Retour-arrière basé sur les divergences pour l'optimisation distribuée

Jonathan Gaudreault, Jean-Marc Frayet, Gilles Pesant

Retour-arrière basé sur les divergences pour l'optimisation distribuée

Jonathan Gaudreault[1,3] **Jean-Marc Frayret**[2,3] **Gilles Pesant**[1]

[1] Département de génie informatique, Ecole Polytechnique de Montréal, Canada
[2] Département de mathématiques et de génie industriel, Ecole Polytechnique de Montréal, Canada
[3] Consortium de recherche FORAC, Université Laval, Québec, Canada

{jonathan.gaudreault,jean-marc.frayret,gilles.pesant}@polymtl.ca

Résumé

Les approches dites DCOP (Distributed Constraint Optimization Problems) sont de plus en plus utilisées pour formaliser des situations où plusieurs agents doivent coopérer pour résoudre un problème d'optimisation. Parfois, les agents constituent une organisation fortement hétérogène. Un réseau de compagnies en est un bon exemple. Dans ce contexte, la structure du réseau et les règles d'affaires régissent les échanges qui sont possibles pendant la résolution du problème. L'espace de recherche d'une solution s'apparente alors à un arbre de recherche classique pour un problème d'optimisation combinatoire centralisé. Or, pour les problèmes centralisés, les stratégies de recherche exploitant un retour-arrière basé sur les divergences (LDS, par exemple) donnent généralement de bons résultats. Nous proposons donc un nouvel algorithme qui permet aux agents de réaliser de manière distribuée une recherche basée sur les divergences. La méthode proposée permet le travail concurrent des agents et elle est tolérante aux délais aléatoires de communication. Elle est complète, mais elle vise d'abord à obtenir de bonnes solutions en un temps très court. L'approche a été évaluée sur des problèmes industriels réels pour lesquels elle a montré de bonnes performances.

1 Introduction

Plusieurs algorithmes génériques pour les problèmes d'optimisation distribués (DCOP) ont été proposés ces dernières années. Dans [9], les auteurs rappellent que les algorithmes génériques n'exploitent pas toujours les particularités de certaines classes de problèmes, notamment la forme des contraintes et l'existence d'heuristiques propres au domaine. Par ailleurs, il se présente des situations pour lesquelles le réseau d'agents représente une organisation existant *a priori* et constituée d'agents hétérogènes (un réseau d'entreprises, par exemple). Or, la structure et le contexte organisationnel peuvent alors définir les interactions permises entre les agents lors de la résolution [5].

Nous nous intéressons à une classe particulière de problèmes d'optimisation distribués, dits hiérarchiques. Comme pour les problèmes classiques, il existe un réseau d'agents désirant minimiser une fonction objectif. Cependant :

- Le problème est divisé en sous-problèmes, et il existe une séquence dans laquelle ils doivent être résolus. Celle-ci est imposée par le contexte d'affaires ;
- Un agent est contraint dans la résolution d'un sous-problème par les solutions retenues pour les sous-problèmes précédents ;
- L'objectif global peut-être formulé en fonction des variables d'un seul sous-problème (généralement le dernier).

Un large éventail de problèmes rencontre ces critères. C'est notamment le cas des problèmes de planification dans un contexte hiérarchique. Schneiwess présente plusieurs de ces problèmes tirés d'applications industrielles réelles [12].

Nous étudierons spécialement le cas des chaînes d'approvisionnement [10], plus spécialement l'exemple suivant dans lequel les agents représentent des compagnies autonomes offrant des produits et services aux autres compagnies. Un client externe lance un appel d'offres pour un produit et le travail de chaque compagnie est nécessaire pour fabriquer et livrer la commande. Différentes alternatives sont possibles, tant à ce qui a trait aux pièces à utiliser, au choix des processus de fabrication, à l'ordonnan-

cement des opérations et au transport. Les partenaires désirent donc produire un plan de production commun (quoi faire, où et quand le faire). Chaque agent n'a qu'une vision locale : il ne peut réfléchir qu'à sa propre production, à ses intrants et à ses extrants. De plus, les règles d'affaires imposent une certaine séquence de résolution. Une usine ne peut calculer ses besoins en matière première et les transmettre à son fournisseur si elle ne sait pas ce qu'elle doit produire. De plus, elle ne pourra pas planifier sa production et transmettre une promesse de livraison à son client sans savoir quel approvisionnement lui est accordé par son fournisseur. Ce sont là les deux sous-problèmes qu'un agent aura à résoudre : (1) calculer la demande à transmettre à son fournisseur et (2) planifier sa production.

Dans ce problème, la fonction objectif globale représente la satisfaction du client externe, i.e. une livraison de la commande finale avec le moins de retard possible. Les agents ne connaissant pas les options de production alternatives des agents qu'ils approvisionnent (ceux-ci sont des compétiteurs sur d'autres dossiers), on ne peut calculer une borne sur la fonction objectif avant d'en être au dernier sous-problème.

Le réseau d'entreprises est en compétition avec d'autres réseaux ; si le client externe n'accepte pas la première solution proposée, il est urgent de proposer au client des solutions alternatives avant qu'il n'accepte une meilleure proposition d'un autre consortium. Pour les cas industriels étudiés plus loin, il est utopique de penser résoudre de manière optimale le sous-problème d'un seul agent. Il en va de même pour le problème global. En pratique, on cherche donc à obtenir de bonnes solutions en un temps raisonnable.

Dans les sections suivantes, nous verrons que peu de méthodes existantes s'appliquent dans un tel contexte. Nous proposerons ensuite une nouvelle méthode (MacDS) permettant aux agents de réaliser une recherche systématique dans l'espace des solutions, mais visant l'obtention rapide de bonnes solutions grâce à l'utilisation d'une stratégie de retour-arrière basée sur le calcul des divergences (ex : LDS). La méthode proposée est complète et permet le travail concurrent des agents. L'algorithme sera ensuite évalué pour des problèmes générés aléatoirement de même que pour des problèmes industriels de taille réelle.

2 Définition formelle du problème

Un problème d'optimisation sous contraintes distribué et hiérarchique (HDCOP) s'exprime via un ensemble de variables $X = \{X_1, \ldots, X_m\}$. Chaque variable X_i peut prendre une valeur parmi un ensemble D_i. L'ensemble X est partitionné en plusieurs sous-ensembles disjoints définissant des sous-problèmes. Chaque variable appartient à un sous-problème $\mathbf{S}(X_i) \in S = [S_1, \ldots, S_n]$. Chacun appartient à un agent (un agent peut posséder plusieurs sous-problèmes). Formellement, nous avons $\mathbf{A}(\mathbf{S}(X_i)) = \mathbf{A}(X_i) \in A = \{A_1, \ldots, A_p\}$. Les sous-problèmes doivent être résolus dans la séquence spécifiée par S. Chaque sous problème S_j est contraint par les décisions précédentes, tel que spécifié par un prédicat $C_j(\cup(S_1, \ldots, S_j))$. Les agents cherchent à minimiser une fonction $\mathbf{F}(S_q)$, où $S_q \in S$.

On suppose que pour chaque sous-problème l'agent dispose d'un algorithme permettant de le résoudre. Il permet d'obtenir plusieurs solutions alternatives, dans un ordre défini implicitement par l'algorithme.

Dans ce contexte, on peut représenter l'espace de solutions à l'aide d'un arbre. Chaque niveau $j = 1, \ldots, n$ correspond au sous-problème S_j. Chaque noeud sur ce niveau représente une instance du sous-problème (définit par les décisions précédentes). Chaque arc représente une solution alternative au sous-problème qui peut être obtenue par l'algorithme local de l'agent responsable du sous-problème. Les arcs sont ordonnés selon l'ordre dans lequel ces solutions sont proposées par l'algorithme.

2.1 DCOP

Plus généralement, un problème distribué d'optimisation sous contraintes (DCOP) [8] est défini pour un ensemble de variables $X = \{X_1, \ldots, X_m\}$. Chaque variable X_i peut prendre une valeur parmi un ensemble D_i. On a également un ensemble de relations $C = \{\mathbf{C}_1, \ldots, \mathbf{C}_n\}$ où $\mathbf{C}_j$ est une fonction $\mathbf{C}_j(Y_1, \ldots, Y_k) : D_{j1} \times \ldots \times D_{jk} \rightarrow \mathbb{R}$ définie pour un sous-ensemble de variables $\{Y_1, \ldots, Y_k\} \subseteq X$. On désire assigner une valeur à chaque variable de manière à ce que la somme des $\mathbf{C}_j$ soit minimale. Chaque variable appartient à un agent $\mathbf{A}(X_i) \in A = \{A_1, \ldots, A_k\}$. Un agent peut affecter une valeur uniquement aux variables qu'il possède. Un agent peut connaître la valeur affectée à une variable d'un autre agent s'il est lié à cette variable par une contrainte. L'algorithme de résolution distribué utilisé spécifie quand et comment cette information est transmise. Certains auteurs/algorithmes n'admettent pour C_j que les relations binaires.

Un problème de HDCOP peut être reformulé à la manière d'un DCOP classique. Pour conserver la notion de sous-problèmes, on doit considérer que chaque sous-problème dans la formulation HDCOP originale est maintenant une variable dans le DCOP. Les variables originales cessent d'exister. Le domaine de chaque nouvelle variable est défini par le produit cartésien des variables originales du sous-problème. Les contraintes dures sont représentées par des relations retournant une valeur infinie lorsque la contrainte est violée. Malheureusement, cette formulation ne permet pas de représenter l'imposition par le contexte d'affaires d'une séquence de résolution. De même, on perd de vue le fait que chaque sous-problème est complexe et qu'un algorithme spécifique doit être utilisé pour le résoudre : à l'opposé, les algorithmes classiques pour les

DCOP spécifient les règles imposant à l'agent l'ordre dans lequel les valeurs de la variable (connues *a priori*) doivent être essayées. Pour ces raisons, nous nous en tiendrons à notre définition des HDCOP.

2.2 Algorithmes pour DCOP classiques

Dans cette section, nous verrons les principales méthodes existantes pour les DCOP et nous discuterons de leur applicabilité dans le contexte des HDCOP.

La méthode la plus simple se nomme Synchronous Backtracking (SyncBT) [15]. Elle vise à reproduire une recherche en profondeur dans un arbre (pour les HDCOP, il s'agirait de l'arbre présenté à la section 2). Les agents solutionnent leurs sous-problèmes à tour de rôle. Les messages transmis entre les agents contiennent l'ensemble des décisions prises jusqu'à ce moment. En cas d'impasse ou après l'obtention d'une solution, on applique un simple retour-arrière chronologique. La méthode est évidemment complète. L'algorithme Synchronous Branch-and-Bound (SyncBB) améliore SyncBT par le calcul d'une borne sur la fonction objectif. Elle sert à réaliser des coupes et à guider l'algorithme pour le choix des valeurs à affecter aux variables [4]. Pour un problème hiérarchique où l'objectif est représenté par une fonction des variables du dernier sous-problème, et lorsque les agents ne disposent pas d'une bonne représentation des autres sous-problèmes, la borne est égale à zéro pour les noeuds internes. SyncBB devient alors équivalent à SyncBT. Dans l'algorithme Asynchronous Forward-Bounding (AFB) [2], lorsqu'un agent assigne une valeur à une variable, il transmet les décisions courantes à tous les agents suivants. Ceux-ci calculent alors leur borne de manière concurrente.

Certaines méthodes permettent à l'agent de changer la valeur de sa variable de manière asynchrone, aussitôt qu'il détecte que ce changement de valeur risque d'améliorer la qualité de la solution globale. Les méthodes réalisant une recherche locale distribuée s'inscrivent dans cette voie, mais elles sont incomplètes [13][4]. L'algorithme Asynchronous Distributed Optimization (ADOPT) [8] a été le premier à la fois asynchrone et complet mais son utilisation suppose que chaque agent est en mesure de calculer une bonne borne et accepte de changer la valeur de sa variable sur cette base. Cela est incompatible avec notre contexte de base (voir section 1).

Une autre approche est la programmation dynamique distribuée. L'algorithme le plus connu est DPOP [11] pour lequel différentes améliorations ont été proposées au fil du temps. Il fait l'hypothèse que chaque agent $\mathbf{A}(S_j)$ peut résoudre S_j avant que S_{j-1} ne soit résolu. $\mathbf{A}(S_j)$ se voit demander de calculer la meilleure solution pour S_j pour toute solution potentielle de S_{j-1}. Cela suppose donc que $\mathbf{A}(S_j)$ dispose d'une très bonne représentation du domaine de S_{j-1} et qu'il peut résoudre S_j de manière optimale en un temps raisonnable.

3 Recherche basée sur les divergences dans un contexte multi-agent

L'espace des solutions dans le contexte introduit précédemment rappelle celui des problèmes d'optimisation combinatoires centralisés. L'arbre de recherche décrit (section 2) est équivalent à celui d'un problème centralisé pour lequel l'ordre des variables et l'ordre des valeurs seraient fixés. Or, en environnement centralisé, il a été montré à plusieurs reprises que le retour-arrière chronologique (i.e. ce que fait SyncBT) peut être surpassé par des méthodes basées sur le calcul des divergences [3, 6] qui ont la caractéristique de ne pas s'appuyer sur le calcul de bornes.

Cette section introduit un nouvel algorithme distribué exploitant un retour-arrière basé sur les divergences : Multi-Agent Concurrent Discrepancy Search (MacDS). Il permet l'exploration systématique de l'espace des solutions (le même que pour SyncBT), mais vise à obtenir de bonnes solutions en un temps court. Il permet le travail concurrent des agents. Les communications et le temps sont gérés de manière asynchrone [7]. Il est tolérant aux délais de transmission variables et aux pannes de réseau temporaires. Il prend de plus avantage des situations où les temps de calcul des différents sous-problèmes sont différents les uns les autres.

3.1 Retour-arrière basé sur les divergences

La première méthode centralisée basée sur les divergences (Limited Discrepancy Search, LDS) a été introduite dans [3]. L'idée principale est que les feuilles de l'arbre (solutions) n'ont pas toutes *a priori* la même qualité espérée ; que celle-ci décroît en fonction du nombre de fois qu'il faut brancher à droite pour aller de la racine jusqu'à cette feuille (i.e. le nombre total de déviations, ou divergences). LDS vise donc à visiter prioritairement les feuilles associées à un nombre de divergences faible. En faisant de la sorte, on cherche d'abord les solutions qui sont le plus en accord avec les heuristiques de choix de valeur utilisées pour chaque variable. Cette technique a également l'effet que les différentes solutions visitées en un temps donné seront plus différentes les unes des autres que dans le cas d'un retour-arrière chronologique. C'est cette caractéristique que nous rechercherons.

Dans la publication originale, LDS était décrite de manière procédurale mais l'idée peut également être exploitée de la manière suivante [1] : lorsque les conditions pour un retour-arrière sont rencontrées, l'engin de recherche doit activer le noeud déjà visité pour lequel le prochain fils non visité a le nombre total de divergences le plus faible. C'est pourquoi LDS peut-être considéré comme une politique de retour-arrière.

LDS a été appliquée à des problèmes d'optimisation notamment dans [6]. Ces derniers ont proposé de compter les divergences de la manière suivante pour un arbre n-aires : la i-ème branche empruntée à un niveau donné compte pour $i-1$ divergences. Différentes variantes de LDS ont été proposées, notamment dans [14] et [1]. Plusieurs de ces méthodes sont intégrées à des solveurs commerciaux, notamment dans ILOG Solver.

3.2 Approches pour la concurrence

Les différents algorithmes classiques pour les DCOP et les DisCSP (problèmes distribués de satisfaction de contraintes) permettent le travail concurrent des agents grâce à deux techniques différentes [16].

La première est l'approche asynchrone exploitée par ADOPT, telle que décrite à la section 2.2. Elle peut également être exploitée pour les DisCSP.

La seconde se nomme Concurrent Search Algorithms (CSA) [16]. Elle est exploitée pour les DisCSP par l'algorithme ConcDB [16]. Selon cette approche, chaque solution est construite de manière séquentielle par les agents. Par contre, le collectif travaille simultanément sur plusieurs solutions. Un algorithme reposant sur cette approche doit donc définir quand et comment de nouveaux "chemins" doivent être explorés. Du point de vue de la concurrence, la méthode que nous proposons s'inscrit dans cette voie.

3.3 Algorithme proposé (MacDS)

Nous présenterons d'abord l'algorithme de manière informelle à l'aide d'un exemple simple. Trois agents $A = \{B, C, D\}$ cherchent à résoudre un problème constitué des sous-problèmes $S = [B, C, D]$. A la manière des CSA, toute solution sera obtenue par la résolution séquentielle de ces sous-problèmes.

Le premier agent (B) utilise son algorithme de résolution local. Lorsqu'il obtient une première solution, l'agent B la transmet à l'agent C dans un message nommé "`B0`" (première solution pour le sous-problème B). L'agent C fait de même et transmet le message "`B0-C0`" (première solution pour C compte tenu de la première solution pour B) au prochain agent (D). Cependant, immédiatement après avoir fourni sa décision "`B0`" au deuxième agent, l'agent B cherche une décision alternative appelée "`B1`"). Dès qu'elle est prête, il la transmet à l'agent C, même si celui-ci n'a pas encore de solution pour "`B0`". Sur la réception de ce message, l'agent C doit se demander s'il est préférable de continuer à chercher une solution "`B0-C0`" ou bien commencer à travailler sur "`B1-C0`". Cette décision sera basée sur les divergences (les noms des messages contiennent toute l'information nécessaire pour les calculer).

Chaque agent gère donc une liste de tâches, alimentée de manière asynchrone par son prédécesseur. Les tâches de l'agent sont priorisées en fonction du profil de divergences du prochain message que cette tâche devrait générer. La politique utilisée par l'agent pour prioriser ses tâches est assimilable à une stratégie de retour-arrière. Elle est implémentée à l'aide d'une fonction permettant de comparer deux profils de divergences. Il est possible de définir quantité de fonctions correspondant à différentes stratégies de retour arrière connues : retour-arrière chronologique (DFS), LDS, DDS, SBS ou autre.

Dans le cas extrême où un seul agent gérerait tous les sous-problèmes (il y aurait une seule liste de tâches), MacDS visiterait alors les noeuds de l'arbre dans le même ordre que ce que ferait un algorithme de recherche centralisée appliquant la même stratégie de retour-arrière.

Dans notre contexte distribué où chaque agent gère sa propre liste de tâches (celles correspondant aux sous-problèmes qu'il a sous sa juridiction), toute solution au problème global qui serait obtenue par la version centralisée de l'algorithme est garantie d'être obtenue dans un temps égal ou inférieur par MacDS (en faisant abstraction des délais de communication) : chaque agent est au travail pour peu qu'il existe une tâche dans sa liste, mais aussitôt qu'une tâche plus prioritaire est ajoutée à sa liste, il s'attaque à cette dernière immédiatement.

L'algorithme est complet et sa terminaison est garantie puisqu'il explore le même espace des solutions que SyncBT ; on modifie uniquement la séquence dans laquelle les noeuds sont visités. En cas de panne de communication isolant deux sous-réseaux (ou bien en présence de délais de transmission aléatoires), les agents travaillent sur les tâches disponibles de plus grande priorité plutôt que d'être en attente. En conséquence, l'algorithme n'oblige pas à ce que l'ordre d'arrivée des messages soit le même que l'ordre d'expédition.

3.3.1 Pseudocode

Les objets suivants sont manipulés par l'algorithme :

- Un message (`msg`) est un couple `<d,p>` où `d` représente les décisions pour les sous-problèmes précédents et `p` est un vecteur d'entiers représentant le profil de divergences. Ce vecteur indique quel chemin devrait être suivi pour passer de la racine au noeud corresponsant à ce message dans l'arbre de recherche. L'élément `p[j]` indique, pour un niveau j, quel arc devrait alors être suivi.
- Une liste de tâches (`tasks`) contient les tâches en attente ou en cours d'exécution pour un agent donné. Une tâche est définie par `d` et `p`, par le nombre de solutions locales produites jusqu'à maintenant pour cette tâche (`i`) et par un booléen indiquant si l'algorithme local utilisé par l'agent sait qu'il n'y a plus d'autre solutions (`noMoreSol`).

Chaque agent possède plusieurs fils d'exécution (threads) : un pour chaque tâche dans la liste de même qu'un fil de contrôle. Pour chaque agent, un seul fil est actif simultanément. Le fil de contrôle (figure 1) est activé quand l'agent reçoit un message (`WhenReceiveMsg`) et lorsqu'une tâche de l'agent vient tout juste de produire une nouvelle solution pour un sous-problème (`WhenNewSolution`). Le fil de contrôle met alors à jour la liste de tâches et transfert le contrôle au fil correspondant à la tâche la plus prioritaire (`ActivateATask`).

```
WhenReceiveMsg(msg)
  if (running ≠ ∅) running.Sleep();
  tasks.insert(<msg.d, msg.p, 0, false>));
  ActivateATask();

WhenNewSolution(task)
  task.Sleep();
  if (task.noMoreSol)
    tasks.Remove(task);
    running ← ∅;
  else
    SendMessage(Successor(task), <task.d +
              task.sol.d, task.p + task.i>);
    task.i++;
  ActivateATask();

ActivateATask()
  if (task.count() > 0)
    running ← tasks[1];
    running.WakeUp();
  else
    running ← ∅;
```

FIG. 1 – Pseudocode pour le fil de contrôle

```
Run(task)
   task.noMoreSol ← false;
   task.sol ← NextSolution(task);
   while (task.sol ≠ ∅)
     SignalNewSolution(task);
     Sleep();
     task.sol ← NextSolution(task);
   task.noMoreSol ← true;
   SignalNewSolution(task);
```

FIG. 2 – Pseudocode pour le fil associé à une tâche

Le pseudocode associé aux fils des tâches est présenté à la figure 2. Lorsqu'une tâche est créée, son fil est inactif. Il devra être activé par le fil de contrôle. Lorsque la tâche produit une nouvelle solution, elle le signale au fil de contrôle (`SignalNewSolution`) et redevient inactive (`Sleep`). C'est le fil de contrôle qui se charge d'envoyer le message à

```
CompareBT(p1, p2)
   depth ← Min(Card(p1), Card(p2));
   j ← 1;
   while (p1[j] = p2[j] && j ≤ depth) j++;
   if (j ≤ depth)
     if (p1[j] ≤ p2[j]) return p1;
     else return p2;
   else
     if (Card(p1)≥Card(p2)) return p1;
     else return p2;

CompareLDS(p1, p2)
   t1 ← Σ(j=1..Card(p1)) p1[j];
   t2 ← Σ(j=1..Card(p2)) p2[j];
   if (t1 < t2) return p1;
   else
     if (t2 < t1) return p2;
     else return CompareBT(p1, p2);
```

FIG. 3 – Fonctions de comparaison pour deux politiques de retour-arrière différentes

l'agent responsable du prochain sous-problème (le dernier agent envoie plutôt la solution au client).

Dans le pseudocode présenté, nous supposons que les tâches de la liste sont triées en ordre décroissant de priorité. La figure 3 montre deux exemples de fonctions pouvant être utilisées pour maintenir ce tri. Chacune permet d'identifier la tâche la plus prioritaire parmi une paire. La première (`CompareBT`) le fait selon une politique de retour-arrière chronologique (DFS). La seconde (`CompareLDS`) définie une politique de type LDS. Au moment d'appeler la fonction de comparaison, le profil de divergences représentant une tâche est obtenu en concaténant `task.p` et `task.i`.

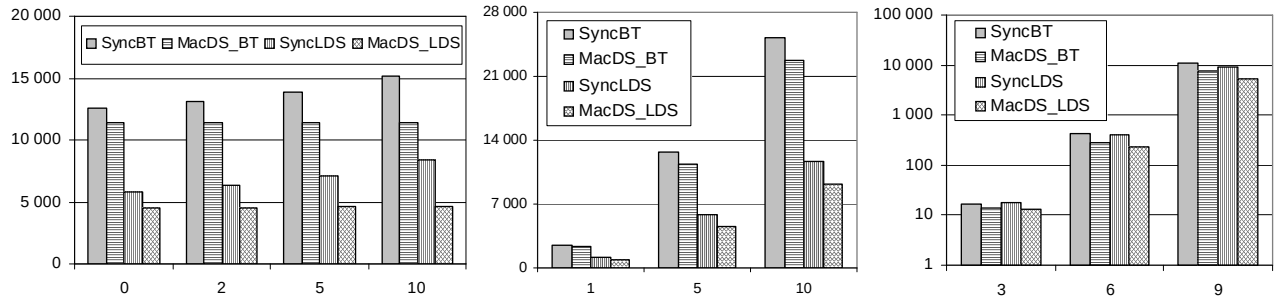
4 Evaluation

MacDS a été évalué avec des données aléatoires de même que pour des problèmes industriels.

4.1 Evaluation avec des données aléatoires

Nous supposerons des arbres n-aires tels que la probabilité qu'une feuille constitue la meilleure solution est proportionnelle à $\delta^{(p)}$, où p est le nombre total de divergences associé à la feuille et δ est un paramètre qui varie de 0.1 à 1.0 (par intervalle de 0.1). Lorsque δ est maximal, toutes les feuilles ont exactement la même probabilité associée.

Les autres paramètres que nous ferons varier sont les suivants : le nombre de sous-problèmes à résoudre ($\mathbf{Card}(S)$), le temps de transmission des messages (τ), le temps de résolution d'un sous-problème (α) et le nombre de solutions par sous-problème (n).



(a) (b) (c)

FIG. 4 – Temps espéré pour obtenir la meilleure solution, en fonction de : (a) délai de transmission des messages [$\mathbf{Card}(S) = 4; n = 10; \alpha = 1; \delta = 0.7$], (b) temps pour résoudre un sous-problème [$\mathbf{Card}(S) = 4; n = 10; \tau = 0; \delta = 0.7$] et (c) nombre de sous-problèmes [$n = 3; \alpha = 1; \tau = 0; \delta = 0.7$]

Nous mesurerons la performance d'un algorithme comme étant l'espérance du temps requis pour trouver la meilleure solution dans cet arbre. Les expérimentations ont été réalisées dans un environnement simulé semblable à celui décrit par [8].

Nous avons comparé MacDS configuré avec une politique de type LDS (MacDS_LDS) avec SyncBT. Pour mesurer quelle part du gain est due à la concurrence et quelle part est due à la politique de retour-arrière, nous avons aussi testé une version de MacDS utilisant le retour-arrière chronologique (MacDS_BT) et créé SyncLDS (synchrone comme SyncBT, mais appliquant LDS).

MacDS_LDS est toujours le meilleur des quatre algorithmes (rattrapé par MacDS_BT lorsque $\delta = 1.0$). Pour les deux politiques de retour-arrière testées (retour-arrière chronologique et LDS), la version concurrente (MacDS) est toujours meilleure que la version synchrone. Les figures suivantes montrent les résultats obtenus pour $\delta = 0.7$, lequel est représentatif du cas moyen. La figure 4(a) montre que le délai de transmission des messages (τ) a un impact linéaire pour les quatre algorithmes, mais qu'il est beaucoup plus petit pour MacDS. Pour MacDS, le temps espéré croît selon $(\mathbf{Card}(S) - 1)\tau$. La figure 4(b) montre que le temps de résolution des sous-problèmes (α) a également un impact linéaire, encore une fois plus petit pour les versions MacDS. La figure 4(c) montre l'impact exponentiel du nombre de sous-problèmes ($\mathbf{Card}(S)$).

4.2 Evaluation dans un contexte industriel

Nous avons ensuite évalué les algorithmes avec des données réelles, pour un cas industriel constitué de plusieurs sous-problèmes complexes. Il s'agit d'une chaîne d'approvisionnement dans l'industrie des produits forestiers. Les données ont été extraites des systèmes informatiques des entreprises à différents moments durant l'année 2005.

Le réseau est formé de trois usines $A = \{A_1, A_2, A_3\}$ et comporte quatre sous-problèmes $S = [V, W, X, Y]$. Le problème centralisé équivalent aurait des millions de solutions. En termes de flux de produits, les trois agents forment une chaîne. Cependant, d'un point de vue décisionnel les flux sont plus complexes (voir figure 5). L'agent A_2 est en contact avec le client externe. Une commande-client est constituée d'un ensemble de tuples <produit, quantité, dateDemandée>. L'agent A_2 calcule d'abord sa demande en matière première qui sera transmise à A_1 (sous-problème V). L'agent A_1 planifie ensuite sa production et transmet à A_2 sa promesse de livraison (sous-problème W). L'agent A_2 planifie alors sa production et transmet à A_3 sa promesse de livraison (sous-problème X). Enfin, l'agent A_3 planifie sa production (sous-problème Y). L'objectif consiste à minimiser la somme des retards (en unité de temps) pour les produits commandés par le client externe, lequel s'exprime par une fonction des variables dans Y.

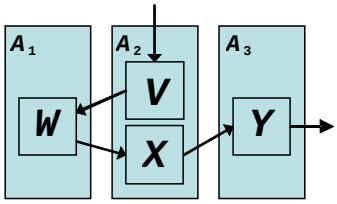


FIG. 5 – Agents, sous-problèmes et séquence de résolution

Chacun des solveurs locaux a été développé antérieurement dans le cadre de l'initiative de recherche FORAC regroupant des chercheurs et des industriels de l'industrie canadienne des produits forestiers. L'agent A_1 planifie ses opérations de sciage de billots à l'aide d'un modèle mathématique linéaire en nombres entiers. L'agent A_2 planifie et ordonnance ses opérations de séchage de bois à l'aide d'un engin de planification utilisant la programmation par contraintes. Il utilise à l'interne une stratégie de retour-arrière de type DDS pour produire des solutions locales alternatives. L'agent A_3 utilise une heuristique de type "pla-

cement au plus tôt" pour ordonnancer ses opérations de rabotage. Il utilise également DDS pour produire des solutions locales alternatives.

Ces expérimentations ont été réalisées dans un environnement réellement distribué, chaque agent fonctionnant sur un ordinateur distinct. Cette approche a permis de mesurer l'impact réel de la concurrence dans le contexte où les sous-problèmes sont difficiles à résoudre et où les temps de résolutions de chaque agent sont très différents. Le temps de calcul pour l'obtention d'une première solution à un sous-problème est très variable d'un type de sous-problème à l'autre (du simple au quintuple). Le temps de production de solutions alternatives est encore plus variable, allant de quelques secondes à plusieurs minutes, pour un même sous-problème. Dans ce contexte, le nombre de messages échangés est relativement petit pour tous les algorithmes évalués et cette métrique est peu pertinente [7].

Pour tous les algorithmes, la première solution obtenue est évidemment toujours la même. C'est également celle qui est serait obtenue en industrie en appliquant les règles d'affaires de manière standard. Conséquemment, nous avons comparé les algorithmes sur la base du gain supplémentaire qu'ils permettent d'obtenir (pourcentage de réduction de la fonction objectif) en fonction du temps de calcul supplémentaire alloué (jusqu'à une heure). La figure 6 présente les résultats pour les quatre cas industriels étudiés (a,b,c,d).

SyncBT et MacDS_LDS donnent des résultats comparables dans les premières secondes. Cela s'explique par le fait qu'ils produisent les mêmes solutions tant que le dernier agent ne reçoit pas une deuxième tâche dans sa liste. Par la suite, MacDS_LDS prend généralement le dessus de manière importante : alors que SyncBT persiste à n'explorer que des variations mineures des premières solutions, MacDS permet l'exploration de zones différentes de l'arbre de recherche. L'exception est le cas (b) pour lequel SyncBT arrive gagnant avec un écart de 0.5% pour presque n'importe quel temps de calcul donné.

On peut voir l'impact de la stratégie de retour-arrière en comparant SyncBT et SyncLDS. Ce dernier surpasse SyncBT, excepté pour des temps de calculs très courts.

En comparant MacDS_LDS avec SyncLDS, on peut voir l'impact du travail concurrent des agents. Pour toute solution atteinte par SyncLDS, MacDS_LDS produit une solution de qualité égale ou supérieure en un temps de calcul égal ou inférieur. La réduction moyenne du temps de calcul pour chacun des cas sont les suivantes : 18.5%, 88.7%, 54.5% et 64.0%. Deux facteurs expliquent ces résultats. Le travail concurrent des agents fait que chaque feuille est visitée en un temps égal ou inférieur à ce qui est nécessaire pour le cas synchrone, mais donne également la possibilité d'explorer plus de solutions pour un même temps de calcul.

MacDS_BT a donné exactement les mêmes résultats que SyncBT. Pour cette raison, il n'apparaît pas sur les figures.

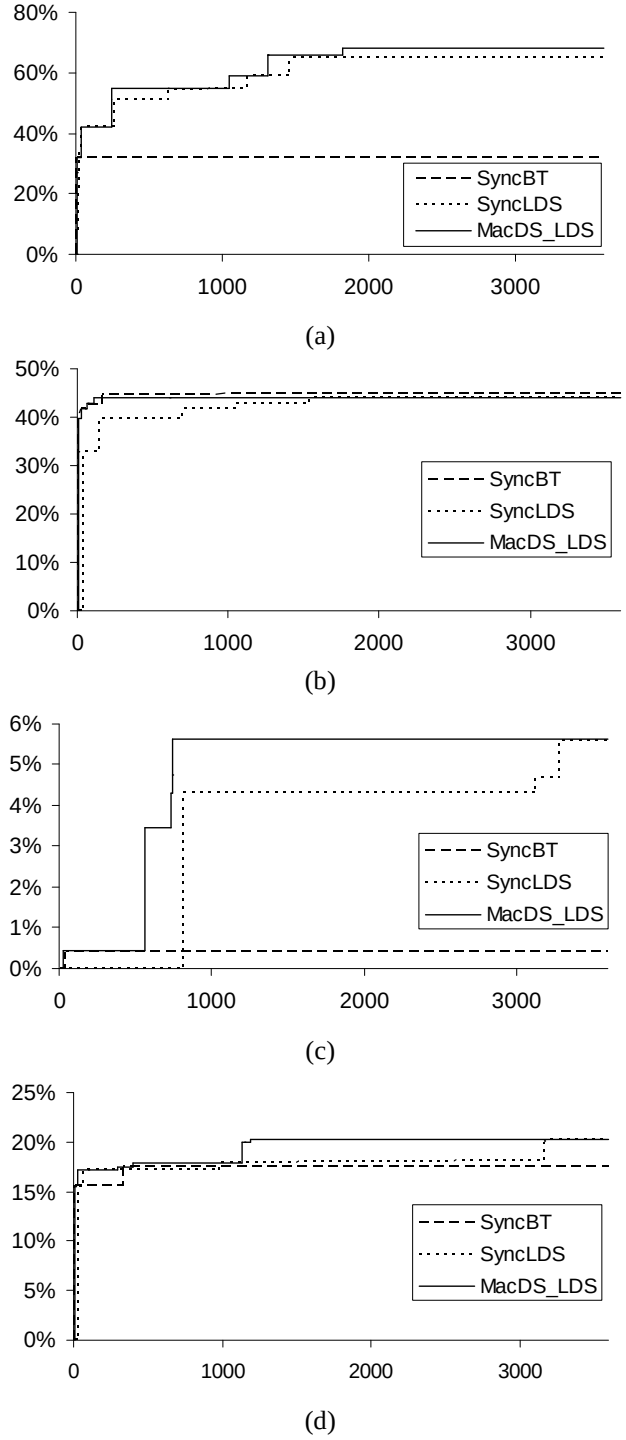


FIG. 6 – Réduction de la fonction objectif en fonction du temps de calcul accordé (en secondes) pour les cas (a), (b), (c) et (d)

L'explication est la suivante. Une recherche en profondeur implique de se concentrer uniquement sur le dernier sous-problème tant que ses solutions ne sont pas toutes énumérées. Pour nos problèmes industriels, cela demande un temps considérable. Dans MacDS_BT, les agents précédents produisent alors des solutions alternatives mais elles n'en viennent jamais à être exploitées par le dernier agent dans le temps de calcul accordé dans l'expérimentation.

5 Conclusion

Les algorithmes actuels pour les DCOP qui surpassent SyncBT s'appuient sur le calcul d'une borne par chaque agent. Ils supposent que l'agent saura calculer cette borne et prendre sa décision locale sur la base de celle-ci. Dans un contexte hétérogène où des agents spécialisés font face à des problèmes complexes, ce n'est pas toujours possible. Nous avons donc proposé l'algorithme MacDS. La méthode surpasse SyncBT grâce à deux principes : (1) l'application d'une stratégie de retour-arrière basée sur les divergences et (2) le travail concurrent des agents. La méthode est tolérante aux failles de communications, elle permet l'obtention de bonnes solutions en un temps très court mais elle est complète. Elle a été évaluée dans un contexte industriel pour des agents hétérogènes résolvant des sous-problèmes locaux complexes, MacDS surpassant généralement SyncBT de manière considérable et permettant une réduction significative des temps de calcul par rapport à un algorithme synchrone non-concurrent appliquant la même stratégie de retour-arrière. L'expérimentation a de plus montré l'impact économique considérable que ces méthodes peuvent avoir en pratique, dans un contexte où toute réduction des retards d'un point de pourcentage est un gain énorme.

Dans l'avenir, il serait intéressant de voir si le retour-arrière basé sur les divergences peut-être bénéfique même dans un contexte où le calcul des bornes est possible, le couplage de ces approches donnant de bons résultats en environnement centralisé [1].

Références

[1] J. C. Beck and L. Perron. Discrepancy-bounded depth first search. In *Workshop on Integration of AI and OR Technologies for Combinatorial Optimization Problems*, pages 7–17, Paderborn, Germany, 2000.

[2] A. Gershman, A. Meisels, and R. Zivan. Asynchronous forward-bounding for distributed constraints optimization. In *European Conference on Artificial Intelligence*, pages 137–139, Amsterdam, 2006. IOS Press.

[3] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *International Joint Conference on Artificial Intelligence*, pages 607–613, Montreal, Can, 1995. Morgan Kaufmann.

[4] K. Hirayama and M. Yokoo. Distributed partial constraint satisfaction problem. In *International Conference on Principles and Practice of Constraint Programming, LNCS #1330*, pages 222–236, Linz, Austria, 1997. Springer.

[5] B. Horling and V. Lesser. A survey of multi-agent organizational paradigms. *Knowledge Engineering Review*, 19(4) :281–316, 2004.

[6] C. Le Pape and P. Baptiste. Heuristic control of a constraint-based algorithm for the preemptive job-shop scheduling problem. *Journal of Heuristics*, 5(3) :305–325, 1999.

[7] N. A. Lynch. *Distributed algorithms*. Morgan Kaufmann, San Francisco, Calif., 1996.

[8] P. J. Modi, W. M. Shen, M. Tambe, and M. Yokoo. Adopt : asynchronous distributed constraint optimization with quality guarantees. *Artificial Intelligence*, 161(1-2) :149–180, 2005.

[9] P. J. Modi and M. Veloso. Bumping strategies for the multiagent agreement problem. In *Proceedings of the International Conference on Autonomous Agents*, pages 527–533, New York, 2005. ACM Press.

[10] T. Moyaux, B. Chaib-draa, S. D'Amours, B. Chaib-draa, and J.P. Muller. Supply chain management and multiagent systems : An overview. In *Multiagent-Based Supply Chain Management*, page 450. Springer, New York, 2006.

[11] A. Petcu and B. Faltings. Dpop : A scalable method for multiagent constraint optimization. In *International Joint Conference on Artificial Intelligence*, Edinburg, Scotland, 2005.

[12] C. Schneeweiss. *Distributed Decision Making*. Springer, New York, 2003.

[13] J. Sun, Y. F. Zhang, and A. Y. C. Nee. A distributed multi-agent environment for product design and manufacturing planning. *International Journal of Production Research*, 39(4) :625–645, 2001.

[14] T. Walsh. Depth-bounded discrepancy search. In *International Joint Conference on Artificial Intelligence*, pages 1388–1393, Nagoya, Japan, 1997. Morgan Kaufmann.

[15] M. Yokoo, E. H. Durfee, T. Ishida, and K. Kuwabara. Distributed constraint satisfaction for formalizing cooperative distributed problem solving and its algorithms. *Transactions of the Institute of Electronics, Information and Communication Engineers*, J75D-I(8) :704–13, 1992.

[16] R. Zivan and A. Meisels. Concurrent search for distributed csps. *Artificial Intelligence*, 170(4-5) :440–61, 2006.