Bidding Algorithms for a Distributed Combinatorial Auction

Benito Mendoza* and José M. Vidal
Computer Science and Engineering
University of South Carolina
Columbia, SC 29208
mendoza2@engr.sc.edu, vidal@sc.edu

ABSTRACT

Distributed allocation and multiagent coordination problems can be solved through combinatorial auctions. However, most of the existing winner determination algorithms for combinatorial auctions are centralized. The PAUSE auction is one of a few efforts to release the auctioneer from having to do all the work (it might even be possible to get rid of the auctioneer). It is an increasing price combinatorial auction that naturally distributes the problem of winner determination amongst the bidders in such a way that they have an incentive to perform the calculation. It can be used when we wish to distribute the computational load among the bidders or when the bidders do not wish to reveal their true valuations unless necessary. PAUSE establishes the rules the bidders must obey. However, it does not tell us how the bidders should calculate their bids. We have developed a couple of bidding algorithms for the bidders in a PAUSE auction. Our algorithms always return the set of bids that maximizes the bidder's utility. Since the problem is NP-Hard, run time remains exponential on the number of items, but it is remarkably better than an exhaustive search. In this paper we present our bidding algorithms, discuss their virtues and drawbacks, and compare the solutions obtained by them to the revenue-maximizing solution found by a centralized winner determination algorithm.

Categories and Subject Descriptors

I.2.11 [**Computing Methodologies**]: Distributed Artificial Intelligence—*Intelligent Agents, Multiagent Systems.*

General Terms

Algorithms, Performance.

Keywords

Combinatorial Auctions, Coordination, Task and resource allocation.

*The first author is a student.

AAMAS'07 May 14–18 2007, Honolulu, Hawai'i, USA.

1. INTRODUCTION

Both the research and practice of combinatorial auctions have grown rapidly in the past ten years. In a combinatorial auction bidders can place bids on combinations of items, called packages or bidsets, rather than just individual items. Once the bidders place their bids, it is necessary to find the allocation of items to bidders that maximizes the auctioneer's revenue. This problem, known as the winner determination problem, is a combinatorial optimization problem and is NP-Hard [10]. Nevertheless, several algorithms that have a satisfactory performance for problem sizes and structures occurring in practice have been developed. The practical applications of combinatorial auctions include: allocation of airport takeoff and landing time slots, procurement of freight transportation services, procurement of public transport services, and industrial procurement [2]. Because of their wide applicability, one cannot hope for a general-purpose winner determination algorithm that can efficiently solve every instance of the problem. Thus, several approaches and algorithms have been proposed to address the winner determination problem. However, most of the existing winner determination algorithms for combinatorial auctions are centralized, meaning that they require all agents to send their bids to a centralized auctioneer who then determines the winners. Examples of these algorithms are CASS [3], Bidtree [11] and CABOB [12]. We believe that distributed solutions to the winner determination problem should be studied as they offer a better fit for some applications as when, for example, agents do not want to reveal their valuations to the auctioneer.

The PAUSE (Progressive Adaptive User Selection Environment) auction [4, 5] is one of a few efforts to distribute the problem of winner determination amongst the bidders. PAUSE establishes the rules the participants have to adhere to so that the work is distributed amongst them. However, it is not concerned with how the bidders determine what they should bid.

In this paper we present two algorithms, PAUSEBID and CACHEDPAUSEBID, which enable agents in a PAUSE auction to find the bidset that maximizes their utility. Our algorithms implement a myopic utility maximizing strategy and are guaranteed to find the bidset that maximizes the agent's utility given the outstanding best bids at a given time. PAUSEBID performs a branch and bound search completely from scratch every time that it is called. CACHEDPAUSEBID is a caching-based algorithm which explores fewer nodes, since it caches some solutions.

978-81-904262-7-5 (RPS)

2. THE PAUSE AUCTION

A PAUSE auction for m items has m stages. Stage 1 consists of having simultaneous ascending price open-cry auctions and during this stage the bidders can only place bids on individual items. At the end of this state we will know what the highest bid for each individual item is and who placed that bid. Each successive stage $k = 2, 3, \ldots, m$ consists of an ascending price auction where the bidders must submit bidsets that cover all items but each one of the bids must be for k items or less. The bidders are allowed to use bids that other agents have placed in previous rounds when building their bidsets, thus allowing them to find better solutions. Also, any new bidset has to have a sum of bid prices which is bigger than that of the currently winning bidset. At the end of each stage k all agents know the best bid for every subset of size k or less. Also, at any point in time after stage 1 has ended there is a standing bidset whose value increases monotonically as new bidsets are submitted. Since in the final round all agents consider all possible bidsets, we know that the final winning bidset will be one such that no agent can propose a better bidset. Note, however, that this bidset is not guaranteed to be the one that maximizes revenue since we are using an ascending price auction so the winning bid for each set will be only slightly bigger than the second highest bid for the particular set of items. That is, the final prices will not be the same as the prices in a traditional combinatorial auction where all the bidders bid their true valuation. However, there remains the open question of whether the final distribution of items to bidders found in a PAUSE auction is the same as the revenue maximizing solution. Our test results provide an answer to this question.

The PAUSE auction makes the job of the auctioneer very easy. All it has to do is to make sure that each new bidset has a revenue bigger than the current winning bidset, as well as make sure that every bid in an agent's bidset that is not his does indeed correspond to some other agents' previous bid. The computational problem shifts from one of winner determination to one of bid generation. Each agent must search over the space of all bidsets which contain at least one of its bids. The search is made easier by the fact that the agent needs to consider only the current best bids and only wants bidsets where its own utility is higher than in the current winning bidset. Each agent also has a clear incentive for performing this computation, namely, its utility only increases with each bidset it proposes (of course, it might decrease with the bidsets that others propose). Finally, the PAUSE auction has been shown to be envy-free in that at the conclusion of the auction no bidder would prefer to exchange his allocation with that of any other bidder [2].

We can even envision completely eliminating the auctioneer and, instead, have every agent perform the task of the auctioneer. That is, all bids are broadcast and when an agent receives a bid from another agent it updates the set of best bids and determines if the new bid is indeed better than the current winning bid. The agents would have an incentive to perform their computation as it will increase their expected utility. Also, any lies about other agents' bids are easily found out by keeping track of the bids sent out by every agent (the set of best bids). Namely, the only one that can increase an agent's bid value is the agent itself. Anyone claiming a higher value for some other agent is lying. The only thing missing is an algorithm that calculates the utility-maximizing bidset for each agent.

3. PROBLEM FORMULATION

A bid b is composed of three elements b^{items} (the set of items the bid is over), b^{agent} (the agent that placed the bid), and b^{value} (the value or price of the bid). The agents maintain a set B of the current best bids, one for each set of items of size $\leq k$, where k is the current stage. At any point in the auction, after the first round, there will also be a set $W \subseteq B$ of currently winning bids. This is the set of bids that covers all the items and currently maximizes the revenue, where the revenue of W is given by

$$r(W) = \sum_{b \in W} b^{\text{value}}. \tag{1}$$

Agent i's value function is given by $v_i(S) \in \Re$ where S is a set of items. Given an agent's value function and the current winning bidset W we can calculate the agent's utility from W as

$$u_i(W) = \sum_{b \in W \mid b^{\text{agent}} = i} v_i(b^{\text{items}}) - b^{\text{value}}. \tag{2}$$

That is, the agent's utility for a bidset W is the value it receives for the items it wins in W minus the price it must pay for those items. If the agent is not winning any items then its utility is zero.

The goal of the bidding agents in the PAUSE auction is to maximize their utility, subject to the constraint that their next set of bids must have a total revenue that is at least ϵ bigger than the current revenue, where ϵ is the smallest increment allowed in the auction. Formally, given that W is the current winning bidset, agent i must find a g_i^* such that $r(g_i^*) \geq r(W) + \epsilon$ and

$$g_i^* = \arg\max_{g \subseteq 2^B} u_i(g), \tag{3}$$

where each g is a set of bids that covers all items and $\forall_{b \in g}$ $(b \in B)$ or $(b^{\text{agent}} = i$ and $b^{\text{value}} > B(b^{\text{items}})$ and $size(b^{\text{items}}) \leq k)$, and where $B(items)$ is the value of the bid in B for the set $items$ (if there is no bid for those items it returns zero). That is, each bid b in g must satisfy at least one of the two following conditions. 1) b is already in B, 2) b is a bid of size $\leq k$ in which the agent i bids higher than the price for the same items in B.

4. BIDDING ALGORITHMS

According to the PAUSE auction, during the first stage we have only several English auctions, with the bidders submitting bids on individual items. In this case, an agent's dominant strategy is to bid ϵ higher than the current winning bid until it reaches its valuation for that particular item. Our algorithms focus on the subsequent stages: $k > 1$. When $k > 1$, agents have to find g_i^*. This can be done by performing a complete search on B. However, this approach is computationally expensive since it produces a large search tree. Our algorithms represent alternative approaches to overcome this expensive search.

4.1 The PAUSEBID Algorithm

In the PAUSEBID algorithm (shown in Figure 1) we implement some heuristics to prune the search tree. Given that bidders want to maximize their utility and that at any given point there are likely only a few bids within B which

```
PAUSEBID(i, k)
 1  my-bids ← ∅
 2  their-bids ← ∅
 3  for b ∈ B
 4      do if b^agent = i or v_i(b^items) > b^value
 5          then my-bids ← my-bids
                          +new Bid(b^items, i, v_i(b^items))
 6          else  their-bids ← their-bids +b
 7  for S ∈ subsets of k or fewer items such that
          v_i(S) > 0 and ¬∃_{b∈B} b^items = S
 8      do my-bids ← my-bids +new Bid(S, i, v_i(S))
 9  bids ← my-bids + their-bids
10  g* ← ∅          ▷ Global variable
11  u* ← u_i(W) ▷ Global variable
12  PBSEARCH(bids, ∅)
13  surplus ← Σ_{b∈g* | b^agent=i} b^value − B(b^items)
14  if surplus = 0
15   then return g*
16  my-payment ← v_i(g*) − u*
17  for b ∈ g* | b^agent = i
18      do if my-payment ≤ 0
19          then b^value ← B(b^items)
20          else  b^value ← B(b^items)
                          + my-payment · b^value − B(b^items) / surplus
21  return g*
```

Figure 1: The PAUSEBID algorithm which implements a branch and bound search. i is the agent and k is the current stage of the auction, for $k \geq 2$.

```
PBSEARCH(bids, g)
 1  if bids = ∅ then return
 2  b ← first(bids)
 3  bids ← bids −b
 4  g ← g + b
 5  Ī_g ← items not in g
 6  if g does not contain a bid from i
 7   then return
 8  if g includes all items
 9   then min-payment ← max(0, r(W) + ε − (r(g) − r_i(g)),
                Σ_{b∈g | b^agent=i} B(b^items))
10        max-utility ← v_i(g) − min-payment
11        if r(g) > r(W) and max-utility ≥ u*
12        then g* ← g
13             u* ← max-utility
14        PBSEARCH(bids, g − b) ▷ b is Out
15   else  max-revenue ← r(g) + max(h(Ī_g), h_i(Ī_g))
16        if max-revenue ≤ r(W)
17        then PBSEARCH(bids, g − b) ▷ b is Out
18        elseif b^agent ≠ i
19        then min-payment ← (r(W) + ε)
                      −(r(g) − r_i(g)) − h(Ī_g)
20             max-utility ← v_i(g) − min-payment
21             if max-utility > u*
22             then PBSEARCH({x ∈ bids |
                      x^items ∩ b^items = ∅}, g) ▷ b is In
23             PBSEARCH(bids, g − b) ▷ b is Out
24        else
25             PBSEARCH({x ∈ bids |
               x^items ∩ b^items = ∅}, g) ▷ b is In
26             PBSEARCH(bids, g − b) ▷ b is Out
27  return
```

Figure 2: The PBSEARCH recursive procedure where *bids* is the set of available bids and g is the current partial solution.

the agent can dominate, we start by defining *my-bids* to be the list of bids for which the agent's valuation is higher than the current best bid, as given in B. We set the value of these bids to be the agent's true valuation (but we won't necessarily be bidding true valuation, as we explain later). Similarly, we set *their-bids* to be the rest of the bids from B. Finally, the agent's search list is simply the concatenation of *my-bids* and *their-bids*. Note that the agent's own bids are placed first on the search list as this will enable us to do more pruning (PAUSEBID lines 3 to 9). The agent can now perform a branch and bound search on the branch-on-bids tree produced by these bids. This branch and bound search is implemented by PBSEARCH (Figure 2). Our algorithm not only implements the standard bound but it also implements other pruning techniques in order to further reduce the size of the search tree.

The bound we use is the maximum utility that the agent can expect to receive from a given set of bids. We call it u^*. Initially, u^* is set to $u_i(W)$ (PAUSEBID line 11) since that is the utility the agent currently receives and any solution he proposes should give him more utility. If PBSEARCH ever comes across a partial solution where the maximum utility the agent can expect to receive is less than u^* then that subtree is pruned (PBSEARCH line 21). Note that we can determine the maximum utility only after the algorithm has searched over all of the agent's own bids (which are first on the list) because after that we know that the solution will not include any more bids where the agent is the winner thus the agent's utility will no longer increase. For example, if an agent has only one bid in *my-bids* then the maximum utility he can expect is equal to his value for the items in that bid minus the minimum possible payment we can make for those items and still come up with a set of bids that has revenue greater than $r(W)$. The calculation of the minimum payment is shown in line 19 for the partial solution case and line 9 for the case where we have a complete solution in PBSEARCH. Note that in order to calculate the *min-payment* for the partial solution case we need an upper bound on the payments that we must make for each item. This upper bound is provided by

$$h(S) = \sum_{s \in S} \max_{b \in B \mid s \in b^{\text{items}}} \frac{b^{\text{value}}}{size(b^{\text{items}})}. \qquad (4)$$

This function produces a bound identical to the one used by the Bidtree algorithm—it merely assigns to each individual item in S a value equal to the maximum bid in B divided by the number of items in that bid.

To prune the branches that cannot lead to a solution with revenue greater than the current W, the algorithm considers both the values of the bids in B and the valuations of the

agent. Similarly to (4) we define

$$h_i(S,k) = \sum_{s \in S} \max_{S' \mid size(S') \leq k \text{ and } s \in S' \text{ and } v_i(S') > 0} \frac{v_i(S')}{size(S')} \quad (5)$$

which assigns to each individual item s in S the maximum value produced by the valuation of S' divided by the size of S', where S' is a set for which the agent has a valuation greater than zero, contains s, and its size is less or equal than k. The algorithm uses the heuristics h and h_i (lines 15 and 19 of PBSEARCH), to prune the just mentioned branches in the same way an A^* algorithm uses its heuristic. A final pruning technique implemented by the algorithm is ignoring any branches where the agent has no bids in the current answer g and no more of the agent's bids are in the list (PBSEARCH lines 6 and 7).

The resulting g^* found by PBSEARCH is thus the set of bids that has revenue bigger than $r(W)$ and maximizes agent i's utility. However, agent i's bids in g^* are still set to his own valuation and not to the lowest possible price. Lines 17 to 20 in PAUSEBID are responsible for setting the agent's payments so that it can achieve its maximum utility u^*. If the agent has only one bid in g^* then it is simply a matter of reducing the payment of that bid by u^* from the current maximum of the agent's true valuation. However, if the agent has more than one bid then we face the problem of how to distribute the agent's payments among these bids. There are many ways of distributing the payments and there does not appear to be a dominant strategy for performing this distribution. We have chosen to distribute the payments in proportion to the agent's true valuation for each set of items.

PAUSEBID assumes that the set of best bids B and the current best winning bidset W remains constant during its execution, and it returns the agent's myopic utility-maximizing bidset (if there is one) using a branch and bound search. However it repeats the whole search at every stage. We can minimize this problem by caching the result of previous searches.

4.2 The CACHEDPAUSEBID Algorithm

The CACHEDPAUSEBID algorithm (shown in Figure 3) is our second approach to solve the bidding problem in the PAUSE auction. It is based in a cache table called *C-Table* where we store some solutions to avoid doing a complete search every time. The problem is the same; the agent i has to find g_i^*. We note that g_i^* is a bidset that contains at least one bid of the agent i. Let S be a set of items for which the agent i has a valuation such that $v_i(S) \geq B(S) > 0$, let g_i^S be a bidset over S such that $r(g_i^S) \geq r(W) + \epsilon$ and

$$g_i^S = \arg \max_{g \subseteq 2^B} u_i(g), \quad (6)$$

where each g is a set of bids that covers all items and $\forall_{b \in g}\ (b \in B)$ or $(b^{\text{agent}} = i$ and $b^{\text{value}} > B(b^{\text{items}}))$ and $(\exists_{b \in g} b^{\text{items}} = S$ and $b^{\text{agent}} = i)$. That is, g_i^S is i's best bidset for all items which includes a bid from i for all S items. In the PAUSE auction we cannot bid for sets of items with size greater than k. So, if we have for each set of items S for which $v_i(S) > 0$ and $size(S) \leq k$ its corresponding g_i^S then g_i^* is the g_i^S that maximizes the agent's utility. That is

$$g_i^* = \arg \max_{\{S \mid v_i(S) > 0 \wedge size(S) \leq k\}} u_i(g_i^S). \quad (7)$$

Each agent i implements a hash table *C-Table* such that *C-Table*$[S] = g^S$ for all S which $v_i(S) \geq B(S) > 0$. We can

CACHEDPAUSEBID$(i, k, k\text{-}changed)$

```
1  for each S in C-Table
2     do if v_i(S) < B(S)
3        then remove S from C-Table
4        else if k-changed and size(S) = k
5                then B' ← B' + new Bid(i, S, v_i(S))
6  g* ← ∅
7  u* ← u_i(W)
8  for each S with size(S) ≤ k in C-Table
9     do S̄ ← Items − S
10       g^S ← C-Table[S]                    ▷ Global variable
11       min-payment ← max(r(W) + ε, Σ_{b∈g^S} B(b^items))
12       u^S ← r(g^S) − min-payment ▷ Global variable
13       if (k-changed and size(S) = k)
                or (∃_{b∈B'} b^items ⊆ S̄ and b^agent ≠ i)
14       then B'' ← {b ∈ B' | b^items ⊆ S̄}
15            bids ← B''
                   + {b ∈ B | b^items ⊆ S̄ and b ∉ B''}
16            for b ∈ bids
17               do if v_i(b^items) > b^value
18                  then b^agent ← i
19                       b^value ← v_i(b^items)
20            if k-changed and size(S) = k
21            then n ← size(bids)
22                 u^S ← 0
23            else n ← size(B'')
24            g ← ∅ + new Bid(S, i, v_i(S))
25            CPBSEARCH(bids, g, n)
26            C-Table[S] ← g^S
27       if u^S > u* and r(g^S) ≥ r(W) + ε
28       then surplus ←
                Σ_{b∈g^S | b^agent=i} b^value − B(b^items)
29            if surplus > 0
30            then my-payment ← v_i(g^S) − u_i(g^S)
31                 for b ∈ g^S | b^agent = i
32                    do if my-payment ≤ 0
33                       then b^value ← B(b^items)
34                       else b^value ← B(b^items) +
                                my-payment · (b^value − B(b^items)) / surplus
35            u* ← u_i(g^S)
36            g* ← g^S
37       else if u^S ≤ 0 and v_i(S) < B(S)
38            then remove S from C-Table
39  return g*
```

Figure 3: The CACHEDPAUSEBID algorithm that implements a caching based search to find a bidset that maximizes the utility for the agent i. k is the current stage of the auction (for $k \geq 2$), and *k-changed* is a boolean that is true right after the auction moved to the next stage.

```
CPBSEARCH(bids, g, n)
 1  if bids = ∅ or n ≤ 0 then return
 2  b ← first(bids)
 3  bids ← bids −b
 4  g ← g + b
 5  Ī_g ← items not in g
 6  if g includes all items
 7   then min-payment ← max(0, r(W) + ε − (r(g) − r_i(g)),
                 Σ_{b∈g | b^agent=i} B(b^items))
 8        max-utility ← v_i(g) − min-payment
 9        if r(g) > r(W) and max-utility ≥ u^S
10        then g^S ← g
11             u^S ← max-utility
12        CPBSEARCH(bids, g − b, n − 1) ▷ b is Out
13   else max-revenue ← r(g) + max(h(Ī_g), h_i(Ī_g))
14        if max-revenue ≤ r(W)
15        then CPBSEARCH(bids, g − b, n − 1) ▷ b is Out
16        elseif b^agent ≠ i
17        then min-payment ← (r(W) + ε)
                      −(r(g) − r_i(g)) − h(Ī_g)
18             max-utility ← v_i(g) − min-payment
19             if max-utility > u^S
20             then CPBSEARCH({x ∈ bids |
                      x^items ∩ b^items = ∅}, g, n + 1) ▷ b is In
21             CPBSEARCH(bids, g − b, n − 1) ▷ b is Out
22        else
23             CPBSEARCH({x ∈ bids |
                      x^items ∩ b^items = ∅}, g, n + 1) ▷ b is In
24             CPBSEARCH(bids, g − b, n − 1) ▷ b is Out
25  return
```

Figure 4: The CPBSEARCH recursive procedure where *bids* is the set of available bids, g is the current partial solution and n is a value that indicates how deep in the list *bids* the algorithm has to search.

then find g^* by searching for the g^S, stored in *C-Table*$[S]$, that maximizes the agent's utility, considering only the set of items S with $size(S) \leq k$. The problem remains in maintaining the *C-Table* updated and avoiding to search every g^S every time. CACHEDPAUSEBID deals with this and other details.

Let B' be the set of bids that contains the new best bids, that is, B' contains the bids recently added to B and the bids that have changed price (always higher), bidder, or both and were already in B. Let $\bar{S} = Items - S$ be the complement of S (the set of items not included in S). CACHEDPAUSEBID takes three parameters: i the agent, k the current stage of the auction, and *k-changed* a boolean that is true right after the auction moved to the next stage. Initially *C-Table* has one row or entry for each set S for which $v_i(S) > 0$. We start by eliminating the entries corresponding to each set S for which $v_i(S) < B(S)$ from *C-Table* (line 3). Then, in the case that *k-changed* is true, for each set S with $size(S) = k$, we add to B' a bid for that set with value equal to $v_i(S)$ and bidder agent i (line 5); this a bid that the agent is now allowed to consider. We then search for g^* amongst the g^S stored in *C-Table*, for this we only need to consider the sets with $size(S) \leq k$ (line 8). But how do we know that the g^S in *C-Table*$[S]$ is still the best solution for S? There are only two cases when we are not sure about that and we need to do a search to update *C-Table*$[S]$. These cases are: i) When *k-changed* is true and $size(S) \leq k$, since there was no g^S stored in *C-Table* for this S. ii) When there exists at least one bid in B' for the set of items $\bar{S}$ or a subset of it submitted by an agent different than i, since it is probable that this new bid can produce a solution better than the one stored in *C-Table*$[S]$.

We handle the two cases mentioned above in lines 13 to 26 of CACHEDPAUSEBID. In both of these cases, since g^S must contain a bid for S we need to find a bidset that cover the missing items, that is $\bar{S}$. Thus, our search space consists of all the bids on B for the set of items $\bar{S}$ or for a subset of it. We build the list *bids* that contains only those bids. However, we put the bids from B' at the beginning of *bids* (line 14) since they are the ones that have changed. Then, we replace the bids in *bids* that have a price lower than the valuation the agent i has for those same items with a bid from agent i for those items and value equal to the agent's valuation (lines 16–19).

The recursive procedure CPBSEARCH, called in line 25 of CACHEDPAUSEBID and shown in Figure 4, is the one that finds the new g^S. CPBSEARCH is a slightly modified version of our branch and bound search implemented in PBSEARCH. The first modification is that it has a third parameter n that indicates how deep on the list *bids* we want to search, since it stops searching when n less or equal to zero and not only when the list *bids* is empty (line 1). Each time that there is a recursive call of CPBSEARCH n is decreased by one when a bid from *bids* is discarded or out (lines 12, 15, 21, and 24) and n remains the same otherwise (lines 20 and 23). We set the value of n before calling CPBSEARCH, to be the size of the list bids (CACHEDPAUSEBID line 21) in case i), since we want CPBSEARCH to search over all *bids*; and we set n to be the number of bids from B' included in bids (CACHEDPAUSEBID line 23) in case ii), since we know that only the those first n bids in *bids* changed and can affect our current g^S.

Another difference with PBSEARCH is that the bound in CPBSEARCH is u^S which we set to be 0 (CACHEDPAUSEBID line 22) when in case i) and $r(g^S)-$*min-payment* (CACHEDPAUSEBID line 12) when in case ii). We call CPBSEARCH with g already containing a bid for S. After CPBSEARCH is executed we are sure that we have the right g^S, so we store it in the corresponding *C-Table*$[S]$ (CACHEDPAUSEBID line 26).

When we reach line 27 in CACHEDPAUSEBID, we are sure that we have the right g^S. However, agent i's bids in g^S are still set to his own valuation and not to the lowest possible price. If u^S is greater than the current u^*, lines 31 to 34 in CACHEDPAUSEBID are responsible for setting the agent's payments so that it can achieve its maximum utility u^S. As in PAUSEBID, we have chosen to distribute the payments in proportion to the agent's true valuation for each set of items. In the case that u^S less than or equal to zero and the valuation that the agent i has for the set of items S is lower than the current value of the bid in B for the same set of items, we remove the corresponding *C-Table*$[S]$ since we know that is not worthwhile to keep it in the cache table (CACHEDPAUSEBID line 38).

The CACHEDPAUSEBID function is called when $k > 1$ and returns the agent's myopic utility-maximizing bidset, if there is one. It assumes that W and B remains constant during its execution.

GENERATEVALUES(i, $items$)

```
1  for x ∈ items
2      do v_i(x) = EXPD(.01)
3  for n ← 1 ... (num-bids − items)
4      do s_1, s_2 ←Two random sets of items with values.
5          v_i(s_1 ∪ s_2) = v_i(s_1) + v_i(s_2) + EXPD(.01)
```

Figure 5: Algorithm for the generation of random value functions. EXPD(x) returns a random number taken from an exponential distribution with mean $1/x$.

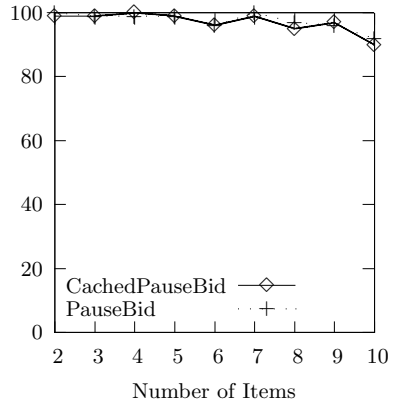

Figure 6: Average percentage of convergence (*y-axis*), which is the percentage of times that our algorithms converge to the revenue-maximizing solution, as function of the number of items in the auction.

5. TEST AND COMPARISON

We have implemented both algorithms and performed a series of experiments in order to determine how their solution compares to the revenue-maximizing solution and how their times compare with each other. In order to do our tests we had to generate value functions for the agents[1]. The algorithm we used is shown in Figure 5. The type of valuations it generates correspond to domains where a set of agents must perform a set of tasks but there are cost savings for particular agents if they can bundle together certain subsets of tasks. For example, imagine a set of robots which must pick up and deliver items to different locations. Since each robot is at a different location and has different abilities, each one will have different preferences over how to bundle. Their costs for the item bundles are subadditive, which means that their preferences are superadditive. The first experiment we performed simply ensured the proper

[1]Note that we could not use CATS [6] because it generates sets of bids for an indeterminate number of agents. It is as if you were told the set of bids placed in a combinatorial auction but not who placed each bid or even how many people placed bids, and then asked to determine the value function of every participant in the auction.

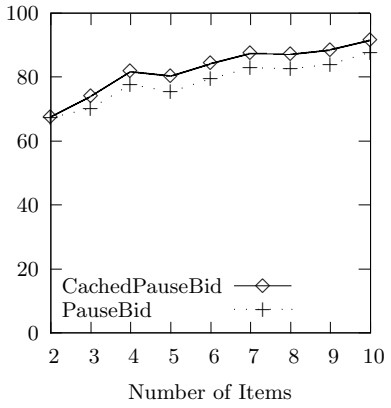

Figure 7: Average percentage of revenue from our algorithms relative to maximum revenue (*y-axis*) as function of the number of items in the auction.

functioning of our algorithms. We then compared the solutions found by both of them to the revenue-maximizing solution as found by CASS when given a set of bids that corresponds to the agents' true valuation. That is, for each agent i and each set of items S for which $v_i(S) > 0$ we generated a bid. This set of bids was fed to CASS which implements a centralized winner determination algorithm to find the solution which maximizes revenue. Note, however, that the revenue from the PAUSE auction on all the auctions is always smaller than the revenue of the revenue-maximizing solution when the agents bid their true valuations. Since PAUSE uses English auctions the final prices (roughly) represent the second-highest valuation, plus ϵ, for that set of items.

We fixed the number of agents to be 5 and we experimented with different number of items, namely from 2 to 10. We ran both algorithms 100 times for each combination. When we compared the solutions of our algorithms to the revenue-maximizing solution, we realized that they do not always find the same distribution of items as the revenue-maximizing solution (as shown in Figure 6). The cases where our algorithms failed to arrive at the distribution of the revenue-maximizing solution are those where there was a large gap between the first and second valuation for a set (or sets) of items. If the revenue-maximizing solution contains the bid (or bids) using these higher valuation then it is impossible for the PAUSE auction to find this solution because that bid (those bids) is never placed. For example, if agent i has $v_i(1) = 1000$ and the second highest valuation for (1) is only 10 then i only needs to place a bid of 11 in order to win that item. If the revenue-maximizing solution requires that 1 be sold for 1000 then that solution will never be found because that bid will never be placed. We also found that average percentage of times that our algorithms converges to the revenue-maximizing solution decreases as the number of items increases. For 2 items is almost 100% but decreases a little bit less than 1 percent as the items increase, so that this average percentage of convergence is around 90% for 10 items. In a few instances our algorithms find different solutions this is due to the different

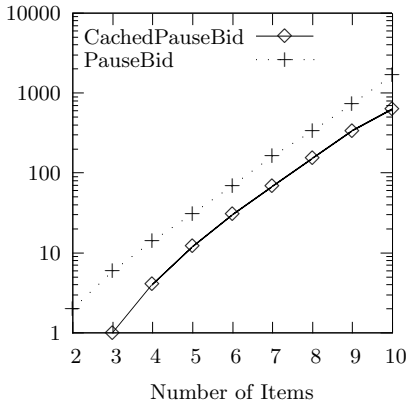

Figure 8: Average number of expanded nodes (*y-axis*) as function of items in the auction.

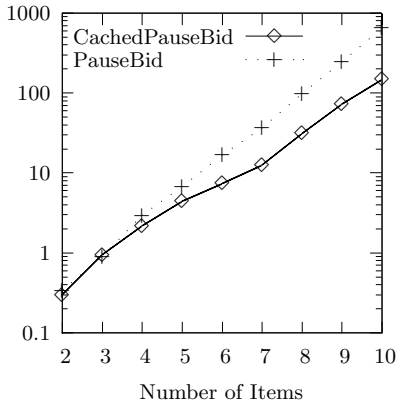

Figure 9: Average time in seconds that takes to finish an auction (*y-axis*) as function of the number of items in the auction.

ordering of the bids in the *bids* list which makes them search in different order.

We know that the revenue generated by the PAUSE auction is generally lower than the revenue of the revenue-maximizing solution, but how much lower? To answer this question we calculated percentage representing the proportion of the revenue given by our algorithms relative to the revenue given by CASS. We found that the percentage of revenue of our algorithms increases in average 2.7% as the number of items increases, as shown in Figure 7. However, we found that CACHEDPAUSEBID generates a higher revenue than PAUSEBID (4.3% higher in average) except for auctions with 2 items where both have about the same percentage. Again, this difference is produced by the order of the search. In the case of 2 items both algorithms produce in average a revenue proportion of 67.4%, while in the other extreme (10 items), CACHEDPAUSEBID produced in average a revenue proportion of 91.5% while PAUSEBID produced in average a revenue proportion of 87.7%.

The scalability of our algorithms can be determined by counting the number of nodes expanded in the search tree. For this we count the number of times that PBSEARCH gets invoked for each time that PAUSEBID is called and the number of times that FASTPAUSEBIDSEARCH gets invoked for each time that CACHEDPAUSEBID, respectively for each of our algorithms. As expected since this is an NP-Hard problem, the number of expanded nodes does grow exponentially with the number of items (as shown in Figure 8). However, we found that CACHEDPAUSEBID outperforms PAUSEBID, since it expands in average less than half the number of nodes. For example, the average number of nodes expanded when 2 items is zero for CACHEDPAUSEBID while for PAUSEBID is 2; and in the other extreme (10 items) CACHEDPAUSEBID expands in average only 633 nodes while PAUSEBID expands in average 1672 nodes, a difference of more than 1000 nodes. Although the number of nodes expanded by our algorithms increases as function of the number of items, the actual number of nodes is a much smaller than the worst-case scenario of n^n where n is the number of items. For example, for 10 items we expand slightly more than 10^3 nodes for the case of PAUSEBID and less than that for the case of CACHEDPAUSEBID which are much smaller numbers than 10^{10}. Notice also that our value generation algorithm (Figure 5) generates a number of bids that is exponential on the number of items, as might be expected in many situations. As such, these results do not support the conclusion that time grows exponentially with the number of items when the number of bids is independent of the number of items. We expect that both algorithms will grow exponentially as a function the number of *bids*, but stay roughly constant as the number of items grows.

We wanted to make sure that less expanded nodes does indeed correspond to faster execution, especially since our algorithms execute different operations. We thus ran the same experiment with all the agents in the same machine, an Intel Centrino 2.0 GHz laptop PC with 1 GB of RAM and a 7200 RMP 60 GB hard drive, and calculated the average time that takes to finish an auction for each algorithm. As shown in Figure 9, CACHEDPAUSEBID is faster than PAUSEBID, the difference in execution speed is even more clear as the number of items increases.

6. RELATED WORK

A lot of research has been done on various aspects of combinatorial auctions. We recommend [2] for a good review. However, the study of distributed winner determination algorithms for combinatorial auctions is still relatively new. One approach is given by the algorithms for distributing the winner determination problem in combinatorial auctions presented in [7], but these algorithms assume the computational entities are the items being sold and thus end up with a different type of distribution. The VSA algorithm [3] is another way of performing distributed winner determination in combinatorial auction but it assumes the bids themselves perform the computation. This algorithm also fails to converge to a solution for most cases. In [9] the authors present a distributed mechanism for calculating VCG payments in a mechanism design problem. Their mechanism roughly amounts to having each agent calculate the payments for two other agents and give these to a secure

central server which then checks to make sure results from all pairs agree, otherwise a re-calculation is ordered. This general idea, which they call the redundancy principle, could also be applied to our problem but it requires the existence of a secure center agent that everyone trusts. Another interesting approach is given in [8] where the bidding agents prioritize their bids, thus reducing the set of bids that the centralized winner determination algorithm must consider, making that problem easier. Finally, in the computation procuring clock auction [1] the agents are given an ever-increasing percentage of the surplus achieved by their proposed solution over the current best. As such, it assumes the agents are impartial computational entities, not the set of possible buyers as assumed by the PAUSE auction.

7. CONCLUSIONS

We believe that distributed solutions to the winner determination problem should be studied as they offer a better fit for some applications as when, for example, agents do not want to reveal their valuations to the auctioneer or when we wish to distribute the computational load among the bidders. The PAUSE auction is one of a few approaches to decentralize the winner determination problem in combinatorial auctions. With this auction, we can even envision completely eliminating the auctioneer and, instead, have every agent performe the task of the auctioneer. However, while PAUSE establishes the rules the bidders must obey, it does not tell us how the bidders should calculate their bids.

We have presented two algorithms, PAUSEBID and CACHEDPAUSEBID, that bidder agents can use to engage in a PAUSE auction. Both algorithms implement a myopic utility maximizing strategy that is guaranteed to find the bidset that maximizes the agent's utility given the set of outstanding best bids at any given time, without considering possible future bids. Both algorithms find, most of the time, the same distribution of items as the revenue-maximizing solution. The cases where our algorithms failed to arrive at that distribution are those where there was a large gap between the first and second valuation for a set (or sets) of items. As it is an NP-Hard problem, the running time of our algorithms remains exponential but it is significantly better than a full search. PAUSEBID performs a branch and bound search completely from scratch each time it is invoked. CACHEDPAUSEBID caches partial solutions and performs a branch and bound search only on the few portions affected by the changes on the bids between consecutive times. CACHEDPAUSEBID has a better performance since it explores fewer nodes (less than half) and it is faster. As expected the revenue generated by a PAUSE auction is lower than the revenue of a revenue-maximizing solution found by a centralized winner determination algorithm, however we found that CACHEDPAUSEBID generates in average 4.7% higher revenue than PAUSEBID. We also found that the revenue generated by our algorithms increases as function of the number of items in the auction.

Our algorithms have shown that it is feasible to implement the complex coordination constraints supported by combinatorial auctions without having to resort to a centralized winner determination algorithm. Moreover, because of the design of the PAUSE auction, the agents in the auction also have an incentive to perform the required computation. Our bidding algorithms can be used by any multiagent system that would use combinatorial auctions for coordination but would rather not implement a centralized auctioneer.

8. REFERENCES

[1] P. J. Brewer. Decentralized computation procurement and computational robustness in a smart market. *Economic Theory*, 13(1):41–92, January 1999.

[2] P. Cramton, Y. Shoham, and R. Steinberg, editors. *Combinatorial Auctions*. MIT Press, 2006.

[3] Y. Fujishima, K. Leyton-Brown, and Y. Shoham. Taming the computational complexity of combinatorial auctions: Optimal and approximate approaches. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 548–553. Morgan Kaufmann Publishers Inc., 1999.

[4] F. Kelly and R. Stenberg. A combinatorial auction with multiple winners for universal service. *Management Science*, 46(4):586–596, 2000.

[5] A. Land, S. Powell, and R. Steinberg. PAUSE: A computationally tractable combinatorial auction. In Cramton et al. [2], chapter 6, pages 139–157.

[6] K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the 2nd ACM conference on Electronic commerce*, pages 66–76. ACM Press, 2000. `http://cats.stanford.edu`.

[7] M. V. Narumanchi and J. M. Vidal. Algorithms for distributed winner determination in combinatorial auctions. In *LNAI volume of AMEC/TADA*. Springer, 2006.

[8] S. Park and M. H. Rothkopf. Auctions with endogenously determined allowable combinations. Technical report, Rutgets Center for Operations Research, January 2001. RRR 3-2001.

[9] D. C. Parkes and J. Shneidman. Distributed implementations of vickrey-clarke-groves auctions. In *Proceedings of the Third International Joint Conference on Autonomous Agents and MultiAgent Systems*, pages 261–268. ACM, 2004.

[10] M. H. Rothkopf, A. Pekec, and R. M. Harstad. Computationally manageable combinational auctions. *Management Science*, 44(8):1131–1147, 1998.

[11] T. Sandholm. An algorithm for winner determination in combinatorial auctions. *Artificial Intelligence*, 135(1-2):1–54, February 2002.

[12] T. Sandholm, S. Suri, A. Gilpin, and D. Levine. CABOB: a fast optimal algorithm for winner determination in combinatorial auctions. *Management Science*, 51(3):374–391, 2005.