Algorithmes de Recherche du Gagnant dans une Négociation Combinée

Graph-Based Search Algorithms for Optimal Winner Determination

Samir Aknine

LIP6, Université Paris 6
8, rue du Capitaine Scott
75015 PARIS Cedex 15
France
Samir.Aknine@lip6.fr

Résumé

Dans cet article, nous présentons deux algorithmes de recherche du gagnant dans une négociation multi-agent combinée. Ces algorithmes sont essentiellement fondés sur l'exploration de graphes très simples pour les items partagés et non-partagés permettant d'accélérer la recherche des solutions optimales. Cet article commence d'abord par illustrer ces algorithmes à travers un exemple pour ensuite rapporter les résultats de leur évaluation expérimentale sur l'application d'organisation de voyages que nous avons implémentée.

Mots Clef

Enchères, algorithmes, négociation.

Abstract

In this paper, we present two extended algorithms for the optimal winner determination problem in combinatorial auction. These algorithms are essentially based on new graphs of shared and unshared items. These graphs proved to be able to speed up the search of optimal solutions. In the first part of this paper, we introduce the new algorithms through an example. Then, we show the results of their experimental evaluation.

Keywords

Auctions, algorithms, negotiation.

1 Introduction

Cet article présente deux algorithmes de recherche du gagnant fondés sur l'utilisation de graphes. Pour illustrer ces deux algorithmes nous avons choisi l'application d'organisation de voyages électronique. Cette application présente, en effet, plusieurs problèmes notamment celui de la négociation combinée [1] et de la recherche des gagnants. Habituellement, un voyageur envoie une requête à une agence de voyage. La requête contient ainsi certains détails tels que le nombre d'arrêts, les dates préférées, le budget, le type et la qualité des services attendus. Le système doit alors fournir un package contenant les services demandés. Par exemple, une requête "Paris-Londres-NewYork à partir du 1/6/2003 avec deux nuits dans un hôtel à Londres, et une réservation de voiture pour une semaine à NewYork, le tout pour un budget de $1500" doit retourner un ensemble de produits, i.e. des billets de train ou d'avion, une réservation d'hôtel et un contrat de location de voiture.

Cette tâche nécessite la recherche des meilleures offres parmi celles retournées par les fournisseurs de services. Ce problème est NP-Complet [7][12], différents travaux ont essayé de trouver un algorithme qui puisse le résoudre en un temps polynomial.

La méthode que nous proposons pour résoudre ce problème se base sur deux étapes : la première consiste à construire des graphes très simples à partir des offres ; la deuxième est une construction de l'arbre des solutions optimales en se servant de ces graphes et des propriétés qu'on définit.

Nos algorithmes s'appuient essentiellement sur (1) de nouveaux graphes pour les items partagés et non-partagés, (2) de nouvelles fonctions d'ordonnancement des offres et (3) la propriété de singularité des items. Ces algorithmes utilisent aussi plusieurs heuristiques pour améliorer leurs performances et sont également basés sur les travaux de [6][9][10]. Nous commençons par illustrer ces algorithmes à travers un exemple et nous donnons ensuite les résultats de leur implémentation sur l'application d'organisation de voyages. Ces algorithmes ont été testés plus de six cent fois. Les résultats obtenus montrent que ces algorithmes améliorent effectivement les performances des meilleurs algorithmes actuels.

Cet article est organisé comme suit. La section 2 présente ces nouveaux algorithmes pour la recherche des solutions optimales et les concepts de graphes d'items partagés et non-partagés. La section 3 discute les expérimentations effectuées en utilisant l'application d'organisation de voyages électronique. Cette section compare ces algorithmes aux travaux actuels. La section 4 conclut ce travail et présente quelques perspectives.

2 Algorithmes de recherche du gagnant

Dans cette application, les items correspondent aux différents composants d'un voyage : déplacements par train ou par avion entre deux arrêts, réservations d'hôtel à chaque arrêt et location de voiture. Certains des items peuvent être utilisés pour définir les contraintes à respecter : le prix, puisque le budget doit être respecté ; les dates, puisque les intervalles d'arrêt et de déplacements ne doivent pas se chevaucher. Premièrement, les relations de dépendance entre les items sont identifiées en se basant sur les requêtes envoyées par les utilisateurs, ensuite elles sont formalisées. Il existe actuellement plusieurs langages de représentation [4][3]. Parmi ces requêtes, on retrouve la forme conjonctive mono-unité pour laquelle l'utilisateur sollicite tous les services dans la requête mais avec une seule unité de chacun. Nous allons donc décrire les algorithmes pour ce genre de requêtes. Les algorithmes proposés étendent ceux développés par [12][10][5] en reprenant certains de leurs pré-traitements et heuristiques. Nous n'allons pas décrire en détails ces algorithmes mais surtout présenter les améliorations apportées.

2.1 Algorithme -I-

Dans cette section, nous présentons notre premier algorithme avec quelques pré-traitements et heuristiques pour améliorer le temps de recherche de la solution optimale. Cet algorithme construit un arbre sur la base des principes de l'algorithme IDA* [6]. Pour faciliter la construction de l'arbre, l'algorithme traite en premier la question de la satisfiabilité de la requête. Une fois la satisfiabilité prouvée, l'algorithme continue de construire l'arbre des solutions optimales. Bien entendu, la recherche de la solution optimale n'est effectuée que si chacun des items de la requête conjonctive figure au moins dans une des offres envoyées par les vendeurs ou les fournisseurs de services. En nous basant sur un exemple, nous présentons les différentes étapes de notre algorithme.

Exemple 1. Un agent acheteur souhaite acquérir sept items {1.2.3.4.5.6.7}. La requête formulée pour les vendeurs est sous la forme conjonctive. Supposons qu'en retour, il reçoit les offres suivantes : *{1, $2}, {2, $1}, {4, $3}, {5, $2}, {6, $2}, {1.2, $4}, {1.5, $3}, {2.5, $3}, {2.4, $3}, {3.7, $4}, {4.5, $3}, {1.4.5, $5}, {2.3.4, $6}, {2.4.5, $5}, {2.4.5.7, $7}, {1.3.4.5, $7}, {1.2.4.7, $6}, {1.2.3.4.5, $9}.*

Par exemple, *{1.2.3.4.5, $9}* signifie que le prix des cinq items ensemble <*1.2.3.4.5*> est de *$9*.

Dans la première étape de cet algorithme, nous commençons par calculer le score associé à chaque item g, en utilisant la fonction f qu'on propose: $f(g) = \sum_{i=1}^{n} \frac{1}{T_i}$, tel que n est le nombre d'offres contenant l'item g et T_i est la longueur de l'offre i contenant g[1].

Cette fonction permet de répartir les différents items et les offres qui leurs correspondent en plusieurs partitions[2]. L'intérêt de cette fonction est qu'elle estime le temps d'exploration des offres de la partition à laquelle l'item appartient. Cette fonction montre que le temps nécessaire à l'exploration d'une partition est proportionnel au nombre d'offres que cette partition contient et que le temps nécessaire à l'exploration d'une offre est inversement proportionnel à sa taille, c'est à dire au nombre d'items qu'elle possède. Moins il y a d'offres dans une partition, moins il faudra de temps pour l'explorer. De la même façon, plus les offres sont de taille élevée, moins l'arbre sera profond. De ce fait, le temps nécessaire pour explorer cette offre sera plus court.

L'application de cette fonction à l'exemple 1 donne le vecteur suivant <3.033, 3.86, 1.28, 3.95, 3.86, 1,1> respectivement pour les items <1.2.3.4.5.6.7>.

Dans la seconde étape de l'algorithme, on commence par la création des différentes partitions (cf. Figure 1). Dans la première partition, nous isolons les offres qui possèdent l'item de plus faible score. Sur cet exemple, l'item 6 a un score de 1, c'est également le même score que pour l'item 7. Pour choisir l'un des deux, nous adoptons une autre propriété sur la singularité des items. On dit qu'un item est un singleton s'il n'existe aucune offre qui porte uniquement sur cet item. L'item 1 n'est pas un singleton puisque nous avons l'offre *{1, $2}*. L'item 7 en est un.

Dans la première partition p_1, nous regroupons donc les offres contenant l'item singleton (7). Notre algorithme favorise les items singletons car ils sont plus difficiles à combiner avec les autres dans les requêtes conjonctives puisqu'ils ne possèdent pas d'offres singletons qui pourraient être ajoutées facilement à n'importe quelle autre combinaison d'offres. En conséquence, si l'algorithme ne parvient pas à former une combinaison d'items avec cet item singleton, la requête conjonctive ne peut être satisfaite. Il n'est plus alors utile de continuer la recherche sur les autres items. La première partition est ainsi formée d'offres contenant l'item 7.

La seconde partition p_2 regroupe les offres ayant l'item de score minimal et ainsi de suite. Les offres S_i de chaque partition sont triées dans un ordre croissant par rapport au prix et à la taille ($prix(S_i)/|S_i|$)[3]. Cette heuristique permet

[1] Rappelons que l'algorithme de [5] utilise également une autre fonction discutées dans la section sur l'évaluation expérimentale.

[2] Une partition est un ensemble d'offres.

[3] Cette heuristique est aussi utilisée dans [5].

d'orienter la recherche des solutions vers les offres les plus prometteuses.

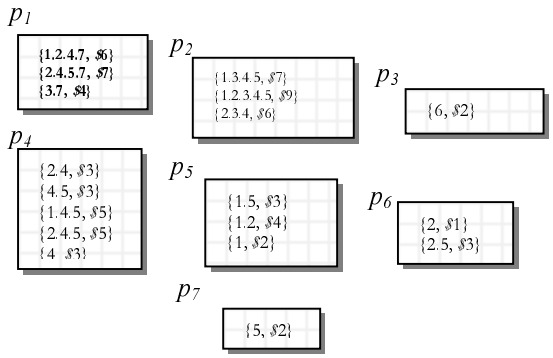

Figure 1 : Partitions d'offres.

L'étape suivante consiste à construire deux graphes. Le premier graphe est pour les items partagés par les offres et le second pour les items non-partagés. Dans le graphe des items partagés, présenté dans la Figure 2, chaque partition des offres est connectée à un nœud contenant les items communs à toutes les offres de cette partition.

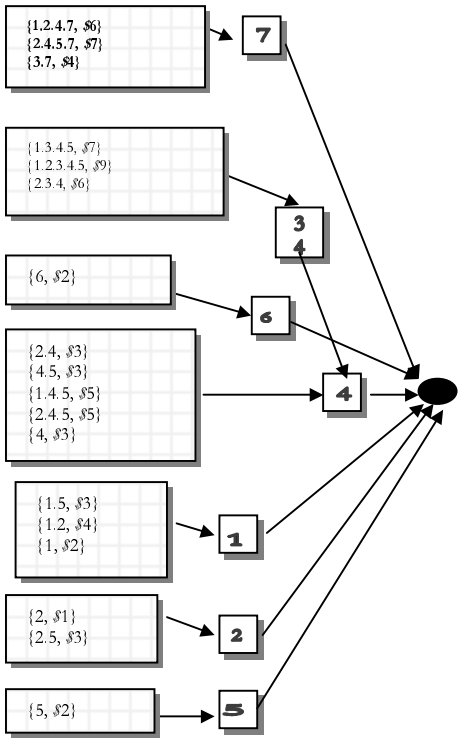

Figure 2: Graphe des items partagés de l'exemple 1.

Par exemple, la première partition est connectée au nœud avec l'item 7 qui est commun aux trois offres. La deuxième partition est connectée au nœud qui a pour items 3 et 4 puisque ces items sont tous les deux communs aux offres de cette partition. La 4^{ième} partition est connectée au nœud dont l'item est 4. Le nœud dont les items sont 3 et 4 est également connecté au nœud dont l'item est 4 puisque cet item est un sous-ensemble de l'ensemble formé par 3 et 4 (par principe d'inclusion).

Dans le graphe des items non-partagés, décrit dans la Figure 3, chaque partition d'offres est connectée à un nœud contenant les items que les offres de cette partition n'ont pas toutes en commun. Par exemple, la deuxième partition est associée au nœud dont les items sont 1, 2 et 5, et qui apparaissent au moins dans une des offres de cette partition. Le même principe d'inclusion des nœuds, décrit précédemment, est également appliqué dans ce graphe.

Ces deux graphes sont efficaces pour accélérer la recherche des solutions car chacun ne contient au plus que *m(m-1)/2* nœuds (avec *m* étant le nombre d'items dans la requête). Comme il existe moins d'items que d'offres, les graphes d'items améliorent la phase de recherche de la solution sans perte de temps dans leur construction.

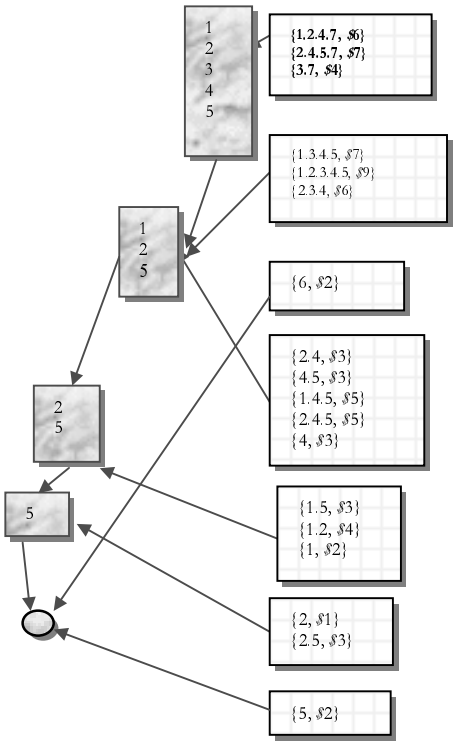

Figure 3: Graphe des items non-partagés de l'exemple 1.

Chacun des deux graphes fournit les conditions nécessaires et suffisantes pour l'exploration d'une partition sur la base des propriétés 1, 2 et 3 énoncées ci-dessous. En d'autres termes, nous ne travaillons plus directement sur les offres mais plutôt sur les nœuds des graphes qui les représentent, ce qui réduit ainsi le nombre

d'opérations. Une offre d'une partition n'est examinée que si toutes ces propriétés sont satisfaites.
Dès qu'une partition est explorée et que l'algorithme retient une offre de cette partition dans une branche de l'arbre des solutions optimales, il marque cette partition avec la référence de la branche pour interdire une seconde recherche dans cette partition.

Propriété 1 *Si une partition p_i partage un nœud avec une autre partition p_j, marquée dans le graphe des items partagés, la partition p_i ne sera pas explorée par l'algorithme. De même si l'un de ses nœuds ancêtres dans le graphe des items partagés possède un item dans la branche alors elle ne sera pas explorée en plus.*

En reprenant l'exemple 1 (cf. Figure 2), la propriété 1 interdit l'exploration de la partition 4. Par exemple, si une des branches de l'arbre des solutions en cours de construction possède une offre de la $2^{ième}$ partition. En effet, toutes les offres de la $2^{ième}$ partition ont en commun les items 3 et 4 qui ont pour ancêtre le nœud avec l'item 4 qui est un ancêtre de la $4^{ième}$ partition.

L'algorithme évite ainsi de rechercher dans cette partition des offres qui ne pourraient pas être utilisées dans la solution.

Une fois la propriété 1 satisfaite, il faut vérifier la seconde propriété.

Propriété 2 *Les offres d'une partition p_i peuvent être examinées uniquement si p_i est connectée à un nœud du graphe des items non-partagés (Figure 3) dont les items ne sont pas déjà inclus dans l'ensemble des items de la branche en cours de construction.*

Propriété 3 *Pour poursuivre la construction d'une branche solution, l'ensemble des items de la branche plus l'ensemble des items des partitions non encore explorées doit être égal au nombre d'items de la requête.*

Cette propriété évite la construction de branches qui ne contiendraient pas de solutions à la requête mais seulement une partie des items.
Grâce à l'utilisation de ces graphes d'items partagés et non-partagés, les propriétés ne seront pas vérifiées nécessairement sur chacune des offres mais uniquement sur des ensembles d'offres que représentent les partitions. Il en résulte que ces graphes limitent significativement la complexité des algorithmes.
Une fois ces graphes construits, la recherche de la solution optimale peut commencer. Dans la troisième étape de résolution, on construit les sous-arbres qui ont uniquement les offres de la première partition dans leurs racines (cf. Figure 4). Pour ce faire, l'algorithme teste à chaque fois les propriétés en se basant sur les graphes.
Chaque solution retenue dans un sous-arbre possède nécessairement un ensemble de nœuds contenant les items de la requête. Chaque nœud contient un sous-ensemble d'items et l'intersection entre les nœuds d'une branche doit nécessairement être vide.

La construction des branches de l'arbre des solutions s'effectue de la façon suivante. La première offre de la première partition, i.e. l'offre {1.2.4.7, *$*6*}* (cf. Figure 3) est initialement retenue comme premier nœud d'un sous-arbre. Un nœud est créé pour cette offre et on recherche ensuite les descendants possibles à ce nœud. Les descendants d'un nœud dans l'arbre des solutions ne peuvent pas contenir un item qui a déjà été sélectionné dans un nœud ancêtre. Les descendants d'un nœud sont les offres appartenant aux partitions suivantes, i.e., de p_2 *à* p_7. La recherche de ces descendants ne va pas s'effectuer directement sur les offres de ces partitions mais en se servant des graphes des items partagés et non-partagés.
Les offres de la partition p_2 ne seront pas examinées puisque dans le graphe des items partagés la partition p_2 est connectée à un nœud ayant les items 3 et 4. L'offre {1.2.4.7, *$*6*}* possède déjà l'item 4.
Les partitions p_3 et p_7 sont les seules qui vérifient la propriété 1. Par contre la recherche sur cette branche ne va pas être poursuivie puisque la propriété 3 n'est pas satisfaite, i.e., le nombre d'items avec ces partitions ne peut pas combler la requête. La branche est donc supprimée de l'arbre des solutions au tout début. Pour les mêmes raisons, le sous-arbre qui a pour racine l'offre suivante {2.4.5.7, *$*7*}* de la partition p_1 sera également supprimé.

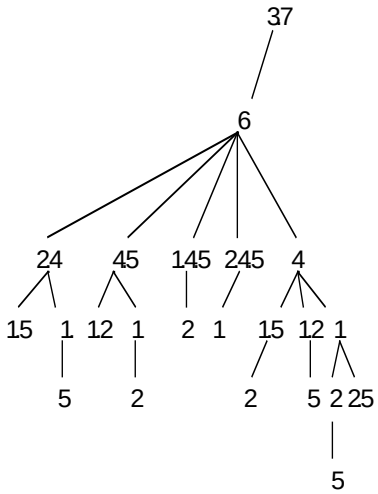

Figure 4: Résultats de l'algorithme -I- sur l'exemple 1.

Le dernier sous-arbre à construire est celui qui a pour racine l'offre {3.7, *$*4*}* toujours dans p_1. La partition p_2 ne peut pas être retenue pour construire ce sous-arbre puisque la propriété 1 l'interdit. La partition p_3 peut être acceptable selon les propriétés 1, 2 et 3. L'offre {6, *$*2*}* de cette partition est donc utilisée comme descendant direct du nœud (Figure 4). De la même façon l'algorithme

poursuit la recherche des descendants suivants dans la prochaine partition p_4 qui donne cinq branches possibles. La recherche est ainsi poursuivie jusqu'à ce que l'arbre soit complété avec toutes les solutions optimales.

2.2 Algorithme -II-

Cet algorithme est une extension de celui présenté par [9, 10] auquel nous avons rajouté le principe de recherche basée sur les graphes ainsi que d'autres heuristiques. En nous appuyant sur le même exemple précédent, nous allons décrire cet algorithme. Dans un premier temps les offres sont triées par rapport à leur taille. La taille correspond au nombre d'items dans une offre. Une partition est alors construite pour toutes les offres de même taille. Nous obtenons ainsi pour notre exemple, cinq partitions de taille allant de 5 à 1.
Dans chacune des partitions, les offres sont ordonnées par prix croissant. De la même façon que pour l'algorithme précédent, nous construisons les graphes d'items partagés et non-partagés. Nous les avons représentés sur la même figure pour simplifier (Figure 5).

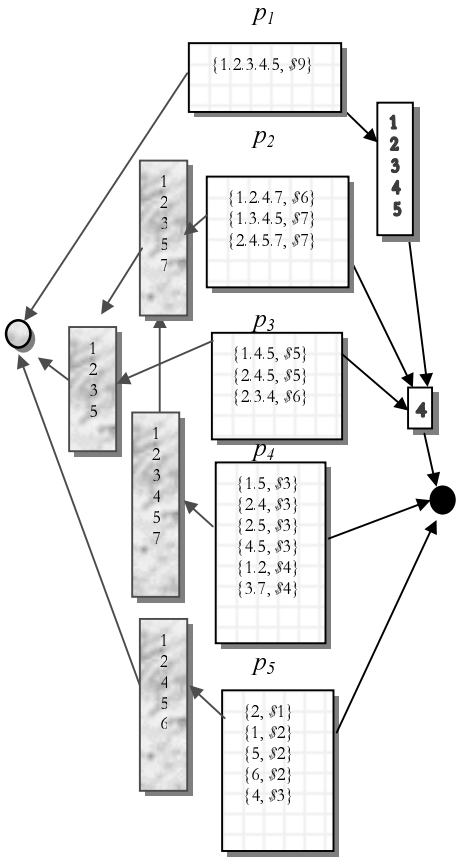

Figure 5: Graphes des items partagés et non-partagés de l'exemple 1.

Le graphe des items partagés est en partie droite de cette figure. Le graphe des items non-partagés est en partie gauche. Une recherche des solutions optimales est alors effectuée sur la base de ces graphes.
L'algorithme commence par construire les différents sous-arbres correspondants à chacune des offres de la première partition.
Cette partition ne comprend que l'offre O_1={1.2.3.4.5, $9} dans cet exemple. Le nœud racine du nouveau sous-arbre est alors construit avec cette offre. On recherche ensuite dans les partitions suivantes les descendants tels que les offres O_i de ces partitions soient de taille $|O_i| \leq min(|O_1|, m-|O_1|)$ (avec m étant la taille de la requête conjonctive).
On utilise le minimum puisque les offres sont déjà triées par rapport à leurs tailles. Bien entendu, une offre est gardée comme descendant d'un nœud uniquement si les propriétés 1 à 3 sont satisfaites (cf. section 2.1).
Pour l'offre O_1={1.2.3.4.5, $9}, l'algorithme ignore les partitions p_2 et p_3 car la propriété 1 n'est pas vérifiée.
De la partition p_4, l'algorithme ne retient pas d'offres puisque chacune de ces offres a en commun un item avec O_1. Les offres de la dernière partition p_5 sont également ignorées puisque la propriété 3 n'est pas satisfaite. En conséquence l'algorithme, supprime la branche de l'offre O_1.
La recherche est poursuivie avec les offres de la partition p_2. L'algorithme sélectionne la première offre O_2={1.2.4.7, $6} comme racine d'un nouveau sous-arbre. Pour cette offre, l'algorithme recherche les descendants O_j tels que $|O_j| \leq min(|O_2|, m-|O_2|)$. L'algorithme ignore également les partitions p_2, p_3 et p_4 en raison de la propriété 1 et 2. Quant aux offres de la partition p_5, elles ne sont pas non plus examinées car elles ne peuvent suffire à produire une solution (propriété 3). Les mêmes résultats sont obtenus sur la base des offres de la partition p_2.

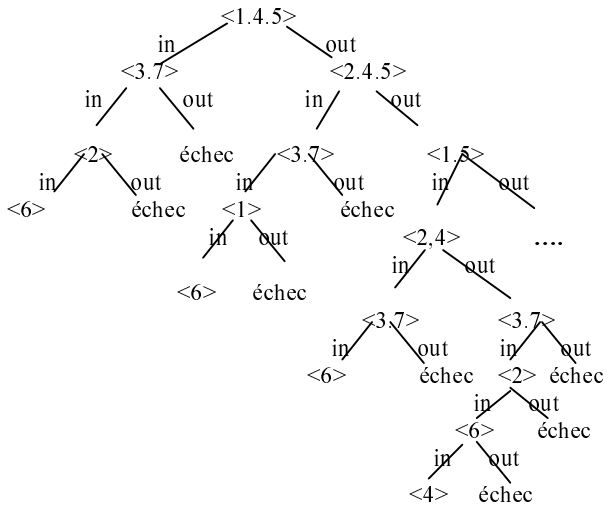

Figure 6 : Construction partielle de l'arbre des solutions.

Une fois que l'algorithme a atteint la partition p_3, il sélectionne l'offre O_5={1.4.5, $5} et en construit un nœud racine du nouveau sous-arbre. On commence alors à considérer toutes les partitions dont les offres O_j sont telles que $|O_j| \leq min(|O_5|, m-|O_5|)$.

De la partition 4, l'algorithme retient l'offre {3.7, $4} qui correspond au descendant immédiat de l'offre {1.4.5, $5} puisque cette offre satisfait l'ensemble des propriétés (cf. section 2.1). Sur la figure 6, on retient cela dans la première branche de l'offre <3.7>, que l'on étiquète avec *in, ce* qui signifie que cette offre est considérée sur cette branche. Les offres {2, $1} et {6, $2} sont aussi rajoutées à la branche pour compléter la solution. Lorsqu'une offre n'est pas retenue dans une branche, cela est indiqué par l'étiquette *out*.
Une solution dans la figure 6 est une succession de branches étiquetées par des *in*. Une branche qui se termine par une feuille *erreur* signifie qu'elle ne donne pas de solution au problème. Les autres solutions sont trouvées de la même façon jusqu'à avoir toutes les branches synthétisées dans la Figure 7.

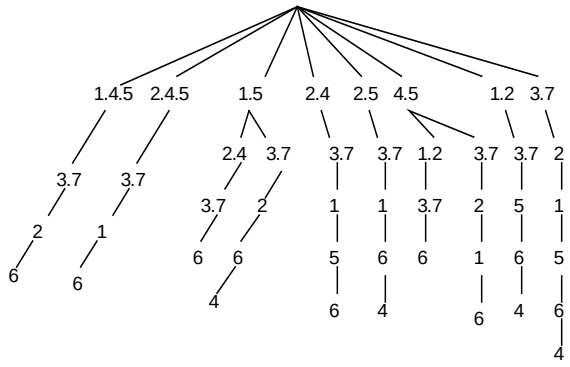

Figure 7 : Résultats de l'algorithme -II- sur l'exemple 1.

3 Evaluation expérimentale

Dans ces expérimentations, nous avons voulu observer la variation du temps nécessaire à nos algorithmes en fonction du nombre d'offres et d'items et aussi analyser l'évolution de la qualité de la solution, à travers le temps, pour un nombre fixé d'items et d'offres. Par ailleurs, nous avons également souhaité montrer qu'en utilisant les nouveaux principes proposés nous parvenons effectivement à améliorer la qualité des algorithmes actuels sur lesquels nous nous sommes également fondés.
Les tests présentés ci-après correspondent aux résultats obtenus avec une moyenne sur 10 exécutions à chaque fois. Ces résultats ont été effectués sur un Pentium III, 500Mhz, sous Windows 2000. Les algorithmes ont été implémentés en JAVA et exécutés plus de six cent fois sur plusieurs exemples. Pour comparer ces deux algorithmes aux algorithmes actuels, nous avons effectué plusieurs tests. Dans une première série, nous présentons les résultats obtenus en comparant notre algorithme -II- à celui de [9]. Nous comparons ensuite l'algorithme -I- à ceux de [8][5]. Nous les avons répartis en deux catégories en fonction des rapprochements qui existent entre eux. Dans la dernière série expérimentale, nous présentons une comparaison synthétique entre les deux algorithmes proposés et les algorithmes de Sandholm *et al.*

3.1 Première série expérimentale

Pour construire les offres, nous générons le nombre d'items *n* aléatoirement. Nous générons ensuite les *n* items d'une offre ainsi que le prix de l'offre sur l'intervalle [0, 1]. Nous avons observé l'évolution du temps d'exécution de notre algorithme -II- en comparaison à celui de [9] sur un ensemble d'items démarrant à 200 et des offres variant entre 100 et 15000. Les tests montrent que les différences entre ces deux algorithmes augmentent avec le nombre d'offres.

La Figure 8 montre les résultats de la comparaison de ces deux algorithmes lorsque le nombre d'items est relevé à 300.

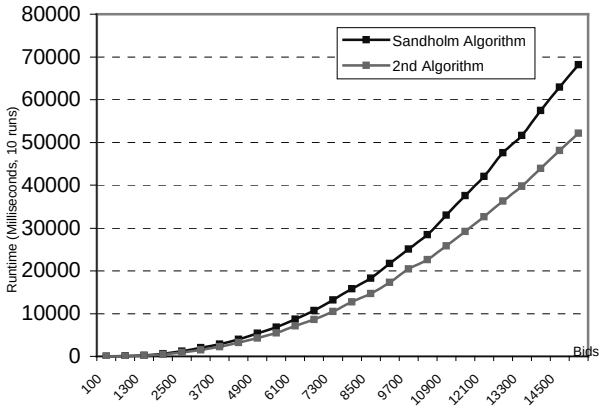

Figure 8: Temps d'exécution vs. nombre d'offres avec 300 items.

On peut voir sur cette figure que la différence est de l'ordre de 20 seconds. Pour vérifier ces observations, nous avons continué de tester ces deux algorithmes sur des exemples avec encore plus d'items. Nous avons fait varier le nombre d'items jusqu'à 10000.

Sur la Figure 9 nous donnons les courbes qui indiquent les écarts en temps d'exécution entre ces algorithmes lorsque le nombre d'items varie de 200 à 800. Pour chacun des cas, nous avons fait varier le nombre d'offres sur plus de 15000 et nous avons effectué au moins 10 tests avant de relever ces résultats. Par exemple, la différence entre le temps d'exécution des deux

algorithmes est de l'ordre de 2000 ms avec 6700 offres et ce quel que soit le nombre d'items.

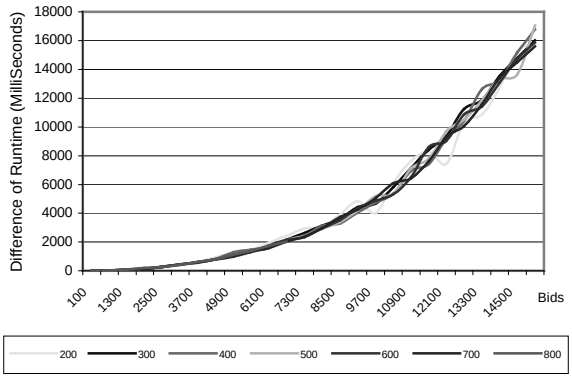

Figure 9: Ecarts entre l'algorithme -II- et l'algorithme de [9] avec une variation du nombre d'items entre 200 et 800.

L'évaluation du temps d'exécution des algorithmes n'était pas suffisante, il était également nécessaire d'évaluer la qualité de la solution fournie par chacun d'eux et de vérifier leur propriété anytime. Cette propriété est importante puisqu'elle garantit que ces algorithmes soient en mesure de proposer une solution même sous-optimale avant la fin de leur exécution, chose importante pour la négociation dans l'application de commerce électronique que nous avons choisie. Nous avons ainsi effectué d'autres essais et avons mesuré la qualité de la solution obtenue par chaque algorithme à différents moments. La Figure 10 présente les résultats de la comparaison de ces deux algorithmes avec 100 items et des offres variant entre 40 et 15000. Dans cette figure, nous pouvons observer que l'algorithme de [9] trouve une solution avec 65,8% d'optimalité après 26,4% du temps d'exécution total et une solution avec l'optimalité 100% après 28% du temps d'exécution.

Cependant, à plusieurs occasions, notre algorithme -II- a trouvé une solution avec plus de 70% d'optimalité avant 15% du temps d'exécution et avec 100% avant 20% du temps d'exécution. Globalement, nous pouvons voir dans cette figure que presque tous les points de notre algorithme sont inférieurs aux points de [9] sur l'axe des Y qui représente le temps et que la qualité de la solution est également préservée sur l'axe des X.

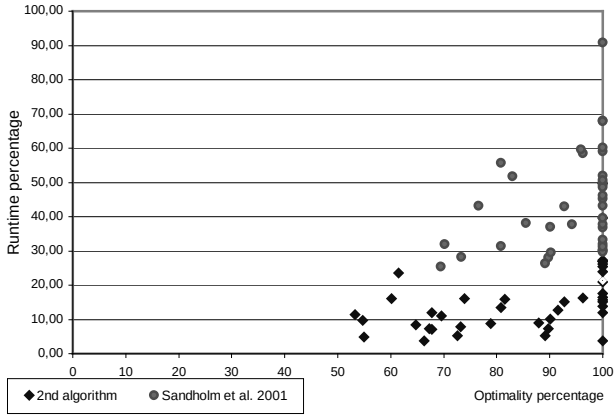

Figure 10 : Pourcentage du temps d'exécution vs. Pourcentage d'optimalité.

3.2 Deuxième série expérimentale

Dans cette deuxième série, nous avons comparé les résultats de notre premier algorithme à [8][5]. La Figure 11 montre les temps d'exécution moyens obtenus en utilisant l'algorithme -I- avec le nombre d'offres variant entre 500 et 3500 et le nombre d'items variant entre 25 et 55.

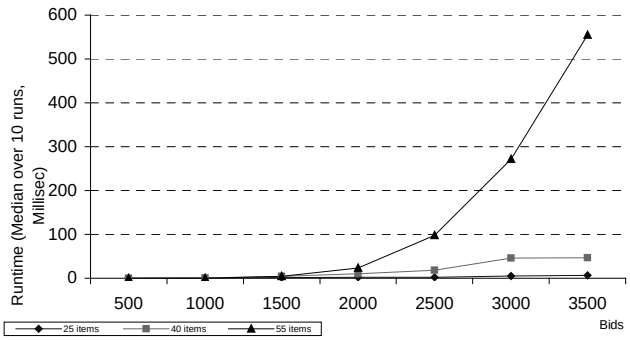

Figure 11: : Temps d'exécution vs. nombre d'offres et d'items.

Ces résultats montrent que la complexité croit exponentiellement par rapport au nombre d'items dès que le nombre d'offres est supérieur à 2500. En utilisant cette précédente distribution, il est difficile de satisfaire une requête avec un nombre élevé d'items. Cependant, les cas sans solution sont rapidement résolus par nos algorithmes. En fait, avant de rechercher les descendants d'un nœud dans l'arbre des solutions, l'algorithme vérifie d'abord que les items qui n'apparaissent pas dans une branche sont présents dans les partitions d'offres qui n'ont pas été déjà explorées.

La Figure 12 donne les résultats obtenus avec plusieurs exécutions. Nous avons appliqué notre premier algorithme avec 6000 offres et une requête contenant 60 items. L'axe des X donne le temps en millisecondes consommé par notre algorithme et l'axe des Y représente la qualité de la solution. Pour calculer cette valeur à un instant donné nous faisons un rapport entre le coût de la meilleure solution trouvée à la fin du processus de résolution du problème et le coût de la solution à cet instant. Nous avons observé qu'à la fin des premières 50ms, l'algorithme avait déjà trouvé une solution optimale à 50%.

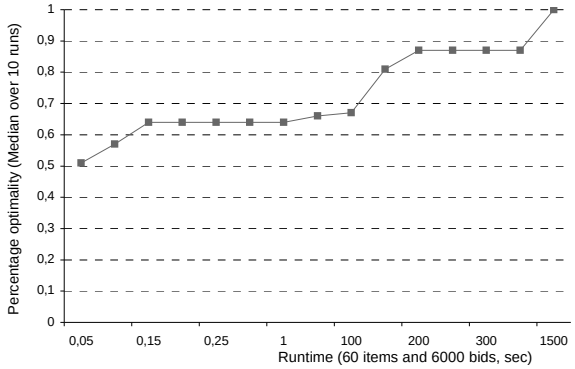

Figure 12. Evolution de la qualité de la solution vs. temps.

Nous avons continué nos expérimentations avec d'autres distributions rendant plus difficile la résolution du problème. Nous présentons ci-après les résultats avec celle présentée dans [5][8]. Comme avec la première distribution, nous avons mesuré le temps d'exécution de chaque algorithme sur la base de plusieurs tests.

Dans la Figure 13, nous indiquons le temps consommé avec la méthode de Sandholm *et al* [12], la méthode Fujishima *et al*. [5] et ensuite la nôtre. Selon ces tests les résultats de Fujishima *et al*. semblent assez proches des nôtres.

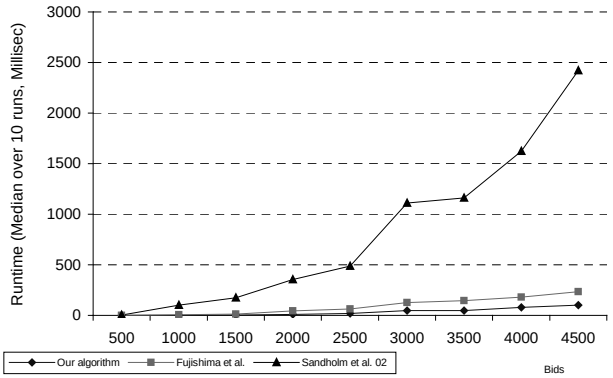

Figure 13. Temps d'exécution pour les algorithmes de Fujishima *et al.,* Sandholm *et al.* et l'algorithme -I- avec 40 items.

La deuxième distribution utilisée est celle de [5]. Le nombre d'items *n* d'une offre est généré aléatoirement avec une probabilité $C\ exp^{-n/p}$ avec $p=5$ et $C=1/p$. Les items composant une offre sont générés aléatoirement et le prix aussi sur l'intervalle $[n*(1-d),\ n*(1+d)]$, avec $d=0.5$.

La Figure 14 compare notre algorithme -I- à celui de [5] sur des requêtes ayant 40 items et un nombre d'offres variant de 200 à 2000. Ces résultats montrent que notre algorithme est plus efficace que celui de [5]. En fait, avec cette distribution le temps requis par chacun de ces algorithmes pour traiter les requêtes est plus élevé que dans la précédente distribution. En effet, cette distribution produit fréquemment des offres de moins de 30 items. Ceci rend le problème plus complexe et la recherche dans l'arbre des solutions devient plus profonde.

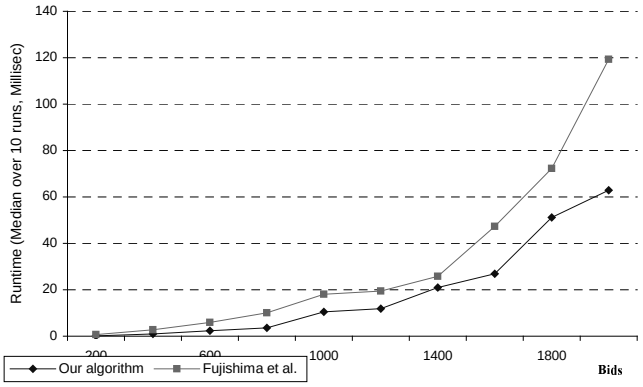

Figure 14. Temps d'exécution pour l'algorithme de [5] et l'algorithme -I- avec 40 items.

3.3 Troisième série expérimentale

Dans cette dernière série, nous avons voulu effectuer une comparaison générale et finale entre nos deux algorithmes et ceux de Sandholm *et al.* Les expérimentations ont été effectuées sur un échantillon d'offres composées de 20 items et un nombre d'offres variant entre 100 et 20000. Les offres ont été générées aléatoirement. Sur la figure 15, nous donnons les résultats obtenus dans cette série.

On remarque clairement sur cette figure le rapprochement entre les deux types d'algorithmes, i.e., l'algorithme -I- et celui de Sandholm [8] présentés dans le haut de la figure, et l'algorithme -II- et celui de 10 présentés dans le bas de cette figure. Il résulte de ces tests que les graphes et les heuristiques proposées améliorent effectivement les algorithmes de Sandholm *et al.*

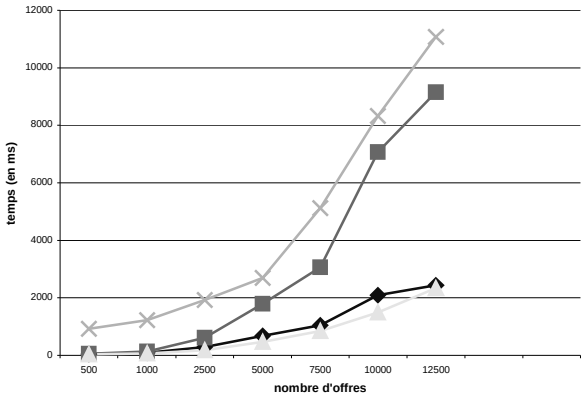

Figure 15. Temps d'exécution pour les algorithmes de Sandholm et les algorithmes -I- et -II- avec 20 items.

4. Conclusion et Perspectives

Cet article a présenté deux nouveaux algorithmes pour le problème de détermination du gagnant dans une négociation combinée. Ces algorithmes ont d'abord été définis pour les requêtes conjonctives et sont basés sur des travaux récents [5][8][9][11]. Ces différents algorithmes ont été implémentés et testés sur différentes distributions d'offres pour évaluer leur performance. Nos algorithmes sont essentiellement basés sur trois nouveaux principes: (1) ils utilisent de nouveaux graphes pour les items partagés et non-partagés ; (2) de nouvelles fonctions pour ordonner les offres et (3) la propriété de singularité des items.

Après avoir testé nos algorithmes plus de six cents fois sur plusieurs problèmes, ils semblent plus rapides que les meilleurs algorithmes puisqu'ils bénéficient de quelques propriétés de [5][9][8] et des heuristiques que nous avons présentées. Plusieurs extensions de ce travail peuvent être considérées.

Dans les travaux actuels, aucun n'a abordé le problème de la l'adaptation de ces algorithmes aux contraintes de la négociation combinée multi-agent [2]. En effet, dans un processus de négociation, les offres des agents sont constamment évolutives, des offres peuvent être annulées, des nouvelles peuvent arriver ou même les valeurs des anciennes offres qui peuvent augmenter ou baisser. Pour chaque changement, une nouvelle solution optimale doit être trouvée dynamiquement. Il est donc nécessaire d'intégrer la contrainte d'évolution dans ces algorithmes. Le travail en cours consiste à définir un algorithme qui effectue cette recherche incrémentale en s'appuyant sur ces graphes.

Remerciement

Merci à Lucie Galand, Sophie Besset, Aurélie Lenoble, Loïc Noach, Sylvia Estivi et Ait-Mohamed Mohand pour avoir implémenté une très grande partie de ce travail.

Références

[1] S. Aknine., "New Multi-Agent Protocols for M-N-P Negotiations in Electronic Commerce", *AAAI, Workshop*, 9-14, Edmonton, Alberta, Canada, July, 2002.

[2] S. Aknine. "Strategies and Behaviours of Agents in Multi-phased Negotiations", Third International Conference on Electronic Commerce and Web Technologies, *EC-WEB, Lecture Notes on Computer Science, LNCS 2455, pp. 17- 26, Springer Verlag, September 2-6, Aix-en-Provence, France*, 2002.

[3] C. Boutilier, Solving Concisely Expressed Combinatorial Auction Problems. *AAAI,* 2002.

[4] C. Boutilier, and H. Hoos. Offreding Languages for Combinatorial Auctions. *IJCAI,* 2001.

[5] Y. Fujishima, K. Leyton-Brown, and Y. Shoham. Taming the computational complexity of combinatorial auctions: Optimal and approximate approaches. *IJCAI*, 548-553, 1999.

[6] R. E. Korf. Depth-first iterative-deepening: An optimal admissible Tree search. *Artificial Intelligence,* 27:97-109, 1985.

[7] M. Rothkopf, A., Pekeč, and R., Harstad. *Computationally manageable combinatorial auctions,* Management Science, 44-8, 1131-1147, 1998.

[8] T. Sandholm. Algorithm for Optimal Winner Determination in Combinatorial Auctions. *Artificial Intelligence, 135:1-54,* 2002.

[9] T. Sandholm, S. Suri, A. Gilpin, and D. Levine. CABOB: A fast optimal algorithm for combinatorial auctions. *IJCAI*, 1102-1108, 2001.

[10] T. Sandholm, and S. Suri, BOB: Improved Winner Determination in Combinatorial Auctions and Generalizations. *Artificial Intelligence,* 145, 2003.

[11] T. Sandholm, and S. Suri. Improved algorithms for optimal winner determination in combinatorial auctions and generalizations. *AAAI*, 2000.

[12] T. Sandholm. An algorithm for optimal winner determination in combinatorial auctions. *IJCAI, 542-547,* 1999.