ELSEVIER

Available at
www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

Artificial Intelligence 145 (2003) 33–58

Artificial Intelligence

www.elsevier.com/locate/artint

BOB: Improved winner determination in combinatorial auctions and generalizations ☆

Tuomas Sandholm [a,*], Subhash Suri [b]

[a] Computer Science Department, Carnegie Mellon University, Pittsburgh, PA 15213, USA
[b] Computer Science Department, University of California, Santa Barbara, CA 93106, USA

Received 15 November 2001

Abstract

Combinatorial auctions can be used to reach efficient resource and task allocations in multiagent systems where the items are complementary or substitutable. 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 of these generalizations support both complementarity and substitutability of the items. Finally, we present algorithms for determining the winners in these generalizations.
© 2003 Elsevier Science B.V. All rights reserved.

Keywords: Auction; Combinatorial auction; Multi-item auction; Multi-object auction; Bidding with synergies; Winner determination; Multiagent systems

1. 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

☆ A shorter early version of this paper appeared in the *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, Austin, TX, July 30–August 3, 2000, pp. 90–97.

* Corresponding author.
E-mail addresses: sandholm@cs.cmu.edu (T. Sandholm), suri@cs.ucsb.edu (S. Suri).

0004-3702/03/$ – see front matter © 2003 Elsevier Science B.V. All rights reserved.
doi:10.1016/S0004-3702(03)00015-8

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 [14,15], transportation exchanges [21,22], pollution right auctions, auctions for airport landing slots [18], and auctions for carrier-of-last-resort responsibilities for universal services [10].

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 [2,22].

Combinatorial auctions can be used to overcome these deficiencies [3,15,18,21]. In a combinatorial auction, 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.

2. 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 nonempty set of items and p_j is the price offer for this set. Assume for now (this is relaxed later), that $p_j \geqslant 0$ for all $j \in \{1, 2, \ldots, n\}$. Assume also that each bid is on a distinct set of items: if multiple bids concern the same set of items, all but the highest bid can be discarded during 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:

$$\begin{aligned} \max \quad & \sum_{j=1}^{n} p_j x_j \\ \text{s.t.} \quad & \sum_{j \mid i \in S_j} x_j \leqslant 1, \quad i = 1, 2, \ldots, m, \\ & x_j \in \{0, 1\}. \end{aligned}$$

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 the number of bids, n [19].

One approach is to solve the problem approximately [6,9,12,18]. However, it was recently shown (via a reduction from the maximum clique problem which is inapproximable [8]) that no polynomial time algorithm for the winner determination problem can guarantee a solution that is close to optimum [23]. Certain special cases can be approximated slightly better, as reviewed in [23].

The second approach is to restrict the allowable bids [17,19,27]. 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 [23] and improvements to the algorithm have been developed since [6].

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 combinatorial auctions 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.

3. 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.

3.1. Structural improvements

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

3.1.1. 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. $BOB(M', g)$

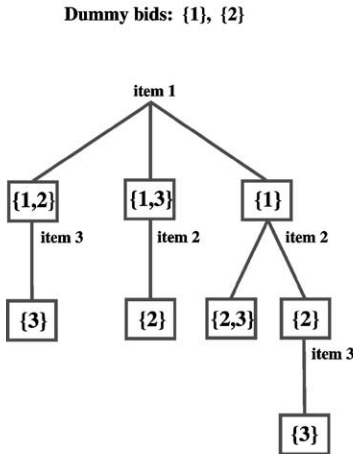
1. If $g > \tilde{f}^*$, then $IN^* \leftarrow IN$ and $\tilde{f}^* \leftarrow g$
2. $h \leftarrow \sum_{i \in M'} c(i)$, where $c(i) \leftarrow \max_{j | i \in S_j, e_j = 0} p_j / |S_j|$ (Any admissible heuristic could be used here)
3. If $g + h \leqslant \tilde{f}^*$, then return **/* Bounding */**

4. Choose a bid B_k for which $e_k = 0$ **/* Choose a bid to branch on */**
   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. $BOB(M' - S_k, g + p_k)$ **/* Branch B_k in */**
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. $BOB(M', g)$ **/* Branch B_k out */**
11. $e_k \leftarrow 0$, return

Both of the previous search algorithms for winner determination, *de facto*, branch on items [6,23]. An example is shown in Fig. 1(left). The children of a search node are those bids that (1) include the lowest-numbered item that is still unallocated, and (2) do not share items with any bid on the path so far. In the branch-on-items formulation, as a preprocessing step, a dummy bid of price zero is submitted on every individual item that received no bids alone (to represent the fact that the auctioneer can keep items) [23]. It is important that dummy bids are used because if they are not, the optimal solution to the winner determination problem might not be represented in the tree. For example

Bids in this example (only items of each bid are shown; prices are not shown):
{1,2}, {2,3}, {3}, {1,3}

Branch-on-items formulation **Branch-on-bids formulation**



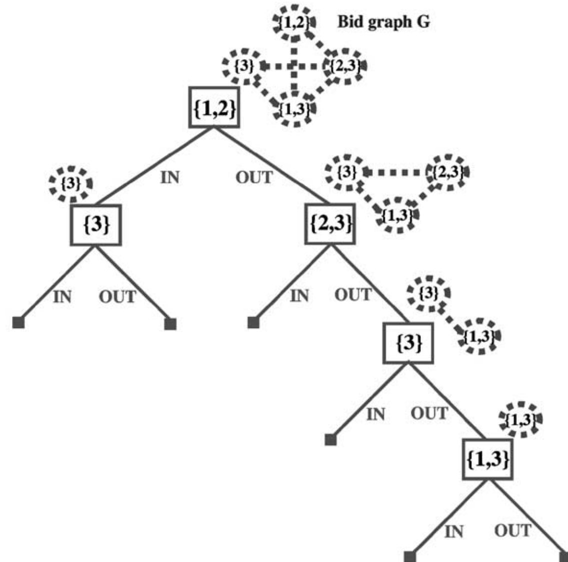


Fig. 1. Branching on items vs. branching on bids.

in Fig. 1(left), if a dummy bid on item 1 were not used, the possibility of the auctioneer keeping item 1 would not be explored (the right-most subtree at the root would be ignored). If dummy bids are used, all optimal solutions will be part of the tree [23]. Let us call the number of dummy bids n_{dummy}. Because there are m items and each bid uses at least one item, the depth of the branch-on-items tree is at most m. Because there are $n + n_{\text{dummy}}$ bids, the branching factor is at most $n + n_{\text{dummy}}$. So, it is trivial to see that the number of leaves is no greater than $(n + n_{\text{dummy}})^m$. However, a tighter upper bound, $((n + n_{\text{dummy}})/m)^m$, on the number of leaves has been established [23]. Because $n_{\text{dummy}} \leqslant m$, the number of leaves in the branch-on-items tree is no greater than $(n/m + 1)^m$.

Unlike the earlier algorithms for winner determination that branched on items, BOB branches on bids (winning or losing, that is, $x_j = 1$ or $x_j = 0$), see Fig. 1(right). So, each interior node of the search tree has two children: the world where the bid chosen at that node is labeled winning, and the world where that bid is labeled losing. The branching factor is 2 and the depth is at most n (the depth of the left branch is at most $\min\{m, n\}$, but the right branch can have depth up to n). No dummy bids are used in the branch-on-bids formulation; the items that are not allocated in bids on the search path are kept by the auctioneer.

Given the branching factor and tree depth, a naive analysis shows that the number of leaves in a branch-on-bids tree is at most 2^n, which is exponential in bids. This is perhaps the reason why this search formulation has not been used for the winner determination problem before. However, with a deeper analysis we establish a significantly tighter worst-case upper bound on the size of the tree:

Theorem 1. *Let κ be the number of items in the bid with the smallest number of items. The number of leaves in the branch-on-bids tree is no greater than*

$$\left(\frac{n}{\lfloor m/\kappa \rfloor} + 1\right)^{\lfloor m/\kappa \rfloor}.$$

The number of nodes in the tree is $2 \cdot \#\textit{leaves} - 1$.

Proof. There is a 1-to-1 correspondence between leaves of the branch-on-bids tree and feasible solutions to the winner determination problem. Therefore, the number of leaves is the same as the number of feasible solutions. We formulate an upper bound on the number of feasible solutions as a problem where, given n, m, and κ, an adversary constructs a problem instance (a set of bids) so as to maximize the number of feasible solutions.

The first key observation is that for each item, only one bid that includes that item can be accepted. So, the first item can be given to at most n alternative bids (or to no bid). Then the second item can be given to at most n alternative bids (or to no bid), and so on. This already gives an upper bound $\prod_{i=1}^{m} (n + 1) = (n + 1)^m$ on the number of solutions. However, we can tighten the upper bound by not counting the same bid multiple times (once for each item), but rather counting each bid once (by associating it with one item that it includes).

We say that an item i has a lower index than item j if $i < j$. Let us denote by N_i the set of bids that include item i as the lowest-indexed item. Let $n_i = |N_i|$. Each bid includes at least one item and the items within a bid are distinct. Therefore each bid has exactly one lowest-indexed item. Thus we have $\sum_{i=1}^{m} n_i = n$.

Now, one can consider constructing the feasible solutions as leaves of a different kind of tree T where the root is at level 1, and at any node at level i, the children correspond to those bids in N_i whose items have not been used by other bids on the path yet, plus an extra child for the possibility that in the feasible solution, the item is not used by any bid.[1] Clearly, the branching factor at level i is at most $(n_i + 1)$. So, $\prod_{i=1}^{m}(n_i + 1)$ is an upper bound on the number of solutions.

Next we incorporate the knowledge that each bid includes at least κ items. This means that at most $\lfloor m/\kappa \rfloor$ bids can be included in a feasible solution. In other words, on any path from the root to a leaf in T, at most $\lfloor m/\kappa \rfloor$ nodes can have a branching factor other than 1 (which represents the fact that there is no choice of bid at that node). We call those nodes *branching nodes*, and the nodes with branching factor 1 *non-branching nodes*.

The adversary (who implicitly chooses T by choosing the bids) would have done at least as well by choosing a tree where on every path from the root, branching nodes are first, and non-branching nodes are last. This is because the branching factor at any level i is at most $(n_i + 1)$ and $\sum_{i=1}^{m} n_i = n$. In this setting, if the adversary "spends" part of n toward n_i at some level i, he achieves the largest number of leaves in the tree by branching as much as possible (branching factor at most $(n_i + 1)$) at *every* node at that level. So, if there is some branching node at level i, then all the nodes at level i should be branching nodes.[2] Therefore, for any given level, all the nodes are either branching or non-branching. If there is some level of non-branching nodes closer to the root than a level of branching nodes, then we can swap (without changing the number of leaves) these levels. This is because a level of non-branching nodes does not change the number of leaves. Now, we can keep applying such swaps until all the levels with branching nodes are adjacent to the root and the levels with non-branching nodes are adjacent to the leaves.

So, from now on we can focus on the case where the adversary picked a tree T' where the branching nodes are adjacent to the root, and the non-branching nodes are adjacent to the fringe. In fact, we can ignore the entire non-branching part of the tree since it does not affect the number of leaves. So, what we have is a tree T'' of depth $\lfloor m/\kappa \rfloor$ where the branching factor at level i is at most $(n_i + 1)$ and $\sum_{i=1}^{m} n_i = n$. An upper bound on

[1] T is not a branch-on-items tree because it incorporates the possibility of not using an item in any bid, even if bids on that singleton item have been submitted. This corresponds to adding a dummy bid on *every* item, not just those items that received no singleton bids.

[2] One might worry about the issue that at different nodes at a given level, different sets of items might be used up by the bids on the path from the root to the node. This would lead to different sets of bids being available at different nodes at the same level. However, we avoid this problem by pessimistically allowing the adversary to submit multiple bids on the same combination of items. This allows the adversary to maximize the set of available bids at *every* node of the level.

If multiple bids on the same combination of items are actually allowed in the input to the search algorithm, the upper bound of the theorem is tight. There is an easy way to construct a worst case. For example, when $n = 12$, $m = 6$, and $\kappa = 3$, a worst case is obtained by submitting six bids on $\{1, 2, 3\}$ and six bids on $\{4, 5, 6\}$. The number of leaves in the corresponding branch-on-bids tree is 49, which equals the upper bound of the theorem.

Even if multiple bids on the same set of items do not exist in the input to the search algorithm (for example, a preprocessor has removed all but the highest bid for each combination of items), our upper bound applies because it is constructed under the pessimistic assumption that gives the adversary too much power.

the number of leaves in T'' is given by the following mathematical program where the adversary is trying to maximize.

$$\max_{n_1,\dots,n_m} \prod_{i=1}^{\lfloor \frac{m}{\kappa} \rfloor} (n_i + 1)$$

$$\text{such that} \quad \sum_{i=1}^{\lfloor \frac{m}{\kappa} \rfloor} n_i = n \quad \forall i \in \left\{1, \dots, \left\lfloor \frac{m}{\kappa} \right\rfloor\right\}, \quad n_i \geqslant 0.$$

The maximum is obtained by distributing n evenly across the variables n_i where $i \in \{1, \dots, \lfloor m/\kappa \rfloor\}$, giving $n_i = n/\lfloor m/\kappa \rfloor$. (Even distribution is not actually possible if n is not divisible by $\lfloor m/\kappa \rfloor$, but it does give an upper bound as desired even in that case.) Therefore, the objective function value of the mathematical program above is at most

$$\left(\frac{n}{\lfloor m/\kappa \rfloor} + 1\right)^{\lfloor m/\kappa \rfloor}.$$

Because the number of leaves in T'' is the same as the number of feasible solutions, which is the same as the number of leaves in the branch-on-bids tree, this upper bound also applies to the number of leaves in the branch-on-bids tree.

Because the branch-on-bids tree is a binary tree, the number of nodes in the tree is $2 \cdot \#\text{leaves} - 1$. □

While this tree size is exponential in items, it is polynomial in bids—unlike the naive upper bound 2^n would suggest. This is desirable because the auctioneer can usually control the items that are for sale in one auction, but not the number of bids that 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.[3] By contrast, the dynamic programming algorithm for winner determination [19] is exponential in items even in the best case.

In the branch-on-bids formulation, there is a 1-to-1 correspondence between leaves and feasible solutions. In the branch-on-items formulation, each leaf corresponds to a distinct feasible solution, but there are feasible solutions that are not represented by any leaf. The leaves correspond 1-to-1 to those feasible solutions where all items have been allocated to bids (actual bids or dummy bids). So, some feasible solutions are not represented by any leaf. Furthermore, some feasible solutions are not represented by any node! For example, if two bids have been submitted, {1} and {2}, then there will be the root node where no items are allocated, the next node where 1 is allocated, and the leaf node where 1 and 2 are allocated. There is no node corresponding to the feasible solution where only 2 is allocated. Finally, multiple nodes of the branch-on-items tree can represent the same feasible solution. For example, on the right-most branch, all nodes correspond to the feasible solution where the auctioneer keeps all of the items.

[3] This occurs, for example, if the bid graph is densely connected (the path from the root to any given leaf will be short in that case). Furthermore, due to bounding (step 3 of the BOB algorithm), only a small portion of the tree is actually visited during the search.

Sometimes the branch-on-bids formulation leads to a larger tree (in leaves and nodes) than the branch-on-items formulation; Fig. 1 shows an example. On the other hand, the reverse can also occur. Examples can easily be constructed by having several items on which no singleton bids have been submitted (dummy bids would be added for them in the branch-on-items formulation).

The main advantage of BOB compared to the branch-on-items formulation is that BOB is in line with the AI principle of least commitment [20]. In a branch-on-items tree, all bids containing an item are committed at a node, while in BOB, choosing a bid to branch on does not constrain future bid selections. BOB allows more refined search control—in particular, better bid ordering. At any search node, the bid to branch on can be chosen in an unconstrained way using information about the subproblem that remains at that node. Many of the techniques of this paper capitalize heavily on that possibility.

3.1.2. 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 contribution of other bids by using up many items. At a search node, we choose a bid that maximizes $p_j/|S_j|^\alpha$ (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. That corresponds to simply choosing the bid that has highest price. But such bids are likely to use up a large number of items in practice, thus reducing significantly the revenue that can be collected from other bids.

Similarly, it seems that $\alpha = 1$ gives too much preference to bids with few items. That corresponds to selecting a bid with the highest per-item price. If a bid with few items is chosen, it seems unlikely that the same (and never higher) per-item revenue can be obtained from the remaining items. So, if there are two bids with close to equal pre-item price, it would be better to choose a bid with a larger number of items so that the high level of per-item revenue could be obtained for a larger number of items.

It was recently shown that in a greedy algorithm that simply inserts bids into IN^* in highest $p_j/|S_j|^\alpha$ first order (as a bid is inserted, bids that share items with it are discarded), $\alpha = 1/2$ gives the best worst case bound over all α [12] (but not necessarily over all possible bid ordering formulas). In other words, $\alpha = 1/2$ strikes the tradeoff outlined above in the worst-case-optimal way for that greedy algorithm. This result does not necessarily mean that $\alpha = 1/2$ is the best setting for a complete search algorithm such as BOB. We suggest the choice of α as an experimental future research problem. Similarly, one could use entirely different bid ordering heuristics in step 4 of BOB.

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

3.1.3. Lower bounding (LOW)

We also prune using a lower bound, L (obtained, for example, 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.

3.1.4. 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 *bid graph*, G, whose vertices V are the bids with $e_j = 0$, and two vertices share an edge if the bids share items (see dotted graphs in Fig. 1). Call the set of edges E. Clearly, $|V| \leqslant n$ and $|E| \leqslant n(n-1)/2$. Using an $O(|E| + |V|)$ time depth-first-search of graph G, the algorithm checks whether G 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 $e_j \leftarrow 1$).[4]

3.1.5. Upper and lower bounding across subproblems (ACROSS)

The straightforward approach is to call BOB on each subproblem with $g = 0$ and $\tilde{f}^* = 0$. 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, for example, 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, for example, 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

$$F^*_{\text{solved}} = g^\theta + \sum_{q=1}^{z-1} f_q^*, \qquad H_{\text{unsolved}} = \sum_{q=z+1}^{k} h_q,$$

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

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

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

and we update IN^* accordingly.

[4] This decomposition check was used as a preprocessor before [23]. That is, it was used at the root of the tree, but not at every node.

Now we can cut the search path whenever

$$F^*_{\text{solved}} + g_z + h_z + H_{\text{unsolved}} \leqslant \tilde{f}^*.$$

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.

3.1.6. 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 [28] for details.) In order to identify articulation bids, we assign to each node v one additional number, $\text{low}(v)$, which is defined inductively:

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

An *internal* node v is an articulation bid if and only if $\text{low}(v) \geqslant d(v)$. (The root node is an articulation bid if and only if it has two or more children in the depth-first-search tree.) Since $\text{low}(v)$ depends only on the low( ) values of v's children and the back edges incident to v, we can calculate $\text{low}(v)$ for all the nodes during the depth-first search. 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. Does one of these schemes dominate the other? To answer this question, we first 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

$$B_k = \arg \max_{B_j \in \mathcal{B} \mid e_j = 0} \nu\big(p_k, |S_k|\big),$$

where ν is a function that is nondecreasing in p_k and nonincreasing in $|S_k|$. Ties can be resolved arbitrarily, for example, preferring bids that articulate.

Theorem 2. *For any given articulation-based bid choosing scheme and any given price-based bid choosing scheme, there are instances where the former leads to less search, as well as instances where the latter leads to less search.*

Proof. We first demonstrate an example where any articulation-based scheme leads to more search than any price-based scheme. Fig. 2 shows an example with 5 bids, labeled A through E. The set of items for each bid is shown in curly brackets above the bid node, and the price for the bid is shown below the bid node. In this bid graph, node D is the only articulation bid. The quantity e is an arbitrarily small positive constant.

Fig. 3 shows the search tree for any articulation-based scheme. We first branch on D. Choosing D excludes bids B, C and E, leaving A as the only remaining bid. We then branch on A. Thus, the best allocation obtained from the "in" branch at D is $\{A, D\}$. We next follow the "out" branch for D. Removing D from the graph disconnects G into two components: $\{A, B, C\}$ and $\{E\}$. The search trees for the two components are shown side-by-side in the figure. The graph for component $\{A, B, C\}$ has an articulation bid A, so we branch on that. Choosing A excludes both B and C, terminating the search, as shown. Next we follow the "out" branch of A, which decomposes the graph into two singleton

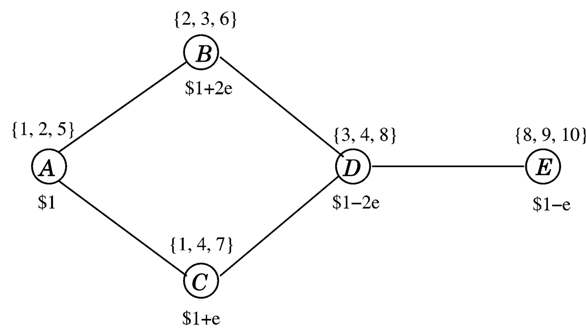


Fig. 2. A set of bids where any articulation-based scheme searches more than any price-based scheme.

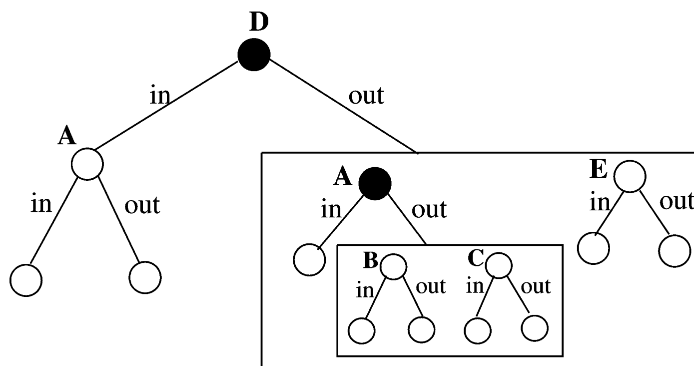


Fig. 3. The search tree for the articulation-based scheme. Articulation bids are shown as filled black circles. Rectangular boxes group together the decomposed subproblems that result from including or excluding a bid.

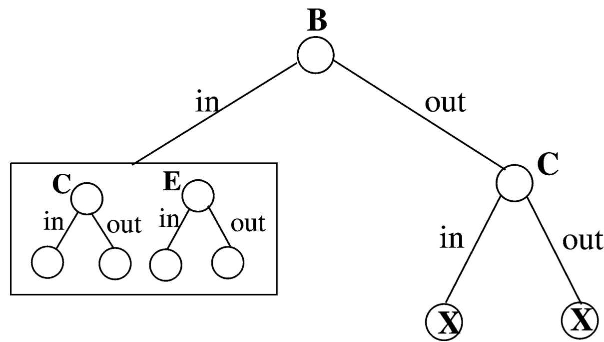


Fig. 4. The search tree for the price-based scheme. X marks nodes where the upper-bounding function cuts off the search.

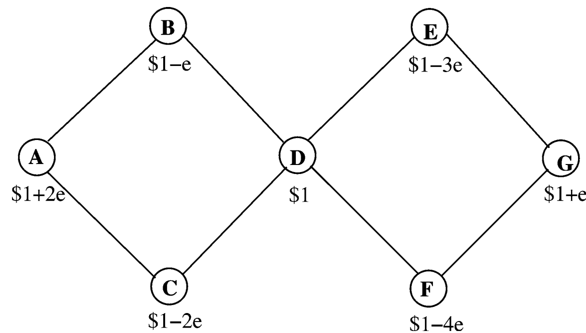


Fig. 5. A set of bids where any articulation-based scheme searches less than any price-based scheme.

components $\{B\}$ and $\{C\}$. Their search trees are shown side-by-side below the out branch of A. The total amount of search (number of edges) is 12.[5]

Since all bids have the same number of items, all price-based orderings will lead to bid ordering B, C, A, E, D. We first branch on B. The "in" branch excludes A, D, leaving two singleton components $\{C\}$ and $\{E\}$. Their search trees are shown side-by-side below the "in" branch at B. The best allocation found during the "in" branch of B is $\{B, C, E\}$, giving a total revenue of $\$(3 + 2e)$. We next follow the "out" branch of B. The next highest bid is C, so we branch on that. Taking C in excludes A and D, leaving only E. At this point, the upper-bounding function cuts off the search since $p(C) + p(E) < (3 + 2e)$. Similarly, when we follow the out branch of C, the upper-bounding function again cuts off the search. In total, the price-based scheme leads to 8 edges of search, that is, fewer than the articulation-based scheme.

[5] Throughout this proof we do not count the cost of the calls to BOB that lead to a decomposition, because those calls only incur negligible effort. In other words, in the figures, there are no edges from a rectangle to the roots of the subtrees at the top of that rectangle. The theorem also applies if one did count such edges, but a different bid graph in Fig. 5 would be required in the proof.

Table 1

Bid	Items included in the bid
A	1, 2, 9, 10
B	2, 3, 11, 12
C	1, 4, 13, 14
D	3, 4, 5, 6
E	6, 7, 15, 16
F	5, 8, 17, 18
G	7, 8, 19, 20

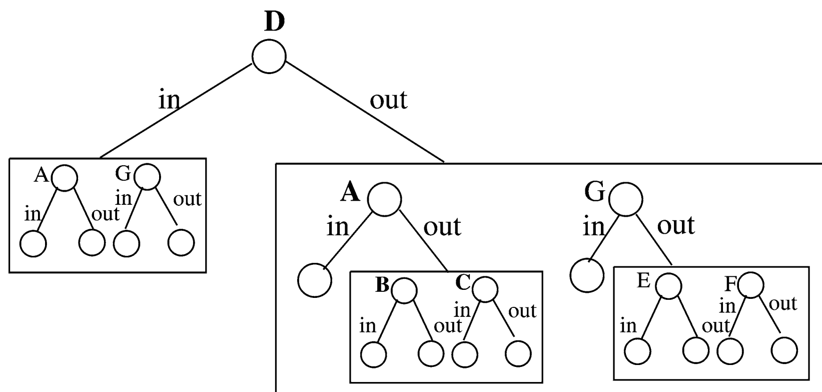


Fig. 6. Articulation-based search tree for the example of Fig. 5.

Next we demonstrate an example (Fig. 5) where any articulation-based scheme leads to less search than any price-based scheme. Again, e is an arbitrarily small positive constant, used to enforce the price-based bid ordering A, G, D, B, C, E, F. Items are not shown in the figure, but Table 1 shows that one can easily construct bids, using 4 items per bid, that lead to the bid graph of Fig. 5.

Figs. 6 and 7, respectively, show the search trees for articulation-based and price-based bid ordering schemes. The articulation-based scheme searches 18 edges, while the price-based scheme searches 20 edges.

This completes the proof.[6] □

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

[6] An obvious minor optimization to the search algorithm is to not try the "out" branch at all if there is only one bid left. This would affect the edge counts, which would be 8, 6, 12, and 15 for the four search trees of this proof (instead of 12, 8, 18, and 20). The theorem clearly still applies under this alternative way of counting.

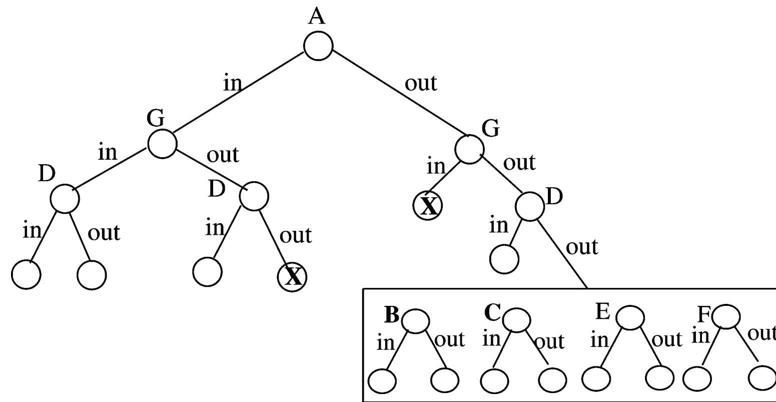


Fig. 7. Price-based search tree for the example of Fig. 5. Nodes where the upper bounding function cuts off the search are marked by X's.

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).

3.2. Tractable subproblems at nodes

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

3.2.1. 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 [23]. We call bids with 1 or 2 items *short* and other bids *long*.[7] Winners can be optimally determined in $O(n_{\text{short}}^3)$ worst case time using a weighted maximal matching algorithm (Edmond's algorithm [4]) if the problem has short bids only [19]. 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 $IN^* \leftarrow IN \cup IN_E$, $\tilde{f}^* \leftarrow g + f_E$.

[7] We define *short* in this way because the problem is $\mathcal{NP}$-complete already if 3 items per bid are allowed [19].

3.2.2. 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.

3.2.3. Interval bids

Rothkopf et al. [19] considered an important special case where the items are 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 $\mathrm{O}(m^2)$ time. We propose a different algorithm that solves this special case in $\mathrm{O}(n + m)$ time. This asymptotic complexity is worst-case optimal because any algorithm must read all of the items and bids as input.

We now describe our algorithm. 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. For instance, bid $[10, 15]$ comes before bid $[5, 20]$, and bid $[1, 20]$ comes before bid $[5, 20]$. Since the set of items has bounded size $[1, m]$, we can bucket sort the bids in $\mathrm{O}(n + m)$ time. Our dynamic program computes optimal solutions for the prefix intervals of the form $[1, i]$, for $i = 1, 2, \ldots, n$. Let $\mathrm{opt}(i)$ 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, $\mathrm{opt}(0) = 0$. Let C_i denote the set of bids whose last item is i; this set is empty if no bid has i as its last item. Then, we have the following recurrence:

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

where p_b is the price of bid b, and f_b is the smallest indexed item in b. The optimal allocation is given by $\mathrm{opt}(m)$. While this dynamic program only computes the value of the optimal allocation, we can easily extend the algorithm to also maintain the actual bid allocation.

Theorem 3. *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 $\mathrm{O}(n + m)$.*

Proof. The maximization in the dynamic programming recurrence above has two terms. 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 $\mathrm{opt}(i)$ in time proportional to the size of C_i. Since $\sum C_i = n$, the total time complexity is $\mathrm{O}(n + m)$. □

If we allow interval wraparound bids (e.g., $S_j = \{m-1, m, 1, 2, 3\}$), the winners can be determined optimally by solving several instances of the interval bids without wraparound, as described below.

Theorem 4. *If interval bids with wraparound are allowed, then winners can be determined in worst-case time* $O(\min\{s, m\}(n+m))$*, where* s *is the smallest number of bids that cross any interval of the form* $[i, i+1]$*, where* $i = 1, 2, \ldots, m$*, and* $m+1$ *is identified with item* 1.[8]

Proof. We can obtain an $O(m(n+m))$ time algorithm by cutting the circle of items at each of the m possible positions in turn, removing the bids that span over the cutting position, and then solving the resulting interval bid problem. The best of these m solutions is the optimal for the wraparound problem. Alternatively, we can obtain an $O(s(n+m))$ time solution by focussing on just one cut, and iterating over all the bids that span that cut. Suppose the cut chosen is $[i, i+1]$. Of all the bids spanning this cut, at most one belongs in the optimal solution. We take each of these bids in turn, remove all the elements covered by this bid, and solve the interval bids problem on the remaining items. If s is the cut with the smallest number of bids spanning it, then the bound is $O(s(n+m))$. Note that s can be $\Omega(m)$ in the worst case, but is likely to be much smaller in practice. Thus, depending on the relative sizes of s or m, we can choose between the two cutting strategies described.

We can determine s in $O(n+m)$ time as follows. Let $BEGIN_i$ be the number of bids that *begin* at item i, and let END_i be the number of bids that *end* at item i. These counts can be calculated in $O(m)$ time by scanning the set of bids once, and updating the *BEGIN* and *END* counters. Next, we start at the cut $[1, 2]$, and determine in $O(m)$ time how many bids span this cut. We then update this count for each new cut in $O(1)$ time as follows. When moving from cut $[i-1, i]$ to $[i, i+1]$, we subtract END_i from the count and add $BEGIN_i$ to the count. Thus, the value of s can be found in $O(n+m)$ time, giving us the claimed bound. □

3.2.4. 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 1-to-1 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 [11] solves this problem in $O(|V| + |E|)$ time. Given the intervals for the bids, one can easily produce a linear ordering of the items.

[8] The fastest prior algorithm for interval wraparound bids takes $O(m^3)$ time [19].

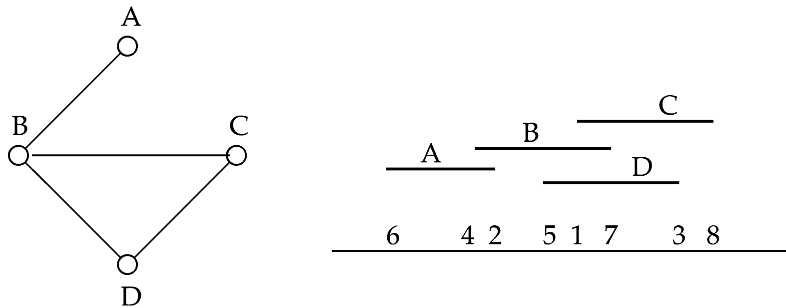


Fig. 8. Bid graph and a valid linear ordering.

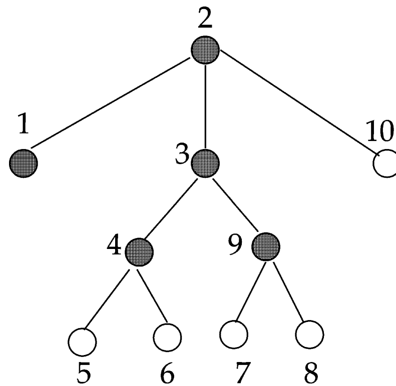


Fig. 9. An example of a subgraph bid: $\{1, 2, 3, 4, 9\}$.

Fig. 8 shows an example with 4 bids: $A = (2, 4, 6)$, $B = (1, 2, 4, 5, 7)$, $C = (1, 3, 7, 8)$, $D = (1, 3, 5, 7)$.

The case where wraparound bids are allowed can be identified in $O(n^2)$ time using an algorithm for recognizing whether the graph G is *circular* [5].

3.2.5. 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 [19]. The items are structured in a tree T, and a valid bid corresponds to a *connected subgraph* of T (see Fig. 9). 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 quite distinct from the "nested structure" model in [19], where the tree nodes correspond to bids.

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 neighbor 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 neighbor 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.

Theorem 5. *The winner determination problem with subgraph bids on tree-structured items can be solved in* $\mathrm{O}(nm)$ *worst-case time.*

Proof. Let r be the root of the item tree T (any node in the graph can be picked as the root of the tree; this choice does not affect the solution value). We assign each node of T a *level*, which is its distance from the root. Thus, the root r has level 0; the children of r have level 1, and so on. The *level* of a bid b, denoted 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 opt(i), for each node i, where 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 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. (For example, suppose we put a bid for items $\{3, 4\}$ in the tree of Fig. 9. Removing the items of this bid breaks the tree rooted at 3 into three subtrees, with roots at 5, 6, and 9). Now, we get the following dynamic programming recurrence:

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

where p_b is the price of bid b. By proceeding bottom up, we compute opt(i) for all nodes of the tree. Again, the proof of correctness follows from the observation that either the winning allocation does not include item i, in which case the second term of the max gives opt(i); otherwise, the first term iterates over all bids that contain item i (and also have level i), and chooses the optimal solution for the subtree that remains after taking that bid.

By proceeding bottom up, we can compute opt(i) for all nodes of the tree in $\mathrm{O}(nm)$ time (each term takes $\mathrm{O}(n)$ time, and there are m terms to compute). □

Because we can *solve* the winner determination problem with subgraph bids on tree-structured items in polynomial time, we can use it in auctions where the special structure is forced. For example in the case of the web store described above, the special structure was imposed via the links among the web pages.

However, we do not know whether this special structure can be *identified* in polynomial time, that is, we do not know how complex it is to determine whether a given general winner determination problem can be converted into a winner determination problem with subgraph bids on tree-structured items. We pose this as an open research problem.

To use this special case to reduce search in our general winner determination algorithm, the identification problem would have to be solved. Then, at every node (or just at the root), the algorithm would check whether the problem is within this special class. If so, it would

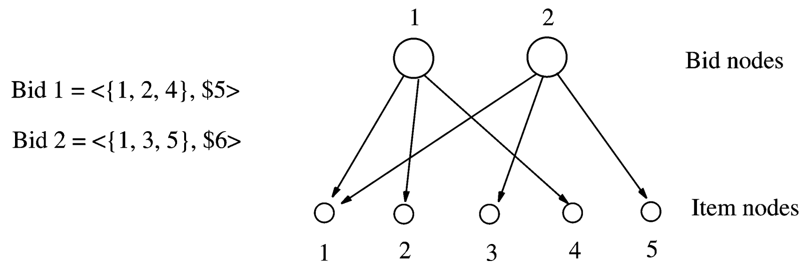


Fig. 10. Illustration for Theorem 6.

be solved using our dynamic program, and no further search would occur in that subtree. If not, the search would proceed further into the subtree.

3.2.6. 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:

Theorem 6. *If the set of items is structured as a directed acyclic graph* (*DAG*) *D, and each bid is a directed subtree of D, then winner determination is $\mathcal{NP}$-complete.*

Proof. We can formulate the general combinatorial auction problem of Section 2 as an instance of the DAG-structured problem. Corresponding to each item of M, we create a "item" node, and for each bid B_i we create a "bid" node. We draw directed edges from bid node B_i to all the item nodes concerning B_i. Fig. 10 shows the construction.

Clearly, the resulting graph is acyclic—all edges are directed from bid nodes to item nodes. Each bid is obviously tree-structured (height one). Thus, an optimal allocation for this DAG instance solves the general combinatorial auction problem, which is $\mathcal{NP}$-complete. This proves that the DAG-structured case is $\mathcal{NP}$-hard. It is also in $\mathcal{NP}$ because the solution (revenue) can be verified in polynomial time. Combining these two facts we have that the problem is $\mathcal{NP}$-complete. □

3.3. Faster data structures

In this section we present fast data structures that support the operations that are used in our winner determination algorithm.

3.3.1. 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'.

3.3.2. Maintaining the heuristic function (MAI)

Any heuristic function could be used in step 2 of BOB. The heuristic function $h \leftarrow \sum_{i \in M'} c(i)$, where $c(i) \leftarrow \max_{j | i \in S_j, e_j = 0} p_j / |S_j|$ is the same as in an earlier winner determination algorithm [23]. In that implementation it took $O(mn)$ time (at every search node) to compute. A faster but rougher approximation of the same heuristic was used in [6]. Here we propose data structures that allow us to compute h fast and exactly. They not only work for the functional form $p_j / |S_j|$, but they also work for any other functional form that depends on the bid only, that is, $c(i) \leftarrow \max_{j | i \in S_j, e_j = 0} \phi(B_j)$.[9]

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 size of $H(i)$ is n in the worst case. The heap supports find-max, extract-max, insert, and delete in $O(\log n)$ 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 $\phi(B_j)$ among the bids B_j 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 $\sum_j |S_j| \cdot O(\log n)$, where the summation is over all the bids that got deleted or reinserted.

As a further optimization, our algorithm uses a *leftist heap* for $H(i)$ [28]. 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 $O(\log n)$. Because the insert and delete operations in BOB are quite frequent, this improves the overall performance.

[9] The complexity results hold if $\phi(B_j)$ can be evaluated in O(1) time for any given B_j. This is usually the case during search since one can precompute $\phi(B_j)$ for every B_j.

3.4. Preprocessing

Four preprocessing techniques were recently proposed for the winner determination problem [23]. Each one of them can be directly used in conjunction with our algorithm.

4. Generalizations of combinatorial auctions

This section discusses generalizations of combinatorial auctions. Our auction server prototype (http://www.cs.cmu.edu/~amem/eMediator) supports all of these generalizations separately and combined, and has been in continuous operation on the web since December 1998 [24].

4.1. 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 $\lambda_j^k \geqslant 0$ is the requested number of units of item k, and p_j is the price. The winner determination problem is:

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

where u_i is the number of units of item i for sale.[10]

Previous winner determination algorithms cannot be used in the multi-unit setting because they branch on items [6,23]. 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 $\exists$ item 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 combinatorial auction. Instead, only those neighbors, j, are removed that demand more units of some item than remain ($\exists$ item k such that

[10] In our basic combinatorial auction, and in every one of the generalizations, if *free disposal* is not possible, an equality constraint should simply be used in the problem formulation in place of the inequality. However, without free disposal the problem is fundamentally different and harder [26].

$\lambda_j^k > u_k - \Lambda_j^k$). 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

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

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 $g + h \leqslant \tilde{f}^*$, the search path is pruned. The lower bounding technique LOW also applies, as do upper and lower bounding across subproblems (ACROSS).

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

4.2. Combinatorial exchanges

In a combinatorial exchange, both buyers and sellers can submit combinatorial bids [24]. 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. Note that a single bid can be buying some items while selling other items.

The winner determination problem is to maximize surplus:[11]

$$\begin{aligned} \max \quad & \sum_{j=1}^n p_j x_j \\ \text{s.t.} \quad & \sum_{j=1}^n \lambda_j^i x_j \leqslant 0, \quad i = 1, 2, \ldots, m, \\ & x_j \in \{0, 1\}. \end{aligned}$$

Unlike earlier algorithms that branch on items [6,23], BOB can be used in this setting. In the basic combinatorial auction and in our other generalizations, the optimal solution occurs in a leaf. In contrast, in our combinatorial exchange, the optimal solution can occur even at 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: $\Lambda_i = \sum_{j | x_j = 1} \lambda_j^i$.

---

[11] If the exchange charges based on transaction volume, as many current exchanges do, it may want to maximize volume instead (subject to the same constraints as above), measured in any of several possible ways. One way is to maximize the number of accepted bids: $\sum_{j \in \{1,\ldots,n\}} x_j$. Another way is to maximize the monetary trade volume. In the common "pay-your-winning-bid" type mechanisms, the monetary trade volume is the amount that the winning bids offered to pay: $\sum_{j \in \{1,\ldots,n\} | p_j > 0} p_j x_j$. Our algorithms apply to such objectives like they do to surplus maximization. However, we advocate surplus maximization since that maximizes social welfare. (This assumes that the participants bid and ask truthfully. If each bidder is charged the prices of her winning bids, then the buyers have the incentive to underbid and the sellers have the incentive to overbid. Even in noncombinatorial exchanges—with only one asset to be exchanged, one seller, and one buyer—no mechanism leads to efficient trade among strategic agents. Sometimes the asset does not trade although the buyer values it more than the seller [16].)

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 $\lambda^i_j \neq 0$ and $\lambda^i_k \neq 0$. However, once a bid is assigned winning and removed from G, the neighbor bids in G cannot always be removed unlike in the basic combinatorial auction. Instead, only those neighbors, j, are removed that cannot possibly be matched any more:

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

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 $\sum_{i | \Lambda_i > 0} \Lambda_i + \lambda^j_i$, or a bid that minimizes $\max_{i | \Lambda_i > 0} \Lambda_i + \lambda^j_i$, the algorithm can reach feasible solutions faster (especially in the case of free disposal), leading again to enhanced pruning from then on.

Additional pruning is achieved by branching on bids with $p_j < 0$ first, and then on bids with $p_j \geqslant 0$.[12] Once $\sum_{j | x_j = 1} p_j > 0$, that branch of the search is pruned.[13] Also, after the switch to bids with $p_j \geqslant 0$ 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.

4.3. 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.[14] 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 prices 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.

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

[13] Alternatively one can do this split of bids into two sets ($\lambda^j_i < 0$ vs. $\lambda^j_i \geqslant 0$) and pruning (when $\Lambda_i > 0$) on any item i instead of price. A reasonable heuristic would be to choose an item (or the price) so that the smaller of these two sets is as small as possible. Then, branching would first occur within bids in that subset. This heuristic is geared toward reducing the size of the search tree.

[14] If some of the items do not have reserve prices, we say $r_i = 0$.

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 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 objective to

$$\max \quad \sum_{j=1}^{n}\left(p_j - \sum_{i \in S_j} r_i\right) x_j.$$

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 combinatorial auctions 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 can then be used for optimally clearing the market.

4.4. 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 combinatorial auctions 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 [23,24]. The second uses dummy items [6]. 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.[15] These additional constraints actually *reduce* the size of the search tree. 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.

[15] As specified earlier in this paper, in the algorithm for exchanges and multi-unit auctions, as a bid is branched in, only some of its neighbors in the bid graph G are removed. However, as a bid is branched in, all neighbors that are connected to it via an *XOR-edge* are removed from G. In this sense, the regular edges and XOR-edges are treated slightly differently in exchanges and multi-unit auctions.

5. Conclusions and future research

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.

We posed the identification of subgraph bids with tree-structured items as an important open problem. Future work also includes experimental evaluation of each of the techniques presented. Since the first version of this paper appeared in AAAI-00, significant experimental work has been done on combinatorial auction winner determination algorithms that use the branch-on-bids formulation rather than the older branch-on-items formulation (e.g., [1,7,13,25,26]). It is fair to say that the branch-on-bids formulation is now the fastest and most prevalent formulation for winner determination. Of the algorithms that have been evaluated experimentally, the *CABOB* algorithm incorporates the largest number of the ideas presented in this paper, and it is currently, to our knowledge, by and large the fastest optimal winner determination algorithm for combinatorial auctions [25].

References

[1] A. Andersson, M. Tenhunen, F. Ygge, Integer programming for combinatorial auction winner determination, in: Proc. Fourth International Conference on Multi-Agent Systems (ICMAS), Boston, MA, 2000, pp. 39–46.
[2] C. Boutilier, M. Goldszmidt, B. Sabata, Sequential auctions for the allocation of resources with complementarities, in: Proc. IJCAI-99, Stockholm, Sweden, 1999, pp. 527–534.
[3] C. DeMartini, A. Kwasnica, J. Ledyard, D. Porter, A new and improved design for multi-object iterative auctions, Technical Report 1054, California Institute of Technology, Social Science, September 1999.
[4] J. Edmonds, Maximum matching and a polyhedron with 0, 1 vertices, J. Res. Nat. Bur. Standards B 69 (1965) 125–130.
[5] E.M. Eschen, J. Spinrad, An $O(n^2)$ algorithm for circular-arc graph recognition, in: Proc. Annual SIAM-ACM Symposium on Discrete Algorithms (SODA), 1993, pp. 128–137.
[6] Y. Fujishima, K. Leyton-Brown, Y. Shoham, Taming the computational complexity of combinatorial auctions: Optimal and approximate approaches, in: Proc. IJCAI-99, Stockholm, Sweden, 1999, pp. 548–553.
[7] R. Gonen, D. Lehmann, Optimal solutions for multi-unit combinatorial auctions: Branch and bound heuristics, in: Proc. ACM Conference on Electronic Commerce (ACM-EC), Minneapolis, MN, 2000, pp. 13–20.
[8] J. Håstad, Clique is hard to approximate within $n^{1-\varepsilon}$, Acta Math. 182 (1999) 105–142.
[9] H. Hoos, C. Boutilier, Solving combinatorial auctions using stochastic local search, in: Proc. AAAI-00, Austin, TX, 2000, pp. 22–29.
[10] F. Kelly, R. Steinberg, A combinatorial auction with multiple winners for universal services, Management Sci. 46 (4) (2000) 586–596.

[11] N. Korte, R.H. Mohring, An incremental linear-time algorithm for recognizing interval graphs, SIAM J. Comput. 18 (1) (1989) 68–81.

[12] D. Lehmann, L.I. O'Callaghan, Y. Shoham, Truth revelation in rapid, approximately efficient combinatorial auctions, J. ACM (2003). To appear. Early version appeared in ACMEC-99.

[13] K. Leyton-Brown, M. Tennenholtz, Y. Shoham, An algorithm for multi-unit combinatorial auctions, in: Proc. AAAI-00, Austin, TX, 2000.

[14] R.P. McAfee, J. McMillan, Analyzing the airwaves auction, J. Economic Perspectives 10 (1) (1996) 159–175.

[15] J. McMillan, Selling spectrum rights, J. Economic Perspectives 8 (3) (1994) 145–162.

[16] R. Myerson, M. Satterthwaite, Efficient mechanisms for bilateral exchange, J. Economic Theory 28 (1983) 265–281.

[17] N. Nisan, Bidding and allocation in combinatorial auctions, in: Proc. ACM Conference on Electronic Commerce (ACM-EC), Minneapolis, MN, 2000, pp. 1–12.

[18] S.J. Rassenti, V.L. Smith, R.L. Bulfin, A combinatorial auction mechanism for airport time slot allocation, Bell J. Economics 13 (1982) 402–417.

[19] M.H. Rothkopf, A. Pekeč, R.M. Harstad, Computationally manageable combinatorial auctions, Management Sci. 44 (8) (1998) 1131–1147.

[20] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, Prentice Hall, Englewood Cliffs, NJ, 1995.

[21] T. Sandholm, An implementation of the contract net protocol based on marginal cost calculations, in: Proc. AAAI-93, Washington, DC, 1993, pp. 256–262.

[22] T. Sandholm, Issues in computational Vickrey auctions, Internat. J. Electronic Commerce 4 (3) (2000) 107–129. Special Issue on Applying Intelligent Agents for Electronic Commerce. A short, early version appeared at the Second International Conference on Multi-Agent Systems (ICMAS), 1996, pp. 299–306.

[23] T. Sandholm, Algorithm for optimal winner determination in combinatorial auctions, Artificial Intelligence 135 (2002) 1–54. First appeared as an invited talk at the First International Conference on Information and Computation Economies, Charleston, SC, October 25–28, 1998. Extended version appeared as Washington Univ., Dept. of Computer Science, Technical Report WUCS-99-01, January 28th, 1999. Conference version appeared at the International Joint Conference on Artificial Intelligence (IJCAI), Stockholm, Sweden, 1999, pp. 542–547.

[24] T. Sandholm, eMediator: A next generation electronic commerce server, Computational Intelligence 18 (4) (2002) 656–676. Early versions appeared in the Conference on Autonomous Agents (AGENTS-00), 2000, pp. 73–96; AAAI-99 Workshop on AI in Electronic Commerce, Orlando, FL, July 1999, pp. 46–55; and as a Washington University, St. Louis, Dept. of Computer Science Technical Report WU-CS-99-02, January 1999.

[25] T. Sandholm, S. Suri, A. Gilpin, D. Levine, CABOB: A fast optimal algorithm for combinatorial auctions, in: Proc. IJCAI-01, Seattle, WA, 2001, pp. 1102–1108.

[26] T. Sandholm, S. Suri, A. Gilpin, D. Levine, Winner determination in combinatorial auction generalizations, in: International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS), Bologna, Italy, 2002, pp. 69–76. Early version appeared at the AGENTS-01 Workshop on Agent-Based Approaches to B2B, Montreal, Canada, May 2001, pp. 35–41.

[27] M. Tennenholtz, Some tractable combinatorial auctions, in: Proc. AAAI-00, Austin, TX, 2000.

[28] M.A. Weiss, Data Structures and Algorithm Analysis in C++, 2nd Edition, Addison-Wesley, Reading, MA, 1999.