Chapter 14: Optimal Winner Determination Algorithms

Tuomas Sandholm

1 Introduction

This chapter discusses optimal winner determination algorithms for combinatorial auctions (CAs). We say the auctioneer has a set of items, $M = \{1, 2, \dots, m\}$, to sell, and the buyers submit a set of package bids, $\mathcal{B} = \{B_1, B_2, \dots, B_n\}$. A package bid is a tuple $B_j = \langle S_j, p_j \rangle$, where $S_j \subseteq M$ is a set of items and $p_j \geq 0$ is a price. (Note that in this chapter, n denotes the number of such bids, not the number of bidders.) The *winner determination problem (WDP)* is to label the bids as winning or losing so as to maximize the sum of the accepted bid prices under the constraint that each item is allocated to at most one bid:

$$\max \sum_{j=1}^{n} p_j x_j \quad \text{s.t.} \quad \sum_{j | i \in S_j} x_j \leq 1, \quad \forall i \in \{1..m\}$$
$$x_j \in \{0, 1\}$$

This problem is computationally complex ($\mathcal{NP}$-complete and inapproximable, see Lehmann, Müller and Sandholm (Chapter 12)). Since 1997, there has been a surge of research addressing it. This chapter focuses on search algorithms that provably find an optimal solution to the general problem where bids are not restricted (e.g., (Sandholm 2002a, Fujishima, Leyton-Brown, and Shoham 1999, Sandholm and Suri 2003, Andersson, Tenhunen, and Ygge 2000, Gonen and Lehmann 2000, Leyton-Brown, Tennenholtz, and

Shoham 2000, Sandholm, Suri, Gilpin, and Levine 2001, Lehmann and Gonen 2001, van Hoesel and Müller 2001, Boutilier 2002, de Vries and Vohra 2003)).[1] Because the problem is $\mathcal{NP}$-complete, any optimal algorithm for the problem will be slow on some problem instances (unless $\mathcal{P} = \mathcal{NP}$). However, in practice modern search algorithms can optimally solve winner determination in the large, and today basically all real-world winner determination problems are being solved using search algorithms.[2]

The goal is to make a set of decisions (e.g., for each bid, deciding whether to accept or reject it). In principle, tree search algorithms work by simulating all possible ways of making the decisions. Thus, once the search finishes, the optimal set of decisions will have been found and proven optimal.

However, in practice, the space is much too large to search exhaustively. The science of search is in techniques that *selectively* search the space while still provably finding an optimal solution. The next section—the bulk of the chapter—studies different design dimensions of search algorithms for winner determination. Section 3 discusses state-of-the-art algorithms in that framework. Section 4 discusses winner determination under fully expressive bidding languages where substitutability is expressed using XOR-constraints between bids. Finally, Section 5 provides pointers to further reading.

2 Design dimensions of search algorithms

Search algorithms for winner determination can be classified under the following high-level design dimensions:

- search formulation,
- search strategy,
- upper bounding techniques (including cutting planes),
- lower bounding techniques and primal heuristics,

- decomposition techniques (including upper and lower bounding across components),
- techniques for deciding which question to branch on,
- techniques for identifying and solving tractable cases at search nodes,
- random restart techniques, and
- caching techniques.

The following subsections study these dimensions in order.

2.1 Search formulations

The most fundamental design dimension is the *search formulation*: what class of questions is the branching question for a node chosen from?

2.1.1 Branching on items

First-generation special-purpose search algorithms for winner determination were based on the branch-on-items search formulation (Sandholm 2002a, Fujishima, Leyton-Brown, and Shoham 1999). At any node, the question to branch on is: "What bid should this item be assigned to?" Each path in the search tree consists of a sequence of disjoint bids, that is, bids that do not share items with each other.

The set of items that are already used on the path is

$$USED = \bigcup_{j \,\mid\, \text{bid } j \text{ is on the path}} S_j \tag{1}$$

and let A be the set of items that are still available: $A = M - USED$. A path ends when no bid can be added to it: for every bid, some of the bid's items have already been used on the path.

As the search proceeds down a path, a tally, g, is kept of the sum of the prices of the bids accepted on the path:

$$g = \sum_{j \mid \text{bid } j \text{ is accepted on the path}} p_j \tag{2}$$

At every search node, the revenue g from the path is compared to the best g-value found so far in the search tree to determine whether the current path is the best solution so far. If so, it is stored as the new *incumbent*. Once the search completes, the incumbent is an optimal solution.

However, care has to be taken to treat the possibility that the auctioneer's revenue can increase by keeping items (Sandholm 2002a). Consider an auction of items 1 and 2. Say there is no bid for 1, a \$5 bid for 2, and a \$3 bid for $\{1, 2\}$. Then it is better to keep 1 and sell 2 than it would be to sell both.

The auctioneer's possibility of keeping items can be implemented by placing *dummy bids* of price zero on those items that received no 1-item bids (Sandholm 2002a). For example in Figure 1, if item 1 had no bids on it alone and dummy bids were not used, the tree under 1 would not be generated and optimality could be lost.

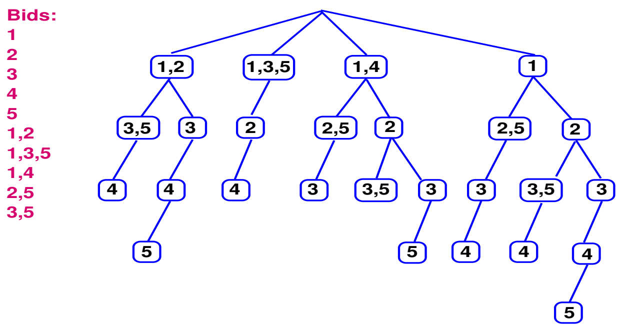

Figure 1: *This example search space corresponds to the bids listed on the left. For each bid, the items are shown but the price is not.*

A naïve method of constructing the search tree would include all bids (that do not include items that are already used 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 2.1 *(Sandholm 2002a) At each node in the search tree, it suffices to let the children of the node be those bids that*

- *include the item with the smallest index among the items that are still available ($i^* = \min\{i \in \{1, \ldots, m\} : i \in A\}$), and*
- *do not include items that have already been used on the path.*

Formally, for any node, θ, of the search tree,

$$children(\theta) = \{j \in \{1, \ldots, n + n_{dummy}\} \mid i^* \in S_j,\ S_j \cap USED = \emptyset\}^{3} \quad (3)$$

The use of that restriction can be seen in Figure 1.

Theorem 2.2 *(Sandholm 2002a) The number of leaves in the tree is no greater than $(\frac{n+n_{dummy}}{m})^m$. Also, $\#leaves \in O(m^m)$. The number of nodes is no greater than $m \cdot \#leaves + 1$.*

So, even in the worst case, the size of the tree is polynomial in the number of bids, but exponential in the number of items.

2.1.2 Branching on bids

Instead of branching on items, newer faster winner determination algorithms use the *branch-on-bids* search formulation (Sandholm and Suri 2003). At any node, the question to branch on is: "Should this bid be accepted or rejected?" When branching on a bid, the children in the search tree are the world where that bid is accepted (IN, $x_j = 1$), and the world where that bid is rejected (OUT, $x_j = 0$), Figure 2 Right.

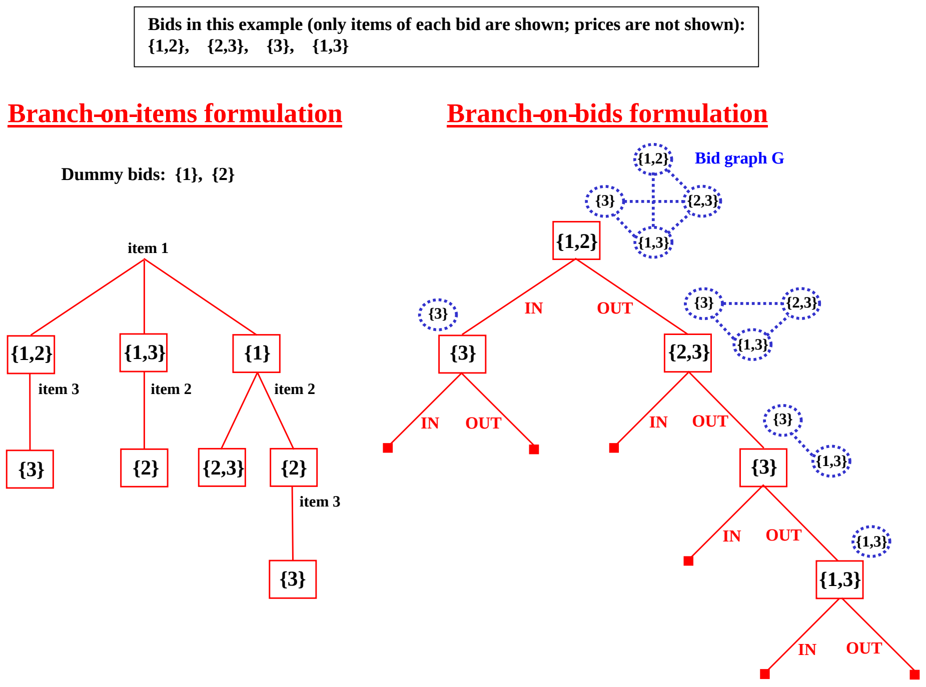

Figure 2: *Branching on items vs. branching on bids.*

The branching factor is 2 and the depth is at most n. (The depth of the left branch is at most $\min\{m, n\}$.) No dummy bids are needed: 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 naïve analysis shows that the number of leaves is at most 2^n. However, a deeper analysis establishes a drastically lower worst-case upper bound:

Theorem 2.3 *(Sandholm and Suri 2003) Let κ be the number of items in the bid with the smallest number of items. The number of leaves is no greater than*

$$\left(\frac{n}{\lfloor \frac{m}{\kappa} \rfloor} + 1\right)^{\lfloor \frac{m}{\kappa} \rfloor} \tag{4}$$

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

While this is exponential in items, it is polynomial in bids—unlike the naïve upper bound 2^n would suggest. This is desirable because the auctioneer can usually control the items that are for sale (if there are too many, she can split the CA into multiple CAs), but does not want to restrict the number of bids submitted.[4] Furthermore, the average performance tends to be significantly better than the worst case.

Sometimes the branch-on-bids formulation leads to a larger tree than the branch-on-items formulation, see Figure 2. Opposite examples can also be constructed—by having 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 the branch-on-bids formulation is that it is in line with the *principle of least commitment* (Russell and Norvig 1995). In a branch-on-items tree, all bids containing an item are committed at a node, while in the branch-on-bids formulation, choosing a bid to branch on does not constrain future bid selections (except that accepting a bid precludes later accepting bids that share items with it). Therefore, the branch-on-bids formulation 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 at that node. Many techniques capitalize on that possibility, as we will see later in this chapter.

2.1.3 Multivariate branching

It is known in the integer programming literature that search algorithms can be made to branch on questions that include multiple variables. That idea can be applied to winner determination as follows (Gilpin and Sandholm 2004). The algorithm can branch on the sum of the values of a set of variables.[5] The branching question could then be, for instance: "Of these 11 bids, are at least 3 winners?"

Consider the remaining problem at a node (it includes only those bids that do not share items with bids that are already accepted on the path). Relaxing the integrality constraints $x_j \in \{0, 1\}$ of that remaining winner determination problem to $x_j \in [0, 1]$ yields a linear program (LP) and it can be solved quickly. Given a set $\mathcal{X}$ of variables and the LP solution $\hat{x}$, one can generate the following two branches:

$$\sum_{i \in \mathcal{X}} x_i \leq k \quad \text{and} \quad \sum_{i \in \mathcal{X}} x_i \geq k + 1. \tag{5}$$

where $k = \left\lfloor \sum_{i \in \mathcal{X}} \hat{x}_i \right\rfloor$. No other value of k should be considered: any other integer value would cause one child to be exactly the same as the node, entailing infinitely deep search on that branch if that branch is ever explored. Similarly, no set of variables $\mathcal{X}$ where $\sum_{i \in \mathcal{X}} \hat{x}_i$ is integral should be a branching candidate: one of the branches will not exclude the current LP solution, so that child will be identical to the node.

While branching on more than one variable at a time may feel less powerful than branching on individual variables because the branch does not seem to make as specific a commitment, we have:

Proposition 2.4 *(Gilpin and Sandholm 2004) The search tree size (measured in terms of the number of nodes or number of leaves) is the same regardless of how many (and which) variables are used in different branches (as long as trivial branches where a child is identical to its parent are not used).*

Thus Theorem 2.3 applies to multivariate branching as well.

Proposition 2.4 is for exhaustive search. If additional techniques (discussed later in this chapter) are used, such as upper bounding, then search tree size can differ based on how many (and which) variables are used for branching at nodes. Experimentally, multivariate branching tends to lead to smaller trees than the other two search formulations discussed above, but it spends more time at each search node (deciding which set $\mathcal{X}$ should be branched on).

The different types of branching question classes could also be merged. For example, a search algorithm could branch on an item at a node, branch on a bid at another node, and branch on a multivariate question at yet another node.

2.2 Search strategies

The second important design dimension of a search algorithm is the *search strategy*: what order is the tree searched in? The following subsections cover the most pertinent search strategies.

2.2.1 Depth-first search

The first special-purpose search algorithm for winner determination (Sandholm 2002a) used the branch-on-items formulation and the *depth-first search* strategy, where the search always proceeds from a node to an unvisited child, if one exists. If not, the search backtracks. Depth-first search is desirable in that only the nodes on one search path (and their children) need to be kept in memory at any one time. So, $O((n + n_{dummy}) \cdot m)$ nodes are in memory.

The strategy yields an *anytime algorithm*: the algorithm is able to output a feasible solution at any time, and solution quality improves over time (because the algorithm keeps track of the best solution found so far (incumbent)). The user can stop the algorithm and use the incumbent if the algorithm is taking too long. In experiments, most of the revenue was generated early on as desired: there were diminishing returns to computation.

Search strategies that are *informed* by upper bounds yield faster winner determination. Specifically, a heuristic function h gives an upper bound on how much revenue the items that are not yet allocated on the current search path can contribute. The following subsections discuss search strategies that use such upper bounding. Later, Section 2.4 presents different ways of computing such bounds.

2.2.2 Depth-first branch-and-bound search

The simplest informed search strategy is *depth-first branch-and-bound (DF-BnB)*. It creates the search tree in depth-first order, but prunes (discontinues) a path if the node's $g + h$ value is no greater than the value of the incumbent. (Recall that g is the sum of the prices of the bids accepted on the path.) This is valid because the condition guarantees that a better incumbent cannot exist in the subtree whose creation was omitted by pruning. The memory usage of DFBnB is as low as that of depth-first search, and pruning leads to significantly smaller search trees. Fujishima, Leyton-Brown, and Shoham (1999) present DFBnB experiments in the branch-on-items formulation, and Sandholm, Suri, Gilpin, and Levine (2001) in the branch-on-bids formulation.

2.2.3 A* and best-bound search

The most famous informed search strategy is *A* search* (Hart, Nilsson, and Raphael 1968). When a search node is expanded, its children are generated and stored in memory. The node itself is removed from memory. The node to expand next is always the node from memory that is *most promising*, as measured by having the highest value of $g + h$. Once a node that has no children (because all bids are decided) comes up for expansion from memory, that is an optimal solution, and the search ends.

A* leads to the smallest possible search tree: no tree search algorithm can provably find an optimal solution without searching all the nodes that A* searches (Dechter and Pearl 1985). A downside is that A* often runs out of memory because the fringe of the search tree is stored. (With depth d and branching factor b, the number of nodes in memory is $O(b^d)$.)

Best-bound search, a common search strategy in the integer programming literature (Wolsey 1998), is identical to A*, except that the following refinements are often used in practice.

- **Approximate child evaluation:** In order to avoid having to carefully compute upper bounds on a node's children at the time when the children are first generated, only approximate upper bounds (that sometimes underestimate) for the children are used in practice. Therefore, the children come up for expansion from memory in *approximate* order of $g + h$. Thus, unlike in A*, once a node with no undecided bids comes up for expansion from memory, that might not be an optimal solution. Rather, the search must continue until all potentially better nodes have been expanded.

- **Diving bias:** A child of the current node is expanded instead of expanding the most promising node if the latter is not too much more promising. While this increases tree size, it can save search time because expanding a child is usually drastically faster than expanding a nonadjacent node. The reason is that data structures (e.g., LP-related ones) can be incrementally updated when moving between a node and a child, while they require significant reconstruction when moving to a nonadjacent node.[6] (All of the search strategies that proceed in depth-first order automatically enjoy the benefits of diving.)

2.2.4 Iterative deepening A search*

Like DFBnB, *iterative deepening A* (IDA*)* search (Korf 1985), achieves the same low memory usage, enjoys the benefits of diving, and takes advantage of upper bounds for pruning. IDA* guesses how much revenue (f-limit) can be obtained, and runs a depth-first search where a path is pruned if $g + h <$ f-limit. If a solution is not found, the guess was too optimistic, in which case a less optimistic guess is carefully made, depth-first search is executed again, and so on.

The following pseudocode shows how Sandholm (2002a) applied IDA* to winner determination. Instead of using depth-first search as the subroutine, this variant uses DFBnB (with an f-limit). The difference manifests itself only in the last IDA* iteration. That is the first iteration where an incumbent is found, and this variant of IDA* saves some search by using the incumbent's value for pruning.

Global variable: f-limit

Algorithm 2.1 (IDA*)
// Returns a set of winning bids that maximizes the sum of the bid prices

```
1. f-limit := $\infty$
2. Loop
    (a) winners, new-f := DFBNB-WITH-F-LIMIT($M$, $\emptyset$, 0)
    (b) if winners $\neq$ null then return winners
    (c) f-limit := $\min$(new-f, 0.95 $\cdot$ f-limit) [7]
```

Algorithm 2.2 DFBNB-WITH-F-LIMIT(*A, winners, g***)**
// Returns a set of winning bids and a new f-cost

```
1. If $g + h(A) <$ f-limit then return null, $g + h(A)$ // Pruning
2. If the current node has no children, then // End of a path reached
    (a) f-limit := $g$ // DFBnB rather than depth-first search
    (b) return winners, $g$
3. maxRevenue := 0, bestWinners := null, next-f := 0
```

```
4. For each bid $b \in$ children of current node
    (a) solution, new-f := DFBNB-WITH-F-LIMIT($A - S_b$, winners $\cup \{b\}$,
        $g + p_b$)
    (b) If solution $\neq$ null and new-f $>$ maxRevenue, then
        i. maxRevenue := new-f
        ii. bestWinners := solution
    (c) next-f := max(next-f, new-f)
5. If bestWinners $\neq$ null then return bestWinners, maxRevenue
   else return null, next-f
```

Experimentally, IDA* is two orders of magnitude faster for winner determination than depth-first search. By setting f-limit to 0 instead of ∞, IDA* turns into DFBnB. IDA* can lead to fewer search nodes than DFBnB because the f-limit allows pruning of parts of the search space that DFBnB would search. Conversely, DFBnB can lead to fewer nodes because IDA* searches nodes close to the root multiple times.

2.2.5 Exotic search strategies

IDA* and DFBnB explore a larger number of nodes than A*. Their run time can be improved by using more memory, while still using much less than A* does. Search strategies that do that include *SMA** (Russell 1992) and *recursive best-first search* (Korf 1993). A downside of SMA*, like A*, is that it requires node-related data structures to be laboriously reconstructed. A downside of recursive best-first search is that it leads to significant amounts of redundant search on problems where the edge costs of the search tree are mostly distinct numbers, as is the case in winner determination.

2.3 An example algorithm: CABOB

Let us now consider an example algorithm within which the other design dimensions of search algorithms for winner determination can be discussed specifically. The *CABOB (Combinatorial Auction Branch on Bids)* (Sandholm, Suri, Gilpin, and Levine 2001) algorithm is a DFBnB search that branches on bids.

The value of the best solution found so far (i.e., the incumbent) is stored in a global variable $\tilde{f}^*$. Initially, $\tilde{f}^* = 0$.

The algorithm maintains a conflict graph structure called the *bid graph*, denoted by G, see Figure 2. The nodes of the graph correspond to bids that are still available to be appended to the search path, that is, bids that do not include any items that have already been allocated. Two vertices in G share an edge whenever the corresponding bids share items.[8, 9] As vertices are removed from G when going down a search path, the edges that they are connected to are also removed. As vertices are re-inserted into G when backtracking, the edges are also reinserted.[10]

For readability, the following pseudocode of CABOB only shows how values are updated, and omits how the incumbent (set of winning bids) is updated in conjunction with every update of $\tilde{f}^*$.

As discussed later, CABOB uses a technique for pruning across independent subproblems (components of G). To support this, it uses a parameter, MIN, to denote the minimum revenue that the call to CABOB must return (not including the revenue from the path so far or from neighbor components) to be competitive with the incumbent. The revenue from the bids that are winning on the search path so far is called g. It includes the lower bounds (or actual values) of neighbor components of each search node on the path so far.

The search is invoked by calling $CABOB(G, 0, 0)$.

Algorithm 2.3 $CABOB(G, g, MIN)$

```
1. Apply cases COMPLETE and NO_EDGES (explained later)
2. Run depth-first search on $G$ to identify the connected components of $G$;
   let $c$ be number of components found, and let $G_1$, $G_2, ..., G_c$ be the $c$
   independent bid graphs
3. Calculate an upper bound $U_i$ for each component $i$
4. If $\sum_{i=1}^{c} U_i \leq MIN$, then return $0$
5. Apply case INTEGER (explained later)
6. Calculate a lower bound $L_i$ for each component $i$
7. $\Delta \leftarrow g + \sum_{i=1}^{c} L_i - \tilde{f}^*$
8. If $\Delta > 0$, then
      $\tilde{f}^* \leftarrow \tilde{f}^* + \Delta$
      $MIN \leftarrow MIN + \Delta$
9. If $c > 1$ then goto (11)
10. Choose next bid $B_k$ to branch on (use articulation bids first if any)
      10.a. $G \leftarrow G - \{B_k\}$
      10.b. For all $B_j$ s.t. $B_j \neq B_k$ and $S_j \cap S_k \neq \emptyset$,
            $G \leftarrow G - \{B_j\}$
      10.c. $\tilde{f}^*_{old} \leftarrow \tilde{f}^*$
      10.d. $f_{in} \leftarrow CABOB(G, g + p_k, MIN - p_k)$
      10.e. $MIN \leftarrow MIN + (\tilde{f}^* - \tilde{f}^*_{old})$
      10.f. For all $B_j$ s.t. $B_j \neq B_k$ and $S_j \cap S_k \neq \emptyset$,
            $G \leftarrow G \cup \{B_j\}$
      10.g. $\tilde{f}^*_{old} \leftarrow \tilde{f}^*$
      10.h. $f_{out} \leftarrow CABOB(G, g, MIN)$
```

```
        10.i. $MIN \leftarrow MIN + (\tilde{f}^* - \tilde{f}^*_{old})$
        10.j. $G \leftarrow G \cup \{B_k\}$
        10.k. Return $\max\{f_{in}, f_{out}\}$
11. $F^*_{solved} \leftarrow 0$
12. $H_{unsolved} \leftarrow \sum_{i=1}^{c} U_i$,    $L_{unsolved} \leftarrow \sum_{i=1}^{c} L_i$
13. For each component $i \in \{1, \ldots, c\}$ do
        13.a. If $F^*_{solved} + H_{unsolved} \leq MIN$, return 0
        13.b. $g'_i \leftarrow F^*_{solved} + (L_{unsolved} - L_i)$
        13.c. $\tilde{f}^*_{old} \leftarrow \tilde{f}^*$
        13.d. $f^*_i \leftarrow CABOB(G_i, g + g'_i, MIN - g'_i)$
        13.e. $MIN \leftarrow MIN + (\tilde{f}^* - \tilde{f}^*_{old})$
        13.f. $F^*_{solved} \leftarrow F^*_{solved} + f^*_i$
        13.g. $H_{unsolved} \leftarrow H_{unsolved} - U_i$
        13.h. $L_{unsolved} \leftarrow L_{unsolved} - L_i$
14. Return $F^*_{solved}$
```

2.4 Upper bounding techniques

As discussed, in all of the informed search methods, upper bounds on how much the unallocated items can contribute are used to prune the search (e.g., in CABOB in steps (3) and (4)). Pruning usually reduces the search time by orders of magnitude.

First-generation special-purpose winner determination algorithms (Sandholm 2002a, Fujishima, Leyton-Brown, and Shoham 1999) used special-purpose upper bounding techniques. The main idea was to use as an upper bound the sum over unallocated items of the item's maximum contribution (Sandholm

2002a):

$$\sum_{i \in A} c(i), \qquad \text{where} \;\; c(i) = \max_{j | i \in S_j} \frac{p_j}{|S_j|} \tag{6}$$

Tighter bounds are obtained by recomputing $c(i)$ every time a bid is appended to the path—because all bids j that share items with that bid can be excluded from consideration (Sandholm 2002a).[11]

The value of the linear program (LP) relaxation of the remaining winner determination problem gives another upper bound.

LP	*DUAL*
$\max \sum_{j=1}^{n} p_j x_j$	$\min \sum_{i=1}^{m} y_i$
$\sum_{j \mid i \in S_j} x_j \le 1, \; \forall i \in \{1..m\}$	$\sum_{i \in S_j} y_i \ge p_j, \; \forall j \in \{1..n\}$
$x_j \ge 0$	$y_i \ge 0$
$x_j \in \mathbb{R}$	$y_i \in \mathbb{R}$

The LP can be solved in polynomial time in the size of the input (which itself is $\Theta(nm)$) using interior point methods, or fast on average using, for example, the simplex method (Nemhauser and Wolsey 1999). Alternatively, one can use the DUAL because its optimal value is the same as LP's. Often the DUAL is used in practice because as the search branches, the parent's DUAL solution is feasible for the child's DUAL and can usually be optimized using a relatively small number of pivots.[12] (The parent's LP solution is infeasible for the child's LP, so the child's LP takes relatively long to solve).

It is not always necessary to run the LP/DUAL to optimality. The algorithm could look at the condition in step (4) of CABOB to determine the threshold revenue that the LP (DUAL) has to produce so that the search branch would not (would) be pruned. If the threshold is reached, LP/DUAL can stop. However, CABOB always runs the LP/DUAL to completion because CABOB uses the solutions for several other purposes beyond upper bounding, as discussed later.

Linear programming-based upper bounding usually leads to faster search times (Sandholm, Suri, Gilpin, and Levine 2001) than any of the other upper bounding methods proposed for winner determination before (Sandholm 2002a, Fujishima, Leyton-Brown, and Shoham 1999, Sandholm and Suri 2003). This is likely due to better bounding, better bid ordering, and the effect of the INTEGER special case, described below. The time taken to solve the linear program is greater than the per-node time with the other bounding methods, but the reduction in tree size usually amply compensates for that. However, on a non-negligible portion of instances the special-purpose bounding heuristics yield faster overall search time (Leyton-Brown 2003).

2.4.1 Branch-and-cut algorithms

The LP upper bound can be tightened by adding *cutting planes* (aka. *cuts*). These are additional constraints that do not affect the solution of the integer program, but do constrain the LP polytope. For example, if the bid graph G contains a set of nodes H that form an odd-length cycle longer than 3, and no nonadjacent pair of nodes in H share an edge (the nodes in H are said to form an odd hole), then it is valid to add the *odd-hole cut* $\sum_{j \in H} x_j \leq (|H| - 1)/2$. There are also cut families, for example *Gomory cuts* (Wolsey 1998), that can be applied to all integer programs, not just winner determination.

It is largely an experimental art as to which cuts, if any, are worth adding. That depends not only on the problem, but also on the instance at hand. The more cuts are added, the fewer nodes the search takes due to enhanced upper bounding. On the other hand, as more cuts are added, the time spent at each node increases due to the time it takes to generate the cuts and solve the LP that now has a larger number of constraints. There is a vast literature on cuts (see, for example, Garfinkel and Nemhauser (1969), Loukakis and Tsouros

(1983), Pardalos and Desai (1991), and the textbooks by Nemhauser and Wolsey (1999) and by Wolsey (1998)). One principle is to only add cutting planes that cut off the currently optimal point from the LP.

Some cuts are *global*: it is valid to leave them in the LP throughout the search. (Nevertheless, it is sometimes worth removing global cuts because they may slow down the LP too much.) Other cuts are *local*: they are valid in the subtree of the search rooted at a given node, but might not be valid globally. Such cuts can be added at the node, but they have to be removed when the search is not within that subtree.

A cut can also be made to constrain the LP polytope more, by carefully moving the cut deeper into the polytope (by including a larger number of the variables in the cutting plane and setting their coefficients), while guaranteeing that it does not cut off any integer solutions. This is called *lifting*. (See, for example, Wolsey (1998).) Again, there is a tradeoff: the more the algorithm lifts, the fewer search nodes are explored due to improved upper bounding, but the more time is spent per search node due to the time it takes to lift.

2.5 Lower bounding techniques and primal heuristics

Lower bounding techniques are another design dimension of search algorithms. At a search node (e.g., in CABOB in step (6)) a lower bound is computed on the revenue that the remaining items can contribute. If this bound is high, it allows the incumbent value, $\tilde{f}^*$, to be updated, leading to more pruning in the subtree rooted at that node. Generating good incumbents early is also desirable from an anytime perspective.

One famous lower bounding technique is rounding (Hoffman and Padberg 1993). CABOB uses the following rounding technique. In step (3), CABOB solves the remaining LP anyway, which gives an "acceptance level" $x_j \in [0, 1]$ for every remaining bid j. CABOB inserts all bids with $x_j > \frac{1}{2}$ into the lower bound solution. It then tries to insert the rest of the bids in decreasing order of x_j, skipping bids that share items with bids already in the lower bound solution. Experiments showed that lower bounding did not help significantly.

Other techniques for constructing a good feasible solution for a node include *local branching* where a tree search of at most k variable changes is conducted from some feasible solution to improve it (Fischetti and Lodi 2002), and *relaxation-induced neighborhood search* where variables that are equal in the feasible solution and LP are fixed, and a search is conducted to optimize the remaining variables (Danna, Rothberg, and Le Pape 2004) (the search can be restricted to k changes). In either method, k can be increased based on allowable time.

Additional lower bounds cannot hurt in terms of the number of search nodes because the search algorithm can use the best (i.e., highest) of the lower bounds. However, there is a tradeoff between reducing the size of the search tree and the time spent computing lower bounds.

2.6 Decomposition techniques

Decomposition techniques are another powerful tool in search algorithms. The idea is to partition the bids into sets (aka. connected components) so that no bid from one set shares items with any bid from any other set. Winner determination can then be conducted in each set separately (and in parallel if desired).

At *every* search node, in step (2) CABOB runs an $O(|E| + |V|)$ time depth-first search in the bid graph G. (Here, E is the set of edges in the graph, and V is the set of vertices.) Each tree in the depth-first forest is a connected component of G. Winner determination is then conducted in each component independently. Since search time is superlinear in the size of G, a decomposition leads to a time savings, and experiments have shown that the savings can be drastic (Sandholm, Suri, Gilpin, and Levine 2001).

2.6.1 *Upper and lower bounding* across components

Perhaps surprisingly, one can achieve further pruning by exploiting information across the components (Sandholm, Suri, Gilpin, and Levine 2001). When starting to solve a component, CABOB checks how much that component would have to contribute to revenue in the context of what is already known about bids on the search path so far *and the connected components that arose from decompositions on the path.* Specifically, when determining the MIN value for calling CABOB on a component, the revenue that the current call to CABOB has to produce (the current MIN value), is decremented by the revenues from solved neighbor components and the lower bounds from unsolved neighbor components. (A neighbor component is a connected component that arose at the same decomposition.) The use of the MIN variable causes the algorithm to work correctly even if on a single search path there are several search nodes where decomposition occurred, interleaved with search nodes where decomposition did not occur.

Every time a better global solution is found and $\tilde{f}^*$ is updated, all MIN values in the search tree are incremented by the amount of the improvement since now the bar of when search is useful has been raised. CABOB handles these updates without separately traversing the tree when an update occurs: CABOB directly updates MIN in step (8), and updates the MIN value of any parent node after the recursive call to CABOB returns.

CABOB also uses lower bounding across components. At any search node, the lower bound includes the revenues from the bids that are winning on the path, the revenues from the solved neighbor components of search nodes on the path, the lower bounds of the unsolved neighbor components of search nodes on the path, and the lower bound on the revenue that the unallocated items in the current search node can contribute.[13]

2.7 Techniques for deciding which question to branch on

The search formulations, search strategies, and the CABOB pseudocode leave open the question: "Which question should the search algorithm branch on at this node?" While any choice maintains correctness, different choices yield orders of magnitude difference in speed in the informed search methods. This section discusses techniques for making that choice.

2.7.1 Forcing a decomposition via articulation bids

In addition to checking whether a decomposition has occurred, CABOB strives for a decomposition. In the bid choice in step (10), it picks a bid that leads to a decomposition, if such a bid exists. Such bids whose deletion disconnects G are called *articulation bids.* Articulation bids are identified in $O(|E| + |V|)$ time by a slightly modified depth-first search in G (Sandholm and Suri 2003). If there are multiple articulation bids, CABOB branches on the one that minimizes the size of the largest connected component.

The strategy of branching on articulation bids may conflict with price-based bid ordering heuristics (which usually suggest branching on bids with high price and a low number of items). Does one of these schemes dominate the other?

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*

$$\arg\max_{j \in V} \; \nu(p_j, |S_j|), \tag{7}$$

where V is the set of vertices in the remaining bid graph G, and ν is a function that is nondecreasing in p_j and nonincreasing in $|S_j|$. Ties can be resolved arbitrarily, for example, preferring bids that articulate.

Theorem 2.5 *(Sandholm and Suri 2003) 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, and instances where the latter leads to less search.*

However, experiments showed that in practice it pays off to branch on articulation bids if they exist (because decomposition tends to reduce search drastically).

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

During the search, the algorithm could also do shallow lookaheads—for the purpose of bid ordering—to identify *combinations* (aka. cutsets) of bids that would disconnect G. Bids within a small cutset should be branched on first. (However, identifying the smallest cutset is intractable.)

To keep the computation at each search tree node linear time in the size of G, CABOB simply gives first priority to articulation bids, and if there are none, uses other bid ordering schemes, discussed in the next three subsecstions.

2.7.2 Should an algorithm branch on confidence or on uncertainty?

It has become clear to me that different search strategies are best served by different branching heuristics, and perhaps surprisingly, the best branching heuristics for A* and DFBnB abide to *opposite* principles.

If good anytime performance is desired, it makes sense to use the DFBnB search strategy. In that context it is best to generate promising branches first because that yields good solutions early, and as a side effect, better pruning of the search tree via upper bounding. So, the principle is that the algorithm should always *branch on a question for which it knows a good answer with high confidence.* For example, in the context of the branch-on-items formulation, Fujishima, Leyton-Brown, and Shoham (1999) renumbered the items before the search so that the items i were in descending order of $max_{j|i \in S_j} \frac{p_j}{|S_j|}$, and that was the branching order. Even better item ordering could be accomplished by *dynamically* reordering the remaining items for every subtree in the search—in light of what bids and items are still available.

On the other hand, if provable optimality is desired, A* tends to be preferable over DFBnB because it searches fewer nodes before proving optimality. Good variable ordering heuristics for A* are the opposite of those for DFBnB: the principle is that the algorithm should *branch on a question about whose correct answer the algorithm is very uncertain*! For example, the best-known branching heuristic in the operations research literature (e.g., (Wolsey 1998, page 99)) is the *most fractional variable heuristic.* In the branch-on-bids formulation of winner determination this translates to the *most fractional bid heuristic*: branching on a bid whose LP value, x_j, is closest to $\frac{1}{2}$ (Sandholm, Suri, Gilpin, and Levine 2001). The idea is that the LP is least sure about these bids, so it makes sense to resolve that uncertainty rather than to invest branching on bids about which the LP is "more certain". More often than not, the bids whose x_j values are close to 0 or 1 tend to get closer to those extreme values as search proceeds down a path, and in the end, LP will give an integer solution. Therefore those bids never end up being branched on.

In both algorithm families, to enhance pruning through upper bounding, it is best to visit the children of a node in most-promising-first order. (In A* this happens automatically due to the order in which nodes come up for expansion from memory, as discussed.) For example, within the branch-on-items formulation, Fujishima, Leyton-Brown, and Shoham (1999) used a child ordering heuristic where bids j are always visited in descending order of $\frac{p_j}{|S_j|}$. Even better child ordering could be accomplished by *dynamically* reordering the children at each search node—in light of what bids and items are still available. (For the particular child-ordering metric above, there is no difference between static and dynamic.)

2.7.3 Sophisticated numeric bid ordering heuristics

An elaborate study has been conducted on bid ordering heuristics for the branch-on-bids formulation with the DFBnB search strategy in the context of CABOB (Sandholm, Suri, Gilpin, and Levine 2001).[14] Experiments were conducted with the following bid ordering heuristics:

- *Normalized Bid Price (NBP) (Sandholm and Suri 2003):* Branch on a bid with the highest $\frac{p_j}{(|S_j|)^\alpha}$.[15]
- *Normalized Shadow Surplus (NSS) (Sandholm, Suri, Gilpin, and Levine 2001):* The problem with NBP is that it treats each item as equally valuable. It could be modified to weight different items differently based on static prices that, for example, the seller guesses before the auction. A more sophisticated approach is to weight the items by their "values" *in the remaining subproblem.* The *shadow price*, y_i, from the linear program DUAL of the remaining problem serves as a proxy for the value of item i.[16] We branch on the bid whose price gives the highest surplus above the value of the items[17] (normalized by the values so the surplus has to be greater if the bid uses valuable items):

$$\frac{p_j - \sum_{i \in S_j} y_i}{(\sum_{i \in S_j} y_i)^\alpha} \tag{8}$$

Experimentally, the following modification to the normalization leads to faster performance:

$$\frac{p_j - \sum_{i \in S_j} y_i}{\log(\sum_{i \in S_j} y_i)} \tag{9}$$

We call this scheme NSS.

- *Bid Graph Neighbors (BGN) (Sandholm, Suri, Gilpin, and Levine 2001):* Branch on a bid with the largest number of neighbors in the bid graph G. The motivation is that this will allow the search to exclude the largest number of still eligible bids from consideration.
- *Number of Items (Sandholm, Suri, Gilpin, and Levine 2001):* Branch on a bid with the largest number of items. The motivation is the same as in BGN.
- *One Bids (OB) (Sandholm, Suri, Gilpin, and Levine 2001):* Branch on a bid whose x_j-value from LP is closest to 1. The idea is that the more of the bid is accepted in the LP, the more likely it is to be competitive.
- *Most fractional bid*, described in the previous section. Branching heuristics of this type are most appropriate for A*-like search strategies (where the search strategy itself drives the search toward promising paths), while CABOB uses DFBnB.

Experiments were conducted on several problem distributions using all possible pairs of these bid ordering heuristics for primary bid selection and tie-breaking, respectively. Using a third heuristic to break remaining ties was also tried, but that never helped. The speed difference between CABOB with the best heuristics and CABOB with the worst heuristics was greater than two orders of magnitude. The best composite heuristic (OB+NSS) used OB first, and broke ties using NSS.

2.7.4 *Choosing the bid ordering heuristic* dynamically

On certain distributions, OB+NSS was best while on distributions where the bids included a large number of items, NSS alone was best. The selective superiority of the heuristics led to the idea of choosing the bid ordering heuristic *dynamically based on the characteristics of the remaining subproblem.* A distinguishing characteristic between the distributions was LP density:

$$\text{density} = \frac{\text{number of nonzero coefficients in LP}}{\text{number of LP rows} \times \text{number of LP columns}} \tag{10}$$

OB+NSS was best when density was less than 0.25 and NSS was best otherwise. Intuitively, when the LP table is sparse, LP is good at "guessing" which bids to accept. When the table is dense, the LP makes poor guesses (most bids are accepted to a small extent). In those cases the price-based scheme NSS (that still uses the shadow prices from the LP) was better. Therefore, in CABOB, at every search node the density is computed, and the bid ordering scheme is chosen dynamically (OB+NSS if density is less than 0.25, NSS otherwise).

2.7.5 *Solution seeding*

As a fundamentally different bid ordering methodology (Sandholm, Suri, Gilpin, and Levine 2001), stochastic local search (e.g., Hoos and Boutilier (2000))—or any other heuristic algorithm for winner determination—could be used to come up with a good solution fast, and then that solution could be forced to be the left branch (IN-branch) of CABOB's search tree. (The technique could be applied at the root, or also at other nodes.) Committing (as an initial guess) to the entire set of accepted bids from the approximate solution in this way would give CABOB a more global form of guidance in bid ordering than conducting bid ordering on a per-bid basis. To refine this method further, CABOB could take hints (for example from the approximation algorithm) as to how "surely" different bids that are accepted in the approximate solution should be accepted in the optimal solution. In the left branch (IN-branch) of CABOB, the "most sure" bids should then be assigned closest to the root of the search tree, because bids near the root will be the last ones to be backtracked in the search. This ordering will allow good solutions to be found early, and (mainly due to upper bounding) avoids unnecessary search later on.

2.7.6 Lookahead

A famous family of techniques for deciding what question to branch on is lookahead. (Some potentially promising subset of) candidate branching questions are considered, and a shallow search below the node is conducted for each one of those questions. The question with the highest "score" is then chosen to be the question to branch on.

Motivated by the goal of getting to a good solution quickly, a traditional scoring method is to take the weighted average of the leaves' values where more promising leaves are weighted more heavily (a leaf's value is $g + h$ where g is the sum of the prices of the accepted bids on that path, and h is an upper bound on how much remaining items can contribute) (Applegate, Bixby, Chvátal, and Cook 1994). This is called *strong branching.* A recent alternative idea, motivated by the role of branching as a means of reducing uncertainty, is to set the score to equal the expected reduction in *entropy* of the leaves' LP values (Gilpin and Sandholm 2004). This is called *information-theoretic branching.* Depending on the problem instance, either method can be superior.

These approaches can be used in all of the search formulations: branch-on-items, branch-on-bids, and multivariate branching.

2.8 Identifying and solving tractable subproblems at nodes

A general approach to speeding up the search is to, at each search node, solve the remaining problem in polynomial time using some special-purpose method rather than continuing search below that node, if the remaining problem happens to be tractable (Sandholm and Suri 2003).[18]

2.8.1 *COMPLETE case*

In step (1), CABOB checks whether the bid graph G is complete: $|E| = \frac{n(n-1)}{2}$. If so, only one of the remaining bids can be accepted. CABOB thus picks the bid with highest price, updates the incumbent if appropriate, and prunes the search path.

2.8.2 *NO_EDGES case*

If G has no edges, CABOB accepts all of the remaining bids, updates the incumbent if appropriate, and prunes the search path.

2.8.3 *INTEGER case*

A classic observation is that if the LP happens to return integer values ($x_j = 0$ or $x_j = 1$) for all bids j (this occurs surprisingly frequently), that solution can be used as the actual solution for the node in question (rather than having to search under that node). CABOB does this in step (5). (If only some of the x_j values are integral, one cannot simply accept the bids with $x_j = 1$ or reject the bids with $x_j = 0$ (Sandholm, Suri, Gilpin, and Levine 2001).)

Sufficient conditions under which the LP provides an integer-valued solution are reviewed by Müller (Chapter 13). For some of those classes there exists algorithms for identifying and solving the problem faster than LP, for instance the NO_EDGES class above, and the class where items can be numbered so that each remaining bid is for consecutive items (possibly with wraparound) (Sandholm and Suri 2003).

2.8.4 Tree-structured items

If the remaining items can be laid out as nodes in a graph so that each remaining bid is for a connected component in the graph, then winners can be determined in time that is exponential only in the treewidth of the graph (and polynomial in the number of items and the number of bids) (Conitzer, Derryberry, and Sandholm 2004). A polynomial algorithm for identifying whether such a graph with treewidth 1 (i.e., a tree) exists has been developed (Conitzer, Derryberry, and Sandholm 2004), so handling of that case can be integrated into tree search algorithms. The question of whether such graphs with treewidth greater than 1 can be identified in polynomial time remains open. (The requirement that each bid is for one component is sharp: even in line graphs, if two components per bid are allowed, winner determination is $\mathcal{NP}$-complete.)

2.8.5 Technique for exploiting **part** *of the remaining problem falling into a polynomially solvable class*

Polynomial solvability can be leveraged even if only *part* of the problem at a search node falls into a tractable class (Sandholm and Suri 2003). This section demonstrates this for one polynomially solvable class.

Bids that include a small number of items can lead to significantly deeper search than bids with many items because the latter exclude more of the other bids due to overlap in items (Sandholm 2002a). Furthermore, bids with a small number of items are ubiquitous in practice. Let us call bids with 1 or 2 items *short* and other bids *long*.[19] Winners can be optimally determined in $O(n^3)$ worst case time using a weighted maximal matching algorithm (Edmonds 1965) if the problem has short bids only (Rothkopf, Pekeč, and Harstad 1998).

To solve problems with both long and short bids efficiently, Edmonds's algorithm can be integrated with search as follows (Sandholm and Suri 2003).

We restrict the branch-on-bids search to branching on long bids only, so it never needs to branch on short bids. At every search node, Edmonds's algorithm is executed using the short bids whose items have not yet been allocated to any accepted long bids on the search path so far. Edmonds's algorithm returns a set of winning short bids. Those bids, together with the accepted long bids from the search path, constitute a candidate solution. If it is better than the incumbent, the candidate becomes the new incumbent.

This technique can be improved further by using a dynamic definition of "short" (Sandholm and Suri 2003). If an item x belongs to only one long bid b in the *remaining* bid graph G, then the size of b can, in effect, be reduced by one. As search proceeds down a path, this method may move some of the long bids into the short category, thereby further reducing search tree size. (When backtracking, the deleted items are reinserted into bids.)

2.9 Random restart techniques

Random restarts have been widely used in local search algorithms, but recently they have been shown to speed up tree search algorithms as well on certain problems (Gomes, Selman, and Kautz 1998). The idea is that if (mainly due to unlucky selection of branching questions) the search is taking a long time, it can pay off to keep trying the search again with different randomly selected branching questions. Sandholm, Suri, Gilpin, and Levine (2001) tested whether sophisticated random restart techniques, combined with careful randomized bid ordering, help in winner determination. The experiments showed that these techniques actually slow CABOB down.

2.10 Caching techniques

Another technique commonly used in search algorithms is *caching.* It can also be applied to search algorithms for winner determination. For example, Fujishima, Leyton-Brown, and Shoham (1999) used caching in the branch-on-items formulation. Here I present how caching can be used in the branch-on-bids formulation, but the same ideas apply to other search formulations.

If the remaining bid graph is the same in different nodes of the search tree (this occurs if the same items are used up, but by different sets of bids), then the remaining subproblems under those nodes are the same. The idea in caching is that the answer from solving the subproblem is stored (aka. cached) in memory so that when that subproblem is encountered again, it does not have to be solved anew—rather, the answer can be looked up from the cache.

One subtlety is that usually the subproblem is not searched completely when it is encountered, due to pruning through upper bounding. Therefore, in many cases only an upper bound h on the subproblem's value is stored in the cache rather than the exact value of the subproblem. The algorithm should therefore also store in the cache, with each solution, whether the value is exact or an upper bound. Then, if the same subproblem occurs in a search node elsewhere in the tree, there are several cases, to be checked in order:

- If the cached value is exact, it can be used as the value of the subproblem, and the search does not need to continue into the subtree. The remaining cases are for cached values that are not exact.
- If the new node has a greater (or equal) MIN-value than the node where the value of the same subproblem was cached (that is, the path to the new node is less promising than the path to the old node was), then the subtree can be pruned.[20] The following cases pertain if the new node has a lesser MIN-value than the old node.
- If the cached h-value is less than (or equal to) the MIN-value, the subtree can be pruned.
- If the cached h-value is greater than the MIN-value, then it is not valid to use the cached value because the optimal solution might be found in

the subtree. In that case, the search has to continue into the subtree. (Once the subtree is solved deeply enough to allow for pruning with that MIN-value—or an exact solution is found—the new solution to the subtree is stored in the cache in place of the old.)

For large search trees, there is not enough memory to cache the values of all subtrees. It is important to decide which subtree solutions the algorithm should cache (and which it should remove from the cache when the cache becomes full). There are at least three principles for this: 1) cache subproblems that are encountered often, 2) cache subproblems that represent a large amount of search effort, and 3) cache subproblems that can be indexed rapidly (i.e., quickly retrieve the value or determine that it is not in the cache).

3 State of the art

Despite the fact that winner determination is $\mathcal{NP}$-complete, modern search algorithms can optimally solve large CAs in practice. The time it takes to solve a problem depends not only on the number of items and bids, but also on the specific structure of the problem instance: which bids include which items, and what the prices of the bids are. The rest of this section summarizes some of the performance results; more detail can be found in Sandholm, Suri, Gilpin, and Levine (2001).

So far in the literature, the performance of winner determination algorithms has been evaluated on problem instances generated randomly from a variety of distributions (Sandholm 2002a, Fujishima, Leyton-Brown, and Shoham 1999, Andersson, Tenhunen, and Ygge 2000), most of which strive to be realistic in terms of how many items package bids tend to include, and in terms of bid prices. In some of the distributions (such as the CATS distributions (Leyton-Brown, Pearson, and Shoham 2000)—see also Chapter 18), the choice of items for each package bid is motivated by potential application domains.

On easier distributions (such as CATS and certain less structured distributions), optimal winner determination scales to hundreds or thousands of items, and tens of thousands of bids in seconds. On hard distributions (such as one where each package bid includes 5 randomly selected items and has price \$1), optimal winner determination only scales to tens of items and hundreds of bids in a minute.

It is interesting to see how the state-of-the-art general-purpose mixed integer program solvers, CPLEX and XPress-MP, fare against state-of-the-art algorithms specifically designed for winner determination like CABOB. All three use the branch-on-bids search formulation (but could be modified to use other formulations), and LP-based upper bounding. CPLEX and XPress-MP use cutting planes, while vanilla CABOB does not. None of the three use random restarts (except some versions of CABOB not compared here) or caching. CPLEX and XPress-MP only have general methods for deciding which variable to branch on, while CABOB uses custom schemes discussed above. CABOB has several techniques for identifying and solving tractable cases at search nodes; the other two use only the INTEGER case. CPLEX and XPress-MP use algebraic preprocessing while CABOB only uses a domination check between bids. (Sophisticated preprocessing techniques specifically for winner determination are discussed in Sandholm (2002a).)

CPLEX and XPress-MP often run out of memory on hard instances because their default strategy is best-bound search (with approximate child evaluation and diving bias). CABOB does not run out of memory due to its DFBnB search strategy.[21] The depth-first order and the tailored variable ordering heuristics also cause CABOB to have significantly better anytime performance than CPLEX. When measuring time to find the optimal solution and to prove its optimality, CPLEX tends to be somewhat faster on many random problem distributions, but CABOB is faster on other random problem distributions. On structured instances that are somewhat decomposable (even if they are not decomposable at the root of the search), CABOB is drastically faster than CPLEX due to its decomposition techniques, identifying and branching on articulation bids, and upper and lower bounding across components. XPress-MP tends to perform similarly to CPLEX.

On real-world winner determination problems—which usually have forms of expressiveness beyond package bids, such as side constraints from the bid taker and bidders, and price-quantity discount schedules—special-purpose search algorithms can be orders of magnitude faster than general-purpose solvers because there is more domain-specific knowledge that the former can capitalize on. Those problems also tend to lead to other important issues such as insufficient numeric stability of CPLEX and XPress-MP (telling whether a real number is an integer, equals another real, or exceeds another number), yielding incorrect answers in terms of feasibility and optimality. Simply tightening the tolerance parameters does not solve this: for certain purposes in those solvers the tolerances must be sufficiently loose.

Interestingly, it was recently shown (Schuurmans, Southey, and Holte 2001) that optimal search algorithms perform favorably in speed on winner determination even against incomplete search algorithms such as stochastic local search (e.g., Hoos and Boutilier (2000)) that do not generally find the optimal solution.

4 Substitutability and XOR-constraints

The winner determination methods discussed so far in this chapter, and most other work on winner determination (e.g., (Rothkopf, Pekeč, and Harstad 1998, DeMartini, Kwasnica, Ledyard, and Porter 1999)), are based on a setting where any number of a bidder's package bids may get accepted. This is called the *OR bidding language* (WDP_{OR} in Lehmann, Müller and Sandholm (Chapter 12)). It suffices when the items are *complementary*, that is, each bidder's valuation function is superadditive. However, when some of the items exhibit *substitutability* (valuations are subadditive), this language is insufficient. Say an agent bids \$4 for an umbrella, \$5 for a raincoat, and \$7 for both. The auctioneer could allocate both to that agent separately, and claim that the agent's bid for the combination would value at $\$5 + \$4 = \$9$ instead of \$7.

This problem was addressed in the context of the *eMediator* Internet auction server prototype (see `http://www.cs.cmu.edu/~amem/eMediator`), where the *XOR bidding language* was introduced (Sandholm 2002a, Sandholm 2002b) (WDP_{XOR} in Lehmann, Müller and Sandholm (Chapter 12)). In effect, each bidder submits exclusive-or (XOR) constraints among all her package bids, so at most one can be accepted. This enables bidders to express general preferences (with complementarity and substitutability): any valuation function $v : 2^m \to \mathbb{R}$. For example, a bidder in a 4-item auction may submit the following:

$$\begin{array}{l}
(\{1\},\$4) \text{ XOR } (\{2\},\$4) \text{ XOR } (\{3\},\$2) \text{ XOR } (\{4\},\$2) \text{ XOR} \\
(\{1,2\},\$8) \text{ XOR } (\{1,3\},\$6) \text{ XOR } (\{1,4\},\$6) \text{ XOR} \\
(\{2,3\},\$6) \text{ XOR } (\{2,4\},\$6) \text{ XOR } (\{3,4\},\$3) \text{ XOR} \\
(\{1,2,3\},\$10) \text{ XOR } (\{1,2,4\},\$10) \text{ XOR } (\{1,3,4\},\$7) \text{ XOR} \\
(\{2,3,4\},\$7) \text{ XOR } (\{1,2,3,4\},\$11)
\end{array}$$

Full expressiveness is important because it allows the bidders to express their valuations exactly, and winner determination uses that information to determine a socially optimal allocation of items. Without full expressiveness, economic efficiency is compromised. Full expressiveness is also important because it is a necessary and sufficient property of a bidding language for motivating truthful bidding (Sandholm 2002a, Sandholm 2002b). Without it, a bidder may not be able to express her preferences even if she wanted to. With it, the VCG mechanism can be used to make truthful bidding a dominant strategy. (VCG is described in Ausubel and Milgrom (Chapter 1).) In the VCG, winners are determined once overall, and once per winning agent without any of that agent's bids. That makes fast winner determination even more crucial. Note that just removing one winning bid at a time would not constitute a truth-promoting mechanism, and truth promotion can also be lost if winner determination is conducted approximately (Sandholm 2002a, Sandholm 2002b).

While the XOR bidding language is fully expressive, representing one's preferences in that language often requires a large number of package bids. To maintain full expressiveness, but at the same time to make the representation more concise, the *OR-of-XORs bidding language* was introduced (Sandholm 2002b). In this language, a set of bids can be combined with XOR, forming an *XOR-disjunct*. These XOR-disjuncts are then combined with non-exclusive ORs to represent independence. For example, a bidder who wants to submit the same offer as in the example above, can submit the following more concise expression:

$$\begin{array}{l} [(\{1\}, \$4)] \\ \text{OR} \\ [(\{2\}, \$4)] \\ \text{OR} \\ [(\{3\}, \$2) \;\; \text{XOR} \;\; (\{4\}, \$2) \;\; \text{XOR} \;\; (\{3,4\}, \$3)] \end{array}$$

The XOR bidding language is a special case of the OR-of-XORs bidding language. Therefore, the shortest way to represent a value function in the OR-of-XORs bidding language is never longer than in the XOR bidding language.

Another fully expressive bidding language is the *OR* bidding language* (Nisan 2000): items are thought of as nodes in a graph, and exclusion edges (i.e., XOR-constraints) can be submitted between arbitrary pairs of nodes. An equivalent encoding is to use the OR bidding language, and have mutual exclusion between a pair of bids encoded by a dummy item that each bid in the pair includes (Fujishima, Leyton-Brown, and Shoham 1999). (However, the latter does not work if (some) bids can be accepted partially.) Bidding languages are discussed in detail in Nisan (Chapter 9).

Search algorithms for winner determination can be easily adapted to handle XOR-constraints between bids (Sandholm 2002a, Sandholm and Suri 2003), be it in the XOR bidding language, the OR-of-XORs language, or the OR* language. If two bids are in the same XOR-disjunct, and one of them is accepted on the search path, the other should not be accepted on that path. This is easy to accomplish. For example, if the bid graph method is used, we simply add an extra edge into the graph for every pair of bids that is combined with XOR.

Constraints actually *reduce* the size of the search space. However, in practice they tend to make winner determination slower because many of the techniques (e.g., upper bounding) in the search algorithms do not work as well: a larger fraction of the search space ends up being searched. This is the case even when the upper bounding is improved by adding to the LP an extra constraint for each XOR-disjunct: $\sum_{j \in D} x_j \leq 1$, where D is the set of bids within the XOR-disjunct. Furthermore, the technique that avoids branching on short bids (Section 2.8.5) does not apply with explicit XOR-constraints, nor does the technique for tree-structured items (Section 2.8.4). If XOR-constraints are encoded using dummy items, those techniques apply, of course, but their effectiveness is compromised.

5 Conclusion

Optimal winner determination is important for economic efficiency and procedural fairness. Optimal winner determination, together with a fully expressive bidding language, is also needed to motivate truthful bidding. While winner determination is $\mathcal{NP}$-complete, modern tree search algorithms can provably optimally solve the problem in the large in practice. Therefore, CAs are now technologically feasible, and have started to be broadly fielded.

One practical issue in iterative CAs is that winners have to be determined multiple times (at the end of each "round", or in some designs even every time a new bid is submitted). In such settings, it is not desirable to solve winner determination anew every time. Instead, *incremental* algorithms use results from earlier winner determinations to speed up the current winner determination (Sandholm 2002a, Parkes and Ungar 2000, Kastner, Hsieh, Potkonjak, and Sarrafzadeh 2002).

Another desideratum of an iterative CA is the ability to provide quotes: "What would I have to bid in order to win bundle S (assuming no further bids are submitted)?" Quotes are subtle in CAs because there generally are no accurate item prices (so bundles have to be priced), and as further bids are submitted, the quote on a given bundle can increase *or decrease* (Sandholm 2002a). Furthermore, computing a quote for a given bundle is $\mathcal{NP}$-complete. Algorithms for quote computation, incremental quote computation, and for computing upper and lower bounds on quotes are presented in Sandholm (2002a).

Many of the techniques of this chapter apply to related and generalized combinatorial markets as well, such as CAs where there are multiple indistinguishable units of each item (Sandholm 2002b, Sandholm and Suri 2003, Leyton-Brown, Tennenholtz, and Shoham 2000, Gonen and Lehmann 2000, Lehmann and Gonen 2001, Sandholm, Suri, Gilpin, and Levine 2002), *combinatorial reverse auctions* (where there is one buyer who tries to fulfill his demand using combinatorial bids from multiple sellers) (Sandholm, Suri, Gilpin, and Levine 2002), and *combinatorial exchanges* (where there are multiple buyers and multiple sellers) (Sandholm 2002b, Sandholm and

Suri 2003, Sandholm, Suri, Gilpin, and Levine 2002). Many of the techniques can also be generalized to capture reserve prices on items, on bundles, and with substitutability (Sandholm and Suri 2003). Many of them can also be used when there is no free disposal (Sandholm 2002a, Sandholm and Suri 2003), but winner determination becomes harder (Sandholm, Suri, Gilpin, and Levine 2002). Finally, the algorithms can be generalized to markets with side constraints (from buyers and/or sellers) and non-price attributes (Sandholm and Suri 2001, Davenport and Kalagnanam 2001). Interesting winner determination issues arise also in the setting where bids can be accepted fractionally (Kothari, Sandholm, and Suri 2003).

Acknowledgments

Parts of this research were funded by CombineNet, Inc., and parts by NSF CAREER Award IRI-9703122 and NSF grants IIS-9800994, ITR IIS-0081246, ITR IIS-0121678, and ITR IIS-0427858.

Notes

[1]These algorithms inherit ideas from search algorithms for related problems such as weighted set packing, weighted maximum clique, and weighted independent set (e.g., (Balas and Yu 1986, Babel and Tinhofer 1990, Babel 1991, Balas and Xue 1991, Nemhauser and Sigismondi 1992, Mannino and Sassano 1994, Balas and Xue 1996, Pardalos and Desai 1991, Loukakis and Tsouros 1983)).

[2]Search algorithms only construct those parts of the search space that are necessary to construct in light of the bids submitted. The method of enumerating exhaustive partitions of items and (at least the naïve execution of) the dynamic programming method—discussed in Lehmann, Müller and Sandholm (Chapter 12)—construct, in effect, the entire search space as if each combination of items had been bid on. That is drastically slower (except when the number of items is tiny).

[3]The correct children can be found quickly using a secondary search in a binary trie data structure (Sandholm 2002a), or using binning techniques (Garfinkel and Nemhauser 1969, Fujishima, Leyton-Brown, and Shoham 1999). The latter approach requires items to be numbered statically, thus reducing the efficiency of item ordering heuristics (discussed later).

[4]Due to limited time and effort of the bidders, the number of bids in all but the tiniest CAs is much smaller than the number of bundles in practice.

[5]More generally, any hyperplane—with (positive or negative) coefficients on the variables—could be used as the branching question.

[6]In principle, such reconstruction could be avoided by maintaining a copy of the data structures with each search node, but that causes the search to usually run out of memory rapidly, and is thus not done in state-of-the-art mixed integer programming solvers.

[7]There is no reason to use an f-limit that is higher than the highest value of f that was lower than the f-limit in the previous IDA* iteration. The 0.95 criterion was used to decrease the f-limit even more quickly. If it is decreased too rapidly, search time increases because the last iteration will have a large number of search nodes in DFBNB-WITH-F-LIMIT. If it is decreased too slowly, search time increases because each iteration repeats a large portion of the search from the previous iteration.

[8]The bid graph can be constructed incrementally as bids are submitted.

[9]The bid graph can be prohibitively large to store if the problem is huge. One can address this by generating the graph explicitly only for those subtrees of the search tree where the graph is small (Sandholm, Suri, Gilpin, and Levine 2001).

[10]Sandholm and Suri (2003) present data structures for representing the bid graph in a way that supports efficient removal and addition of a bid's neighbors (and the connecting edges). Efficient conflict graph data structures for cut generation are presented in Atamtürk, Nemhauser, and Savelsbergh (2000).

[11]Fast ways of maintaining such upper bounds are discussed in Sandholm and Suri (2003), and approximations of this heuristic in Fujishima, Leyton-Brown, and Shoham (1999).

[12]CABOB does not make copies of the LP table. Instead, it incrementally deletes (reinserts) columns corresponding to the bids being deleted (reinserted) in the bid graph G as the search proceeds down a path (backtracks).

[13]Due to upper and lower bounding across components (and due to updating of $\tilde{f}^*$), the order of tackling the components can potentially make a difference in speed.

[14]In terms of ordering the answers to try for a given question (aka. *value ordering*), the basic version of CABOB always tries the IN-branch first. The reason is that it tries to include good bids early so as to find good solutions early. This enables more pruning through upper bounding. It also improves the anytime performance. On the other hand, the most prominent general-purpose mixed integer program solvers, CPLEX and XPress-MP, (being variants of best-bound search) sometimes try the OUT-branch first. Future research could experiment with that in CABOB as well.

[15]$\alpha = 0$ (selecting a bid that has highest price) gives too much preference to bids with many items. Such bids are likely to use up a large number of items, thus reducing significantly the revenue that can be collected from other bids. Conversely, it seems that $\alpha = 1$ (selecting a bid with the highest per-item price) gives too much preference to bids with few items. If there are two bids with close to equal per-item price, it would be better to choose a bid with a larger number of items so that the high per-item revenue could be obtained for many items. Experimentally, $\alpha \in [0.8,\ 1]$ yields fastest performance (Sandholm, Suri, Gilpin, and Levine 2001).

[16]In the binary case (where bids have to be accepted entirely or not at all), individual items cannot generally be given prices (in a way that would motivate bidders to self-select packages so that the overall optimal allocation is achieved), but each y_i value from the continuous DUAL gives an upper bound on the price of item i. (The solution to DUAL is generally not unique).

[17]A similar numerator can be used for a different purpose in another technique called *column generation*, which is sometimes used in other search applications (Barnhart, Johnson, Nemhauser, Savelsbergh, and Vance 1998). It is best-suited when the problem is so huge that not even a single LP fits in memory. CAs of that magnitude do not exist currently.

[18]If the identification and solving using the special-purpose method is still slow, one could use it in selected search nodes only.

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

[20]One could generalize the application of caching using the observation that an upper bound on a set of bids is also an upper bound on any subset of those bids (and a lower bound is also a lower bound on any superset).

[21]CPLEX and XPress-MP also support DFBnB, but that option usually makes them slower.

References

Andersson, Arne, Mattias Tenhunen, and Fredrik Ygge (2000), "Integer Programming for Combinatorial Auction Winner Determination," in *International Conference on Multi-Agent Systems*, pp. 39–46, Boston.

Applegate, David, Robert Bixby, Vasek Chvátal, and William Cook (1994), "The traveling salesman problem," Discussion paper, DIMACS.

Atamtürk, Alper, George Nemhauser, and Martin Savelsbergh (2000), "Conflict Graphs in Solving Integer Programming Problems," *European Journal of Operational Research*, 121, 40–55.

Babel, Luitpold (1991), "Finding Maximal Cliques in Arbitrary and Special Graphs," *Computing*, 46, 321–341.

Babel, Luitpold, and Gottfried Tinhofer (1990), "A Branch and Bound Algorithm for the Maximum Weighted Clique Problem," *ZOR - Methods and Models of Operations Research*, 34, 207–217.

Balas, Egon, and Jue Xue (1991), "Minimum Weighted Coloring of Triangulated Graphs, with Application to Maximum Weighted Vertex Packing and Clique Finding in Arbitrary Graphs," *SIAM Journal on Computing*, 20, 209–221.

Balas, Egon, and Jue Xue (1996), “Weighted and Unweighted Maximum Clique Algorithms with Upper Bonds from Fractional Coloring,” *Algorithmica*, 15, 397–412.

Balas, Egon, and Chang Sung Yu (1986), “Finding a Maximum Clique in an Arbitrary Graph,” *SIAM Journal on Computing*, 15, 1054–1068.

Barnhart, Cynthia, Ellis Johnson, George Nemhauser, Martin Savelsbergh, and Pamela Vance (1998), “Branch-and-price: column generation for solving huge integer programs,” *Operations Research*, 46, 316–329.

Boutilier, Craig (2002), “Solving Concisely Expressed Combinatorial Auction Problems,” in *National Conference on Artificial Intelligence*, pp. 359–366, Edmonton.

Conitzer, Vincent, Jonathan Derryberry, and Tuomas Sandholm (2004), “Combinatorial Auctions with Structured Item Graphs,” in *National Conference on Artificial Intelligence*, pp. 212–218, San Jose.

Danna, Emilie, Edward Rothberg, and Claude Le Pape (2004), “Exploring relaxation induced neighborhoods to improve MIP solutions,” *Mathematical Programming*, Forthcoming.

Davenport, Andrew J, and Jayant Kalagnanam (2001), “Price Negotiations for Procurement of Direct Inputs,” Discussion Paper RC 22078, IBM.

de Vries, Sven, and Rakesh Vohra (2003), “Combinatorial Auctions: A Survey,” *INFORMS Journal on Computing*, 15, 284–309.

Dechter, Rina, and Judea Pearl (1985), “Generalized best-first search strategies and the optimality of A*,” *Journal of the ACM*, 32, 505–536.

DeMartini, Christine, Anthony Kwasnica, John Ledyard, and David Porter (1999), "A New and Improved Design For Multi-Object Iterative Auctions," Discussion Paper 1054, CalTech, Social Science.

Edmonds, Jack (1965), "Maximum matching and a polyhedron with 0,1 vertices," *Journal of Research of the National Bureau of Standards*, B, 125–130.

Fischetti, Matteo, and Andrea Lodi (2002), "Local branching," *Mathematical Programming*, 98, 23–47.

Fujishima, Yuzo, Kevin Leyton-Brown, and Yoav Shoham (1999), "Taming the Computational Complexity of Combinatorial Auctions: Optimal and Approximate Approaches," in *International Joint Conference on Artificial Intelligence*, pp. 548–553, Stockholm.

Garfinkel, Robert, and George Nemhauser (1969), "The Set Partitioning Problem: Set Covering with Equality Constraints," *Operations Research*, 17, 848–856.

Gilpin, Andrew, and Tuomas Sandholm (2004), "Information-Theoretic Approaches to Branching in Search," Mimeo.

Gomes, Carla, Bart Selman, and Henry Kautz (1998), "Boosting Combinatorial Search Through Randomization," in *National Conference on Artificial Intelligence*, Madison.

Gonen, Rica, and Daniel Lehmann (2000), "Optimal Solutions for Multi-Unit Combinatorial Auctions: Branch and Bound Heuristics," in *ACM Conference on Electronic Commerce*, pp. 13–20, Minneapolis.

Hart, Peter, Nils Nilsson, and Bertram Raphael (1968), "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, 4, 100–107.

Hoffman, Karla, and Manfred Padberg (1993), “Solving Airline Crew-Scheduling Problems by Branch-and-Cut,” *Management Science*, 39, 657–682.

Hoos, Holger, and Craig Boutilier (2000), “Solving Combinatorial Auctions using Stochastic Local Search,” in *National Conference on Artificial Intelligence*, pp. 22–29, Austin.

Kastner, Ryan, Christina Hsieh, Miodrag Potkonjak, and Majid Sarrafzadeh (2002), “On the Sensitivity of Incremental Algorithms for Combinatorial Auctions,” in *IEEE workshop on Advanced Issues of E-Commerce and Web-Based Information Systems.*

Korf, Richard (1985), “Depth-first iterative-deepening: An optimal admissible tree search,” *Artificial Intelligence*, 27, 97–109.

Korf, Richard (1993), “Linear-Space Best-First Search,” *Artificial Intelligence*, 62, 41–78.

Kothari, Anshul, Tuomas Sandholm, and Subhash Suri (2003), “Solving Combinatorial Exchanges: Optimality via a Few Partial Bids,” in *ACM Conference on Electronic Commerce*, pp. 236–237, San Diego.

Lehmann, Daniel, and Rica Gonen (2001), “Linear Programming Helps Solving Large Multi-unit Combinatorial Auction,” Mimeo, Leibniz Center for Research in Computer Science, Hebrew University.

Leyton-Brown, Kevin (2003), “Resource Allocation in Competitive Multiagent Systems,” Ph.D. thesis, Stanford University.

Leyton-Brown, Kevin, Mark Pearson, and Yoav Shoham (2000), “Towards a Universal Test Suite for Combinatorial Auction Algorithms,” in *ACM Conference on Electronic Commerce*, pp. 66–76, Minneapolis.

Leyton-Brown, Kevin, Moshe Tennenholtz, and Yoav Shoham (2000), "An Algorithm for Multi-Unit Combinatorial Auctions," in *National Conference on Artificial Intelligence*, Austin.

Loukakis, Emmanuel, and Constantine Tsouros (1983), "An Algorithm for the Maximum Internally Stable Set in a Weighted Graph," *International Journal of Computer Mathematics*, 13, 117–129.

Mannino, Carlo, and Antonio Sassano (1994), "An Exact Algorithm for the Maximum Stable Set Problem," *Computational Optimization and Application*, 3, 242–258.

Nemhauser, George, and Gabriele Sigismondi (1992), "A Strong Cutting Plane/Branch-and-Bound Algorithm for Node Packing," *Journal of the Operational Research Society*, 43, 443–457.

Nemhauser, George, and Laurence Wolsey (1999), *Integer and Combinatorial Optimization*, John Wiley & Sons.

Nisan, Noam (2000), "Bidding and Allocation in Combinatorial Auctions," in *ACM Conference on Electronic Commerce*, pp. 1–12, Minneapolis.

Pardalos, Panos, and Nisha Desai (1991), "An Algorithm for Finding A Maximum Weighted Independent Set in An Arbitrary Graph," *International Journal of Computer Mathematics*, 38, 163–175.

Parkes, David, and Lyle Ungar (2000), "Iterative combinatorial auctions: Theory and practice," in *National Conference on Artificial Intelligence*, pp. 74–81, Austin.

Rothkopf, Michael, Aleksandar Pekeč, and Ronald Harstad (1998), "Computationally Manageable Combinatorial Auctions," *Management Science*, 44, 1131–1147.

Russell, Stuart (1992), "Efficient Memory-Bounded Search Methods," in *European Conference on Artificial Intelligence*, pp. 1–5, Vienna.

Russell, Stuart, and Peter Norvig (1995), *Artificial Intelligence: A Modern Approach*, Prentice Hall.

Sandholm, Tuomas (2002a), "Algorithm for Optimal Winner Determination in Combinatorial Auctions," *Artificial Intelligence*, 135, 1–54. Early versions: ICE-98 talk, WUCS-99-01 1/28/99, IJCAI-99.

Sandholm, Tuomas (2002b), "eMediator: A Next Generation Electronic Commerce Server," *Computational Intelligence*, 18, 656–676. Early versions: AGENTS-00, AAAI-99 Workshop on AI in Electronic Commerce, WU-CS-99-02 1/99.

Sandholm, Tuomas, and Subhash Suri (2001), "Side Constraints and Non-Price Attributes in Markets," in *IJCAI-2001 Workshop on Distributed Constraint Reasoning*, pp. 55–61, Seattle.

Sandholm, Tuomas, and Subhash Suri (2003), "BOB: Improved Winner Determination in Combinatorial Auctions and Generalizations," *Artificial Intelligence*, 145, 33–58. Early version: AAAI-00.

Sandholm, Tuomas, Subhash Suri, Andrew Gilpin, and David Levine (2001), "CABOB: A Fast Optimal Algorithm for Combinatorial Auctions," in *International Joint Conference on Artificial Intelligence*, pp. 1102–1108, Seattle. Forthcoming in *Management Science.*

Sandholm, Tuomas, Subhash Suri, Andrew Gilpin, and David Levine (2002), "Winner Determination in Combinatorial Auction Generalizations," in *International Conference on Autonomous Agents and Multi-Agent Systems*, pp. 69–76, Bologna. Early version: AGENTS-01 Workshop on Agent-Based Approaches to B2B.

Schuurmans, Dale, Finnegan Southey, and Robert Holte (2001), "The Exponentiated Subgradient Algorithm for Heuristic Boolean Programming," in *International Joint Conference on Artificial Intelligence*, pp. 334–341, Seattle.

van Hoesel, Stan, and Rudolf Müller (2001), "Optimization in electronic marketplaces: Examples from combinatorial auctions," *Netnomics*, 3, 23–33.

Wolsey, Laurence (1998), *Integer Programming*, John Wiley & Sons.