RESOURCE ALLOCATION IN COMPETITIVE MULTIAGENT SYSTEMS

ALGORITHMS FOR COMBINATORIAL AUCTION WINNER DETERMINATION

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Kevin Leyton-Brown
August 2003

Chapter 5

Combinatorial Auctions

5.1 Motivation

Auctions are the most widely studied mechanism in the mechanism design literature in economics and game theory [Fudenberg & Tirole, 1991]. This is due to the fact that auctions are basic protocols, serving as the building blocks of more elaborated mechanisms. Given the wide popularity of auctions on the Internet and the emergence of electronic commerce, where auctions serve as the most popular game-theoretic mechanism, efficient auction design has become a subject of considerable importance for researchers in multi-agent systems (*e.g.*, [Wellman *et al.*, 1998; Monderer & Tennenholtz, 2000]). The use of auctions in business-to-business trades is also increasing rapidly. Within AI there is growing interest in using auction mechanisms to solve resource allocation problems in competitive multiagent systems. For example, auctions and other market mechanisms are used in network bandwidth allocation, distributed configuration design, factory scheduling, and operating system memory allocation [S. H. Clearwater, 1996; Wellman, 1993].

5.1.1 Complementarity

The value of a good to a potential buyer can depend on what other goods s/he wins. We say that there exists *complementarity* between goods γ_1 and γ_2 to agent a if

$u_a(\{\gamma_1, \gamma_2\}) > u_a(\{\gamma_2\}) + u_a(\{\gamma_2\})$, where $u_a(S)$ is the utility to a of acquiring the set of goods S. If goods γ_1 and γ_2 were auctioned separately, it is likely that neither of the typically desired properties for auctions—efficiency or revenue maximization—would hold. One way to accommodate complementarity in auctions is to allow bids for combinations of goods as well as for individual goods. *Combinatorial auctions* are a class of auctions that accommodate bidders whose valuations exhibit complementarities: multiple goods are auctioned simultaneously and bidders place as many bids as they want for different bundles of goods with the guarantee that these bundles will be allocated "all-or-nothing". For example, imagine an auction of used electronic equipment. A bidder might value a particular TV at x and a particular VCR at y, but value the pair at $z > x + y$.

5.1.2 Substitutability

It is also common for bidders to desire a second good less once they have won a first good. We say that there exists *substitutability* between goods γ_1 and γ_2 to agent a when $u_a(\{\gamma_1, \gamma_2\}) < u_a(\{\gamma_1\}) + u_a(\{\gamma_2\})$. A common example of substitutability is for a bidder to be indifferent between several goods but not to want more than one. In order to be useful, a combinatorial auction mechanism should provide some way for bidders to indicate that goods (or, more generally, bundles of goods) are substitutable. Combinatorial auctions that allow the description of valuation functions involving both complementarity and substitutability can be seen as providing a general framework for allocation and decision-making problems among agents in competitive multiagent systems.

5.1.3 Applications

Combinatorial auctions are applicable to many real-world situations. In an auction for the right to use railroad segments a bidder desires a bundle of segments that connect two particular points; at the same time, there may be alternate paths between these points and the bidder needs only one [Brewer & Plott, 1996]. Similarly, in the FCC spectrum auction bidders may desire licenses for multiple geographical regions at the same frequency band while being indifferent to which particular band they receive [Milgrom, 1998]. These and other real-world applications of combinatorial auctions are described in more detail in Chapter 7. While economics and game theory provide many insights into the potential uses of such auctions, they have little to say about computational considerations. This is an obstacle to the practical use of combinatorial auctions, because it turns out that combinatorial auctions often give rise to very hard computational problems.

5.2 Combinatorial Auction Winner Determination

In a combinatorial auction, a seller is faced with a set of price offers for various bundles of goods, and his aim is to allocate the goods in a way that maximizes his revenue. The *winner determination problem* (WDP) is choosing the subset of bids that maximizes the seller's revenue, subject to the constraint that each good can be allocated at most once.

5.2.1 Formal definition

Let $G = \{\gamma_1, \gamma_2, \ldots, \gamma_m\}$ be a set of goods, and let $B = \{b_1, \ldots, b_n\}$ be a set of bids. Bid b_i is a pair $(p(b_i), g(b_i))$ where $p(b_i) \in \mathbb{R}^+$ is the price offer of bid b_i and $g(b_i) \subseteq G$ is the set of goods requested by b_i. For each bid b_i define an indicator variable x_i that encodes the inclusion or exclusion of bid b_i from the allocation.

Problem 5.1 *The* single-unit WDP *is the following integer program:*

$$\begin{array}{lll} \text{maximize:} & \sum_i x_i p(b_i) & \\ \text{subject to:} & \sum_{i \mid \gamma \in g(b_i)} x_i \leq 1 & \forall \gamma \in G \\ & x_i \in \{0, 1\} & \forall i \end{array}$$

5.2.2 XOR constraints

Problem 5.1 allows bidders to describe complementarities in their valuations; however, it does not explicitly allow the expression of substitutabilities. To do so, bidders must have some way of indicating that their interest in two bundles is mutually exclusive. Let $S = \{s_1, \ldots, s_\kappa\}$, where each s_i denotes a set of bids which are not allowed to be allocated together by the optimization algorithm.

Problem 5.2 *The* single-unit WDP with XOR constraints *is the following integer program:*

$$\begin{array}{lll} \text{maximize:} & \sum_i x_i p(b_i) & \\ \text{subject to:} & \sum_{i|\gamma \in g(b_i)} x_i \leq 1 & \forall \gamma \in G \\ & \sum_{i|b_i \in s_j} x_i \leq 1 & \forall j \in [1, \kappa] \\ & x_i \in \{0, 1\} & \forall i \end{array}$$

Some algorithms may not provide a way of specifying sets of mutually exclusive bundles s. Luckily, it is possible to use an encoding trick, introducing *dummy goods* [Fujishima *et al.*, 1999]. Dummy goods do not correspond to actual goods in the auction, but serve to enforce mutual exclusion between bids. For example, if bids b_1 and b_2 are intended to be mutually exclusive, we add a dummy good d to each bundle, defining new bids b'_i where $g(b'_i) = g(b_i) \cup d$. Since the good d can be allocated only once, at most one of b'_1 and b'_2 will allocated. More generally, it is possible to introduce n-unit dummy goods to enforce the condition that no more than n of a set of bids may be allocated (see Section 5.2.3 for a definition of units), and to use multiple dummy goods with the same bundles to enforce other complex constraints. This technique can lead to a combinatorial explosion in the number of bids if many goods are substitutable, but in many interesting cases this does not occur. While dummy goods increase the expressive power of the bidding language, making use of them requires no changes to an optimization algorithm. In fact, observe that Problem 5.2 is exactly the same as Problem 5.1 when XOR constraints are expressed using dummy goods. Hence, in what follows we do not discuss explicit XOR constraints, but assume that dummy goods are used where required.

5.2.3 Multi-unit auctions

In some cases, the set of goods at auction will contain subsets of goods among which all bidders are indifferent. We call these subsets *units* of a single good. While it is possible to auction each unit as a separate good, this forces bidders interested in a subset of the units to specify unnecessary XOR bids. For example, consider an electronics manufacturer auctioning 100 identical TVs and 100 identical VCRs. A retailer who wants to buy 70 TVs and 30 VCRs would be indifferent between all bundles having 70 TVs and 30 VCRs. Rather than having to bid on each of the $\binom{100}{70} \cdot \binom{100}{30}$ distinct bundles, she would prefer to place the single bid (price, {70 TVs, 30 VCRs}). This can be achieved by generalizing Problem 5.1. Let $q(\gamma)$ denote the number of units of good γ.

Problem 5.3 *The* multi-unit WDP *is the following integer program:*

$$\begin{array}{rll} \text{maximize:} & \sum_i x_i p(b_i) & \\ \text{subject to:} & \sum_{i \mid \gamma \in g(b_i)} x_i \le q(\gamma) & \forall \gamma \in G \\ & x_i \in \{0,1\} & \forall i \end{array}$$

5.2.4 Asymptotic Hardness

With or without XOR constraints, the WDP is equivalent to weighted set-packing and is therefore $\mathcal{NP}$-hard even in its single-unit variant: see *e.g.*, [Rothkopf *et al.*, 1998]. Furthermore, it is known that the WDP is inapproximable within any constant factor: see *e.g.*, [Sandholm, 1999].

5.3 Related Work on the WDP

In recent years many researchers have been interested in the combinatorial auction winner determination problem. For a survey, see [de Vries & Vohra, 2003].

5.3.1 Tractable Subcases

Rothkopf *et al.* [1998] identified the following subcases of the single-unit WDP which may be solved in polynomial time:

1. bids contain no more than two goods;
2. for any two bids, either they are disjoint or one is a subset of the other; or
3. bids only name goods that are consecutive given a one-dimensional ordering.[1]

More recent work on tractable subcases may be found in [Nisan, 2000; Tennenholtz, 2000; de Vries & Vohra, 2003]. The case of infinitely divisible goods may be solved in polynomial time by using linear programming techniques: we solve the linear programming relaxation of Problem 5.1, where the integrality constraint is replaced by the linear constraint $0 \leq x_i \leq 1$.

5.3.2 Approximation algorithms

Some researchers have studied algorithms to approximate the WDP, despite the fact that the WDP cannot be approximated with guarantees. For example, Hoos and Boutilier [2000] and Zurel and Nisan [2000] have proposed algorithms with good empirical performance despite their lack of theoretical guarantees. Furthermore others,

[1]In fact, this case can be extended to the case where goods are placed around a ring and each bid requests only consecutive goods. Consider adding each bid in turn (and removing all other bids that conflict with this bid): the remaining subproblem to be solved has only bids that request consecutive goods given a one-dimensional ordering, because the selection of the first bid breaks the ring.

notably Nisan and Ronen [2000] and Lehmann *et al.* [1999], have proposed alternative economic mechanisms that are built around approximation algorithms. It is also possible to make bidders responsible for improving the quality of the approximation. Banks *et al.* [1989] and Bykowsky *et al.* [1995] have reported a mechanism called AUSM in which non-winning bids are pooled in a stand-by queue. Bidders can combine their bids with other bids currently in the queue to form new allocations. A new allocation is adopted if it generates more revenue than the previously best allocation.

5.3.3 Solving the WDP to optimality

Although the WDP is $\mathcal{NP}$-hard, in practice it is possible to address interestingly-large datasets with heuristic methods [Fujishima *et al.*, 1999; Sandholm, 1999; Gonen & Lehmann, 2000; Leyton-Brown *et al.*, 2000b; Nisan, 2000; Sandholm & Suri, 2000; Gonen & Lehmann, 2001; Sandholm *et al.*, 2001]. Furthermore, there are reasons why it can be important to solve the WDP to optimality, and why restrictions to a tractable subcase may not be acceptable. For example, optimal solutions to the WDP are required in order for the Vickrey-Clarke-Groves mechanism [Mas-Colell *et al.*, 1995; Varian, 1995] give rise to dominant strategies. In fact, Nisan and Ronen [2000] show that the Vickrey-Clarke-Groves mechanism can give rise to arbitrarily bad outcomes when agents suspect that there is any possibility that any non-optimal solution to the WDP will be used. As another example, Parkes [1999], among others, has proposed an ascending auction mechanism that requires a provably-optimal solution to the WDP. Because of the importance of such applications, we concern ourselves in this work only with provably-optimal solutions to the WDP.

Chapter 6

Algorithms for Solving the Combinatorial Auction Winner Determination Problem

We present two branch-and-bound algorithms that exploit instances' particular bid structures to find optimal solutions to the WDP, using contextual information to make upper bounds tighter. Upper bounds are further tightened by online caching of results from unpruned subtrees. The first algorithm, CASS, considers only the single-unit WDP (*i.e.*, it solves WDP Problem 5.1). The second algorithm, CAMUS, generalizes CASS, addressing the multi-unit WDP (Problem 5.3).

6.1 CASS Algorithm

In this section we present an algorithm, Combinatorial Auction Structured Search (CASS), a branch-and-bound search algorithm with a novel heuristic. Most importantly, CASS structures the search space in a way that provides context to this heuristic in order to allow more pruning during the search and that avoids consideration of most infeasible allocations. CASS also caches the results of partial searches and prunes the search tree. Finally, CASS may be used as an anytime algorithm, as it tends to find good allocations quickly.

Before proceeding, we must introduce additional notation pertaining to allocations. An allocation $\pi \subseteq B$ is a subset of the bids where $\forall b_1 \in B, b_2 \neq b_1 \in B$, $g(b_1) \cap g(b_2) = \{\}$. We overload the functions $g()$ and $p()$ to apply to allocations: $g(\pi) = \bigcup_{b \in \pi} g(b)$ and $p(\pi) = \sum_{b in \pi} p(b)$. A full allocation π_{full} is an allocation for which $g(\pi_{full}) = G$, and a partial allocation is an allocation that is not full.

6.1.1 Dominated Bids

Some bids may be removed in a polynomial-time preprocessing step before search begins. For each pair of bids (b_1,b_2) where $g(b_1) \subseteq g(b_2)$ and $p(b_1) \geq p(b_2)$, we may remove b_2 from the list of bids to be considered during the search as b_2 is never preferable to b_1 (hence we say that b_1 *dominates* b_2).

6.1.2 Branch-and-Bound Search

Branch-and-bound search is a general search strategy that is widely used in the operations research community (see *e.g.*, [Nemhauser & Wolsey, 1988]). We explain it here using the terminology of combinatorial auctions.

Whenever a bid is encountered that does not conflict with the current partial allocation then the search tree branches, where one branch adds the bid to the partial allocation and the other does not. CASS performs a depth-first search, meaning that one branch of the tree is fully explored before the other is considered. (This has the advantage that CASS requires only linear space to store the search tree.) When a full allocation π is reached CASS records this allocation as π_{best} if $p(\pi) > p(\pi_{best})$, and then backtracks.

CASS also computes a function $h(\pi)$ at each node, which gives an upper bound on the revenue that can be collected from the goods $G \setminus g(\pi)$. When $h()$ indicates that the current subtree cannot lead to a solution better than π_{best} then the search tree can be *pruned*: we can backtrack before a full allocation has been constructed. More precisely, we backtrack whenever $p(\pi) + h(\pi) \leq p(\pi_{best})$. We will describe the construction of the function $h(\pi)$ in Section 6.1.4, but first we must introduce the concept of bins.

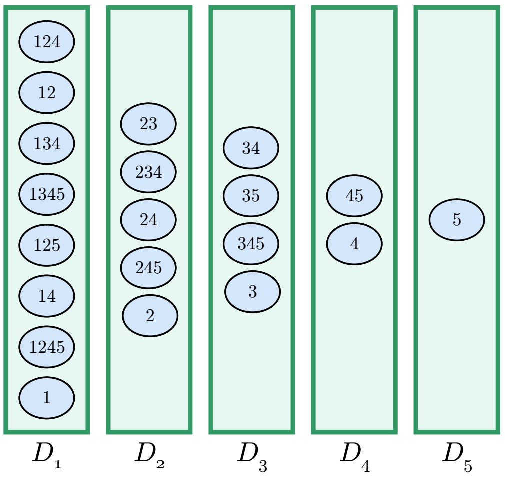


Figure 6.1: Partition into Bins

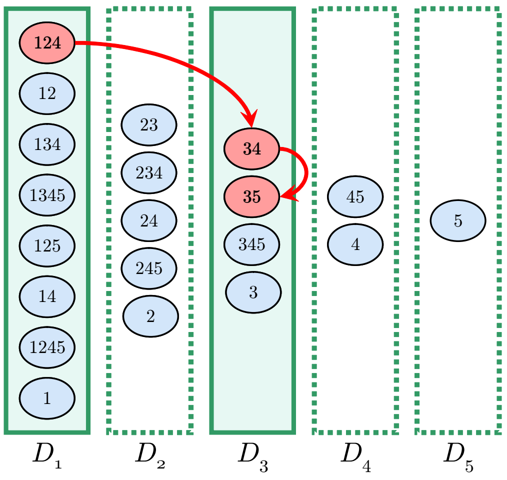


Figure 6.2: Skipping bins

6.1.3 Bins

We can reduce the number of infeasible allocations considered by partitioning bids into *bins*. First, we must choose an ordering on the goods. We create one bin for each good, and we place each bid into the bin corresponding to its lowest-order good. For an example, see Figure 6.1. The example shows input from an auction with five goods, $G = \{1, 2, 3, 4, 5\}$. Circles in the figure represent bids, the concatenated numbers in the circles represent the goods named in each bid, and prices are omitted from the bids for clarity.

Rather than always trying to add each bid to our allocation, we add at most one bid from every bin since all bids in a given bin are mutually exclusive. In fact, we can often skip bins entirely. While considering bin D_i, if we observe that good $j > i$ is already part of the allocation then we do not need to consider any of the bids in D_j. In general, instead of considering each bin in turn, skip to D_k where $k \notin g(\pi)$ and $\forall i < k$, $i \in g(\pi)$. To avoid the situation where it is possible to enter a bin that we cannot leave, we augment the set of bids. If there is no bid requesting a single unit of any good $\gamma_j \in G$ then we add a dummy bid $b = (0, \gamma_j)$.

Continuing our example, see Figure 6.2. The first bid we add contains goods $\{1, 2, 4\}$, so we skip all remaining bids in bin D_1 and all bids in bin D_2, since all of these bids will conflict. We thus consider bin D_3. The first bid we encounter requests goods $\{3, 4\}$, which also conflicts with our partial allocation. We move to the next bid in D_3, which requests goods $\{3, 5\}$. We have found our first full allocation, so we update our lower bound and backtrack.

The main benefit of bins is not the ability to avoid consideration of conflicting bids, however. Bins are powerful because they allow the pruning function to consider context without significant computational cost. If bids in bin D_i are currently being considered then the pruning function must only take into account bids in bins $\{D_i, \ldots, D_m\}$. Since most bids will belong to a low-order bin[1] but the search will spend most of its time in higher-order bins, this can allow us to generate very tight upper bounds. Furthermore, because the partitioning of bids into bins is fixed we can compute the upper bound information for each bin in a preprocessing step; this makes the upper bound fast to evaluate during search.

6.1.4 Upper Bound

We now describe the construction of the function $h(\pi)$. For every remaining good j we calculate a value $v(j)$, the maximum over all the remaining bids requesting good j that do not conflict with π of the bid's price divided by the number of goods requested by the bid. The sum of $v(j)$ values for all goods is an upper bound on optimal revenue because it relaxes the constraint that the bids in the optimal allocation may not conflict.

More formally, consider that we have built up a partial allocation π and reached bin D_σ. For each bid b_i, let $a(b_i) = p(b_i)/m$ be the average price per good of bid b_i. Notice that the average price per good may change dramatically from bid to bid, and it is a non-trivial notion; our technique will work for any arbitrary average price per good. Let $L_\sigma(j)$ be a list of the bids that refer to good j and that belong to bins D_ϕ with $\phi \geq \sigma$ (*i.e.*, the bids that can be encountered in the remainder of the search). The list is sorted in a monotonically decreasing manner according to the a_i's. Observe

[1]To see why bids are not spread evenly across the bins, consider what happens when we receive a bid for *every* bundle. Half of the bids will involve good γ_1 and will thus belong to bin D_1; half the bids that do not involve γ_1 will involve γ_2 and belong to bin D_2, and so on. Clearly, each bin D_{i+1} will contain half as many bids as its predecessor D_i.

that $L_\sigma(j)$ can be precomputed before search begins. $v(j)$ is defined as $a(b_j^{\text{ok}})$, where b_j^{ok} is the first bid in $L_\sigma(j)$ that does not conflict with π.

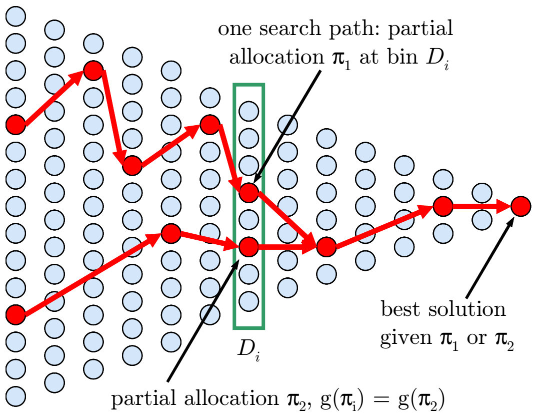
Theorem 6.1 *Let* $B^* = \{b_1^*, \ldots, b_s^*\}$ *be the bids in an optimal allocation. Then,* $r^* = \Sigma_{b \in B^*} p(b) \leq \Sigma_{1 \leq j \leq m} v(j)$.

Remark: In the multi-unit special case where when all goods happen to have only one unit, the upper bound function in Section 6.2.4 computes exactly the same upper bounds as the function presented above. Thus Theorem 6.2 considers a more general case, and so the proof to Theorem 6.1 can be deduced from that proof.

6.1.5 Caching

CASS's caching scheme is a form of dynamic programming that allows the algorithm to use experience from earlier in the search to tighten its upper bound function; it is illustrated in Figure 6.3. Consider a partial allocation π_1 that is reached during the search phase. If the search proceeds beyond π_1 then $h(\pi_1)$ was not sufficiently small to allow us to backtrack. Later in the search we may reach an allocation π_2 which, by combining different bids, achieves the same allocation π_1. CASS incorporates a mechanism for caching the results of the search beyond π_1 to generate a better estimate for the revenue given π_2 than is given by $h(\pi_2)$. (Since π_1 and π_2 do not differ in the units of goods that remain, $h(\pi_1) = h(\pi_2)$.) Consider all the allocations extending π_1 upon consideration of which the algorithm backtracked, denoted $s_1, \ldots, s_f$. When we backtracked at each s_i we did so because $p(s_i) + h(s_i) \leq p(\pi_{best})$, as explained above, or because s_i was a full allocation. It follows that $max_i(p(s_i) + h(s_i)) - p(\pi_1)$ is an overestimate of the revenue attainable beyond π_1, and that it is a smaller overestimate than $h(\pi_1)$. If it were not, we would have backtracked at π_1 rather than searching this subtree. We cache the value $c(g(\pi)) = max_i(p(s_i) + h(s_i)) - p(\pi_1)$ and backtrack when $p(\pi_2) + c(g(\pi_2)) \leq p(\pi_{best})$.

Our cache is implemented as a hash table, since caching is only beneficial to the overall search if lookup time is inconsequential. A consequence of this choice of data structure is that cache data may sometimes be overwritten; we overwrite an old entry



one search path: partial allocation π_1 at bin D_i
best solution given π_1 or π_2
D_i
partial allocation π_2, $g(\pi_1) = g(\pi_2)$

Figure 6.3: Caching

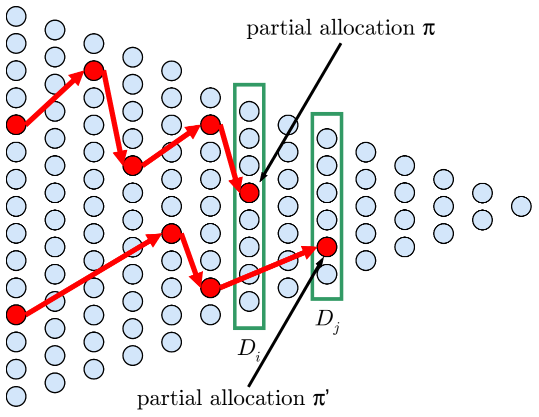


Figure 6.4: Cache Pruning

in the cache when the search associated with the new entry examined more nodes.[2] Even when we do overwrite useful data the error is not catastrophic, however: in the worst case we must simply search a subtree that we might otherwise have pruned. In order to reduce the cost of writing useless entries to the cache, and to reduce the chance that useful entries will be overwritten, partial allocations are only stored in the cache if the search tree below that point involved at least one backtrack. (Because of the exponential character of search trees, the vast majority of nodes that are eligible for caching will be one step away from leaf nodes, which means that caching these nodes would give a negligible performance gain.)

We can also use the cache to prune even when we reach a new partial allocation π_2 that has never been reached before, as shown in Figure 6.4. The search path is provably unable to lead to a new best allocation whenever $g(\pi_1) \subset g(\pi_2)$ and $p(\pi_2)+c(g(\pi_1)) \leq p(\pi_{best})$. (Since our cache is implemented as a hash table, we detect this case by checking the cache for each π_1 that differs from π_2 by the exclusion of each a single good; this requires a linear number of cache lookups.) In this case, the sum of the revenue from the cached path beyond π_1 and the revenue leading up to π_2 is less than the revenue from π_{best}. Note that since π_2 allocates a superset of the goods allocated in π_1, the goods in $g(\pi_2) \setminus g(\pi_1) \cap g(\pi_2)$ are counted both in $p(\pi_2)$

[2]We must, however, store $g(\pi)$ in the hash table along with $c(g(\pi))$, so that we can detect cache collisions.

and $c(g(\pi_1))$, corresponding to an overestimate of revenue.[3] Therefore, no allocation better than π_{best} could be found by expanding the search tree below π_2.

6.1.6 Good Ordering Heuristic

We must determine an ordering of the goods; that is, decide which good will correspond to the first bin, which will correspond to the second, etc. For each good i we compute $score_i = numbids_i/avggoods_i$, where $numbids_i$ is the number of bids that request good i and $avggoods_i$ is the average number of goods requested by these bids. We designate the lowest-order good as the good with the lowest score, then we recalculate the score for the remaining goods and repeat. The intuition behind this heuristic is as follows:

- We want to minimize the number of bids in low-order bins, to minimize early branching and thus to make each individual prune more effective.
- We want to maximize the total number of goods requested by bids in low-order bins. Taking these bids moves us more quickly towards the leaves of the search tree, again providing the pruning function with more contextual information.

6.1.7 Bid Ordering Heuristic

Our second heuristic determines the ordering of bids within bins. We sort bids dynamically—*i.e.*, bids may be ordered differently when there is a different partial allocation π. Bids are sorted in a given bin in descending order of $score(b_j)$, where (abusing notation slightly):

$$score(b_j) = \frac{p(b_j)}{|g(b_j)|} + h(\pi \cup b_j).$$

The intuition behind this heuristic is that the average price per good of b_j is a measure of how promising the bid is, while the upper bound $h(\pi \cup b_j)$ is an estimate of

[3]In fact, this bound can be tightened. Let $\mathcal{G} = g(\pi_2) \setminus g(\pi_1) \cap g(\pi_2)$. We can actually backtrack whenever $p(\pi_2) + c(g(\pi_1)) - \ell(\pi_2, \mathcal{G}) \leq p(\pi_{best})$, where $\ell(\pi_2, \mathcal{G})$ is a lower bound on the revenue that could be achieved from the goods $\mathcal{G}$ given the allocation π. A simple implementation of $\ell()$ would be the sum of the singleton bids for each of the goods in $\mathcal{G}$.

```
Process dominated bids.
Determine an ordering on the goods, according to the good-ordering
 heuristic.
Partition all bids into bins, according to the good ordering.
Precompute pruning information for each bin.
Set $i = 1$ and $\pi = \{\}$.
Recursive entry point:
    $\pi = \pi \cup b_j$.
    If $(p(\pi) + c(g(\pi)) \leq p(\pi_{best}))$ backtrack.
    If $(p(\pi) + h(\pi) \leq p(\pi_{best}))$ backtrack.
    If $(|goods(\pi)| = total)$ update $\pi_{best}$ if necessary; backtrack.
    Set $i$ to the index of the lowest-order good absent from $\pi$.
    Dynamically order the bids in bin $i$, and remove bids that
     conflict with $\pi$.
    Recurse to the recursive entry point above.
    $\pi = \pi \setminus b_j$.
Return the optimal allocation: $\pi_{best}$.
```

Figure 6.5: CASS Pseudocode

how promising the unallocated units are, given the partial allocation. This heuristic helps CASS to find good allocations quickly, improving anytime performance and also increasing π_{best}, making pruning more effective. Observe that dynamic reordering of the goods in each bin allows us to make use of an upper bound which depends on π.

6.2 CAMUS Algorithm

We now present a generalization of CASS which can solve Problem 5.3: the general multi-unit WDP. This algorithm is termed CAMUS (Combinatorial Auction Multi-Unit Search), and was introduced in [Leyton-Brown *et al.*, 2000b]. We explain characteristics of CAMUS that differ from CASS, and then give pseudocode for CAMUS.

In this section we make use of the following notation specific to the multi-unit case. Let $units(j)$ denote the total number of units of good j. Redefine the set of bids $B = \{b_1, \ldots, b_n\}$ so that bid b_i is a pair $(p(b_i), e(b_i))$, where $e(b_i) = (e(b_i)_1, e(b_i)_2, \ldots, e(b_i)_m)$ and $e(b_i)_j$ is the number of requested units of good γ_j in b_i. We overload the function *units* to refer to allocations: given an allocation π we denote the total number of units allocated as $units(\pi)$, and given both an allocation π and a good γ_i we denote the total number of units of good γ_i allocated in π by $units_i(\pi)$.

6.2.1 Dominated Bids

In the multi-unit case we must handle domination in a different way than we did in the single-unit case. We say that bid b_1 dominates b_2 whenever $p(b_1) \geq p(b_2)$ and $e(b_1)_j \leq e(b_2)_j$ for every good j. Although b_2 is never preferable to b_1, it is possible that an optimal allocation could contain both b_1 *and* b_2. For this reason we store b_2 in a secondary data structure associated with b_1, and consider adding it to those allocations which include b_1.

6.2.2 Subbins

In the multi-unit setting, we will often need to select more than one bid from a given bin. This leads to the idea of *subbins*. A subbin is a subset of the bids in a bin that is constructed during the search. Since subbins are created dynamically they cannot provide precomputed contextual information; rather, they facilitate the efficient selection of multiple bids from a given bin. Every time we add a bid to our partial allocation we create a new subbin containing the next set of bids to consider. If the search moves to a new bin, the new subbin is generated from the new bin by removing all bids that conflict with the current partial allocation. If the search remains in the same bin, the new subbin is created from the current subbin by removing conflicting bids as above, and additionally: if $b_1, b_2, \ldots, b_i$ is the ordered set of elements in the current subbin and b_j is the bid that was just chosen, then we remove all $b_k, k \leq j$. In this way we consider all combinations of non-conflicting bids in each bin rather than all permutations.

6.2.3 Dynamic Programming

Singleton bids (that is, bids that name units from only one good) deserve special attention. These bids will generally be among the most computationally expensive to consider—the number of nodes to search after adding a very short bid is nearly the same as the number of nodes to search after skipping the bid, because a short bid allocates few units and hence conflicts with few other bids. Unfortunately, we expect that singleton bids will be quite common in a variety of real-world multi-unit CA's. CAMUS simplifies the problem of singleton bids by applying a polynomial-time dynamic programming technique as a preprocessing step. We construct a vector $singleton_\gamma$ for each good γ, where each element of the vector is a set of singleton bids naming only good γ. $singleton_\gamma(j)$ evaluates to the revenue-maximizing set of singleton bids totaling j units of good γ. This frees us from having to consider singleton bids individually; instead, we consider only elements of the singleton vector and treat these elements as atomic bids during the search. Also, there is never a need to add more than one element from each singleton vector. To see why, imagine that we add both $singleton_\gamma(j)$ and $singleton_\gamma(k)$ to our partial allocation. These two elements may have bids in common, and additionally there may be singleton bids with more than $max(j,k)$ elements that would not conflict with our partial allocation but that we have not considered. Clearly, we would be better off adding the single element $singleton_\gamma(j+k)$.

We now show how to construct the *singleton* vector. Let $b_1, \ldots, b_\ell$ be bids for a single good γ. Our aim is to compute the optimal selection of b_i's in order to allocate k units of good γ, for $1 \leq k \leq units(\gamma)$. Consider a two dimensional grid of size $[1 \ldots \ell] \times [1 \ldots units(\gamma)]$ where the (i,j)-th entry, denoted by $U(i,j)$, is the optimal allocation of j units considering only bids $b_1, \ldots, b_i$. The value of $U(i,j)$, denoted by $V(i,j)$, is the sum of the price offers of the bids in $U(i,j)$. $U(1,j)$ will be b_1 if b_1 requests no more than j units, and otherwise will be the empty set. A recursive definition of $U(i,j)$ is given in Figure 6.6. This dynamic programming procedure is polynomial, and yields the desired result; the optimal allocation of k units is given by $U(\ell,k)$. Set $singleton_\gamma(k) = U(\ell,k)$, $1 \leq k \leq units(\gamma)$.

```
$e(b_i) > j$:
    $U(i,j) = U(i-1,j)$;
$e(b_i) = j$:
    if $p(b_i) > V(i-1,j)$
        then $U(i,j) = b_i$.
    Else $U(i,j) = U(i-1,j)$.
$e(b_i) < j$:
    if $V(i-1,j) \geq p(b_i) + V(i-1,j-e(b_i))$
        then $U(i,j) = U(i-1,j)$.
    Else $U(i,j) = b_i \cup U(i-1,j-e(b_i))$.
```

Figure 6.6: Singleton Pre-processing Algorithm

```
Let $v(j)$=0
Let $m(j)$=0
For $i = 1$ to $|L_\sigma(j)|$:
    if $m(j) < units_j(\pi)$ and $L_\sigma(j)_k \cap \pi = \emptyset$ then
        let $d := min(e(L_\sigma(j)_i)_j, units(j) - m(j))$
        $m(j) = m(j) + d$
        $v(j) = v(j) + a(L_\sigma(j)_i) \cdot d$
```

Figure 6.7: Upper Bound Algorithm

6.2.4 Upper Bound

CAMUS's upper bound function generalizes the CASS upper bound function described in Section 6.1.4, considering average price per unit rather than average price per good. Let π be the current allocation; recall that $units_i(\pi)$ denotes the number of available units of good j. Redefine $a(b_i) = \frac{p(b_i)}{\Sigma_{j=1}^m e(b_i)_j}$: the average price per unit of bid b_i. Define $L_\sigma(j)$ as before. Let $|L_\sigma(j)|$ denote the number of elements in $L_\sigma(j)$, and let $L_\sigma(j)_k$ denote the k^{th} element of $L_\sigma(j)$. $v(j)$ is determined by the algorithm given in Figure 6.7.

Theorem 6.2 *Let $B^* = \{b_1^*, b_2^*, \ldots, b_s^*\}$ be the bids in an optimal allocation. Then, $r^* = \Sigma_{b \in B^*} p(b) \leq \Sigma_{1 \leq j \leq m} v(j)$.*

Proof. Consider the bid $b^* \in B^*$. Then, $p(b^*) = \Sigma_{1 \leq j \leq m} a(b^*) e(b^*)_j$. Hence, $r^* = \Sigma_{b \in B^*} p(b) = \Sigma_{b \in B^*} \Sigma_{1 \leq j \leq m} a(b) e(b)_j$. By changing the order of summation we get that $r^* = \Sigma_{1 \leq j \leq m} \Sigma_{b \in B^*} a(b) e(b)_j$. Notice that, given a particular j, the contribution of bid b to $\Sigma_{b \in B^*} a(b) e(b)_j$ is $a(b) e(b)_j$. Recall now that $v(j)$ has been constructed from the set of all bids that refer to good j by choosing the maximal available units of good j from the bids in $L_\sigma(j)$, where these bids are sorted according to the average price per unit of good. Hence, we get $v(j) \geq \Sigma_{b \in B^*} a(b) e(b)_j$. Given that the above holds for every good j, this implies that $\Sigma_{1 \leq j \leq m} v(j) \geq \Sigma_{b \in B^*} p(b)$, as requested. ■

6.2.5 Heuristics

We must update our good- and bid-ordering heuristics for the multi-unit case. For our good-ordering heuristic we compute $score_i = \frac{numbids_i \cdot units(i)}{avgunits_i}$, where $numbids_i$ is the number of bids that request good i and $avgunits_i$ is the average number of *total* units (*i.e.*, not just units of good i) requested by these bids. The intuition here is similar to the intuition described in Section 6.1.6:

- We want to minimize the number of bids in low-order bins, to minimize early branching and thus to make each individual prune more effective.
- We want to minimize the number of units of goods corresponding to low-order bins, so that we will more quickly move beyond the first few bins. As a result, the pruning function will be able to take into account more contextual information.
- We want to maximize the total number of units requested by bids in low-order bins. Taking these bids moves us more quickly towards the leaves of the search tree, again providing the pruning function with more contextual information.

Given current partial allocation π, we sort bids in a given bin in descending order of $score(b_j)$, where $score(b_j) = \frac{p(b_j)}{units(b_j)} + h(\pi \cup b_j)$, which is a direct generalization of the heuristic discussed in Section 6.1.7.

6.3 Conclusions

We have presented CASS and CAMUS, algorithms that solve the single-unit and multi-unit WDPs respectively. In the next chapter, we discuss test data for evaluating such algorithms. We go on to evaluate CASS using this data in Chapter 8.

```
Process dominated bids.
Determine an ordering on the goods, according to the good-ordering
  heuristic.
Using the dynamic programming technique, determine the optimal
  combination of singleton bids totaling $1 \ldots units(j)$ for each good
  $j$.
Partition all non-singleton bids into bins, according to the good
  ordering.
Precompute pruning information for each bin.
Set $i = 1$ and $\pi = \{\}$.
Recursive entry point:
    For $j$ = 1 ...number of bids in the current subbin of
      $bin_i$.
        $\pi = \pi \cup b_j$.
        If $(p(\pi) + c(g(\pi)) \leq p(\pi_{best}))$ backtrack.
        If $(p(\pi) + h(\pi) \leq p(\pi_{best}))$ backtrack.
        If $(units(\pi) = \sum_{\gamma \in G} units(\gamma))$ record $\pi$ if it is the best;
          backtrack.
        Set $i$ to the index of the lowest-order good in $\pi$ where
          $units_i(\pi) < units(i)$.  ($i$ may or may not change)
        Construct a new subbin based on the previous subbin of $bin_i$
          (which is $bin_i$ itself if $i$ changed above):
            Include all $b_k$ from current subbin, where $k > j$.
            Include all dominated bids associated with $b_j$.
            Include $singleton_i(units(i) - units_i(\pi))$.
            Sort the subbin according to the subbin-ordering
              heuristic.
            Recurse to the recursive entry point, above, and search
              this new subbin.
        $\pi = \pi \setminus b_j$.
    End For
Return the optimal allocation: $\pi_{best}$.
```

Figure 6.8: CAMUS Pseudocode

Chapter 8

Evaluating Combinatorial Auction Algorithms

This chapter presents experimental results for CASS.[1] First, scaling experiments demonstrate that CASS can scale exponentially in the number of goods, but appears to scale subexponentially in the number of bids. Second, CASS's anytime performance is examined: CASS can find nearly-optimal solutions orders of magnitude sooner than it terminates, and can spend considerable time proving optimality after finding the optimal solution. Third, CASS is contrasted with the Bidtree algorithm, another widely-cited WDP algorithm. Finally, CASS is contrasted with the latest version of ILOG's CPLEX software.

8.1 Original CASS Experiments

All the experiments in this section were run a 450MHz Pentium II with 256MB of RAM running Windows NT 4.0. CASS was implemented in ANSI C++, and is publicly available. While these experiments were run on older hardware, they are still useful for gaining an understanding of the algorithm. They examine all legacy

[1]We do not present experimental results for CAMUS here, since Chapter 7 focused on single-unit distributions, and Chapter 9 will go on to consider only the single-unit WDP. For an experimental evaluation of CAMUS, please see [Leyton-Brown *et al.*, 2000b].

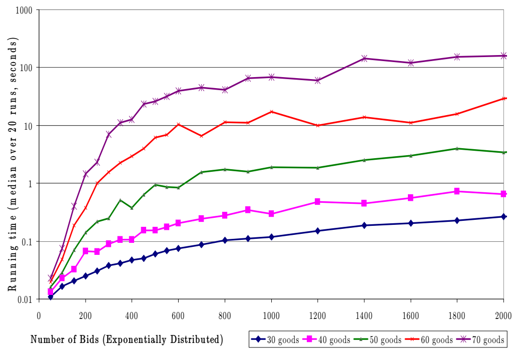


Figure 8.1: CASS Scaling: L6

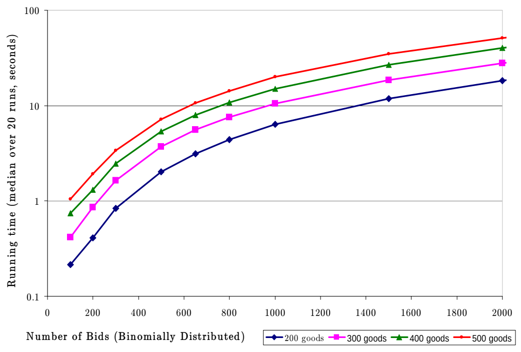


Figure 8.2: CASS Scaling: L7

distributions discussed in Section 7.3.7 except for L5 and L8.

8.1.1 Scaling Performance

The experiments in this section help us to understand CASS's performance as the number of bids[2] and goods is varied. Figures 8.1 and 8.2 show CASS's performance on the L6 and L7 distributions. Observe that for L6 runtime appears to scale exponentially in the number of goods (the four plots, representing linear increases in the number of goods, are spaced roughly equally on the log plot); for L7 runtime appears to scale sub-exponentially in the number of goods. For both L6 and L7 runtime clearly scales polynomially in the number of bids, as all the curves are sublinear on a linear axis.

8.1.2 Anytime Performance

Figure 8.3 shows CASS's anytime performance. Observe that the time it takes CASS to find the optimal solution is nearly of an order of magnitude smaller than the time at which it terminates: this is because *proving* optimality occupies most of the search. (Contrast this behavior with other optimization algorithms such as A*, which never finds the optimal solution before optimality is proven.) Also observe that the time it

[2]Unlike the data in Section 8.2 and in later chapters, the experiments in this section consider raw numbers of bids rather than numbers of undominated bids.

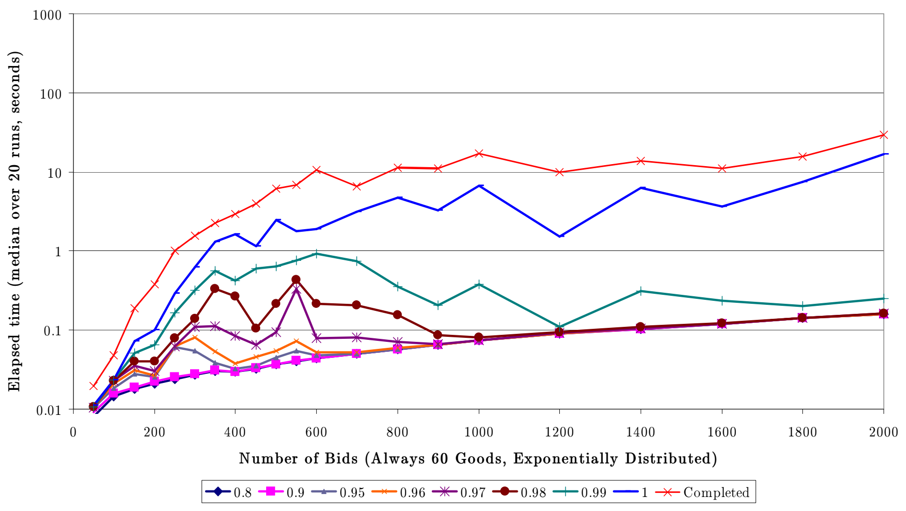


Figure 8.3: CASS Anytime Performance: L6

takes CASS to finds a solution within 99% of optimality is usually *much* smaller—as much as two orders of magnitude. This means that CASS can be very useful as an anytime algorithm even in situations where it does not terminate in a reasonable amount of time. Of course, however, there is no theoretical guarantee that CASS will always find a solution that is close to optimal.

8.1.3 CASS vs. Bidtree

Besides CASS, Bidtree is the other special-purpose WDP algorithm that has been most widely studied and cited in the literature. It was presented in the same conference proceedings as CASS [Sandholm, 1999]. The Bidtree algorithm is similar to CASS in several ways, but important differences hold. In particular, Bidtree performs a secondary depth-first search to identify non-conflicting bids, whereas CASS's structured approach provides context to the upper bound function as well as allowing it to avoid considering most conflicting bids. Bidtree also performs no caching or cache pruning. On the other hand, Bidtree uses an IDA* search strategy rather than CASS's branch-and-bound approach, and does more preprocessing.

Figures 8.4, 8.5, 8.6 and 8.7 contrast CASS and Bidtree's performance on the L1, L2, L3 and L4 distributions respectively. The Bidtree algorithm has never been made

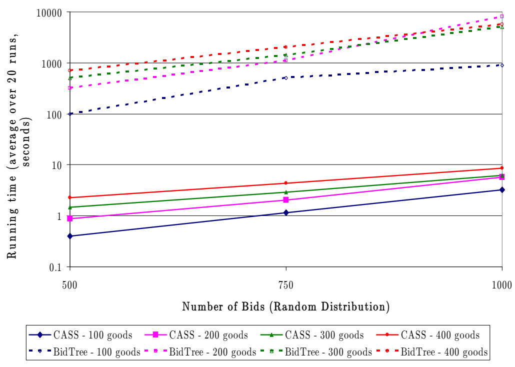


Figure 8.4: CASS vs. Bidtree: L1

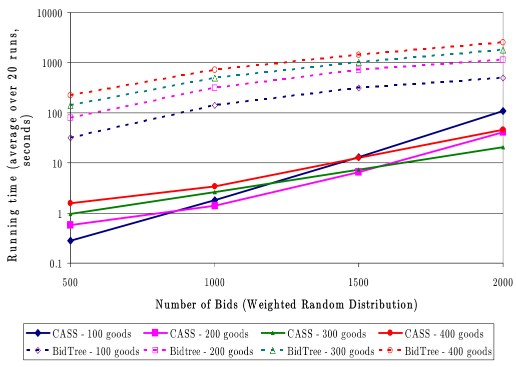


Figure 8.5: CASS vs. Bidtree: L2

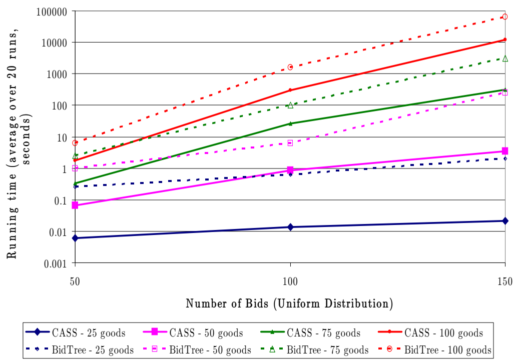


Figure 8.6: CASS vs. Bidtree: L3

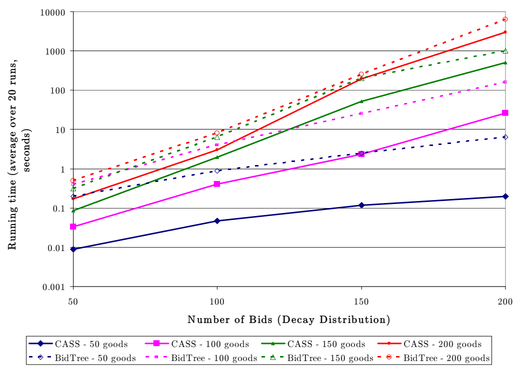


Figure 8.7: CASS vs. Bidtree: L4

publicly available; Bidtree performance for these figures was taken from [Sandholm, 1999]. It is therefore impossible to rerun these experiments on other distributions or varying the number of nondominated bids instead of the number of raw bids. Observe that overall, CASS dramatically outperforms Bidtree: CASS is between 2 and 500 times faster than Bidtree on the data points shown here, and is never slower.

8.2 CASS vs. CPLEX

When CASS and Bidtree were proposed, ILOG's CPLEX 5 mixed integer programming package (the industry standard) was unable to solve most WDP problems within a reasonable amount of time. Since that time, however, CPLEX's mixed integer programming module improved substantially with version 6 (released 2000), and substantially again with version 7 (released 2001). Now that CPLEX is in version 8, there has been general convergence in the research community towards using CPLEX as the default approach for solving the WDP.

The only ongoing effort at competition with CPLEX has come from the authors of Bidtree, who have written an updated algorithm called CABOB which they claim is much faster [Sandholm *et al.*, 2001]. However, like Bidtree, CABOB is not available to researchers. This is a serious impediment because published runtime data for CABOB is insufficient for our purposes: from this point forward we analyze algorithms in more detail than simply comparing average or median running times. In any case, CABOB's reported performance is similar to CPLEX's, and CABOB is also similar to CPLEX in its construction: it makes use of CPLEX's linear programming package as a subroutine and uses a similar search strategy. For these reasons, we present no experiments with CABOB.

In this section we compare the performance of CASS and CPLEX 8.0. These experiments were run on a cluster of 12 dual 2.4 Ghz Xeon machines with 1 GB RAM running Redhat Linux. We tested on 10 of the distributions provided by the CATS suite: all those distributions capable of generating an arbitrary number of nondominated bids. Specifically, we used all five of the CATS distributions (paths, regions, arbitrary, matching, scheduling) as well as five of the legacy distributions (L2, L3, L4, L6, L7). We sampled each of the distributions' parameters from a hand-chosen range of "reasonable" parameters as described in Section 7.4.2. We also sampled the problem size parameters: number of goods was chosen uniformly from $[40, 400]$ and number of nondominated bids was chosen uniformly from $[50, 2000]$. The full dataset had roughly 100 data points per distribution for a total of about 1000 data points, and took nearly 8 months of CPU time to collect. In order to increase the number of data points we were able to collect during this time, we capped CASS's runtime at 12 hours.

Figure 8.8 shows the mean runtime of CASS and CPLEX on each distribution. Note that the vertical axis uses a logarithmic scale so that all the bars can be seen on

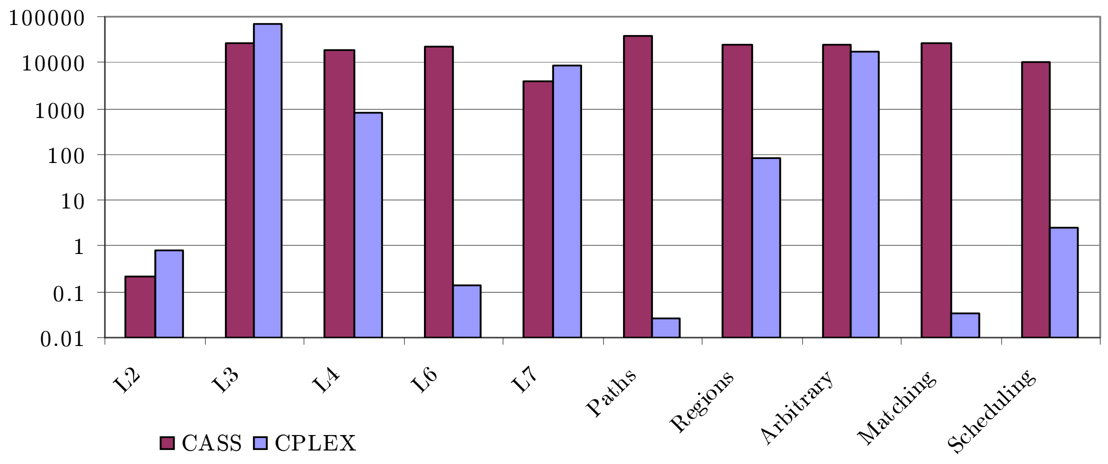


Figure 8.8: CASS vs. CPLEX: mean runtime per distribution

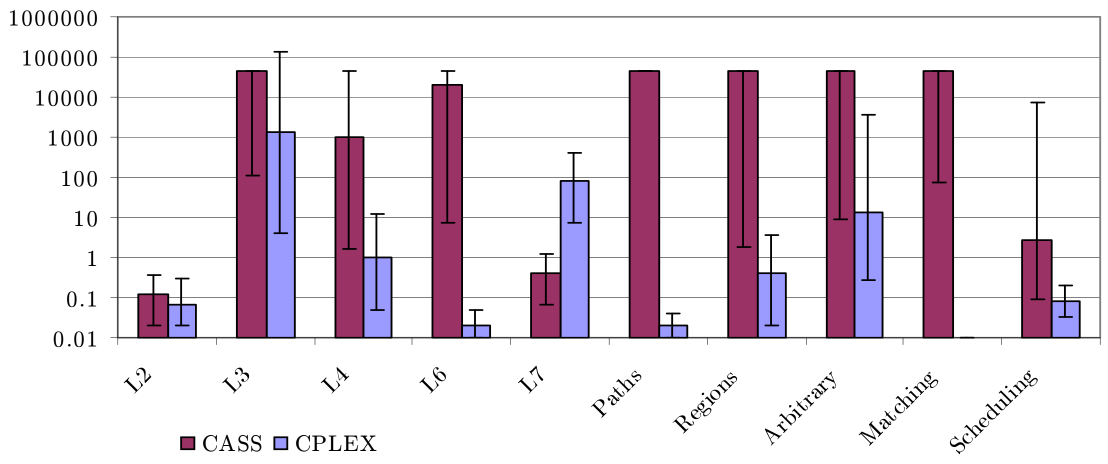


Figure 8.9: CASS vs. CPLEX: first, second and third quartiles

the same graph. Judging from this picture CPLEX seems to be a better choice than CASS most of the time. Exceptions are L3 and L7, where CASS does much better than CPLEX, and L2 where the difference is less pronounced.

Average performance can be overwhelmed by a relatively small fraction of runs that take a very long time. To learn more about how CASS and CPLEX 8.0 compare, we can graph medians instead of means. Figure 8.9 shows medians for each distribution; the error bars indicate first and third quartiles (of course, median is the second quartile). We can see several things from this graph. First, on the whole CPLEX still appears to be much faster than CASS. Note L3: while CASS had better average performance on this distribution, CPLEX has better median performance. On L7 CASS's advantage is now shown to be much larger, with the two algorithms' error bars not even overlapping. On this distribution it does seem that a small number of very hard instances skewed CASS's average upwards; the same is true for CASS on L4 and scheduling, and for CPLEX on L4, L7, regions and arbitrary. Overall, we can see from the error bars that most distributions exhibit substantial runtime variation.

In Figure 8.9 it is also the case that the error bars often overlap. This raises the question of the extent to which the algorithms' runtimes are uncorrelated. If this level is high then there could be great benefit to running both CASS and CPLEX in parallel (or in choosing between the algorithms in a more sophisticated way: see Chapter 10). To investigate the level of correlation between CASS and CPLEX on a per-instance basis, we plotted the algorithms' runtimes on separate axes of a scatter plot in Figure 8.10. For 5% of the instances, CASS and CPLEX had the same running time within two significant digits, and on 6% of the instances both CASS and CPLEX took longer than CASS's cap time of 12 hours, making it impossible to compare the algorithms. As we expected given its much better running times in Figures 8.8 and 8.9, CPLEX outperformed CASS a large fraction of the time (67%). The surprise is that there remained a substantial fraction (22%) of instances on which CASS outperforms CPLEX; on many instances the performance difference was very significant.

8.3 Conclusions

This chapter detailed experimental investigations of the CASS algorithm's performance. First, it was shown to have good anytime performance, finding good solutions almost immediately and finding an optimal solution long before optimality is proven. Second, it was compared to Sandholm's Bidtree algorithm, which it consistently and significantly outperformed. Finally, it was compared to a more modern algorithm, ILOG's CPLEX 8.0. On most (but not all) test distributions, CPLEX exhibited considerably better performance. However, a more careful analysis showed

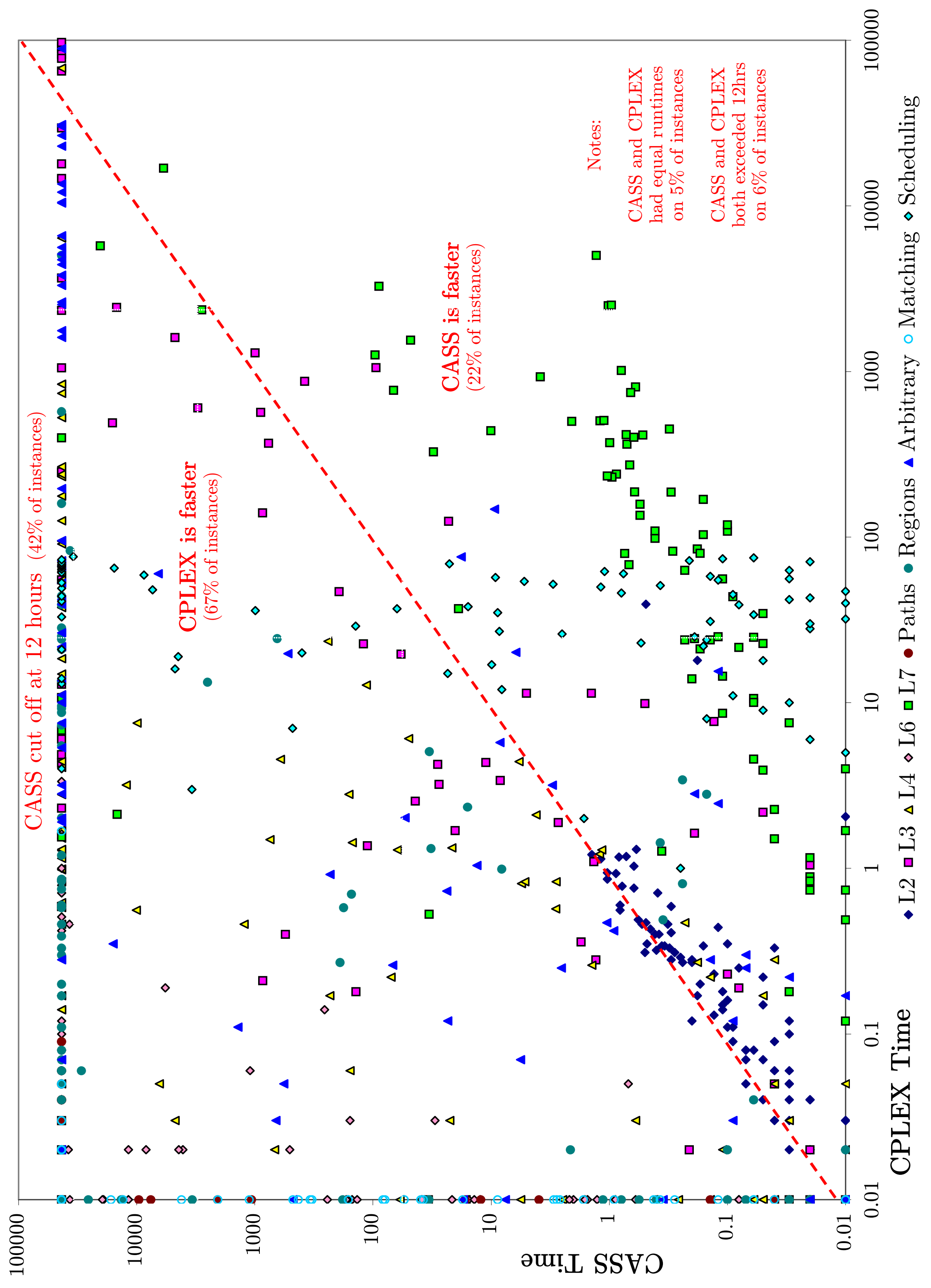


Figure 8.10: CASS vs. CPLEX: Scatter Plot

that the two algorithms' performance was often uncorrelated, and that there were a substantial fraction of these instances on which CASS dramatically outperformed CPLEX. In Chapter 10 we will revisit this uncorrelation between CASS and CPLEX and explore ways of leveraging it to build a better WDP algorithm. First, however, we will study ways of explaining why an algorithm (such as CPLEX) shows so much runtime variation on similar problems.

Bibliography

Achlioptas, D., Gomes, C. P., Kautz, H. A., & Selman, B. (2000). Generating satisfiable problem instances. *AAAI*.

Anderson, A., Tenhunen, M., & Ygge, F. (2000). Integer programming for combinatorial auction winner determination. *ICMAS* (pp. 39–46).

Ausubel, L., Cramption, P., McAfee, R., & McMillan, J. (1997). Synergies in wireless telephony: Evidence from the broadband PCS auctions. *Journal of Economics and Management Strategy*, *6*(3), 497–527.

Banks, J., Ledyard, J., & Porter, D. (1989). Allocating uncertain and unresponsive resources: An experimental approach. *RAND Journal of Economics*, *20*, 1–23.

Billings, D., Burch, N., Davidson, A., Holte, R., Schaeffer, J., Schauenberg, T., & Szafron, D. (2003). Approximating game-theoretic optimal strategies for full-scale poker. *IJCAI*.

Blair, D. (1998). Impact of the Internet on core switching network. *Proc. of ENPW'98*. Les Arcs, France.

Blum, B., Shelton, C., & Koller, D. (2003). A continuation method for Nash equilibria in structured games. *IJCAI*.

Boutilier, C. (1996). Planning, learning and coordination in multiagent decision processes. *Theoretical Aspects of Rationality and Knowledge* (pp. 195–201).

Boutilier, C., Goldszmidt, M., & Sabata, B. (1999). Sequential auctions for the allocation of resources with complementarities. *IJCAI*.

Brewer, P., & Plott, C. (1996). A binary conflict ascending price (BICAP) mechanism for the decentralized allocation of the right to use railroad tracks. *International Journal of Industrial Organization*, *14*, 857–886.

Bykowsky, M., Cull, R., & Ledyard, J. (1995). *Mutually destructive bidding: The FCC auction design problem* (Technical Report Social Science Working Paper 916). California Institute of Technology, Pasadena.

CATS Website (2000). http://robotics.stanford.edu/CATS.

Cheeseman, P., Kanefsky, B., & Taylor, W. M. (1991). Where the Really Hard Problems Are. *IJCAI-91*.

Conitzer, V., & Sandholm, T. (2002). *Complexity Results about Nash Equilibria* (Technical Report CM-CS-02-135). CMU.

Cramton, P., & Palfrey, T. (1990). Cartel enforcement with uncertainty about costs. *International Economic Review*, *31*(1), 17–47.

Crawford, V., & Sobel, J. (1982). Strategic information transmission. *Econometrica*, *50*(6), 1431–1451.

de Vries, S., & Vohra, R. (2003). Combinatorial auctions: A survey. *INFORMS Journal on Computing*, *15*(3).

DeMartini, C., Kwasnica, A., Ledyard, J., & Porter, D. (1998). *A new and improved design for multi-object iterative auctions* (Technical Report Social Science Working Paper 1054). California Institute of Technology, Pasadena.

Demers, A., Keshav, S., & Shenker, S. (1990). Analysis and simulation of a fair queuing algorithm. *Journal of internetworking research and experience*, 3–26.

Doucet, A., de Freitas, N., & (ed.), N. G. (2001). *Sequential monte carlo methods in practice*. Springer-Verlag.

Feinstein, J., Block, M., & Nold, F. (1985). Asymmetric behavior and collusive behavior in auction markets. *American Economic Review*, *75*(3), 441–460.

Floyd, S. (1994). TCP and explicit congestion notification. *ACM Computer Communication Review*, *24*(5), 10–23.

Floyd, S., & Jacobson, V. (1993). Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, *1*(4), 397–413.

Friedman, J. (1991). Multivariate adaptive regression splines. *Annals of Statistics*, *19*.

Fudenberg, D., & Tirole, J. (1991). *Game theory*. MIT Press.

Fujishima, Y., Leyton-Brown, K., & Shoham, Y. (1999). Taming the computational complexity of combinatorial auctions: Optimal and approximate approaches. *IJCAI*.

Gibbens, R., & Kelly, F. (1999). Resource pricing and the evolution of congestion control. *Automatica*, *35*, 1969–1985.

Gibbens, R., & Key, P. (1999). *The use of games to assess user strategies for differential quality of service in the internet* (Technical Report). Microsoft Research, Cambridge.

Gomes, C., & Selman, B. (2001). Algorithm portfolios. *Artificial Intelligence*, *126*(1-2), 43–62.

Gomes, C. P., & Selman, B. (1997). Problem structure in the presence of perturbations. *AAAI/IAAI*.

Gonen, R., & Lehmann, D. (2000). Optimal solutions for multi-unit combinatorial auctions: Branch and bound heuristics. *ACM Conference on Electronic Commerce*.

Gonen, R., & Lehmann, D. (2001). *Linear programming helps solving large multi-unit combinatorial auctions* (Technical Report TR-2001-8). Leibniz Center for Research in Computer Science.

Graham, D., & Marshall, R. (1987). Collusive bidder behavior at single-object second-price and English auctions. *Journal of Political Economy*, *95*, 579–599.

Graham, D., Marshall, R., & Richard, J.-F. (1990). Differential payments within a bidder coalition and the Shapley value. *American Economic Review*, *80*(3), 493–510.

Grether, D., Isaac, R. M., & Plott, C. (1989). *The allocation of scarce resources: Experimental economics and the problem of allocating airport slots.* Boulder, CO: Westview Press.

Guestrin, C. E., Koller, D., & Parr, R. (2001). Multiagent planning with factored MDPs. *14th Neural Information Processing Systems (NIPS-14)* (pp. 1523–1530). Vancouver, Canada.

Harstad, R., Kagel, J., & Levin, D. (1990). Equilibrium bid functions for auctions with an uncertain number of bidders. *Economic Letters*, *33*(1), 35–40.

Hastie, T., Tibshirani, R., & Friedman, J. (2001). *Elements of statistical learning.* Springer.

Hendricks, K., & Porter, R. (1989). Collusion in auctions. *Annales d'Économie et de Statistique*, *15/16*, 216–229.

Holte, R. C. (2001). Combinatorial auctions, knapsack problems, and hill-climbing search. *Canadian Conference on AI.*

Hoos, H., & Boutilier, C. (2000). Solving combinatorial auctions using stochastic local search. *The 17th national conference on artificial intelligence* (pp. 22–29).

Horvitz, E., Ruan, Y., Gomes, C., Kautz, H., Selman, B., & Chickering, M. (2001). A Bayesian approach to tackling hard computational problems. *UAI.*

Hotelling, H. (1929). Stability in competition. *Economic Journal*, *39*, 41–57.

Kastner, R., Hsieh, C., Potkonjak, M., & Sarrafzadeh, M. (2002). On the sensitivity of incremental algorithms for combinatorial auctions. UCLA CS Tech. Report 020000.

Kearns, M., Littman, M., & Singh, S. (2001). Graphical models for game theory. *UAI.*

Kearns, M., & Mansour, Y. (2002). Efficient nash computation in large population games with bounded influence. *UAI.*

Key, P., & McAuley, D. (1999). Differential QoS and pricing in networks: Where flow control meets game theory. *IEE Proc Software 146.*.

Kohavi, R., & John, G. (1997). Wrappers for feature subset selection. *Artificial Intelligence Journal, special issue on relevance*, *97*(1–2), 273–324.

Koller, D., & Milch, B. (2001). Multi-agent influence diagrams for representing and solving games. *IJCAI.*

Korf, R., & Reid, M. (1998). Complexity analysis of admissible heuristic search. *AAAI-98.*

Lagoudakis, M., & Littman, M. (2000). Algorithm selection using reinforcement learning. *ICML.*

Lagoudakis, M., & Littman, M. (2001). Learning to select branching rules in the DPLL procedure for satisfiability. *LICS/SAT.*

Ledyard, J., & Szakaly, K. (1994). Designing organizations for trading pollution rights. *Journal of Economic Behavior and Organization*, *25*, 167–196.

Ledyard, J. O., Porter, D., & Rangel, A. (1997). Experiments testing multiobject allocation mechanisms. *Journal of Economics & Management Strategy*, *6*(3), 639–675.

Lehmann, D., O'Callaghan, L., & Shoham, Y. (1999). Truth revalation in rapid, approximately efficient combinatorial auctions. *ACM Conference on Electronic Commerce.*

Leyton-Brown, K., Nudelman, E., Andrew, G., McFadden, J., & Shoham, Y. (2003a). Boosting as a metaphor for algorithm design. *Constraint Programming.*

Leyton-Brown, K., Nudelman, E., Andrew, G., McFadden, J., & Shoham, Y. (2003b). A portfolio approach to algorithm selection. *IJCAI.*

Leyton-Brown, K., Nudelman, E., & Shoham, Y. (2002a). Learning the empirical hardness of optimization problems: The case of combinatorial auctions. *CP*.

Leyton-Brown, K., Pearson, M., & Shoham, Y. (2000a). Towards a universal test suite for combinatorial auction algorithms. *ACM EC*.

Leyton-Brown, K., Porter, R., Venkataraman, S., & Prabhakar, B. (2001). Smoothing out focused demand for network resources (short paper). *ACM Conference on Electronic Commerce*.

Leyton-Brown, K., Porter, R., Venkataraman, S., Prabhakar, B., & Shoham, Y. (2003c). Incentive mechanisms for smoothing out a focused demand for network resources. *Computer Communications*, *26*, 237–250.

Leyton-Brown, K., Shoham, Y., & Tennenholtz, M. (2000b). An algorithm for multi-unit combinatorial auctions. *Proceedings of AAAI-00*.

Leyton-Brown, K., Shoham, Y., & Tennenholtz, M. (2000c). Bidding clubs: institutionalized collusion in auctions. *ACM Conference on Electronic Commerce*.

Leyton-Brown, K., Shoham, Y., & Tennenholtz, M. (2002b). Bidding clubs in first-price auctions. *The 19th National Conference on Artificial Intelligence*.

Leyton-Brown, K., & Tennenholtz, M. (2003). Local-effect games. *IJCAI*.

Lobjois, L., & Lemaître, M. (1998). Branch and bound algorithm selection by performance prediction. *AAAI*.

MacKie-Mason, J., & Varian, H. (1994). Pricing the internet. In B. Kahin and J. Keller (Eds.), *Public access to the internet*. Prentice-Hall.

Mailath, G., & Zemsky, P. (1991). Collusion in second-price auctions with heterogeneous bidders. *Games and Economic Behavior*, *3*, 467–486.

Mas-Colell, A., Whinston, M. D., & Green, J. R. (1995). *Microeconomic theory*. New York: Oxford University Press.

McAfee, R., & McMillan, J. (1987). Auctions with a stochastic number of bidders. *Journal of Economic Theory*, *43*, 1–19.

McAfee, R., & McMillan, J. (1992). Bidding rings. *American Economic Review*, *82*, 579–599.

Milgrom, P. (1998). Putting auction theory to work: The simultaneous ascending auction. Technical Report 98-0002, Department of Economics, Stanford University.

Mitchell, B. (1978). Pricing policies in selected European telephone systems. *Proceedings of 6th Conference on Telecommunications Policy Research* (pp. 437–475).

Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., & Troyansky, L. (1998). Determining computational complexity for characteristic 'phase transitions'. *Nature*, *400*.

Monderer, D., & Shapley, L. (1996). Potential games. *Games and Economic Behavior*, *14*, 124–143.

Monderer, D., & Tennenholtz, M. (2000). Optimal Auctions Revisited. *Artificial Intelligence*, *120*(1), 29–42.

Mura, P. L. (2000). Game networks. *UAI*.

Nash, J. (1950). Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences of the United States of America*, *36*, 48–49.

Nemhauser, G. L., & Wolsey, L. A. (1988). *Integer and combinatorial optimization*. New York, NY: Wiley.

Nisan, N. (2000). Bidding and allocation in combinatorial auctions. *ACM Conference on Electronic Commerce*.

Nisan, N., & Ronen, A. (2000). Computationally feasible VCG mechanisms. *ACM Conference on Electronic Commerce*.

Odlyzko, A. (1997). *A modest proposal for preventing internet congestion* (Technical Report TR 97.35.1). AT&T Research.

Osborne, M., & Pitchik, C. (1987). Equilibrium in Hotelling's model of competition. *Econometrica*, *55*, 911–922.

Osborne, M., & Rubinstein, A. (1994). *A course in game theory.* MIT Press.

Pan, R., Breslau, L., Prabhakar, B., & Shenker, S. (2001). Approximate fairness through differential dropping. *Submitted.*

Pan, R., Prabhakar, B., & Psounis, K. (2000). CHOKe: A stateless active queue management scheme for approximating fair bandwidth allocation. *Proceedings of IEEE INFOCOM 2000* (pp. 942–951).

Parkes, D. C. (1999). iBundle: An efficient ascending price bundle auction. *ACM Conference on Electronic Commerce.*

Plott, C., & Cason, T. (1996). EPA's new emissions trading mechanism: A laboratory evaluation. *Journal of Environmental Economics and Management*, *30*, 133–160.

Quan, D. (1994). Real estate auctions: A survey of theory and practice. *Journal of Real Estate Finance and Economics*, *9*, 23–49.

Rassenti, S., Reynolds, S., & Smith, V. (1994). Cotenancy and competition in an experimental auction market for natural gas pipeline networks. *Economic Theory*, *4*, 41–65.

Rassenti, S., Smith, V., & Bulfin, R. (1982). A combinatorial auction mechanism for airport time slot allocation. *Bell Journal of Economics*, *13*, 402–417.

Reeves, D., & Wellman, M. (2003). Computing equilibrium strategies in infinite games of incomplete information. *Fifth Workshop on Game Theoretic and Decision Theoretic Agents at the 2nd Conference on Autonomous Agents and Multi-Agent Systems.*

Rice, J. R. (1976). The algorithm selection problem. *Advances in Computers*, *15*, 65–118.

Riley, J., & Samuelson, W. (1981). Optimal auctions. *American Economic Review*, *71*, 381–392.

Robinson, M. (1985). Collusion and the choice of auction. *Rand Journal of Economics*, *16*(1), 141–145.

Rosenthal, R. (1973). A class of games possessing pure-strategy Nash equilibria. *International Journal of Game Theory*, *2*, 65–67.

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

Roughgarden, T., & Tardos, E. (2001). *Bounding the inefficiency of equilibria in nonatomic congestion games* (Technical Report TR2002-1866). Cornell, Ithaca.

Ruan, Y., Horvitz, E., & Kautz, H. (2002). Restart policies with dependence among runs: A dynamic programming approach. *CP*.

S. H. Clearwater, e. (1996). *Market-based control: A paradigm for distributed resource allocation*. World Scientific.

Sandholm, T. (1999). An algorithm for optimal winner determination in combinatorial auctions. *IJCAI-99*.

Sandholm, T., & Suri, S. (2000). Improved algorithms for optimal winner determination in combinatorial auctions and generalizations. *AAAI-00*.

Sandholm, T., Suri, S., Gilpin, A., & Levine, D. (2001). CABOB: A fast optimal algorithm for combinatorial auctions. *IJCAI*.

Sanholm, T. (1993). An implementation of the contract net protocol based on marginal cost calculations. *Proceedings of AAAI-93* (pp. 256–262).

Schapire, R. (1990). The strength of weak learnability. *Machine Learning*, *5*, 197–227.

Schuurmans, D., Southey, F., & Holte, R. C. (2001). The exponentiated subgradient algorithm for heuristic boolean programming. *IJCAI-01*.

Selman, B., Mitchell, D. G., & Levesque, H. J. (1996). Generating hard satisfiability problems. *Artificial Intelligence*, *81*(1-2), 17–29.

Shenker, S. (1995). Making greed work in networks: A game-theoretic analysis of switch service disciplines. *IEEE/ACM Transactions on Networking*, *3*, 819–831.

Slaney, J., & Walsh, T. (2001). Backbones in optimization and approximation. *IJCAI-01*.

Songhurst, D., Stamoulis, G., & Stoer, M. (1999). Usage-based charging using effective bandwidths: studies and reality. *Proceedings of the International Teletraffic Congress, ITC-16*.

Tennenholtz, M. (2000). Some tractable combinatorial auctions. Proceedings of AAAI-2000.

Varian, H. R. (1995). Economic mechanism design for computerized agents. *Proceedings of the First Usenix Conference on Electronic Commerce*.

Vickrey, D., & Koller, D. (2002). Multi-agent algorithms for solving graphical games. *AAAI*.

von Ungern-Sternberg, T. (1988). Cartel stability in sealed bid second price auctions. *The Journal of Industrial Economics*, *18*(3), 351–358.

Wellman, M. (1993). A market-oriented programming environment and its application to distributed multicommodity flow problems. *Journal of Artificial Intelligence Research*, *1*, 1–23.

Wellman, M., Greenwald, A., Stone, P., & Wurman, P. (2003). The 2001 trading agent competition. *Electronic Markets*, *13*(1).

Wellman, M., Wurman, P., Walsh, W., & MacKie-Mason, J. (1998). Auction protocols for distributed scheduling. Games and Economic Behavior.

Zhang, W. (1999). *State-space search: Algorithms, complexity, extensions, and applications.* Springer.

Zurel, E., & Nisan, N. (2000). An efficient approximate allocation algorithm for combinatorial auctions. *ACM Conference on Electronic Commerce.*