien avec le destinataire, x_i. Ce dernier, à la réception du message, ajoute un lien vers x_j. Cependant, si la valeur rapportée dans la requête est valide, aucun message n'est envoyé : tant que x_j ne reçoit rien de x_i, il sait que cette valeur est correcte. Dés que x_i change de valeur il informe tous ses enfants et les agents liés, puis supprime les liens ajoutés. A l'autre bout du fil, x_j et tous les autres agents liés à x_i peuvent défausser leur jeton lien ainsi que tous les nogoods invalidés par ce changement.

Cette nouvelle procédure est appelée ABT_2. Elle garantit qu'un lien sera disponible à chaque fois que ce sera nécessaire, mais sans exiger que les agents de haut niveau diffusent en permanence leurs informations vers un voisinage étendu.

4.4 Sans liens

Le problème de la pertinence des nogoods peut être abordé différemment. Au lieu de mettre en oeuvre des mécanismes qui garantissent qu'un agent est informé lorsque l'un de ses nogoods cesse d'être pertinent, on peut demander à chaque agent d'évaluer l'impact de ses propres décisions sur la valeur des informations qu'il détient, et de mettre à jour ces données en conséquence. Plus précisément, lorsque l'ensemble des nogoods est résolu par rapport au coupable le plus proche pour lui envoyer un nouveau nogood, l'expéditeur sait que ce message peut atteindre, par retours successifs, tous les agents qui y apparaissent. Cela signifie que tous ces agents sont susceptibles de changer de valeur. Il n'y a pas lieu de s'inquiéter pour les parents de x_i, qui l'informeront en cas de modification. En revanche, s'il y a dans le nogood des agents auxquels x_i n'est pas lié, leur modification peut invalider tous les nogoods dans lesquels ils apparaîssent, sans moyen pour x_i d'être informé. Pour palier à ce problème, x_i peut oublier ces nogoods peu sûrs, éliminant à la base le risque de garder des nogoods invalides.

Mais qu'advient-il lorsqu'un changement survient, avec une cause autre que x_i ? Y a-t-il un risque pour que x_i, mal informé, conserve indéfiniment un nogood invalide ? En fait, non. En effet, si l'un de ces agents supérieurs x_k change de valeur, il y a deux cas disjoints. Soit la valeur de x_i supprimée par le nogood devenu obsolète appartient à toutes les solutions, soit il existe une solution avec une autre valeur de x_i. S'il existe une autre solution, x_i conservera son nogood invalide jusqu'à la fin de la recherche, mais le système produira une solution. Si la valeur masquée de x_i est un passage obligé pour une solution, x_i sera forcé d'essayer toutes les autres options avant de faire un retour arrière. Et comme le nouveau nogood fait mention de x_k, x_i oublie le nogood correspondant avant de reprendre la recherche.

Cet algorithme fait l'objet d'une description approfondie dans [1], sous le nom de $DisDB$. Pour des raisons d'homogénéité, nous l'appelerons ABT_3 dans la suite de cet article.

Il est intéressant de remarquer que le fonctionnement de ces algorithmes ne dépend pas d'un environnement homogène : différents agents peuvent opter pour différentes stratégies, puisqu'au final chaque agent est seul responsable de la validité de son ensemble de nogoods. La seule limite à ceci est l'addition de liens : pour permettre à chaque agent de choisir son protocole, il est nécessaire que tous les agents puissent traiter les requêtes d'ajout de lien, définitif ou temporaire.

5 Expériences

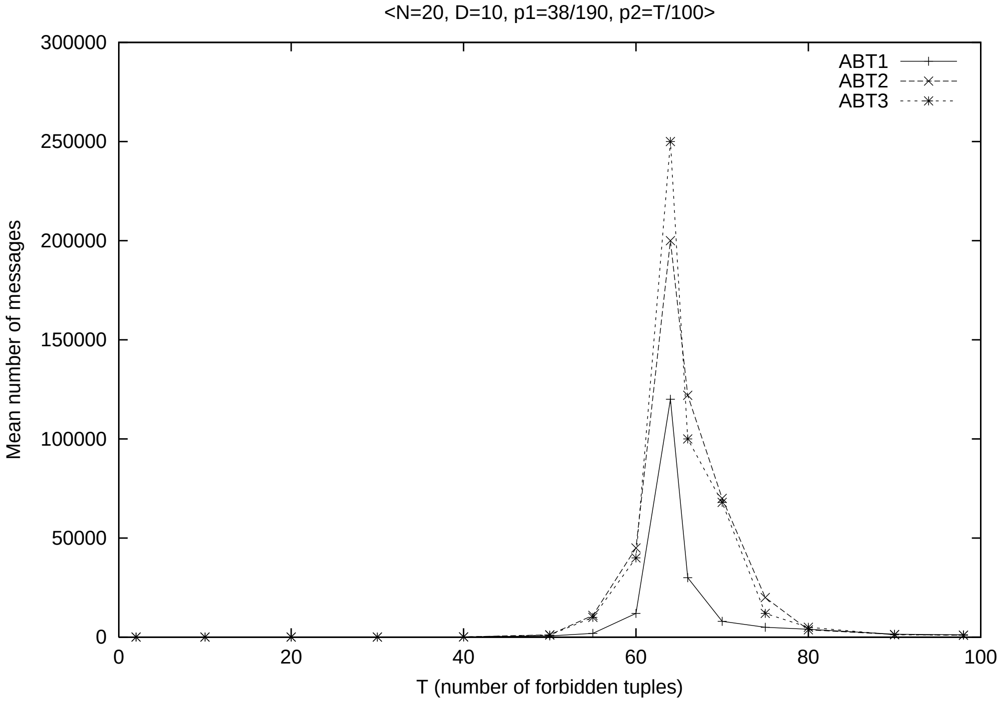

FIG. 2 – Comparaison entre ABT_1, ABT_2, et ABT_3 sur des réseaux de 20 variables, 10 valeurs par domaine, et 38 contraintes parmi les 190 possibles d'un graphe complet

Nous avons mené quelques expériences préliminaires pour comparer le comportement des différents versions d'ABT présentées en Section 4. Pour cette évaluation, nous avons utilisé un simulateur à évènements discrets. 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. On suppose que tout message expédié au temps t est disponible pour son destinataire au temps $t + 1$, et que le processus système se charge de la détection de la terminaison. Pour renforcer le caractère asynchrone de la simulation, les messages reçus au cour d'un cycle sont ordonnés de manière aléatoire.

Nos agents ont pour tâche de résoudre des CSP aléatoires uniformes générés conformément au modèle B [5]. Le générateur admet quatre paramètres : le nombre de variables, noté N, la taille initiale des domaines, notée D, la proportion de contraintes dans le réseau, $p1$, et la proportion de paires interdites dans une contrainte, $p2$. Un problème peut donc être désigné comme un réseau $<N, D, p1, p2>$. On affecte à chaque agent une variable et les contraintes qui la relient à ses voisines.

Des réseaux relativement peu denses $<20, 10, 0.20, p2>$ sont générés, et résolus pour différentes valeurs de $p2$ par incrément de 5 pourcents. Cent problèmes sont générés à chaque point, et étudiés par un ensemble d'agents exécutant ABT_1, un autre exécutant ABT_2, et un dernier utilisant ABT_3. Tous les agents mémorisent au plus un nogood par valeur, et la priorité entre agents est déterminée par l'ordre lexicographique.

Attendu que la distribution était simulée, tous les calculs ayant lieu sur un seul processeur, nous avons mesuré le nombre de messages nécessaires à la résolution de chaque problème.

La Figure 2 illustre les résulats. La moyenne du nombre de messages est calculée sur 100 instances différentes pour chaque valeur de $p2$. Au pic de complexité, qui se situe au seuil entre la satisfiabilité et l'incohérence, on peut voir que les performances d'$ABT1$ sont meilleures que celles d'ABT_2, qui elles mêmes sont meilleures que celles d'ABT_3. Ceci peut probablement s'expliquer par le fait que lorsqu'on va de ABT_1 à ABT_3, les agents sont de moins en moins informés des valeurs actuelles de leurs homologues. On peut donc penser que sur les instances difficiles qui nécessittent un grand nombre de retours arrière, diffuser plus largement l'information, comme le fait ABT_1, économise suffisament de travail aux agents de faible priorité pour compenser les messages supplémentaires, et finit globalement par payer.

En analysant les résultats instance par instance, on remarque que le comportement moyen au seuil est dominé par un petit nombre de problèmes très difficiles. Sur les instances moins difficiles du seuil, et celles à l'extérieur du seuil, le classement est inversé, ABT_3 domine légèrement ABT_2, qui fait un peu mieux qu'ABT_1.

6 Conclusion

Nous avons décrit une procédure élémentaire pour la recherche asynchrone. Ce protocole est correct mais n'offre aucune garantie de terminaison. Nous avons donc décrit un certain nombre d'extensions de la procédure de base, qui traitent les nogoods et les communications entre agents de manière à assurer un résultat en temps fini. Certaines de ces extensions sont des algorithmes connus, comme le classique ABT de Yokoo, d'autres sont originales, mais elles ne diffèrent que dans la manière dont elles étendent et complètent notre noyau élémentaire. Cette caractérisation de la recherche asynchrone est intéressante car elle permet une meilleure compréhension de mécanismes non triviaux. Finalement, nous avons procédé à quelques expériences qui montrent qu'actualiser le contexte des agents aussi tôt que possible est fructueux sur les problèmes difficiles, malgré les messages supplémentaires que cela nécessite.

Références

[1] C. Bessière, A. Maestre, and P. Meseguer. Distributed dynamic backtracking. In M.C. Silaghi, editor, *Proceedings of the IJCAI'01 workshop on Distributed Constraint Reasoning*, pages 9–16, 2001.

[2] K.M. Chandy and L. Lamport. Distributed snapshots : determining global states of distributed systems. *ACM Transactions on Computing Systems*, 3 :63–75, 1985.

[3] Y. Hamadi. *Traitement des problèmes de satisfaction de contraintes distribués.* PhD thesis, University of Montpellier II, July 1999. in French.

[4] Y. Hamadi, C. Bessière, and J. Quinqueton. Backtracking in distributed constraint networks. In *Proceedings ECAI'98*, pages 219–223, Brighton, UK, 1998.

[5] P. Prosser. An empirical study of phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81 :81–109, 1996.

[6] M.C. Silaghi, D. Sam-Haroud, and B. Faltings. Asynchronous search with aggregations. In *Proceedings AAAI'00*, pages 917–922, Austin TX, 2000.

[7] M.C. Silaghi, D. Sam-Haroud, and B. Faltings. Consistency maintenance for abt. In *Proceedings CP 2001*, pages 917–922, Paphos, Cyprus, 2001.

[8] M.C. Silaghi, D. Sam-Haroud, and B. Faltings. Hybridizin abt and awc into a polynomial space, complete protocol with reordering. Technical Report 364, EPFL, 2001.

[9] M. Yokoo. Asynchronous weak-commitment search for solving distributed constraint satisfaction problems. In *Proceedings CP'95*, pages 88–102, Cassis, France, 1995.

[10] M. Yokoo. Private communication, 2000.

[11] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing distributed problem solving. In *Proceedings DCS*, pages 614–621, 1992.

[12] M. Yokoo, E.H. Durfee, T. Ishida, and K. Kuwabara. The distributed constraint satisfaction problem : Formalization and algorithms. *IEEE Trans. Knowledge and Data Engineering*, 10 :673–685, 1998.

[13] M. Yokoo and T. Ishida. Search algorithms for agents. In G. Weiss, editor, *Multiagent Systems*, pages 165–199. MIT Press, 1999.