Improved Algorithms for Optimal Winner Determination in Combinatorial Auctions and Generalizations

Tuomas Sandholm and Subhash Suri

Department of Computer Science
Washington University
St. Louis, MO 63130-4899
{sandholm,suri}@cs.wustl.edu

Abstract

Combinatorial auctions can be used to reach efficient resource and task allocations in multiagent systems where the items are complementary. Determining the winners is $\mathcal{NP}$-complete and inapproximable, but it was recently shown that optimal search algorithms do very well on average. This paper presents a more sophisticated search algorithm for optimal (and anytime) winner determination, including structural improvements that reduce search tree size, faster data structures, and optimizations at search nodes based on driving toward, identifying and solving tractable special cases. We also uncover a more general tractable special case, and design algorithms for solving it as well as for solving known tractable special cases substantially faster. We generalize combinatorial auctions to multiple units of each item, to reserve prices on singletons as well as combinations, and to combinatorial exchanges—all allowing for substitutability. Finally, we present algorithms for determining the winners in these generalizations.

Introduction

Auctions are important mechanisms for resource and task allocation in multiagent systems. In many auctions, a bidder's valuation for a combination of items is not the sum of the individual items' valuations—it can be more or less. This is often the case for example in electricity markets, equities trading, bandwidth auctions, transportation exchanges (Sandholm 1993), pollution right auctions, and auctions for airport landing slots (Rassenti, Smith, & Bulfin 1982).

In a traditional auction format where the items are auctioned separately (sequentially or in parallel), to decide what to bid on an item, an agent needs to estimate which other items it will receive in the other auctions, requiring intractable lookahead in the game tree. Even after lookahead, residual uncertainty would remain due to incomplete information about the other bidders. This leads to inefficient allocations where bidders do not get the combinations that they want and get combinations that they do not (Sandholm 1996).

Combinatorial auctions (CAs) can be used to overcome these deficiencies (Rassenti, Smith, & Bulfin 1982; Sandholm 1993). In a CA, bidders may submit bids on combinations of items. This allows the bidders to express complementarities between items instead of having to speculate into an item's valuation the impact of possibly getting other, complementary items.

Winner determination problem

The auctioneer has a set of items, $M = \{1, 2, \ldots, m\}$, to sell, and the buyers submit a set of bids, $\mathcal{B} = \{B_1, B_2, \ldots, B_n\}$. A bid is a tuple $B_j = \langle S_j, p_j \rangle$, where $S_j \subseteq M$ is a set of items and p_j is a price. Assume for now (this is relaxed later), that $p_j \geq 0$ for all $j \in \{1, 2, \ldots, n\}$. Assume also that no two bids concern the same set of items: $S_j \neq S_k$ (if multiple bids do concern the same set of items, all but the highest bid can be discarded as a preprocessing step, breaking ties arbitrarily). The *winner determination problem* is to label the bids as winning ($x_j = 1$) or losing ($x_j = 0$) so as to maximize the auctioneer's revenue under the constraint that each item can be allocated to at most one bidder:

$$\max \sum_{j=1}^{n} p_j x_j \quad \text{s.t.} \quad \sum_{j|i \in S_j} x_j \leq 1 \quad i = 1, 2, \ldots, m$$

$$x_j \in \{0, 1\}$$

This problem is intractable: it is equivalent to weighted set packing, a well-known $\mathcal{NP}$-complete problem. It can be solved via dynamic programming, but that takes $\Omega(2^m)$ and $O(3^m)$ time independent of n (Rothkopf, Pekeč, & Harstad 1998).

One approach is to solve the problem approximately (Rassenti, Smith, & Bulfin 1982; Fujishima, Leyton-Brown, & Shoham 1999; Lehmann, O'Callaghan, & Shoham 1999). However, it was recently shown (using the inapproximability of maximum clique) that no polynomial time algorithm can guarantee a solution that is close to optimum (Sandholm 1999). Certain special cases can be approximated slightly better, as reviewed in (Sandholm 1999).

The second approach is to restrict the allowable bids (Rothkopf, Pekeč, & Harstad 1998; Nisan 1999; Tennenholtz 2000). For certain restrictions, which are severe in the sense that only a vanishingly small fraction of the combinations can be bid on, winners can be determined in polynomial time. Restrictions on the bids give rise to the same economic inefficiencies that prevail in noncombinatorial auctions because bidders may not be able to bid on the combinations they prefer.

The third approach is to solve the unrestricted problem using search. This was shown to work very well on average, scaling optimal winner determination up to hundreds of items and thousands of bids depending on

the problem instance distribution (Sandholm 1999) and improvements to the algorithm have been developed since (Fujishima, Leyton-Brown, & Shoham 1999).

In the vein of the third approach, this paper presents a more sophisticated algorithm for optimal winner determination. The enhancements include structural improvements that reduce search tree size, faster data structures, and optimizations at search nodes based on driving toward, identifying and solving tractable special cases. We also uncover a more general tractable case, and design algorithms for solving it as well as for solving known tractable cases substantially faster. We generalize CAs to auctions with multiple units of each item, to auctions with reserve prices on singletons as well as combinations, and to combinatorial exchanges—all allowing for substitutability. We also give algorithms for determining the winners in these generalizations.

While we present our results for auctions where the auctioneer is the seller and the bidders are buyers, all of the results apply directly to reverse auctions where the auctioneer buys and the bidders sell.

A sophisticated search algorithm

In this section we present an algorithm for optimal winner determination. The improvements over previous algorithms are classified into structural improvements, capitalizing on tractable subproblems at nodes, and faster data structures.

Structural improvements

This section presents improvements that reduce search tree size by changing its structure.

Branching on bids (BOB) The skeleton of our algorithm is a depth-first branch-and-bound tree search that branches on bids. The set of bids that are labeled winning on the path to the current search node is called IN, and the set of bids that are winning in the best allocation found so far is IN^*. Let $\tilde{f}^*$ be the value of the best solution found so far. Initially, $IN = \emptyset$, $IN^* = \emptyset$, and $\tilde{f}^* = 0$. Each bid, B_j, has an exclusion count, e_j, that stores how many times B_j has been excluded by bids on the path. Initially $e_j = 0$ for all $j \in \{1, 2, \ldots, n\}$. M' is the set of items that are still unallocated, and g is the revenue from the bids with $x_j = 1$ on the search path so far. h is an upper bound on how much the unallocated items can contribute (let $\max\{\emptyset\} = 0$). The search is invoked by calling $BOB(M, 0)$.

Algorithm 1 $BOB(M', g)$

1. If $g > \tilde{f}^*$, then $IN^* \leftarrow IN$ and $\tilde{f}^* \leftarrow g$
2. $h \leftarrow \sum_{i \in M'} c(i)$, where $c(i) \leftarrow \max_{j | i \in S_j, e_j = 0} \frac{p_j}{|S_j|}$
3. If $g + h \leq \tilde{f}^*$, then return /* bounding */
4. Choose a bid B_k for which $e_k = 0$ /* branching */
   If no such bid exists, then return
5. $IN \leftarrow IN \cup \{B_k\}$, $e_k \leftarrow 1$

6. For all B_j such that $B_j \neq B_k$ and $S_j \cap S_k \neq \emptyset$,
   $e_j \leftarrow e_j + 1$
7. $BOB(M' - S_k, g + p_k)$
8. $IN \leftarrow IN - \{B_k\}$
9. For all B_j such that $B_j \neq B_k$ and $S_j \cap S_k \neq \emptyset$,
   $e_j \leftarrow e_j - 1$
10. $BOB(M', g)$
11. $e_k \leftarrow 0$, return

Both of the previous search algorithms for winner determination, *de facto*, branch on items (Sandholm 1999; Fujishima, Leyton-Brown, & Shoham 1999). The children of a search node are those bids that include the smallest item that is still unallocated, and do not share items with any bid on the path so far. If, as a preprocessing step, a dummy bid of price zero is submitted for every individual item that received no bids alone (to represent the fact that the auctioneer can keep items), then it was proven that the leaves of this tree correspond to feasible solutions to the winner determination problem (Sandholm 1999). Clearly, the branching factor is at most $n + m$ (the m comes from the dummy bids), and the depth is at most m, so the complexity is $O((n+m)^m)$.

On the other hand, BOB branches on bids (winning or losing, i.e., $x_j = 1$ or $x_j = 0$) instead of items. The branching factor is 2 and the depth is at most n, so a naive analysis shows that BOB is $O(2^n)$, which is exponential in bids. However, the nodes (both interior and leaf) correspond to feasible solutions to the winner determination problem. Therefore, the number of nodes in this tree is the same as the number of leaves in the old formulation. This proves that BOB is $O((n+m)^m)$, i.e., polynomial in bids. This is desirable since the auctioneer can usually control how many items are auctioned, but she cannot control how many bids are submitted. Furthermore, even though the complexity is exponential in items, this is only a worst-case bound and the average case tends to be significantly better. By contrast, the dynamic programming algorithm (Rothkopf, Pekeč, & Harstad 1998) is exponential in items even in the best case.

The main advantage of BOB compared to the earlier search formulation is that BOB is in line with the AI principle of least commitment. In BOB, the choice in step 4 only commits one bid, while in the old formulation the choice of an item committed all the remaining bids that include the item. BOB allows more refined search control—in particular, better bid ordering. Many of the techniques of this paper capitalize heavily on that possibility. A secondary advantage of BOB is that there is no need to use dummy bids.

Bid ordering heuristics (HEU) Search speed can be increased by improving the pruning that occurs in step 2. Our algorithm does this by constructing many high-revenue allocations early. We do this by bid ordering in step 4. We choose bids that contribute a lot to the revenue, and do not retract from the potential contribu-

tion of other bids by using up many items. At a search node, we choose a bid that maximizes $\frac{p_j}{|S_j|^\alpha}$ (to avoid scanning the list of bids repeatedly, the bids are sorted in descending order before the search begins) and has $e_j = 0$. Intuitively, $\alpha = 0$ gives too much preference to bids with many items, and $\alpha = 1$ gives too much preference to bids with few items. It was recently shown that in a greedy algorithm that simply inserts bids into IN^* in highest $\frac{p_j}{|S_j|^\alpha}$ first order (as a bid is inserted, bids that share items with it are discarded), $\alpha = \frac{1}{2}$ gives the best worst case bound over all α (Lehmann, O'Callaghan, & Shoham 1999) (but not necessarily over all possible bid ordering formulas). As in the greedy algorithm, we want to construct high-valued allocations. Unlike in the greedy algorithm, we also want to construct many allocations early to increase the chance of high-valued ones. Since bids with few items lead to deeper search than bids with many items (because bids with many items exclude more of the other bids due to overlap in items) (Sandholm 1999), preference for bids with many items increases the number of allocations seen early. Therefore, we set α slightly below $\frac{1}{2}$.

In addition to finding the optimal solution faster via more pruning, such bid ordering improves the algorithm's anytime performance: $\tilde{f}^*$ increases faster.

Lower bounding (LOW) We also prune using a lower bound, L, (obtained, e.g., using the greedy algorithm described above) at each node. If $g + L > \tilde{f}^*$, then $\tilde{f}^* \leftarrow g + L$ and IN^* is updated. This reduces search by enhanced pruning in the subtree rooted at the current search node.

Exploiting decomposition (DEC) If the set of items can be divided into subsets such that no bid includes items from more than one subset, the winner determination can be done in each subset separately. Because the search is superlinear in the size of the problem (both n and m), such decomposition leads to a speedup.

At every search node (between steps 1 and 2), our algorithm checks whether the problem has decomposed. We maintain a graph, G, where the vertices V are the bids with $e_j = 0$, and two vertices share an edge if the bids share items. Call the set of edges E. Clearly, $|V| \leq n$ and $|E| \leq \frac{n(n-1)}{2}$. Via an $O(|E| + |V|)$ depth-first-search in G, the algorithm checks whether the graph has decomposed. Every tree in the depth-first-forest corresponds to an independent subproblem (subset of bids and the associated subset of items). The winners are determined by calling BOB on each subproblem separately (bids not in that subproblem are marked $e_j \leftarrow 1$).[1]

Upper and lower bounding across subproblems (ACROSS) The straightforward approach is to call BOB on each subproblem with $g = 0$ and $\tilde{f}^* = 0$.

---

[1]This decomposition check was used as a preprocessor before (Sandholm 1999), not at every node.

Somewhat unintuitively, we can achieve further pruning, without compromising optimality, by exploiting information across the independent subproblems. Say there are k subproblems at the current search node θ: $1, \ldots, k$. Let g^θ be the g-value of θ before any of the subproblems have been solved. Let f_q^* be the value of the optimal solution found for subproblem q. Let h_q be the h-value of subproblem q. Let L_q be a lower bound (obtained, e.g., using the greedy algorithm described above, but even $L_q = 0$ works) for subproblem q.

Now, consider what to do to solve subproblem z after subproblems $1, \ldots, z - 1$ have been solved and the other subproblems have not. Let l_z be a lower bound (obtained, e.g., using the greedy algorithm described above) on the value that the *unallocated* items of subproblem z can contribute. Let g_z be the g-value within subproblem z only, and let h_z be the h-value within subproblem z only. Let

$$F_{solved}^* = g^\theta + \sum_{q=1}^{z-1} f_q^*$$

$$H_{unsolved} = \sum_{q=z+1}^{k} h_q$$

$$LO_{unsolved} = \sum_{q=z+1}^{k} L_q$$

At every search node within the subproblem z, we update the global lower bound $\tilde{f}^*$ as follows:

$$\tilde{f}^* \leftarrow \max\{\tilde{f}^*, \ F_{solved}^* + g_z + l_z + LO_{unsolved}\}$$

and we update IN^* accordingly.

Now we can cut the search path whenever

$$F_{solved}^* + g_z + h_z + H_{unsolved} \leq \tilde{f}^*$$

Since both the straightforward approach and this approach are correct, we use both. If either one allows the search path to be cut, the algorithm does so in step 3.

Due to the upper and lower bounding across subproblems, the order of tackling the subproblems makes a difference in speed, providing further opportunities for optimization via subproblem ordering.

Forcing a decomposition via articulation bids (ART) In addition to checking whether a decomposition has occurred, our algorithm strives for a decomposition. In the bid choice in step 4, we pick a bid that leads to a decomposition, if such a bid exists. Such bids whose deletion disconnects G are called *articulation bids*. Articulation bids can be identified during the depth-first-search of G in $O(|E| + |V|)$ time, as follows.

The depth-first-search assigns each node v of G a number $d(v)$, which is the order in which nodes of G are "discovered". The root has number 0. (See (Weiss 1999) for details.) In order to identify articulation bids, we assign to each node v one additional number, low(v), which is defined inductively:

$$x = \min\{low(w) \mid w \text{ is a child of } v\}$$
$$y = \min\{d(z) \mid (v, z) \text{ is a back edge}\}$$
$$low(v) = \min(x, y)$$

A node v is an articulation bid if and only if $\text{low}(v) \geq d(v)$. If there are multiple articulation bids, we branch on the one that minimizes the size of the larger subproblem, measured by the number of bids.

The strategy of branching on articulation bids may conflict with our price-based branching. Is one scheme necessarily dominant over the other? To answer this question, we define the two classes of schemes:

Definition. 1 *In an* articulation-based bid choosing scheme*, the next bid to branch on is an articulation bid if one exists. Ties can be resolved arbitrarily, as can cases where no articulation bid exists.*

Definition. 2 *In a* price-based bid choosing scheme*, the next bid to branch on is $B_k = \arg\max_{B_j \in \mathcal{B} | e_j = 0} \frac{p_k}{\nu(|S_k|)}$, for any given positive function ν. Ties can be resolved arbitrarily, e.g., preferring bids that articulate.*

Proposition 1 *For any given articulation-based bid choosing scheme and price-based bid choosing scheme, there are instances where the former leads to fewer search nodes, as well as instances where the latter leads to fewer search nodes.*[2]

Even if a bid is not an articulation bid, and would not lead to a decomposition if the bid is assigned losing, it might lead to a decomposition if it is assigned winning because that removes the bid's neighbors from G as well. This is yet another reason to assign a bid that we branch on to be winning before assigning it to be losing (value ordering). Also, in bid ordering (variable ordering), we can give first preference to articulation bids, second preference to these bids that articulate on the winning branch only, and third preference to bids that do not articulate on either branch (among them, the price-based bid ordering is used).

During the search, the algorithm could also do shallow lookaheads—for the purpose of bid ordering—to identify *combinations* of bids that would disconnect G. Such *cutsets* of bids can also be identified in a preprocessor, and then the bids within a small cutset should be branched on first in the search (however, identifying the smallest cutset is intractable).

Tractable subproblems at nodes

The following techniques, used at each search node, drive toward, identify and solve tractable special cases.

Avoiding branching on short bids Bids that include a small number of items lead to significantly deeper search than bids with many items because the latter exclude more of the other bids due to overlap in items. A previous search algorithm scaled to thousands of bids when bids had many items, and only hundreds of bids when bids had few items each (Sandholm 1999). We call bids with 1 or 2 items *short* and other

bids *long*.[3] Winners can be optimally determined in $O(n_{short}^3)$ worst case time using a weighted maximal matching algorithm (Edmonds 1965) if the problem has short bids only (Rothkopf, Pekeč, & Harstad 1998). To solve problems with both long and short bids efficiently, we integrate Edmond's algorithm with search. Our algorithm achieves optimality without ever branching on short bids. In step 4, bid choice is restricted to long bids. At every node, before step 1, Edmond's algorithm is executed using the short bids with $e_j = 0$. It returns a set of winning bids, IN_E, and the revenue they provide, f_E. The only remaining change is to step 1:

1. If $g + f_E > \tilde{f}^*$, then $IN^* \leftarrow IN \cup IN_E$, $\tilde{f}^* \leftarrow g + f_E$

Deleting items included in only one bid In the previous optimization, short bids are statically defined. We can improve on this by a more dynamic size determination. If an item x belongs to only one long bid b, then the size of b can be effectively reduced by one. This optimization may move some of the long bids into the short category, thereby further reducing search tree size. This optimization can be done at each search node, by keeping track of bids concerning each item.

Interval bids (Rothkopf, Pekeč, & Harstad 1998) considered an important special case where the items can be linearly ordered, and each bid concerns a contiguous interval of items. Specifically, assume that items are labeled $\{1, 2, \ldots, m\}$, and each bid b is for some interval $[i, j]$ of items. Using dynamic programming, Rothkopf et al. solved the problem in $O(m^2)$ time. We propose a different algorithm that solves this special case in $O(n + m)$ time. This asymptotic complexity is worst-case optimal because any algorithm may need to read all of the items and bids as input.

We briefly describe our algorithm here. Given a bid b on the interval $[f, l]$, let us call item f the *first item* of b, and item l the *last item* of b. We sort the bids in increasing order of their *last item*; if two bids have the same last item, the one with the smaller first item comes earlier in the sorted order. Since the set of items has bounded size $[1, m]$, we can bucket sort the bids in $O(n + m)$ time. Our dynamic program computes optimal solutions for the prefix intervals of the form $[1, i]$, for $i = 1, 2, \ldots, n$. Let $\text{opt}(i)$ denote the optimal solution for the problem considering only those bids that contain items in the range $[1, i]$; that is, bids whose last item is no later than i. Initially, $\text{opt}(0) = 0$. Let C_i denote the set of bids whose last item is i. Then, we have the following recurrence:

$$\text{opt}(i) = \max_{b \in C_i} \{p_b + \text{opt}(f_b - 1), \ \text{opt}(i-1)\},$$

where p_b is the price of bid b, and f_b is the smallest indexed item in b. The maximization has two terms.

---

[2]Proofs are omitted in this version due to limited space.

[3]We define *short* in this way because the problem is $\mathcal{NP}$-complete already if 3 items per bid are allowed (Rothkopf, Pekeč, & Harstad 1998).

The first term corresponds to accepting bid b, in which case we need an optimal solution for the subproblem $[1, f_b - 1]$. The second term corresponds to not accepting b, in which case we use the optimal allocation for items in $[1, i - 1]$. By solving these problems in increasing order of i, we can compute each $\text{opt}(i)$ in time proportional to the size of C_i. Since $\sum C_i = n$, the total time complexity is $O(n + m)$. The optimal allocation is $\text{opt}(m)$.

Proposition 2 *If all n bids are interval bids in a linearly ordered set of items $[1, m]$, then an optimal allocation can be computed in worst-case time $O(n + m)$.*

If we allow interval wraparound bids (e.g, $S_j = \{m - 1, m, 1, 2, 3\}$), the winners can be determined optimally in $O(m(n + m))$ time by cutting the circle of items in each of the m possible positions separately, and solving the associated problem (ignoring bids that span over the cutting position) using our algorithm described above. The fastest prior algorithm for this case took $O(m^3)$ time (Rothkopf, Pekeč, & Harstad 1998).

Identifying linear ordering Our interest is not to limit the auctions to interval bids only, but rather to recognize whether the remaining problem at any search node falls under this special case and to solve it by our specialized fast algorithm. This requires an algorithm to check whether *there exists some linear ordering of items* for which the given set of bids are all interval bids. It turns out this problem can be phrased as the *interval graph recognition* problem, for which a linear-time solution exists.

Given a graph $G = (V, E)$, we say that G is an *interval graph* if the vertices V can be put in one-to-one correspondence with intervals of the real line such that two intervals overlap if and only if there is an edge between the vertices corresponding to those intervals. The interval graph recognition problem is to decide whether G is an interval graph, and to also construct the intervals. The algorithm in (Korte & Mohring 1989) solves this problem in $O(|V| + |E|)$ time. Given the intervals for the bids, one can easily produce a linear ordering of the items. Figure 1 shows an example with 4 bids: $A = (2, 4, 6)$, $B = (1, 2, 4, 5, 7)$, $C = (1, 3, 7, 8)$, $D = (1, 3, 5, 7)$.
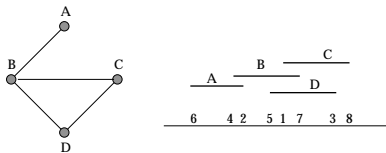

Figure 1: *Bid graph and a valid linear ordering.*

The case where wraparound bids are allowed can be identified in $O(n^2)$ time using an algorithm for recognizing whether the remaining graph G is *circular* (Eschen & Spinrad 1993).

Subgraph bids on tree-structured items We now propose a fast algorithm for another case that subsumes and substantially generalizes the interval bid model

of (Rothkopf, Pekeč, & Harstad 1998). The items are structured in a tree T, and a valid bid corresponds to a *connected subgraph* of T (see Figure 2). This is a strict generalization of the linear ordering model, which corresponds to the special case where T is a path. Our tree model is also distinct from the "nested structure" model in (Rothkopf, Pekeč, & Harstad 1998), where the tree nodes corresponds to bids.
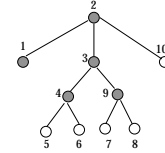

Figure 2: *An example of a subgraph bid:* $\{1, 2, 3, 4, 9\}$.

An example application where this special structure prevails is the following web shopping scenario. The goods are structured in a tree, where a web page contains the description of a good and links to children goods. For example, the page of a tent could have links to a heater, camping stove, and bug spray. The stove could have links to fuel refills and pots, etc. On any page, the user can 1. buy the item and be allowed to continue to any number of the children goods, or 2. not buy the item and backtrack, or 3. submit the bid by specifying a price for the subgraph that the user has chosen so far, or 4. exit without submitting the bid.

We developed an $O(nm)$ algorithm for solving the winner determination problem with subgraph bids on tree-structured items. We pick an arbitrary node r as the root of the item tree T. We assign each node of T a *level*, which is its distance from the root. The *level* of a bid b, denoted $\text{level}(b)$, is the smallest level of any item in b. We sort the bids in increasing order of level, breaking ties arbitrarily. We use a dynamic program to compute the optimal solutions at nodes of T in decreasing order of level.

Given a node i of T, let C_i denote the set of bids that include i and whose level is the same as the level of i. Our algorithm computes the function $\text{opt}(i)$, for each node i, where $\text{opt}(i)$ is the optimal solution for the problem considering only those bids that contain items in the subtree below i. Our goal is to compute $\text{opt}(r)$.

Consider a bid b, and suppose that the item giving b its level is x. Removing all items of b disconnects the tree rooted at x, namely T_x, into several subtrees. Let U_b be the set of roots of this forest of subtrees. Now,

$$\text{opt}(i) = \max\left\{ \max_{b \in C_i}\left\{ p_b + \sum_{j \in U_b} \text{opt}(j) \right\}, \sum_{j \in \text{children}(i)} \text{opt}(j) \right\}$$

where p_b is the price of bid b. By proceeding bottom up, we compute $\text{opt}(i)$ for all nodes of the tree.

Proposition 3 *The recurrence above correctly computes the optimal solution for subtree bids in tree-structured items in $O(nm)$ worst-case time.*

Subtree bids in DAGs Our special case of subtree bids on tree-structured item is sharp in the sense that a slight generalization makes the problem intractable:

Proposition 4 *If the set of items is structured as a directed acyclic graph D, and each bid is a subtree of D, then winner determination is $\mathcal{NP}$-complete.*

Faster data structures

Bid graph representation (GRA) We use an adjacency list representation of the bid graph G for efficient insert and delete operations on bids. We do not actually keep track of exclusion counts e_j. Instead, a bid j having been deleted corresponds to $e_j > 0$, and a bid j not having been deleted corresponds to $e_j = 0$. We use an array to store the nodes of G. The array entry for node j points to a doubly-linked list of bids that share items with j. Thus, an edge (j, k) creates two entries: one for j in the list of k, and the other for k in the list of j. We use cross-pointers with these entries to be able to access one from the other in $O(1)$ time. To delete node j whose current neighbor list is $\{b_1, b_2, \ldots, b_k\}$, we mark the node j "deleted" in the node array. Then, we use the linked list of j to access the position of j in each of the b_i's list, and delete that entry, at $O(1)$ cost each. When reinserting a node j with edges E_j into G, node j's "deleted" label is first removed in the node array. Then, for each $(j, k) \in E_j$, j is inserted at the front of k's neighbor list, k is inserted at the front of j's neighbor list, and the cross-pointer is set between them, all at $O(1)$ cost.

As BOB branches by $x_j = 1$, j and its neighbors in G are deleted. We also store in the search node a list of the edges that were in effect removed: the edges E' that include j, and the edges E'' that include j's neighbors but not j. When backtracking to that node, we reinsert j's neighbors into G using the edges E''. Then BOB branches by $x_j = 0$. When backtracking from that branch, j is reinserted into G using edges E'.

Maintaining the heuristic function (MAI) Our heuristic function, h, is the same as in an earlier winner determination algorithm (Sandholm 1999). In that implementation it took $O(mn)$ time per search node to compute. A faster but rougher approximation of the same heuristic was used in (Fujishima, Leyton-Brown, & Shoham 1999). Here we propose data structures that allow us to compute h fast and exactly.

We store the items in a dynamic list which supports insert and delete in $O(\log m)$ time each. Each item i points to a *heap* $H(i)$ that maintains the bids that include i. The heap supports find-max, extract-max, insert, and delete in $O(\log m)$ time each (delete requires a pointer to the item being deleted, which we maintain).

The heuristic function requires us to compute, for each item i, the maximum value $\frac{p_j}{|S_j|}$ among the bids that have not been deleted and concern item i. We keep a tally of the current heuristic function and update it each time a bid gets deleted or reinserted into G. Consider the deletion of bid j that has k items; each

item points to its position in the item list. We delete j's entry from the heap of each of these k items. For each of these k items, we update the heuristic function, by calculating the difference in its c value before and after the update. When j is reinserted, we reinsert j into the heaps of all the items that concern j. The cost, per search node, of updating the heuristic function is $\sum_j |S_j| \log m$, where the summation is over all the bids that got deleted or reinserted.

As a further optimization, our algorithm uses a *leftist heap* for $H(i)$ (Weiss 1999). A leftist heap has the same worst-case performance as an ordinary heap, but improves the amortized complexity of insert and delete to $O(1)$, while extract-max and find-max remain $O(\log m)$. Because the insert and delete operations in BOB are quite frequent, this improves the overall performance.

Preprocessing

Four preprocessing techniques were recently proposed for the winner determination problem (Sandholm 1999). Each one of them can be directly used in conjunction with our algorithm.

Generalizations of CAs

This section discusses generalizations of CAs. Our auction server prototype (`http://ecommerce.cs.wustl.edu/emediator`) supports all of these generalizations separately and combined (Sandholm 2000), and has been in continuous operation since December 1998.

Multiple units of each item

In some auctions, there are multiple indistinguishable units of each item for sale. One can compress the bids and speed up winner determination by not treating every unit as a separate item, since the bidders do not care which units of each item they get, only how many. We define a bid in this setting as $B_j = \langle (\lambda_j^1, \lambda_j^2, \ldots \lambda_j^m), p_j \rangle$, where $\lambda_j^k \geq 0$ is the requested number of units of item k, and p_j is the price. The winner determination problem is:

$$\max \sum_{j=1}^{n} p_j x_j \quad \text{s.t.} \quad \sum_{j=1}^{n} \lambda_j^i x_j \leq u_i \quad i = 1, 2, \ldots, m$$
$$x_j \in \{0, 1\}$$

where u_i is the number of units of item i for sale. In our basic CA, and in every one of the generalizations, if *free disposal* is not possible, we use an equality constraint in place of the inequality.

Previous winner determination algorithms cannot be used in the multi-unit setting because they branch on items (Sandholm 1999; Fujishima, Leyton-Brown, & Shoham 1999). Even if each unit is treated as a separate item, the earlier algorithms cannot be used if the demands, λ_j^k, are real-valued instead of integer.

BOB can be used. A tally of the number of units allocated on the search path is kept for each item: $\Lambda_i = \sum_{j|x_j=1} \lambda_j^i$.

The decomposition techniques DEC and ART apply on the bid graph G where two vertices, j and k, now

share an edge if $\exists i$ s.t. $\lambda_j^i > 0$ and $\lambda_k^i > 0$. However, once a bid is assigned winning and removed from G, the neighbor bids in G cannot always be removed unlike in the basic CA. Instead, only those neighbors, j, are removed that demand more units of some item than remain ($\exists$ item k such that $\lambda_j^k > u_k - \Lambda_k^k$). The removed bids are reinserted into G when backtracking. The data structure improvements GRA and MAI apply with this change.

One admissible heuristic for this setting is

$$h = \sum_{i \in M} [(u_i - \Lambda_i) \max_{j \in V_G | \lambda_j^i > 0} \frac{p_j}{\sum_{i \in S_j} \lambda_j^i}]$$

where V_G is the set of bids that remain in G. More refined heuristics can be constructed by giving different items different weights. Once $g + h \le \tilde{f}^*$, the search path is cut. The lower bounding technique LOW also applies, as do upper and lower bounding across subproblems (ACROSS).

Bid ordering can be used, e.g., by presorting the bids in descending order of $\frac{p_j}{(\sum_{i=1}^m \lambda_j^i)^\alpha}$.

Combinatorial exchanges

In a combinatorial exchange, both buyers and sellers can submit combinatorial bids (Sandholm 2000). Bids are as in the multi-unit case, except that the λ_j^i values can be negative, as can the prices p_j, representing selling instead of buying. The winner determination problem is to maximize surplus:[4]

$$\max \sum_{j=1}^n p_j x_j \quad \text{s.t.} \quad \sum_{j=1}^n \lambda_j^i x_j \le 0 \qquad i = 1, 2, \dots, m$$
$$x_j \in \{0, 1\}$$

Unlike earlier algorithms that branch on items (Sandholm 1999; Fujishima, Leyton-Brown, & Shoham 1999), BOB can be used in this setting. In the basic CA and in our other generalizations, the optimal solution occurs in a leaf. In contrast, in our combinatorial exchange, the optimal solution can occur even in an interior node of the search tree. In the search, a tally of the net number of units demanded (units supplied are negative numbers) on the path is kept for each item: $\Lambda_i = \sum_{j | x_j = 1} \lambda_j^i$.

The decomposition techniques DEC and ART apply on bid graph G where two vertices, j and k, share an edge if $\exists$ item i such that $\lambda_j^i \ne 0$ and $\lambda_k^i \ne 0$. However, once a bid is assigned winning and removed from G, the neighbor bids in G cannot always be removed unlike in the basic CA. Instead, only those neighbors, j, are removed that cannot possibly be matched any more:

---

[4]If the exchange charges based on transaction volume, as most current exchanges do, it may want to maximize volume instead: $\max \sum_{j \in \{1, \dots, n\} | p_j > 0} p_j x_j$ with the same constraints. Our algorithms apply to this case as they do to surplus maximization. However, we advocate surplus maximization since that maximizes social welfare (assuming that bidders are truthful).

- $\exists$ item i s.t. $\lambda_j^i > 0$ and $\lambda_j^i + \Lambda_i + \sum_{k \in V_G | \lambda_k^i < 0} > 0$, or

- $\exists$ item i s.t. $\lambda_j^i < 0$ and $\lambda_j^i + \Lambda_i + \sum_{k \in V_G | \lambda_k^i > 0} < 0$,

where V_G is the set of remaining bids in G. The removed bids are reinserted into G when backtracking. The data structure improvements GRA and MAI apply with this modification.

The upper bounding and lower bounding (LOW) techniques discussed earlier in the paper can be used after constructing functions that compute an upper bound h and a lower bound L. Then, also the upper bounding and lower bounding techniques across subproblems (ACROSS) apply.

Bid ordering can also be used. For example, by branching on a bid, j, that maximizes p_j (the bids can be sorted in this order as a preprocessing step to avoid sorting during search), the algorithm can strive to high-surplus allocations early, leading to enhanced pruning. As another example, by branching on a bid, j, that minimizes $\sum_{i | \Lambda_i > 0} \Lambda_i + \lambda_i^j$, or a bid that minimizes $\max_{i | \Lambda_i > 0} \Lambda_i + \lambda_i^j$, the algorithm can reach feasible solutions faster (especially in the case of free disposal), leading again to enhanced pruning later.

Additional pruning is achieved by branching on bids with $p_j < 0$ first, and then on bids with $p_j \ge 0$.[5] Once $\sum_{j | x_j = 1} p_j > 0$, that branch of the search is cut.[6] Also, after the switch to bids with $p_j \ge 0$ has occurred on a path, h, LOW, ACROSS, and bid ordering from the multi-unit case can be used among the remaining bids to achieve further pruning.

Reserve prices

In some auctions, the seller has a reserve price r_i for every item i, below which she is not willing to sell. This could be easily incorporated into our algorithm by adding a constraint: the revenue collected from the bids is no less than the sum of the reserve prices of the items that are allocated to bidders. A stricter way of interpreting reserve prices as a constraint is to require that the auctioneer's payoff (revenue collected from the bidders plus reserve price of the items kept) would not increase by keeping an additional item or by allocating an additional item to one of the bidders. This could also be easily incorporated into our algorithm.

However, this raises the concern that the auctioneer's payoff might increase by keeping or allocating a *set* of items. It turns out that requiring that it does not coincides with maximizing social welfare (sum of the auctioneer's payoff plus the bidder's payoffs; each bidder's payoff is her valuation for the bundle of goods that she gets minus what she has to pay), assuming

---

[5]Alternatively one can branch on bids with $p_j > 0$ first, and reverse the tests respectively.

[6]Alternatively one can do this split of bids into two sets ($\lambda_i^j < 0$ vs. $\lambda_i^j \ge 0$) and cutting (when $\Lambda_i > 0$) on any item i instead of price.

that bidders enter their true valuations and the auctioneer enters his true reserve prices. This is done not as a constraint, but by changing the maximization criterion to max $\sum_{j=1}^{n} (p_j - \sum_{i \in S_j} r_i)x_j$. This is trivial to incorporate into our algorithm: the item's reserve prices are simply subtracted from the bid prices as a preprocessing step.

This method can also be used for exchanges where only one side (buyers or sellers) is allowed to place combinatorial bids. The other side has to bid noncombinatorially. The bids of the noncombinatorial side are considered reserve prices, allowing the fast winner determination algorithm for one-to-many CAs to be used in many-to-many exchanges for optimal clearing.

Auctions where the seller is allowed to submit reserve prices on combinations of items or is allowed to express substitutability in the reserve prices, cannot be handled by the one-to-many algorithm. Instead, they are treated as exchanges where the seller's reserve prices are her bids. Our algorithm for combinatorial exchanges is then used for optimally clearing the market.

Substitutability

In the auctions discussed so far in the paper, bidders can express superadditive preferences: the value of a combination is greater or equal to the sum of the values of its parts. They cannot express subadditive preferences, aka. substitutability. For example, by bidding \$5 for $\{1, 2\}$, \$3 for $\{1\}$, and \$4 for $\{2\}$, the bidder may get $\{1, 2\}$ for \$7. Two solutions have been proposed that allow any preferences to be expressed. They extend directly to all the generalized CAs presented in this paper: the multi-unit case, the exchange, and the case of reserve prices. In the first, bidders can combine their bids with XORs, potentially joined by ORs (Sandholm 2000; 1999). The second uses dummy items (Fujishima, Leyton-Brown, & Shoham 1999). If two bids share a dummy item, they cannot be in the same allocation.

BOB can be used with the first method by adding edges in G for every pair of bids that is combined with XOR. These additional constraints actually speed up the search. However, only some of the optimization apply: HEU, LOW, DEC, ART, ACROSS, GRA, and MAI. BOB supports the second method directly and all of the optimization apply. Unfortunately, certain preferences require exponentially many dummy items to express.

Acknowledgments

Conclusions

Combinatorial auctions can be used to reach efficient resource and task allocations in multiagent systems where the items are complementary. Determining the winners is $\mathcal{NP}$-complete and inapproximable, but it was recently shown that optimal search algorithms do very well on average. This paper presented a more sophisticated search algorithm for optimal (and anytime) winner determination, including structural improvements that reduce search tree size, faster data structures, and optimizations at search nodes based on driving toward, identifying and solving tractable special cases. We also discovered a more general tractable special case, and designed algorithms for solving it as well as for solving known tractable special cases substantially faster. We generalized combinatorial auctions to multiple units of each item, to reserve prices on singletons as well as combinations, and to combinatorial exchanges—all allowing for substitutability. Finally, we developed algorithms for determining the winners in these generalizations.

References

Edmonds, J. 1965. Maximum matching and a polyhedron with 0,1 vertices. *J. Res. Nat. Bur. Standards* B(69):125–130.

Eschen, E. M., and Spinrad, J. 1993. An $O(n^2)$ algorithm for circular-arc graph recognition. In *SIAM-ACM Sym. on Discrete Algorithms (SODA)*, 128-137.

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

Korte, N., and Mohring, R. H. 1989. An incremental linear-time algorithm for recognizing interval graphs. *SIAM Journal on Computing* 18(1):68–81.

Lehmann, D.; O'Callaghan, L. I.; and Shoham, Y. 1999. Truth revelation in rapid, approximately efficient combinatorial auctions. In *ACM Conference on Electronic Commerce (ACM-EC)*, 96–102.

Nisan, N. 1999. Bidding and allocation in combinatorial auctions: Preliminary version. Hebrew U., Sept.

Rassenti, S. J.; Smith, V. L.; and Bulfin, R. L. 1982. A combinatorial auction mechanism for airport time slot allocation. *Bell J. of Economics* 13:402–417.

Rothkopf, M. H.; Pekeč, A.; and Harstad, R. M. 1998. Computationally manageable combinatorial auctions. *Management Science* 44(8):1131–1147.

Sandholm, T. W. 1993. An implementation of the contract net protocol based on marginal cost calculations. In *AAAI*, 256–262.

Sandholm, T. W. 1996. Limitations of the Vickrey auction in computational multiagent systems. In *IC-MAS*, 299–306.

Sandholm, T. W. 1999. An algorithm for optimal winner determination in combinatorial auctions. In *IJCAI*, 542–547. Extended version first appeared as Washington U., Comp. Sci. WUCS-99-01, Jan. 28th.

Sandholm, T. W. 2000. eMediator: A next generation electronic commerce server. In *AGENTS*. Early version: AAAI-99 Workshop on AI in Electronic Commerce, Orlando, FL, pp. 46–55, July 1999.

Tennenholtz, M. 2000. Some tractable combinatorial auctions (preliminary report). Draft. Technion, Israel.

Weiss, M. A. 1999. *Data structures and algorithm analysis in C++*. Addison-Wesley, 2nd edition.