An Algorithm for Optimal Winner Determination in Combinatorial Auctions

Tuomas Sandholm*
sandholm@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130-4899

Abstract

Combinatorial auctions, i.e. auctions where bidders can bid on combinations of items, tend to lead to more efficient allocations than traditional auctions in multi-item auctions where the agents' valuations of the items are not additive. However, determining the winners so as to maximize revenue is $\mathcal{NP}$-complete. We present a search algorithm for optimal winner determination. Experiments are shown on several bid distributions. The algorithm allows combinatorial auctions to scale up to significantly larger numbers of items and bids than prior approaches to optimal winner determination by capitalizing on the fact that the space of bids is necessarily sparsely populated in practice. We do this via provably sufficient selective generation of children in the search and by using a method for fast child generation, heuristics that are accurate and optimized for speed, and four methods for preprocessing the search space.

1 Introduction

Auctions are popular, efficient, and autonomy preserving ways of allocating items among agents. This paper focuses on auctions with multiple items to be allocated.

In a *sequential auction*, the items are auctioned one at a time. If a bidder has preferences over bundles, i.e. combinations of items (as is often the case e.g. in electricity markets, equities trading, bandwidth auctions [McAfee and McMillan, 1996], and transportation exchanges [Sandholm, 1993]), bidding in such auctions is difficult. To determine her valuation for an item, the bidder needs to guess what items she will receive in later auctions. This requires speculation on what the others will bid in the future because that affects what items she will receive. Furthermore, what the others bid in the future depends on what they believe others will bid, etc. This counterspeculation introduces computational cost and other wasteful overhead. Moreover, in auctions with a reasonable number of items, such lookahead in the game tree is intractable, and then there is no known way to bid rationally. Bidding rationally would involve optimally trading off the cost of lookahead against the gains it provides, but that would again depend on how others strike that tradeoff. Furthermore, even if lookahead were computationally manageable, usually uncertainty remains about the others' bids because agents do not have exact information about each other. This often leads to inefficient allocations where bidders fail to get the combinations they want and get ones they do not.

In a *parallel auction* the items are open for auction simultaneously and bidders may place their bids during a certain time period. This has the advantage that the others' bids partially signal to the bidder what the others' bids will end up being so the uncertainty and the need for lookahead is not as drastic as in a sequential auction. However, the same problems prevail as in sequential auctions, albeit in a mitigated form.

Combinatorial auctions can be used to overcome the need for lookahead and the inefficiencies that stem from the uncertainties [Rassenti *et al.*, 1982, Sandholm, 1993]. In a combinatorial auction bidders may place 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. For example, the Federal Communications Commission saw the desirability of combinatorial bidding in their bandwidth auctions, but it was not allowed due to perceived intractability of winner determination. This paper focuses on winner determination in combinatorial auctions where each bidder can bid on bundles of indivisible items, and any number of her bids can be accepted.

2 Winner determination

Let M be the set of items to be auctioned, and let $m = |M|$. Then any agent, i, could place any bid $b_i(S)$ for any combination $S \subseteq M$. The relevant bids are:

$$\bar{b}(S) = \max_{i \in \text{ bidders}} b_i(S)$$

Let n be the number of these bids. Winner determination is the following problem, where the goal is to maximize the auctioneer's revenue:

$$\max_{\mathcal{X}} \sum_{S \in \mathcal{X}} \bar{b}(S)$$

where $\mathcal{X}$ is a valid outcome, i.e. an outcome where each item is allocated to only one bidder: $\mathcal{X} = \{S \subseteq M | S \cap S' = \emptyset \text{ for every } S, S' \in \mathcal{X}\}$.

If each combination S has received at least one bid of positive price, the search space will look like Fig. 1.

Proposition 2.1 *The number of allocations is $O(m^m)$ and $\omega(m^{m/2})$.*

The proof is long, and is presented in [Sandholm, 1999].

The graph can be searched more efficiently than exhaustive enumeration by dynamic programming, which takes $\Omega(2^m)$ and $O(3^m)$ steps [Rothkopf *et al.*, 1998]. This is still too complex to scale up above about 25 items. Also, dynamic programming executes the same

*Patent pending since 10/27/1998.

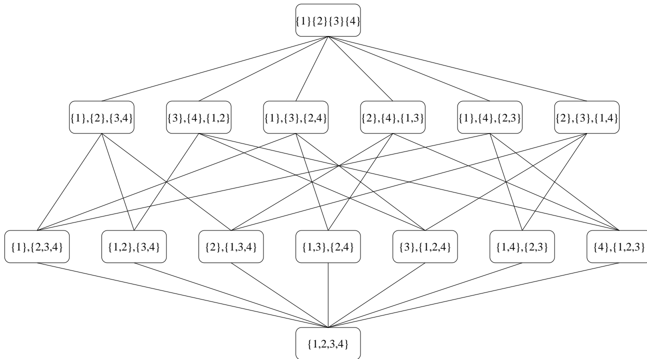

Figure 1: Space of allocations in a 4-item example. Each node represents one possible allocation $\mathcal{X}$.

algorithmic steps regardless of which bids have actually been submitted.

Some combinations of items may not have received any bids, so some of the allocations in the graph need not be considered. Unfortunately no algorithm can find the optimal allocation in polynomial time in n, the number of bids submitted, unless $\mathcal{P} = \mathcal{NP}$:

Proposition 2.2 *Winner determination is $\mathcal{NP}$-complete.*

Proof. Winner determination is weighted set packing, and set packing is $\mathcal{NP}$-complete [Karp, 1972]. □

Even approximate winner determination is hard:

Proposition 2.3 *No polytime algorithm can guarantee an allocation within a bound $\frac{1}{n^{1-\epsilon}}$ from optimum for any $\epsilon > 0$ (unless $\mathcal{NP}$ equals probabilistic polytime).*

The proof is based on [Håstad, 1999], and is presented in the full length version of this paper [Sandholm, 1999].

If the bids exhibit special structure, better approximations can be achieved in polynomial time [Chandra and Halldórsson, 1999, Halldórsson, 1998, Hochbaum, 1983, Halldórsson and Lau, 1997], but even these guarantees are so far from optimum that they are irrelevant for auctions in practice [Sandholm, 1999].

Polynomial time winner determination can be achieved by restricting the combinations on which the agents are allowed to bid [Rothkopf *et al.*, 1998]. However, because the agents may then not be able to bid on the combinations they want, similar economic inefficiencies prevail as in the non-combinatorial auctions.

3 Our optimal search algorithm

The goals of our approach to winner determination are:

- allow bidding on all combinations.
- strive for the optimal allocation.
- completely avoid loops and redundant generation of vertices when searching the allocation graph, Fig. 1.
- capitalize heavily on the sparseness of bids. In practice the space of bids is necessarily extremely sparsely populated. For example, if there are 100 items, there are $2^{100} - 1$ combinations, and it would take longer than the life of the universe to bid on all of them even if every person in the world submitted a bid per second. Sparseness of bids implies sparseness of the allocations $\mathcal{X}$ that need to be checked. Our algorithm constructively checks each allocation $\mathcal{X}$ that has positive value exactly once, and does not construct the other allocations. Therefore, unlike dynamic programming, the algorithm only generates those parts of the search space which are actually populated by bids. The disadvantage then is that the run time depends on the bids received.

To achieve these goals, we use a search algorithm that generates a tree, Fig. 2. Each path in the tree consists of a sequence of disjoint bids, i.e. bids that do not share items. As a bid is added to the path, the bid price is added to the g-function. A path terminates when all items have been used on that path. At that point the path corresponds to a feasible allocation, and the revenue from that allocation, i.e. the g-value, can be compared to the best one found so far to determine whether the allocation is the best one so far. The best so far is stored, and once the search completes, that allocation is optimal.

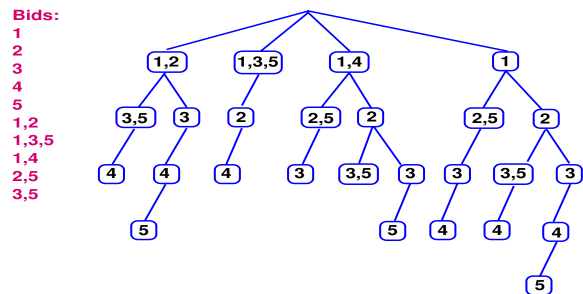

Figure 2: A search tree generated by our algorithm.

The naive method of constructing the search tree would include all bids (that do not include items that are already on the path) as the children of each node. Instead, the following proposition enables a significant reduction of the branching factor by capitalizing on the fact that the order of the bids on a path does not matter.

Proposition 3.1 *Every allocation will be explored exactly once in the tree if the children of a node are those bids that*

- *include the item with the smallest index among the items that are not on the path yet, and*
- *do not include items that are already on the path.*

Proof. We first prove that each allocation is generated at most once. The first bullet leads to the fact that an allocation can only be generated in one order of bids on the path. So, for there to exist more than one path for a given allocation, some bid would have to occur multiple times as a child of some node. However, the algorithm uses each bid as a child for a given node only once.

What remains to be proven is that each allocation is generated. Assume for contradiction that some allocation is not. Then, at some point, there has to be a bid in that allocation such that it is the bid with the item with the smallest index among those not on the path, but that bid is not inserted to the path. Contradiction □

Our search algorithm restricts the children according to the proposition, Fig. 2. This can be seen for example at the first level because all the bids considered at the

first level include item 1. The minimal index does not coincide with the depth of the search tree in general.

The auctioneer's revenue can increase if he can keep items. That can be profitable if some item has received no bids on its own. For example, if there is no bid for item 1, a $5 bid for item 2, and a $3 bid for the combination of 1 and 2, it is more profitable for the auctioneer to keep 1, and to allocate 2 alone. Such optimization can be implemented by placing dummy bids of price zero on those individual items that received no bids alone, Fig. 2. For example, if item 1 had no bids on it alone and dummies were not used, the tree under 1 would not be explored and optimality could be lost. When dummy bids are used, the resulting search generates each allocation that has positive revenue exactly once (and searches through no other allocations). This guarantees that the algorithm finds the optimal solution. Throughout the rest of the paper, we use this dummy bid technique.

3.1 Optimized generation of children

The main search algorithm uses a secondary depth-first-search (DFS) to quickly determine the children of a node. The secondary search occurs in a data structure which we call the *Bidtree*. It is a binary tree in which the bids are inserted up front as the leaves. Only those parts of the tree are generated for which bids are received, Fig. 3. What makes the data structure special is the use of a

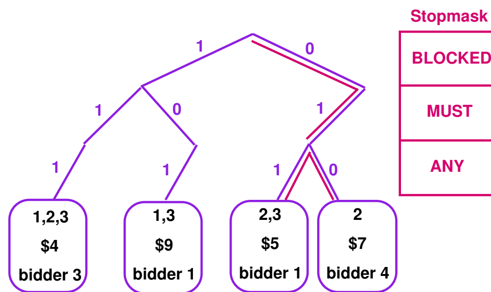

Figure 3: The Bidtree data structure.

Stopmask. The Stopmask is a vector with one variable for each auctioned item. If the variable corresponding to an item has the value BLOCKED, those parts of the Bidtree are pruned instantaneously (and in place) that contain bids containing that item. In other words, search in the Bidtree will never progress left at that level. If the item's variable has the value MUST, all other parts of the Bidtree are pruned instantaneously and in place, i.e. search cannot progress right at that level. The value ANY corresponds to no pruning based on that item: the search may go left or right.

To start, the first item has value MUST in the Stopmask, and the others have ANY. The first child of any given node in the main search is determined by a DFS from the top of the Bidtree. The siblings of that child are determined by backtracking in the Bidtree after the main search has explored the tree under the first child. As a bid is appended to the path of the main search, BLOCKED is inserted in the Stopmask for each item of that bid. That implements the branching reduction of the main search based on the second bullet of Prop. 3.1. MUST is inserted at the unallocated item with the smallest index. That implements the branching reduction of the main search based on the first bullet of Prop. 3.1. These MUST and BLOCKED values are changed back to ANY when backtracking a bid from the path of the main search, and MUST is reallocated to the place where it was before that bid was appended to the path.

The secondary search can be implemented to execute in place, i.e. without memory allocation during search. That is accomplished via the observation that recursion or an open list is not required because in DFS, to decide where to go next, it suffices to know where the search focus is now, and from where it most recently came.

3.2 Anytime winner determination via depth-first-search (DFS)

We first implemented the main search as DFS which executes in linear space. The depth-first strategy causes feasible allocations to be found quickly (the first one is generated in linear time when the first search path ends), and the solution improves monotonically since the algorithm keeps track of the best solution found so far. This implements the anytime feature: if the algorithm does not complete in the desired amount of time, it can be terminated prematurely, and it guarantees a feasible solution that improves monotonically over time. When testing the anytime feature, it turned out that in practice most of the revenue was generated early on as desired, and there were diminishing returns to computation.

3.3 Preprocessing

Our algorithm preprocesses the bids in four ways to make the main search faster without compromising optimality. The next subsections present the preprocessors in the order in which they are executed.

PRE1: Keep only the highest bid for a combination

As a bid arrives, it is inserted into the Bidtree. If a bid for the same S already exists in the Bidtree, only the bid with the higher price is kept, and the other bid is discarded. We break ties in favor of the earlier bid.

PRE2: Remove provably noncompetitive bids

This preprocessor removes bids that are provably noncompetitive. A bid (*prunee*) is noncompetitive if there is some disjoint collection of subsets of that bid such that the sum of the bid prices of the subsets exceeds or equals the price of the prunee bid. For example, a $10 bid for items 1, 2, 3, and 4 would be pruned by a $4 bid for items 1 and 3, and a $7 bid for items 2 and 4.

To determine this we search, for each bid (potential prunee), through all combinations of its disjoint subset bids. This is the same DFS as the main search except that it restricts the search to those bids that only include items that the prunee includes (Fig. 4): BLOCKED is kept in the Stopmask for other items.

Especially with bids that contain a large number of items, PRE2 can take more time than it saves in the main search. In the extreme, if some bid contains all items, the preprocessing search with that bid as the prunee is the same as the main search (except for one main search

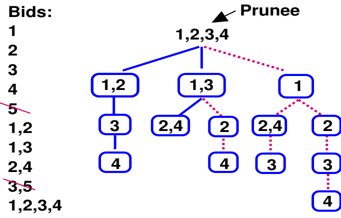

Figure 4: A search tree generated for one prunee in PRE2. The dotted paths are not generated because pruning occurs before they are reached.

path that contains that bid only). To save preprocessing time, PRE2 is carried out partially. Some of the noncompetitive bids are left unpruned, but that will not affect optimality of the main search—although it can make it slower. We implemented two ways of restricting PRE2:

1. A cap, Γ, on the number of pruner bids that can be combined to try to prune a particular prunee bid. This limits the depth of the search in PRE2 to Γ.
2. A cap, Φ, on the number of items in a prunee bid. Longer bids would then not be targets of pruning. This entails a cap, Φ, on tree depth. It also tends to exclude wide trees because long prunees usually lead to trees with large branching factors.

With either method PRE2 takes $O(nn^{cap}m)$ time, which is polynomial for a constant cap (there are n prunees, the tree for each is $O(n^{cap})$, and finding a child in the Bidtree is $O(m)$). The latter method is usually preferable. It does not waste computation on long prunees which take a lot of preprocessing time and do not significantly increase the main search time. This is because the main search is shallow along the branches that include long bids: each item can only occur once on a path and a long bid uses up many items. Second, if the bid prices are close to additive, the former method does not lead to pruning when a path is cut prematurely based on the cap.

PRE3: Decompose bids into connected sets

The bids are partitioned into sets such that no item is shared by bids from different sets. PRE4 and the main search are then done in each set of bids independently, and using only items included in the bids of the set. The sets are determined as follows. We define a graph where bids are vertices, and two vertices share an edge if the bids share items. We generate an adjacency list representation of the graph in $O(mn^2)$ time. We use DFS to generate a depth-first forest of the graph in $O(n+m)$ time. Each tree is then a set with the desired property.

PRE4: Mark noncompetitive tuples of bids

Noncompetitive tuples of disjoint bids are marked so that they need not be considered on the same path in the main search. For example, the pair of bids \$5 for items 1 and 3, and \$4 for items 2 and 5 is noncompetitive if there is a bid of \$3 for items 1 and 2, and a bid of \$7 for items 3 and 5. Noncompetitive tuples are determined as in PRE2 except that now each prunee is a virtual bid that contains the items of the bids in the tuple, and the prunee price is the sum of the prices of those bids.

For computational speed, we only mark 2-tuples, i.e. pairs of bids. A pair of bids is excluded also if the bids share items. PRE4 is used as a partial preprocessor like PRE2, with caps Γ' or Φ' instead of Γ or Φ. PRE4 runs in $O(n^2 n^{cap} m)$ time. Handling 3-tuples would increase this to $O(n^3 n^{cap} m)$, etc. Handling large tuples also slows the main search because it needs to ensure that noncompetitive tuples do not exist on the path.

As a bid is appended to the path, it excludes from the rest of the path those other bids that constitute a noncompetitive pair with it. Our algorithm determines this quickly as follows. For each bid, a list of bids to exclude is determined in PRE4. In the main search, an exclusion count is kept for each bid, starting at 0. As a bid is appended to the path, the exclusion counts of those bids that it excludes are incremented. As a bid is backtracked from the path, those exclusion counts are decremented. Then, when searching for bids to append to the main search path from the Bidtree, only bids with exclusion count 0 are accepted.[1]

3.4 IDA* and heuristics

We sped up the main search by using an iterative deepening A* (IDA*) search strategy [Korf, 1985] instead of DFS. The search tree, use of the Bidtree, and the preprocessors stay the same. At each iteration of IDA*—except the last—the IDA* threshold gives an upper bound on solution quality. It can be used, for example, to communicate search progress to the auctioneer.

Since winner determination is a maximization problem, the heuristic function h should never underestimate the revenue from the items that are not yet allocated in bids on the path because that could lose optimality. We designed two heuristics that never underestimate:

1. $h = \sum_{i \in \text{unallocated items}} c(i)$ where $c(i) = \max_{S | i \in S} \frac{\bar{b}(S)}{|S|}$
2. As above, but accuracy is increased by recomputing $c(i)$ every time a bid is appended to the path since some combinations S are excluded: some of their items are on the path, or they constitute a noncompetitive pair with some bid on the path.

We use (2) with several methods for speeding it up. A tally of h is kept, and only some of the $c(i)$ values in h need to be updated when a bid is appended to the path. In PRE4 we precompute for each bid the list of items that must be updated: items included in the bid and in bids that are on the bid's exclude list. To make the update even faster, we keep a list for each item of the bids in which it belongs. The $c(i)$ value is computed by traversing that list and choosing the highest $\frac{\bar{b}(S)}{|S|}$ among the bids that have exclusion count 0. So, recomputing h

[1] PRE2 and PRE4 could be converted into anytime preprocessors without compromising optimality by starting with a small cap, conducting the searches, increasing the cap, reconducting the searches, etc. Preprocessing would stop when it is complete (cap $= n$), the user decides to stop it, or some other stopping criterion is met. PRE2 and PRE4 could also be converted into approximate preprocessors by allowing pruning when the sum of the pruners' prices exceeds a fixed fraction of the prunee's price. This would allow more bids to be pruned which can make the main search faster, but it can compromise optimality.

takes $O(\underline{m}\underline{n})$ time, where $\underline{m}$ is the number of items that need to be updated, and $\underline{n}$ is the (average or greatest) number of bids in which those items belong.[2]

On the last IDA* iteration, the IDA* threshold is always incremented to equal the revenue of the best solution found so far in order to avoid futile search. In other words, once the first solution is found, the algorithm converts to branch-and-bound with the same heuristic.

4 Experimental setup

Not surprisingly, the worst case complexity of the main search is exponential in the number of bids. However, unlike dynamic programming, this is complexity in the number of bids actually received, not in the number of allowable bids. To determine how the algorithm does in practice, we ran experiments on a regular uniprocessor workstation (360MHz Sun Ultra 60 with 256 MRAM) in C++ with four different bid distributions:

- **Random:** For each bid, pick the number of items randomly from $1, 2, ..., m$. Randomly choose that many items without replacement. Pick the price randomly from $[0, 1]$.
- **Weighted random:** As above, but pick the price between 0 and the number of items in the bid.
- **Uniform:** Draw the same number of randomly chosen items for each bid. Pick the prices from $[0, 1]$.
- **Decay:** Give the bid one random item. Then repeatedly add a new random item with probability α until an item is not added or the bid includes all m items. Pick the price between 0 and the number of items in the bid.

If the same bid was generated twice, the new version was deleted and regenerated. So if the generator was asked to produce e.g. 500 bids, it produced 500 different bids.

We let all the bids have the same bidder. This conservative method causes PRE1 to prune no bids. In practice, the chance that two agents bid on the same combination of items is often small anyway because the number of combinations is large ($2^m - 1$). However, in some cases PRE1 is very effective. For example, it prunes all of the bids except one if all bids are placed on the same combination by different bidders.

5 Experimental results

We focus on IDA* because it was two orders of magnitude faster than DFS. We lower the IDA* threshold between iterations to 95% of the previous threshold or to the highest $f = g + h$ that subceeded the previous threshold, whichever is smaller. Experimentally, this tended to be a good rate of decreasing the threshold. If it is decreased too fast, the overall number of search nodes increases because the last iteration becomes large. If it is decreased too slowly, the number of search nodes increases because new iterations repeat a large portion of the search from previous iterations.

For PRE2, the cap $\Phi = 30$ gave a good compromise between preprocessing time and main search time. For PRE4, $\Phi' = 20$ led to a good compromise. These values are used in the rest of the experiments. With these caps, the hard problem instances with short bids get preprocessed completely, and PRE2 and PRE4 take negligible time compared to the main search because the trees under such short prunees are small. The caps only take effect in the easy cases with long bids. In the uniform distribution all bids are the same length, so PRE2 does not prune any bids because no bid is a subset of another.

PRE3 saved significant time on the uniform and decay distributions by partitioning the bids into sets when the number of bids was small compared to the number of items, and the bids were short. In almost all experiments with random and weighted random, all bids fell in the same set because the bids were long. In real world combinatorial auctions it is likely that the number of bids will significantly exceed the number of items which would suggest that PRE3 does not help. However, most bids will usually be short, and the bidders' interests often have special structure which leads to some items being independent of each other, and PRE3 capitalizes on that.

The main search generated 35,000 nodes per second when the number of items was small, e.g. 25, and the bids were short. This rate decreased slightly with the number of bids, but significantly with the number of items and bid size. With the random distribution with 400 items and 2000 bids, the search generated only 9 nodes per second. However, the algorithm solved these cases easily because the search paths were short and the heuristic focused the search well. Long bids make the heuristic and exclusion checking slower but the search tree shallower which makes them easier for our algorithm than short bids overall. This observation is further supported by the results below. Each point in each graph represents an average over 20 problem instances. The search times presented include all preprocessing times.

The random distribution was easy, Fig. 5, since the search was shallow because the bids were long. The weighted random distribution was even easier. The curves become closer together on the logarithmic value axis as the number of items increases, which means that search time is polynomial in items. In the weighted random case, the curves are sublinear meaning that search time is polynomial in bids as well, while in the unweighted case they are roughly linear meaning that search time is exponential in bids.

The uniform distribution was harder, Fig. 6 left. The bids were shorter so the search was deeper. The curves are roughly linear so complexity is exponential in bids. The spacing of the curves does not decrease significantly indicating that complexity is exponential in items also. Fig. 6 right shows complexity decrease as bids get longer.

[2] PRE2 and PRE4 use DFS because due to the caps their execution time is negligible compared to the main search. Alternatively they could use IDA*. Unlike in the main search, the $c(i)$ values should be computed using only combinations S that are subsets of the prunee. The threshold for IDA* can be set to the prunee bid's price (or a fraction thereof in the case of approximation), so IDA* will complete in one iteration. Finally, care needs to be taken that the heuristic and the tuple exclusion are handled correctly since they are based on the results of the preprocessing itself.

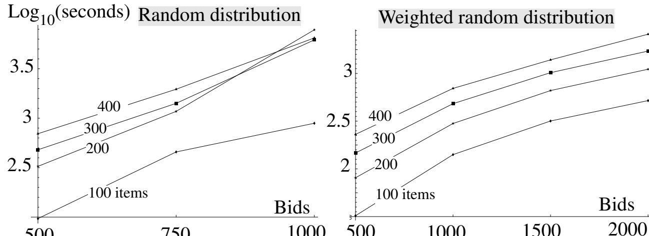

Figure 5: Search time for the random and weighted random distributions. In the random distribution, the point with 1,000 bids and 200 items is unusually high due to one hard outlier among the 20 problem instances.

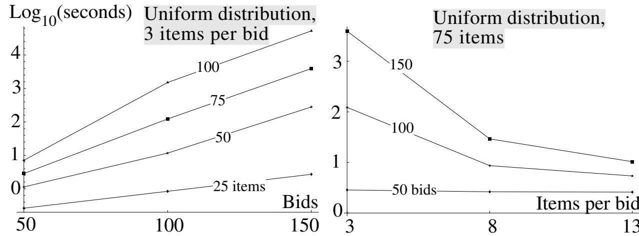

Figure 6: Search time for the uniform distribution.

The decay distribution was also hard, Fig. 7 left. However, the curves get closer together as the number of items increases: complexity is polynomial in items. Complexity first increases in α, and then decreases, Fig. 7 right. Left of the maximum, PRE3 decomposes the problem leading to small, fast searches. The hardness peak moves left as the number of bids grows because the decomposition becomes less successful. Right of the maximum, all bids are in the same set. The complexity then decreases with α because longer bids lead to shallower search.

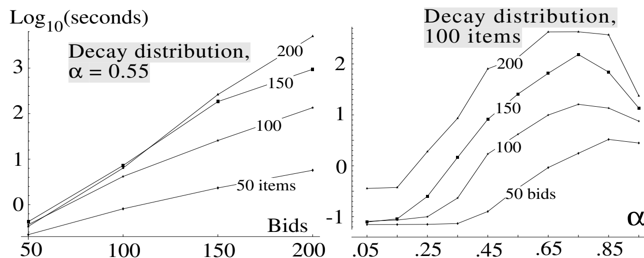

Figure 7: Search time for the decay distribution.

6 Conclusions and generalizations

We presented a search algorithm for optimal winner determination in combinatorial auctions. It allows combinatorial auctions to scale up to significantly larger numbers of items and bids than previous approaches to optimal winner determination. The IDA* search can also be distributed across multiple computers for additional speed. We believe that our algorithm will make the difference between being able to use a combinatorial auction design in many practical markets and not.

The algorithm can also be used to solve weighted set packing, independent set, and maximum clique problems because they are in fact the same problem. So is coalition structure generation in characteristic function games.

The methods discussed so far are based on the common assumption that bids are superadditive: $\bar{b}(S \cup S') \geq \bar{b}(S) + \bar{b}(S')$. But what happens if agent 1 bids $b_1(\{1\}) = \$5$, $b_1(\{2\}) = \$4$, and $b_1(\{1,2\}) = \$7$, and there are no other bidders? The auctioneer could allocate items 1 and 2 to agent 1 separately, and that agent's bid for the combination would value at $\$5 + \$4 = \$9$ instead of $7. So, the current techniques focus on capturing synergies (positive complementarities) among items. In practice, local subadditivities can occur as well. For example, when bidding for a landing slot for a plane, the bidder is willing to take any one of a host of slots, but does not want more than one. To address this we developed a protocol where the bidders can submit *XOR-bids* in our auction server, i.e. bids on combinations such that only one of the combinations can get accepted. This allows the bidders to express general preferences with both positive and negative complementarities, see also [Rassenti *et al.*, 1982]. The winner determination algorithm of this paper can be easily generalized to XOR-bids by marking (as in PRE4) noncompetitive those pairs of bids that are mutually exclusive. These extra constraints cause the algorithm to run faster for XOR-bids than for the same number of nonexclusive bids.

Our server also allows there to be multiple units of each item. The winner determination algorithm then needs to keep track of the sum of the units consumed for each item separately on the main search path. For the multi-unit setting, the h-function can be improved to differentiate between the potential future contributions of units of different items.

Currently we are developing winner determination algorithms for combinatorial double auctions which include multiple buyers and multiple sellers.

References

[Chandra and Halldórsson, 1999] B Chandra and M Halldórsson. Greedy local search and weighted set packing approximation. In *SIAM-ACM Symposium on Discrete Algorithms.*

[Halldórsson and Lau, 1997] M Halldórsson and H Lau. Low-degree graph partitioning via local search with applications to constraint satisfaction, max cut, and 3-coloring. *J.of GraphAlg.Appl.* 1(3):1–13.

[Halldórsson, 1998] M Halldórsson. Approximations of independent sets in graphs. In *Intl. Workshop on Approximation Algorithms for Combinatorial Optimization Problems*, p. 1–14, Aalborg, Denmark. Springer LNCS 1444.

[Håstad, 1999] Johan Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 1999. To appear. Draft: Royal Institute of Tech., Sweden, 8/11/98. Early version: FOCS-96, 627—636.

[Hochbaum, 1983] Dorit S. Hochbaum. Efficient bounds for the stable set, vertex cover, and set packing problems. *Discrete Applied Mathematics*, 6:243–254.

[Karp, 1972] R Karp. Reducibility among combinatorial problems. In Raymond Miller and James Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, NY.

[Korf, 1985] R Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109.

[McAfee and McMillan, 1996] R P McAfee and J McMillan. Analyzing the airwaves auction. *J. of Econ. Perspectives*, 10(1):159–175.

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

[Rothkopf *et al.*, 1998] M Rothkopf, A Pekeč, and R Harstad. Computationally manageable combinatorial auctions. *Management Science*, 44(8):1131–1147. Draft: Rutgers Center for OR report 13-95.

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

[Sandholm, 1999] T Sandholm. An algorithm for optimal winner determination in combinatorial auctions. WUCS-99-01, Washington University, Dept. of Computer Science, January.