An Optimal Multiprocessor Combinatorial Auction Solver

Shouxi Yang[†]
syang@cs.uiowa.edu

Alberto Maria Segre[†]
segre@cs.uiowa.edu

Bruno Codenotti[‡]
bruno.codenotti@iit.cnr.it

[†]Department of Computer Science
The University of Iowa
Iowa City, IA 52242

[‡]Istituto di Informatica e Telematica
Consiglio Nazionale delle Ricerche
Pisa, Italy

Abstract

A combinatorial auction is an auction that permits bidders to bid on bundles of goods rather than just a single item. Unfortunately, winner determination for combinatorial auctions is known to be *NP*-hard. In this paper, we propose a distributed algorithm to compute optimal solutions to this problem. The algorithm uses *nagging*, a technique for parallelizing search in heterogeneous distributed computing environments. Here, we show how nagging can be used to parallelize a branch-and-bound algorithm for this problem, and provide empirical results supporting both the performance advantage of nagging over more traditional partitioning methods as well as the superior scalability of nagging to larger numbers of processors.

1. Introduction

Combinatorial auctions (CAs) are auctions where participants may bid on *combinations* or *bundles* of items, as opposed to single items only. Compared to other auction mechanisms, CAs maximize the efficiency of the auction, while reducing risk for each individual bidder [19]. This is because combinatorial auctions allow bidders to attribute greater value to combinations of items, whose constituent items may have significantly lesser or no value alone. In contrast, using traditional auction mechanisms, the bidder would have to bid sequentially on each desired item, hoping to obtain all of them, while risking being unable to complete the bundle. Despite this theoretical advantage, however, the determination of an optimal winning bid combination is an *NP*-hard problem, a fact that has been widely noted [18, 20, 21]. In contrast, determination of winning bids in a sequential series of traditional auctions, where each item is awarded separately is, computationally speaking, much easier.

From: *Computers and Operations Research* **36**:1 (2009)

In this paper, we present a parallel optimal solution algorithm for the CA winner determination problem. Our algorithm employs *nagging*, a paradigm for parallelizing search algorithms by playing multiple reformulations of the problem (or reformulations of portions of the problem) against each other [24]. The intuition here is that, while you may not be able to beat the exponent, a parallel solution that scales cleanly to many processors can, in practice, significantly increase the size of problems which can be solved in reasonable time. Thus, the message of this paper is that when optimal solutions are required, nagging should be the method of choice.

The paper is organized as follows. After a concise problem description in Section 2, we describe a serial solution algorithm in Section 3. Section 4 describes nagging, and Section 5 describes the NICE distributed infrastructure used here. In Section 6, we show how the algorithm of Section 3 can be parallelized with nagging, and Section 7 recounts an empirical study exploring the advantage of nagging over partitioning as well as the scalability of nagging to larger numbers of processors.

2. Combinatorial Auction Winner Determination

The problem of determining the winning set of bids in a CA is the *winner determination problem* (WDP). Given a set of bids, find an allocation of items to bidders that maximizes the sum of the accepted bids.

Let N be the set of n bidders and M the set of m distinct items available. For every subset S of M, let $v_i(S)$ be the bid that agent $i \in N$ is willing to pay for the collection of items represented by S, let $x_i(S) = 1$ mean agent i is assigned subset S in the winning solution, and let $y_j(S) = 1$ mean that item j is included in set S. We can then formulate WDP as an integer programming problem:

$$\textbf{maximize:} \quad \sum_{i \in N} \sum_{S \subseteq M} v_i(S)\, x_i(S)$$

$$\textbf{subject to:} \quad \sum_{i \in N} \sum_{S \subseteq M} x_i(S) y_j(S) \leq 1 \qquad \forall\, j \in M$$
$$\sum_{S \subseteq M} x_i(S) \leq 1 \qquad \forall\, i \in N$$
$$x_i(S) \in \{0,1\} \qquad \forall\, S \subseteq M, \forall\, i \in N$$
$$y_j(S) \in \{0,1\} \qquad \forall\, S \subseteq M, \forall\, j \in S$$

The first constraint ensures that no overlapping sets of items are ever assigned (*i.e.*, no single item is ever sold twice; see [15] for a discussion of multi-unit combinatorial auctions, where more than one copy of each item may be available). The second constraint prevents any bidder from being assigned more than one set of items in the winning solution; this condition trivially ensures that the auctioneer cannot sell individual items from a bundle to a given bidder for a greater price than can be obtained for the items in combination.[1]

Using this formulation, WDP is *NP*-hard by reduction from the weighted set packing problem [4, 8, 15, 20]. Since WDP is *NP*-hard, one possible means of solving the problem in practice is to find approximate solutions instead [7, 11, 13, 19]. In particular, the algorithm of [30] seems to perform very well on random instances, getting to within 1% of the optimal solution. However, in the general case, no approximation to within $n^{1-\varepsilon}$ (with parameter $\varepsilon > 0$) can be guaranteed to be found in polynomial time (unless $P = NP$), although optimal solutions in certain special cases can be approximated slightly more effectively [21]. A major drawback of any approximation algorithm is that it may cause the underlying mechanism (if the mechanism is of the commonly used Vickrey-Clarke-Groves type) to lose the strategyproofness property if the approximation does not yield the optimal solution (an auction is strategyproof when it is in the best interest of each bidder to bid their true valuations). Another alternative is to solve the problem exactly using search, assuring an optimal solution. This approach has been shown to work very well on average, scaling optimal winner determination to hundreds of items and

[1] This property is usually called *substitutability*, and is simply dropped in some formulations [22], or enforced via alternative means, such as the incorporation of additional dummy items in some bids, as described in [4].

thousands of bids (depending on the problem instance distribution) [1, 4, 22]. As expected, however, performance still deteriorates exponentially as problem size increases.

This paper also takes the latter approach, presenting an optimal solution algorithm that exploits many processors in order to solve large WDPs. Our solution involves parallelizing a branch-and-bound search algorithm and distributing the computation over a network of processors. Our solution relies on both more traditional search partitioning methods as well as an unusual parallelization strategy, called nagging, that has been shown to provide supralinear speedups on many similar *NP*-hard problems.

3. An Optimal Sequential Solver for WDP

A branch-and-bound algorithm finds optimal solutions to discrete optimization problems by exhaustive enumeration of the solution space. We can think of branch-and-bound search as exploring a tree of nodes, where the root node represents the set of all possible solutions, and each successor node represents an incremental refinement (*i.e.*, a subset) of its parent node's solution set. Since the solution space will, of course, grow exponentially with problem size, this approach becomes feasible in practice only if a significant portion of the space can be safely excluded from exploration through some sort of bounding procedure. Different branch-and-bound implementations can be characterized by (1) how successor nodes are generated, (2) how they choose which node to branch on next, (3) how they calculate bounds at a given node, and (4) what criteria are used to recognize and prune nodes that will not lead to an optimal solution.

In its simplest form, our solution explores, depth-first, a tree of solutions starting from a root node representing all possible bid combinations (*i.e.*, no bids accepted or rejected as yet). At each step, we partition the active node by choosing, possibly at random, a bid from the set of as-yet-unconsidered bids. We then generate at most two successor nodes, one that accepts the chosen bid, and the other that rejects it (if accepting the bid conflicts with one or more previously accepted bids, only the rejection successor is

generated). In this space, each leaf node corresponds to a (legal) Boolean combination of accepted bids. Our task is to find the leaf corresponding to the largest-valued combination of accepted bids: since this simple algorithm is complete (*i.e.*, every possible legal bid combination is exhaustively generated), we are guaranteed that the optimal bid combination will eventually be found.

We can improve the performance of the algorithm by reducing the number of nodes examined. Entire subtrees of the search space may be pruned without compromising completeness if the partial solution represented by the root of the subtree has no completion that is better than the best solution discovered by the search so far. This requires (1) the ability to make good upper bound estimates of the maximal completion value of any partial solution, and (2) the ability to find a good, high-value solution quickly, so that more subtrees may be excluded from the search. If we can guarantee that the completion estimate is a true upper bound for the best solution value found in that subtree (this is the *admissibility condition* for A* heuristics [17]), and if that estimate fails to reach the lower bound on the global solution established by the best solution found so far, then it is safe to conclude that the globally optimal solution is not to be found within the subtree in question.

The procedure is outlined in Figure 1. The **WDP**() function takes a set of bids *H* as input and returns the highest-valued consistent subset of *H* according to the utility function **value**(). The heart of the algorithm is the recursive function **search**(), which takes three bid sets, *G*, *H*, and *R*, representing the accepted bids for the current partial solution, the set of bids yet to be considered, and the highest-valued (initially null) consistent bid set found so far, respectively. Each recursive call to **search**() selects a candidate bid *b* from *H* and considers extending *G* both by including *b* (but only if *b* is consistent, as determined by **feasible**(), with the other bids in *G*) and by excluding *b*. Whenever a partial solution better than *R* is found, it replaces *R*.

```
WDP(H: bidset): bidset;
  return (search(∅, H, ∅);

search(G: bidset, H: bidset, R: bidset): bidset;                    /* Find best bidset */
  b: bid;
  if (value(G) > value(R))                                          /* Found better solution */
    R ← G;
  if (H = ∅)                                                        /* Leaf: R still best solution */
    return (R);
  if (value(G) + estimate(G, H) ≤ value(R))                         /* Prune (see Figure 2) */
    return (R);
  b ← select(H);                                                    /* Consider next bid */
  H ← H\{b};                                                        /* Update pending bids H */
  if (feasible(G ∪ {b}))
    R ← search((G ∪ {b}), H, R);                                    /* Accept bid */
  return (search(G, H, R));                                         /* Reject bid */
```

Figure 1: Sequential search algorithm for the combinatorial auction problem. The **WDP**() function takes a set of bids H as input and returns the highest-valued consistent subset of H according to the utility function **value**().

As mentioned previously, the number of nodes searched can be reduced using a pruning function: extensions of partial solutions whose completions can be guaranteed not to exceed the value of the best solution found so far can be safely ignored (see Figure 2). Here, the **estimate**(G, H) function is used to estimate the maximal completion of a partial solution G using remaining bidsets H. If this function is admissible, that is, represents a true upper bound on the best completion of G within H, then the procedure can still guarantee that the optimal solution will be found. Here, **estimate**() involves solving G's corresponding linear programming problem on remaining bidsets H and comparing the result to R, the best solution found so far (see Figure 2). Since the solution to the linear programming problem is always an upper bound on the solution to corresponding integer programming problem, we know that this heuristic estimate is admissible.

Empirical tests show that bid selection (*i.e.*, the order in which bids are considered by the **select**() function in Figure 1 and the relative ordering with which the two successor nodes are explored) as well as

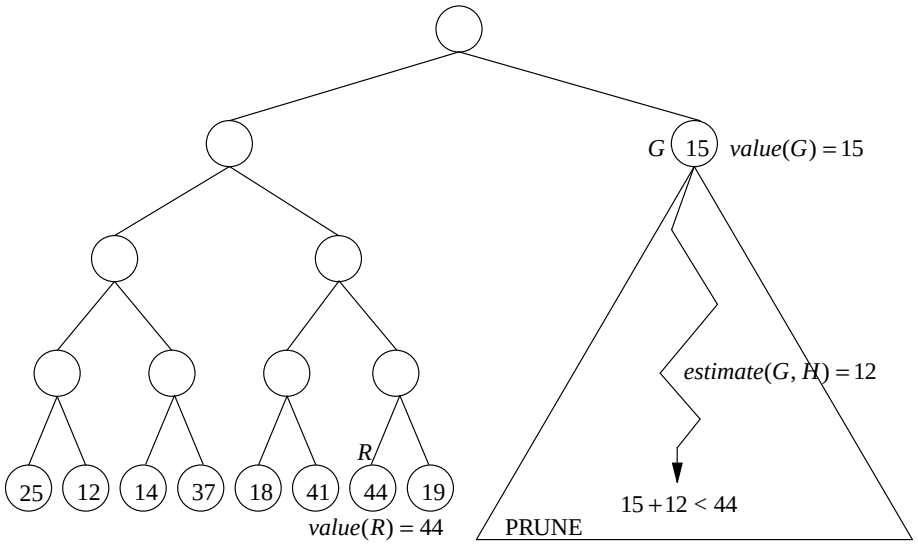

Figure 2: Subtree pruning: the subtree rooted at node G cannot lead to an optimal solution. The **estimate**(G, H) function returns an upper bound on the completion value of partial solution G. Here, the value of the best completion of G is guaranteed not exceed $15+12$, which is less than 44, the value of R, the best solution found so far: it is therefore safe to ignore all completions of G.

the initial distribution of bids have great effect on the expected solution time. To see why this is so, consider a search that, by sheer good fortune, stumbles immediately upon the optimal solution. Since **value**(R) will be maximal, a large number of candidate bid sets will be pruned by comparison with R, leading to improved search times. Various bid ordering heuristics are studied in [22], where it is shown that no certain strategy dominates on all distributions (this result is hardly surprising, given that WDP is NP-hard). Note that there are also a variety of heuristic modifications that can be made to counterbalance a poor bid selection strategy (*i.e.*, one that does not produce an ''early'' good solution for use as a lower bound). For instance, one could occasionally use a greedy strategy to complete the current partial solution in an attempt to quickly increase the lower bound. While such a strategy is often employed at the root before starting the search altogether in order to establish an early lower bound, it can also be applied on random partial solutions; its effectiveness depends on how often a new, larger lower bound is found, how

many times the new lower bound provides for pruning of the search space, and the cost of computing the greedy completion.

4. Nagging and Combinatorial Search Problems

Nagging, described in [24], is a paradigm for parallelizing search problems. First introduced in the context of automated theorem proving, nagging has also been applied to other search problems from the AI literature [12, 25, 27, 28]. In contrast with more traditional partitioning methods, where portions of the search space are parceled out to individual processors, nagging employs multiple processes operating on different transformations of the search space (or portions thereof). The underlying assumption is that since different search orderings lead to potentially large differences in search time, nagging can often achieve speedups far beyond those achieved by partitioning by playing multiple reformulations of a problem against each other. Moreover, nagging, unlike partitioning, is inherently load balancing and fault tolerant, reducing the computational overhead typically required by partitioning strategies to maintain these important properties.

The hierarchy of nagging nodes form a tree topology. The root, or topmost master processor, carries out an exhaustive search procedure, such as that outlined in Figure 1, thereby guaranteeing the optimal solution will be found (this search should be exhaustive, but may well exploit some heuristic ordering strategy). As the root searches the space, it is aided by some number of nagging processors, each repeatedly searching portions of reformulated or transformed search space assigned by the master (see Figure 3). Since a nagger may also be helped recursively by some number of additional naggers (who then search reformulated portions of their own master's assigned search space), the master/nagger groupings form a tree of processors, with the root processor serving as master for its immediate descendants, and so on down the tree. A master and its naggers race to exhaust their respective search spaces, with each race leading to three possible outcomes (as outlined in [24]):

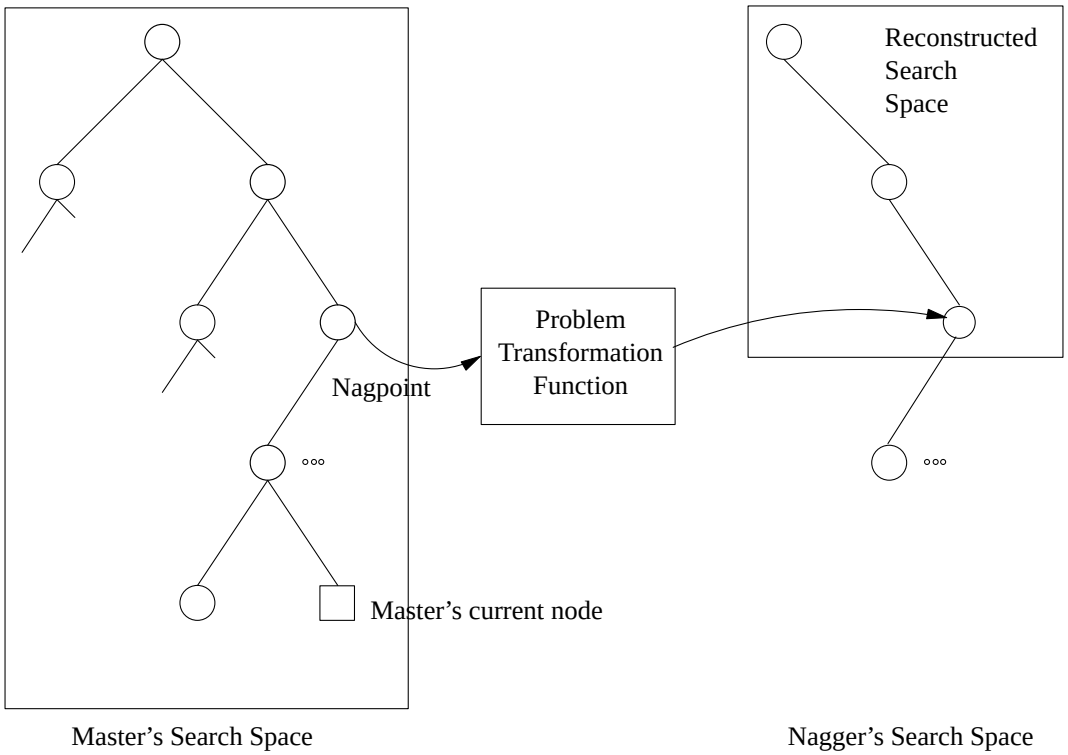

Figure 3: Nagging episode. The square node indicates the current position of the master process, which is executing a depth-first search. A nagpoint is selected along the master's search path, and is described to the nagging process by communicating the series of choices from the root of the search tree to the nagpoint. The nagger reconstructs the master's search space up to the nagpoint and then commences exploring its own, transformed, version of the space rooted at the nagpoint.

(1) *Abort:* If the master backtracks over the partial solution allocated to the nagger before the nagger completes its own search, the master signals the nagger to abort its search, which causes the nagger to once again become idle and request new work from its master.

(2) *Prune:* If the nagger completes its search and finds there are no better solutions than that already found by the master, then the nagger interrupts its master and instructs it to backtrack to beyond the partial solution assigned to the nagger, resulting in a reduction in the master's search space.

(3) *Solve:* Finally, should the nagger find a better solution than that known by its master, it will report the new value to the master so that the master might use this new value to reduce its own search space. The nagger may then elect to either abort and ask for a new search from its master, or continue its current search until one of the other outcomes is attained.

The more often outcomes (2) and (3) are encountered in practice, the more effective nagging will be in speeding up search performance. Using appropriate transformation functions can help make these positive

outcomes more likely, as can the application of recursive nagging, where nagging processors serve as masters to additional naggers.

5. NICE: A Distributed Search Infrastructure

Our distributed WDP solver implementation is based on a software middleware layer that manages access to other processors over a network. The *Network Infrastructure for Combinatorial Exploration*, or NICE, models the distributed computing resource as a hierarchy, or tree, of processors. Developed explicitly to support nagging, NICE also supports more traditional partitioning methods.[2] The NICE distribution includes *niced*, a resource management daemon, *niceq*, a daemon status query program, and *niceapi*, the applications programmer's interface library. The code is written in ANSI C with BSD sockets over TCP/IP, and is known to run on multiple variants of the Unix operating system, including Linux, Sun OS, and HPUX, as well as Windows XP using the Cygwin runtime library.

The NICE daemon, or *niced*, must be running on every participating machine. Typically, it is started automatically as part of the system boot process, and runs as long as its host CPU is running (the daemon itself is single threaded and extremely lightweight, having no noticeable impact on system performance). NICE daemons are arranged hierarchically, with each daemon reporting to a single parent daemon while answering to zero or more child daemons. The NICE daemon has three primary responsibilities:

(1) The daemon maintains contact with the NICE hierarchy, occasionally exchanging host load and availability information with its parent and child daemons, and managing failure recovery should its parent and/or child daemons become unreachable or unresponsive. Each daemon may also initiate local reorganization of the NICE hierarchy in a greedy attempt to enhance overall search performance according to each host's current actual load.

[2] NICE represents a mature software effort: versions of the NICE daemon have been running continuously on our systems for several years, with no noticeable impact on performance. The code is quite robust, with NICE daemons running reliably and unobtrusively for periods of many months between system restarts. But more to the point, while the NICE infrastructure represents a necessary enabling technology that directly supports research in distributed search algorithms without the additional features provided in more general toolkits such as PVM [2] or MPI [10], the more interesting parallelization issues are algorithmic ones.

(2) The daemon provides the interface through which a qualifying application may request additional processors. It is also responsible for security, certifying which applications on which hosts are allowed to request support from other processors, which executables can be run on the local host, as well as which files, if any, can be accessed locally.

(3) The daemon manages the local host's resources according to prespecified host-specific constraints: for example, some hosts may be available only during specified times, such as during nighttime hours. Thus the daemon must decide when a processor can respond to requests for new processes, and must also manage previously spawned but still running processes, putting them to sleep and later waking them when the host becomes available once more.

A NICE-enabled application communicates with the NICE infrastructure via a set of callable functions contained in the NICE applications programmer's interface, or API. The API contains functions that, when invoked, *e.g.*, request that new copies of the application be spawned on the same or other machines. It also contains functions to support communication between these multiple copies of the application once they are established. Note that the library does not actually contain code specific to a search algorithm, but rather only the handful of functions that are needed to parallelize — via either nagging or partitioning — appropriately designed serial search algorithms.

6. An Optimal Distributed Solver for WDP

We now describe the nagging version of our winner determination algorithm (see Figures 4 and 5). The general idea is to run the sequential search procedure on the root node, with nagging processors engaging in a series of nagging episodes. Once the **nagWDP**() function is launched on the root node, the call to **niceInit**(H) connects to the NICE infrastructure and ensures copies of the same process are forked on the root processor's children (if the executable is not locally available, the appropriate digitally-signed binary is automatically downloaded and installed by the NICE infrastructure). The infrastructure also provides each copy of **nagWDP**() with the initial problem description, here consisting of H, the initial set of bids offered.

While the root processor (the function **niceRoot**() returns true only on the original root processor) performs depth-first search, each child processor engages in a series of nagging episodes. Each nagging

```
nG[ ] : array of bidsets                                   /* Global stack. */
nH[ ] : array of bidsets

nagWDP(H : bidset) : bidset ;
  R, S : bidset ;
  n : integer ;

  niceInit(H) ;                                  /* Connect to infrastructure. */
  if (niceRoot( ))                                /* Are you the root node? */
    nG[0] ← ∅ ;                                     /* Initialize stack. */
    nH[0] ← H ;
    return (searchExplicit(∅, 0, 0) ;
  else                                                          /* Nagger. */
    while ((n, nG[n], nH[n], S) ← niceIdle( ))          /* Request nagpoint. */
      R ← searchExplicit(S, n, n) ;                       /* Solve nagpoint. */
      if (R ∧ value(R) > value(S))
        niceSolve(R) ;                                /* Found better result. */
      elseif (R ≠ ∅)
        nicePrune(n) ;                              /* Found no better result. */
```

Figure 4: Parallelized search algorithm for WDP. The **nagWDP**() function takes a set of bids H as input and returns the highest-valued consistent subset of H according to the utility function **value**().

episode is initiated when a new nagpoint is requested from the corresponding parent process via **niceIdle**(), where a nagpoint is represented by $(g, nG[g], nH[g], S)$, consists of the base depth of the nagpoint g, the search node at that depth $nG[g]$, the remaining bids to consider $nH[g]$, and the best solution found so far, S. The search space rooted at the nagpoint is then transformed by the nagger via application of the **transform**() function prior to starting the search.

In both root and child, the bulk of the work is carried out by the **searchExplicit**() function (see Figure 5), analogous to the **search**() function of Figure 1, except with function stack operations made explicit (here, the $nG[i]$ and $nH[i]$ arrays of bidsets are analogous to the G and H variables in Figure 1, where the index i indicates the corresponding search depth). The explicit stacks are necessary in order to allow processes to access and exchange information about their own current search state. For the root

process, **searchExplicit**() performs the same recursive descent search as **search**(), while handling incoming messages.

As copies of the process propagate down the NICE hierarchy, each process must handle incoming messages from its own naggers (**niceIdle**(), **niceSolve**(), and **nicePrune**()) and their parent (**niceAbort**()).[3] A nagging episode is initiated when the parent process receives a **niceIdle**() message. The parent selects a nagpoint at some intermediate level g and returns a description of the search node at level g to the nagger. This will lead to one of three outcomes. First, the nagger may find a better solution (R) than that currently known by its parent process (S); here, **searchExplicit**() will immediately return R, and **nagWDP**() will then use **niceSolve**() to report the new solution to the parent process, abandoning its current search and requesting a new nagpoint.[4] Second, the nagger may complete its search without finding a result better than S: in this case, the value returned, R, will be the same as the S obtained in the nagpoint description, and **nagWDP**() will use **nicePrune**() to apprise the parent process that it may safely abandon its own current search, backtracking over the originally assigned nagpoint g. Finally, when the parent process backtracks over an assigned nagpoint, it will tell the appropriate nagger to abandon its current nagpoint, forcing **searchExplicit**() to return a null value. In each case, the nagger, once idle, simply requests a new nagpoint from its master and begins a new nagging episode.

[3] For expository purposes, we represent message handling in Figure 5 by explicitly polling for incoming messages each time the search process explores a new node of the space. In practice, the NICE API messaging system is somewhat more sophisticated.

[4] Having the nagger abandon its search when it finds a solution is a design decision. One might instead return the solution (so that the parent processor can make use of the new bound to improve its own search) but continue the nagger's search until the space rooted at the nagpoint is exhausted, or a **niceAbort**() message is received by the nagger.

```
backtrack←0 : integer                                   /* Unwind indicator. */
nagpoint[ ] : array of integer                          /* Current nagpoints. */

searchExplicit( R : bidset , g : integer , d : integer ) : bidset ;
  b : bid ;
  m : message ;
  n : integer ;
  while( m←niceCheck( ) )                               /* Incoming message. */
    if ( m . type = Idle )
      n←random( d − g ) + g ;                           /* Select nagpoint. */
      nagpoint[ n ]←m . address ;                       /* Note nagger address. */
      niceReply( n , nG[ n ] , transform( nH[ n ] ) , R ) ;   /* Send nagpoint. */
    elseif ( m . type = Solve ∧ m . value > value( R ) )
      R←m . value ;                                     /* Retain solution. */
    elseif ( m . type = Prune )
      backtrack←max( backtrack , d − m . value ) ;      /* Unwind to nagpoint. */
    elseif ( m . type = Abort )
      R←∅ ;                                             /* Abandon solution. */
      backtrack←d − g ;                                 /* Unwind completely. */
  if ( backtrack = 0 )                                  /* Explore current node. */
    if ( value( nG[ d ] ) > value( R ) )
      R←nG[ d ] ;                                       /* Found better solution. */
    if ( nH[ d ] =∅ )
      return ( R ) ;                                    /* Leaf node. */
    if ( value( nG[ d ] ) + estimate( nG[ d ] , nH[ d ] ) < value( R ) )
      return ( R ) ;                                    /* Prune. */
    b←select( nH[ d ] ) ;                               /* Consider bid. */
    nH[ d + 1 ]←nH[ d ] \ { b } ;
    if ( feasible( G∪{ b } ) )                          /* Include bid. */
      nG[ d + 1 ]←nG[ d ]∪{ b } ;
      R←searchExplicit( R , g , d + 1 ) ;
    if ( backtrack = 0 )                                /* Keep looking. */
      nG[ d + 1 ]←nG[ d ] ;
      R←searchExplicit( R , g , d + 1 ) ) ;             /* Exclude bid. */
  if ( nagpoint[ d ] ≠ 0 )                              /* Cleanup and return. */
    niceAbort( nagpoint[ d ] ) ;                        /* Abort irrelevant nagger. */
    nagpoint[ d ]←0 ;
  backtrack←max( 0 , backtrack − 1 ) ;
  return ( R ) ;                                        /* Return best solution. */
```

Figure 5: The function **searchExplicit()** is logically equivalent to the **search()** function of Figure 1, but with explicit stack manipulation and interrupt handling capabilities for processing messages to/from parent/child processes.

7. An Empirical Evaluation of Nagging for the Winner Determination Problem

In this section, we present a series of four empirical tests designed to evaluate the effectiveness of nagging for CA/WDP. In particular, we would like to use these tests to answer specific questions about the performance of nagging. First, is nagging an effective means of speeding up the calculation of optimal solutions to the combinatorial auction problem? Of particular interest is a direct comparison of nagging with more traditional search space partitioning methods (Experiment 1), and an exploration of problem characteristics that might be useful in predicting which method is more likely to be advantageous in a particular domain (Experiment 2). Second, how well does nagging scale with the number of processors? Here, we are interested in determining if the advantages of nagging are likely to be noted as more and more processors are brought to bear on the problem (Experiment 3). Finally, if these empirical tests are to be meaningful, we wish to ascertain whether our implementation, in its serial incarnation, is competitive with those used by others in prior work. If the single-processor version of our system should prove to be unusually slow, then the performance improvement attributed to parallelization may instead be an unfortunate artifact of our implementation (Experiment 4).

7.1. Data Sets

Any empirical evaluation relies on access to sample data that are representative of the kinds of problems that might be encountered in the real world. Unfortunately, data from large, real-life, combinatorial auctions are difficult to obtain; for this reason, prior work in this area has typically relied on the use of simulated data, obtained from problem generation systems, such as CATS [14], The expectation is that the simulated data are representative of the sort of scenarios one is likely to encounter in practice. Of course, such problem generation systems necessarily rely on implicit structural assumptions about the problem space made by their designers. Thus, to help ensure that the results obtained on simulated data extrapolate reliably to real-world applications, we also use simulated problems generated by a second system with significantly different structural assumptions [3]. Problems generated by this system exhibit

an explicit *sunflower structure*, where goods fall into one of two categories: *strategic items*, which have a considerable singleton value, and *non-strategic items*, which complement strategic items and are used to complete high value bundles. Examples of application areas with this kind of structure include real-estate auctions, where strategic items or parcels of land populate the core, or receptacle, of the flower, and non-strategic items or parcels form the petals.

Thus, in the absence of large test data sets culled from real-world applications, our experiments use six artificially-generated data sets from both the CATS simulator and the sunflower structure generator. Five sets of 100 problems each are generated according to five different CATS distributions (denoted *arbitrary*, *paths*, *regions*, *matching* and *scheduling* in [14]), while the sixth set of 100 problems is generated using the sunflower distribution test generator of [3]. For each data set, the generation parameters (*i.e.*, the number of items and number of bids) were adjusted in order to generate problems that typically require no more than roughly an hour of CPU time to solve.

7.2. Experiment 1

In this experiment, we compare a single-processor serial system to two distinct two-processor systems. The first two-processor system employs its second slave processor as a nagger, while the second two-processor system uses its slave processor to search a portion of the master processor's search space (*i.e.*, space partitioning). In both cases, once the slave processor finishes searching its assigned subspace, it shares the result with the master, and requests another subspace.

The experimental procedure is simple. First, the problems in each of the six data sets just described are solved serially using an arbitrarily fixed bidset ordering on a 2.4GHz Intel Xeon machine with 512KB of on-chip cache and 1GB of RAM running the Linux operating system. Next, a second, identical, processor is added to the NICE hierarchy and each problem is solved once again. For the two parallel trials, the slave processor, when idle, prompts the master for some work. In the first trial, the slave

processor is configured to perform nagging, applying a simple random bidset reordering problem transformation strategy to randomly-selected nagpoints (*i.e.*, the nagpoint is selected from among the ancestors of the current node using a uniform distribution). In the second parallel trial, the slave processor is configured as an asynchronous partitioning process by searching the highest open available sibling node on the master's search path.

For each problem, the solution found and elapsed processor time used by the master processor (as returned by the ANSI C **clock**() function) are recorded. Any single problem that is not solved by a prespecified resource limit (here, 100,000 CPU-seconds, or just over one CPU-day) is marked as *censored*, and the best solution found so far is returned for comparison. For the 600 problems attempted from six datasets, a total of only four problems were censored (*i.e.*, remained unsolved due to resource constraints) by the serial trial, with three of these problems occurring in the CATS scheduling domain and the last in the CATS paths domain. Of these, three were solved to optimality by the nagging system, but only one was solved by the partitioning system.

But aside from these small differences in the number of problems solved by each system, we are really more interested in measuring *speedup*, σ, a standard measure of performance. Speedup is defined as:

$$\sigma = \frac{t_s}{t_p}$$

where t_s is the serial solution time and t_p is the parallel solution time. A value $\sigma = 1$ implies no difference between the serial and parallel systems, while speedups greater than 1 imply the parallel system is faster. Comparing sets of speedup values requires some care. First, since speedups are ratios, geometric means should be used, since arithmetic means of ratios will not be meaningful (although they are still all too often reported) [5]. Unfortunately, geometric means of speedups treat improvements obtained on ''small'' problems to be equal in importance to improvements on ''large'' problems. For example, a 10 millisecond improvement (same order of magnitude as system clock resolution) on a problem solved serially in 1

second is considered equal to a 30 CPU-hour improvement on harder problems in these distributions. One alternate metric that makes the appropriate differentiation is the speedup ratio computed on arithmetic means of runtimes (as opposed to arithmetic means of speedup ratios) [26]. Finally, one must be especially careful when making comparisons of data sets that contain censored datapoints [23]. Since identical resource limits are imposed on both parallel and serial trials, doubly-censored datapoints will have unit speedup values but, if the same doubly-censored problems do not occur in all tested systems, may still lead to invalid results.

The following table reports minimum and maximum speedups, the (geometric) mean speedup ϕ and the speedup ratio of (arithmetic) means ψ:

$$\phi = \left(\prod_{i=1}^{N} \frac{t_{s_i}}{t_{p_i}} \right)^{\frac{1}{N}} \qquad \psi = \frac{\frac{1}{N} \sum_{i=1}^{N} t_{s_i}}{\frac{1}{N} \sum_{i=1}^{N} t_{p_i}}$$

for all tested systems. Singly-censored datapoints were included in all of the summary statistics shown here, since all singly-censored problems were censored by the serial system and not the parallel system. Their contributions, therefore, always represent underestimates of true speedup, and can never inflate apparent parallel system performance (doubly-censored datapoints are noted in the discussion, but excluded from the statistic).

	Nagging					**Partitioning**				
	N	$\sigma_{\min}$	$\sigma_{\max}$	ϕ	ψ	N	$\sigma_{\min}$	$\sigma_{\max}$	ϕ	ψ
Arbitrary	100	1.00	7.10	1.23	1.46	100	0.98	1.54	1.03	1.03
Matching	100	0.80	22.25	1.17	1.60	100	0.80	3.76	1.09	1.27
Paths	100	0.80	96.03	1.43	2.06	99	0.98	3.56	1.11	1.06
Regions	100	1.00	2.68	1.15	1.23	100	0.99	1.51	1.02	1.02
Scheduling	99	0.75	156,250.00	1.70	15.09	98	0.75	46,948.40	1.16	5.09
Sunflower	100	1.00	1.03	1.02	1.03	100	0.98	1.12	1.01	1.00

The values shown in the table merit closer scrutiny. First of all, based on values of ϕ and ψ shown in the table, it seems clear that nagging outperforms partitioning, on average, in all tested domains (more on this

point later). Second, it is also clear that the problem domain and its associated distribution have a great effect on the performance of both systems. Some distributions (*e.g.*, CATS scheduling and CATS paths) yield excellent expected speedups for nagging ($\psi = 15.09$ and $\psi = 2.06$, respectively) while other distributions (*e.g.*, sunflower) yield essentially no expected speedups at all ($\psi = 1.03$ and $\psi = 1.00$ for nagging and partitioning, respectively). Finally, we note that the huge maximum speedups observed for the CATS scheduling distribution are actually observed on singly-censored problems, and therefore represent underestimates of the true speedup on those problems. For the partitioning system, the cost of solving its lone singly-censored problem went from more than 100,000 CPU seconds (the resource bound) to 2.13 seconds; the same problem was solved in only 0.64 seconds by the nagging system (the nagging system also solved a second singly-censored problem that was not solved by the partitioning system in only 1.88 seconds). In fact, the time advantage obtained by nagging on just one of the two singly-censored scheduling problems exceeds the cumulative time used by the serial system for all of the solved scheduling problems. These very high speedups obtained on just a few problems have an understandably large impact on ψ (ϕ is much less sensitive to outliers).

While descriptive statistics can be a good way to summarize an experimental outcome, they fail to provide a more intuitive overview of the data. Figures 6 and 7 plot the results of this experiment for nagging and partitioning, respectively. What this graphical representation makes clear are the qualitative differences between nagging and partitioning. For nearly every domain, partitioning provides some speedup, as measured visually by the drop below $f(x) = x$, but rarely provides speedup that exceeds the number of processors applied (here, this would correspond to datapoints appearing below $f(x) = \frac{x}{2}$). In contrast, nagging leaves some datapoints on the $f(x) = x$ line, where effectively no speedup is obtained, but when nagging is faster, it is often supralinearly faster. In other words, when nagging works, it seems to provide greater speedup than the incremental improvements offered by partitioning. Moreover, the plots highlight how sub-unit speedups (*i.e.*, speedup values below 1.0, corresponding to datapoints above

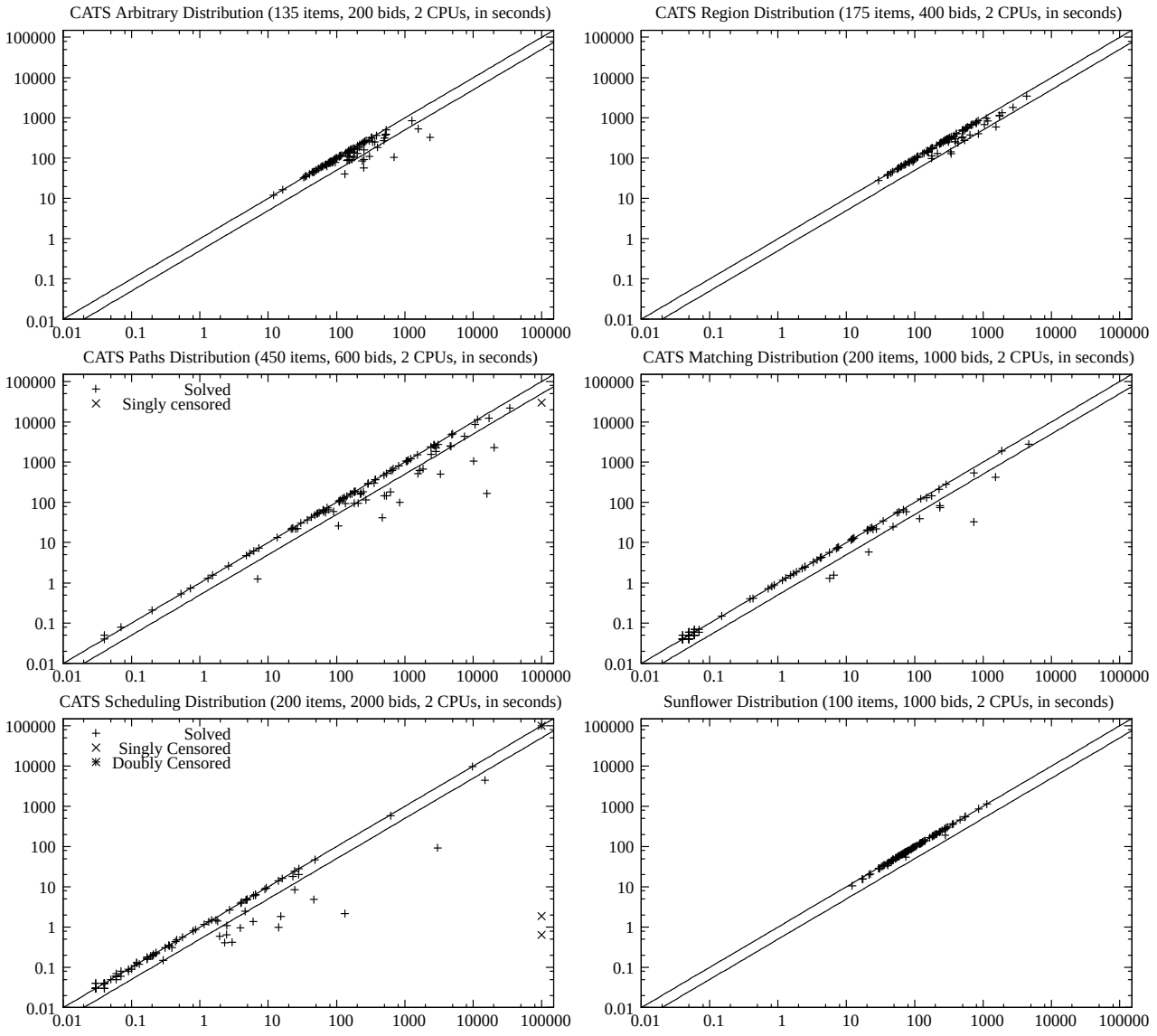

Figure 6: Nagging results obtained running randomly-generated problems from six different problem distributions. Performance is shown in log-log space, with serial solution time, in seconds, on the abscissa and the parallel solution time on the ordinate. Datapoints falling below the upper diagonal line ($f(x) = x$) are faster in parallel, while datapoints falling below the lower diagonal line ($f(x) = \frac{x}{2}$) are supralinearly faster (*i.e.*, more than twice as fast on two processors) in parallel. Doubly-censored datapoints, where neither the serial or parallel system was able to solve the problem in the allotted time, fall at the upper end of the upper diagonal line. Singly-censored datapoints (or those problems solved within the time limit by the parallel, but not the serial, system) systematically underestimate true speedup: their position in the plot therefore appears artificially displaced to the left of their true position.

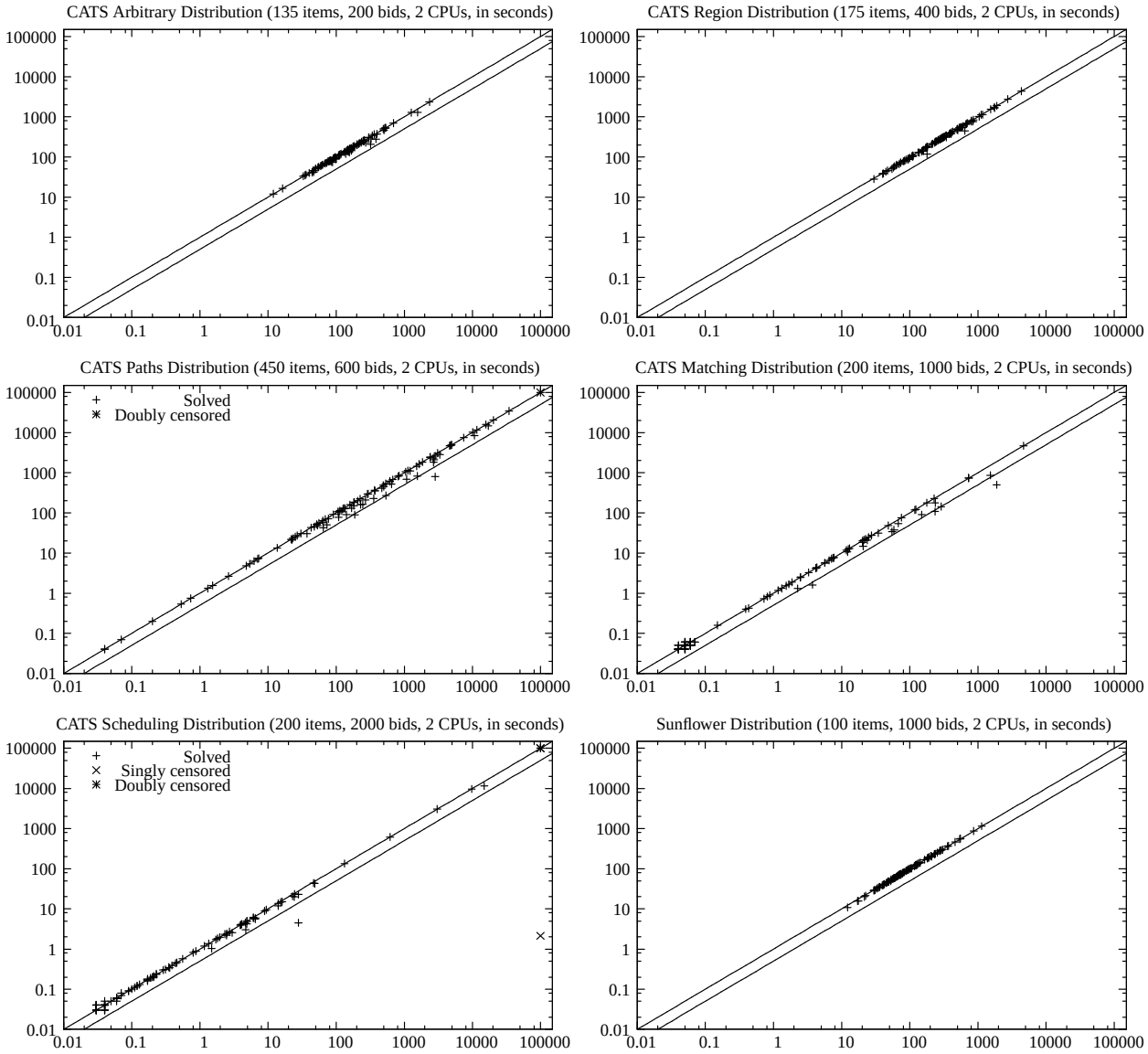

Figure 7: Partitioning results obtained running randomly-generated problems from six different problem distributions. Performance is shown in log-log space, with serial solution time, in seconds, on the abscissa and the parallel solution time on the ordinate. Datapoints falling below the upper diagonal line ($f(x) = x$) are faster in parallel, while datapoints falling below the lower diagonal line ($f(x) = \frac{x}{2}$) are supralinearly faster (*i.e.*, more than twice as fast on two processors) in parallel. Doubly-censored datapoints, where neither the serial or parallel system was able to solve the problem in the allotted time, fall at the upper end of the upper diagonal line. Singly-censored datapoints (or those problems solved within the time limit by the parallel, but not the serial, system) systematically underestimate true speedup: their position in the plot therefore appears artificially displaced to the left of their true position.

$f(x) = x$) occur only for very ‘‘small’’ problems, where the startup overhead of parallel search is not as easily amortized over longer solution times. Conversely, the greatest speedups are generally observed on ‘‘larger’’ problems, where the computational advantage, in terms of time saved, is most beneficial.

We can also use the data from this experiment to test, statistically, the null hypothesis ‘‘nagging is no faster than partitioning.’’ According to the logic of hypothesis testing, if we can reject the null hypothesis, then we can conclude that nagging is indeed a better choice than partitioning for the tested domain. To test our null hypothesis without making any distributional assumptions, we’ll use a nonparametric statistic, the paired Wilcoxon signed-ranks test [29]; what such nonparametric statistics may sacrifice in terms of power is more than counterbalanced by their broad applicability. Using the 100 paired samples from each problem domain as the input, the paired signed-ranks test easily rejects the null hypothesis using the traditional critical value for statistical significance ($p < 0.05$) for five of the six tested domains, including the sunflower domain where the speedups are small, but still favor nagging over partitioning. Only in the CATS matching domain do we fail to meet the critical value; in this domain, the relative performance merits of nagging and partitioning cannot be resolved with these data.

But what is most strikingly apparent from Figures 6 and 7 is that not all problem domains are created equal. In particular, the sunflower distribution shows little speedup is obtained with either nagging or partitioning, while some domains, like the CATS scheduling or paths distribution, provide a particularly rich environment for nagging to do well. A possible formal characterization of the differences between these problem domains, proposed in [24], relies on the expected problem solution times meeting certain distributional criteria under the problem transformation function applied by nagging. Examining whether our experiments support the proposed model is the focus of the next experiment.

7.3. Experiment 2

From previous work, we know that the distribution of solution times observed under the problem transformation function is critical in predicting whether nagging is likely to provide good speedups [24]. In particular, we know that heavy-tailed distributions generally correspond to better nagging performance.

While it is difficult to formally characterize the distribution of solution times under a particular problem transformation, it is easy to explore the distribution empirically, thus providing informal support for the proposed model. Here, we generate ten problems from two of the experimental problem distributions used in the first experiment: the CATS scheduling and CATS matching domains (recall that the former is a domain where nagging performs well, while the latter is the only domain where our paired Wilcoxon ranked-sign test failed to reject the null hypothesis). Each problem is then solved ten times using a single processor after first applying a random bidset ordering problem transformation function (the same problem transformation function used by the nagging processor in the first experiment). The observed solution times are sorted and converted to unitless *z*-scores for comparison. In this fashion, we can get a picture of how random problems drawn from the domain in question can be expected to behave under the problem transformation function.

Figure 8 plots the experimentally-obtained unitless distributions (in the form of unitless CDFs, or cumulative density functions computed from Z-scores) for both the CATS scheduling and matching domains, along with normal and lognormal standards for comparison (the lognormal distribution is an example of a heavy-tailed distribution). Recall that the theoretical model of [24] predicts, loosely speaking, that the heavier the tail, the better the performance of nagging. As the plot makes clear, the scheduling curve has the heavier tail (*i.e.*, its shape more closely matches the sample lognormal curve in Figure 8) than the matching curve, so the model would predict better nagging performance in the scheduling domain with respect to the matching domain, which is consistent with our results of our experiments.

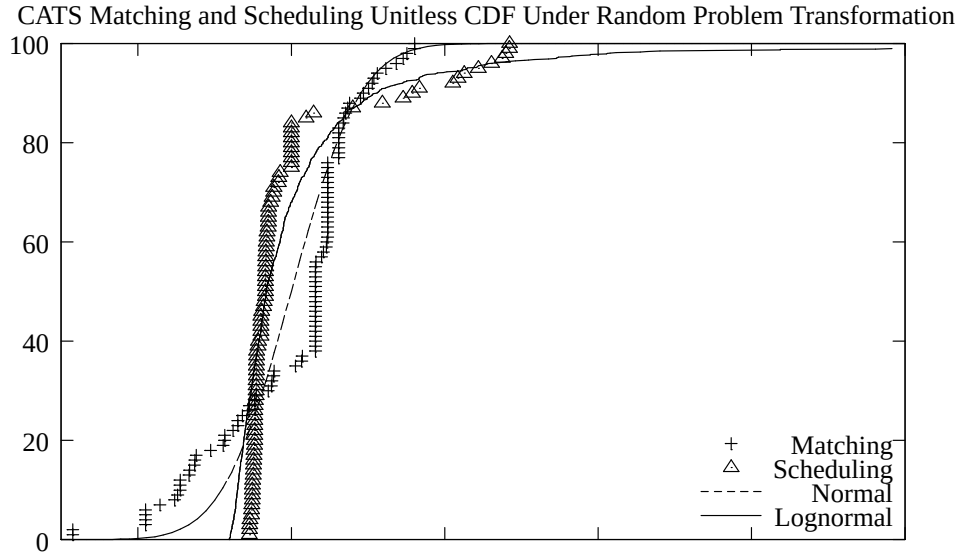

Figure 8: Empirically observed unitless cumulative density functions computed from Z-scores for the CATS scheduling and matching distributions. CDF's for the normal and lognormal distributions are included for comparison; note that the lognormal distribution is an example of a heavy-tailed distribution.

7.4. Experiment 3

In this experiment, we explore the scalability of nagging as applied to CA/WDP. We are particularly interested in seeing if the performance improvements observed in the first experiment increase with the number of processors. The experimental procedure follows that of the first experiment, except that here we will be using more processors ($P = 4$, $P = 8$, and $P = 16$), dynamically arranged by the NICE daemon in a hierarchy with no more than three descendants per processor. For this experiment, we focus on the CATS scheduling and paths distributions.

In terms of problems solved, for the CATS paths distribution, 99 were solved by both the serial system and all of the nagging systems, with the remaining problem censored by the serial system but solved by all of the nagging systems. The situation is slightly more complicated for the CATS scheduling distribution, where only 97 problems were solved by all of the systems: none of the remaining three

problems were solved by the serial system, resulting in two singly- and one doubly-censored problems for the $P = 2$, $P = 8$ and $P = 16$ CPU systems and one singly-censored and two doubly-censored problems for the $P = 4$ CPU system (the fact that the $P = 4$ CPU system censored a problem that was solved easily by the other nagging configurations — including one with fewer CPUs — is hardly surprising given the random problem transformation function employed).

As before, however, we are primarily interested in measuring speedup over the problem suite and not just the number of problems solved without censoring. The following table reports minimum and maximum speedups $\sigma_{\min}$ and $\sigma_{\max}$, the (geometric) mean speedup ϕ and the speedup ratio of (arithmetic) means ψ (two processor results are replicated from Section 7.2 for ease of comparison; moreover, as in the first experiment, we again include singly-censored problems but exclude doubly-censored problems, thus the values reported in all tests where censoring occurs necessarily represent lower bounds on the true speedup values).

Nagging	**Paths**					**Scheduling**				
P	N	$\sigma_{\min}$	$\sigma_{\max}$	ϕ	ψ	N	$\sigma_{\min}$	$\sigma_{\max}$	ϕ	ψ
2	100	0.8	96.03	1.43	2.06	99	0.75	156,250	1.70	15.09
4	100	0.97	14.67	1.39	1.75	98	0.75	126,582	1.85	6.49
8	100	0.99	74.72	2.16	3.76	99	0.75	126,582	1.85	2.67
16	100	0.99	56.69	2.42	5.57	99	0.75	153,846	2.18	22.90

Recall the value of ϕ can be used to compare the relative improvement between systems, while the value of ψ, by virtue of differentiating speedups according to actual time saved, gives a better picture of expected speedups. Here, both ϕ and ψ generally increase with the number of CPUs used; while ϕ increases nearly monotonically, the increase in ψ, which is more affected by outliers, is less so (since the transformation function is random, we would expect to observe quite a bit of fluctuation in the outliers, and this is indeed the case, with, *e.g.*, $126,582 \leq \sigma_{\max} \leq 156,250$ for the CATS scheduling domain). Note that improvements observed with partitioning were much lower than those found for nagging ($\phi_2 = 1.11$

and $\phi_{16} = 1.46$ for the CATS paths domain, $\phi_2 = 1.16$ and $\phi_{16} = 1.26$ for the CATS scheduling domain).

Partitioning	**Paths**					**Scheduling**				
P	N	$\sigma_{\min}$	$\sigma_{\max}$	ϕ	ψ	N	$\sigma_{\min}$	$\sigma_{\max}$	ϕ	ψ
2	99	0.98	3.56	1.11	1.06	98	0.75	46,948	1.16	5.09
16	99	0.96	20.36	1.46	1.32	98	0.75	47,393	1.26	7.55

A more intuitive interpretation of these data follows from Figure 9. A quick Gestalt of Figure 9 shows that increasing the number of processors does effectively cause the datapoints to move smoothly downwards, reflecting improved performance. This effect is especially apparent in the CATS paths domain (left side of Figure 9), and somewhat less so in the CATS scheduling domain if only because so many of the scheduling problems are solved very quickly anyway. The different balance of problem sizes between CATS paths and CATS scheduling domains also accounts for the measured improvement differences in ϕ: since the CATS scheduling domain has a larger proportion of ‘‘small’’ problems that are not helped much by partitioning, we expect it to produce smaller values of ϕ. In fact, the reported $\sigma_{\min} = 0.75$ correspond to very ‘‘small’’ problems, those solved serially in time close to the 10 msec resolution of the system clock (these datapoints appear just above the $f(x) = x$ line in the lower left corner of the plots; in this region, the limited clock resolution results in a grid-like cluster of datapoints). On the other hand, Figure 9 also makes clear that the largest speedups were in fact observed on ‘‘large’’ problems in the CATS scheduling domain (hence the relatively larger values of ψ). These are precisely the problems where a reduction in solution time is most beneficial.

7.5. Experiment 4

In the experiments just reported, we compare the performance of our parallel solver against the performance of the same solver running on one system. This final experiment is meant to ensure that the performance improvements noted are real; by comparing our serial system against another system from the literature, we ensure that our results are not due to some sort of implementation anomaly that results

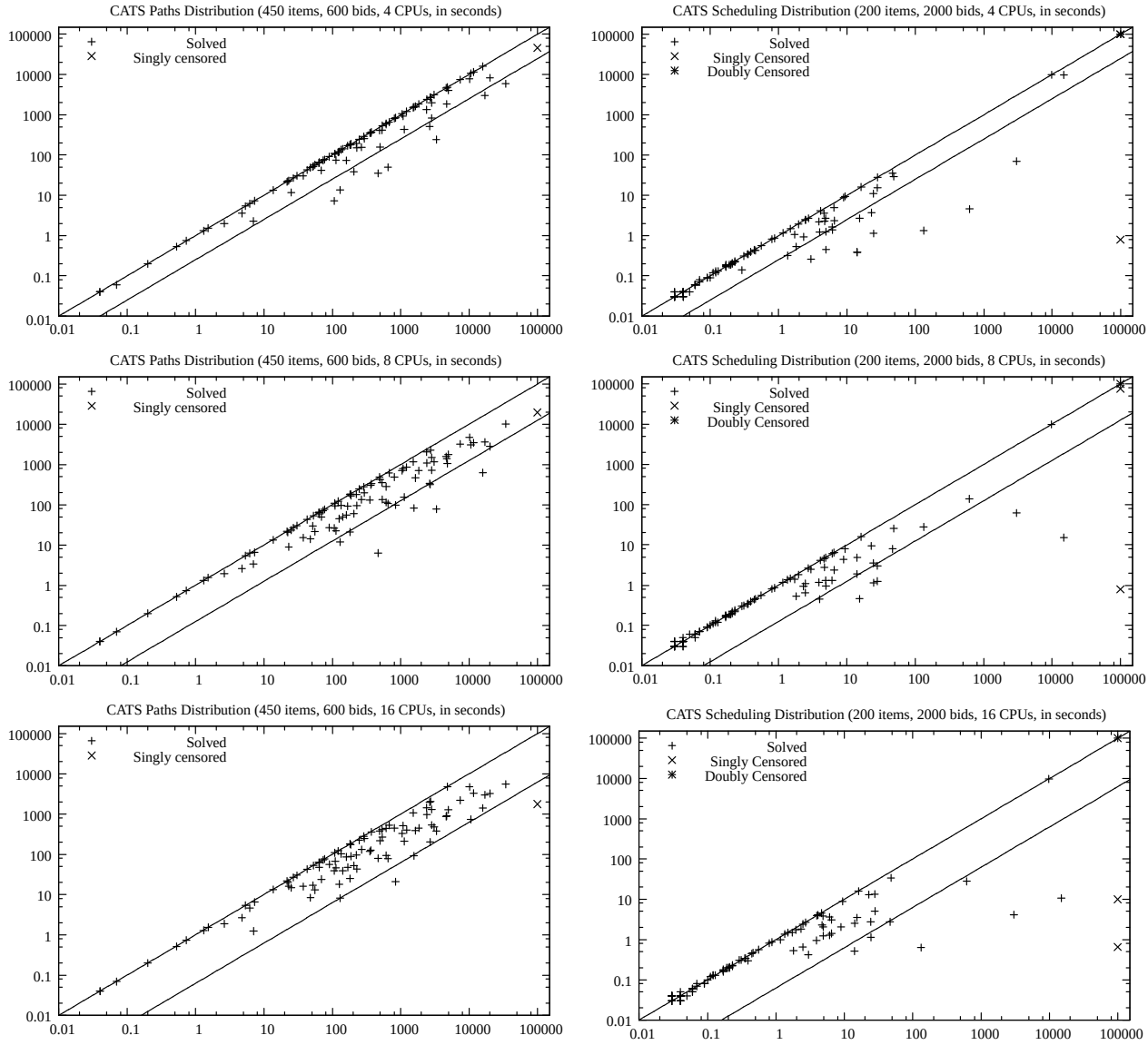

Figure 9: Nagging results obtained on the CATS paths and CATS scheduling problems of Experiment 1 with 4, 8, and 16 CPUs arranged in a NICE hierarchy with branching factor 3. Performance is shown in log-log space, with serial solution time, in seconds, on the abscissa and the parallel solution time on the ordinate. Datapoints falling below the upper diagonal line ($f(x) = x$) are faster in parallel, while datapoints falling below the lower diagonal line ($f(x) = \frac{x}{P}$) are supralinearly faster (*i.e.*, more than twice as fast on two processors) in parallel. Doubly-censored datapoints, where neither the serial or parallel system was able to solve the problem in the allotted time, fall at the upper end of the upper diagonal line. Singly-censored datapoints (or those problems solved within the time limit by the parallel, but not the serial, system) systematically underestimate true speedup: their position in the plot therefore appears artificially displaced to the left of their true position.

in an unusually slow serial solver.

As noted in Section 1, several optimal winner-determination solvers have been described elsewhere, usually with associated empirical test results (see, *e.g.*, CASS [7], CAMUS [15], and BOB [22]). Still others have used standard mixed integer programming formulations [1], coupled with high-quality optimized software, such as CPLEX.[5] Unfortunately, direct comparisons between our system and results published elsewhere are difficult to make, due to hardware differences, differences in test problem sets, and limited availability of expensive commercial software. Fortunately, the CASS system is freely available, so that meaningful comparison tests can, in fact, be performed on the same problem instances and hardware.

Here, we compare the performance of CASS with a our serial solver using 20 randomly-selected problem instances drawn from each of our six 100-element problem distributions, for a total of 120 problem instances. Each test problem was solved twice on identical AMD Athlon64 3400+ CPUs with 2GB of RAM; once with CASS (configured to use 2×10^8 bytes of cache), and once with our serial solver. Both systems were set to censor unsolved problems after 1200 seconds of CPU time had elapsed.

A quick look at observed the solution times indicates our system is faster than CASS in four domains, slower than CASS in one domain, and roughly as fast as CASS in the last domain. We can make this informal comparison more rigorous by testing the null hypothesis ‘‘our serial solver is no faster than CASS,’’ adopting the traditional critical value for statistical significance ($p < 0.05$). As before, we use the paired Wilcoxon signed-ranks test so as to avoid making any distributional assumptions about the observed solution times. Under these experimental conditions, we are able to reject the null hypothesis in four of the six problem domains (arbitrary, path, matching, and scheduling), while the dual null hypothesis (*i.e.*, ‘‘CASS is no faster than our serial solver’’) can be rejected for the sunflower domain. The

[5] See http://www.cplex.com for more information.

final domain (regions) is roughly a tie; neither null hypothesis can be rejected within the traditional critical value for statistical significance. Note that these results do not indicate *how much* faster one system is over the other; in fact, for those five domains where one system provides better performance than the other, the solution time advantage ranges from 1 to 3 orders of magnitude; interestingly enough, at least some of the domains where our serial solver is faster than CASS are precisely those domains where the parallel version of our solver provides its best speedups.[6]

The point of this experiment is not to compare CASS to our system; indeed, we are limiting our comparison here to our serial solver and not the (even faster) parallel version that is the subject of this paper. Rather, the point is to ensure that the results of the prior experiments are not artifacts of an unusual implementation. By calibrating against an existing system, and showing that our serial system is competitive with other systems of this ilk, we maintain that the positive performance effects observed due to parallelism are indeed realistic.

8. Discussion

The message of this paper is that nagging provides effective performance improvement for combinatorial auction winner determination problems. Empirical results demonstrate how additional processors reduce the time to solution and often provide supralinear benefits, particularly on ‘‘larger’’ problems, where serial solution times are especially prohibitive. We have also shown how the use of nagging as a search parallelization technique provides greater performance advantage than more

[6] A similar comparison with the CPLEX mixed integer programming solution is both more difficult to perform and less favorable in outcome. The CPLEX solution appears to be faster than our serial solver in all six domains, although a good bit of its performance advantage is due simply to faster hardware (at our site, CPLEX is licensed only to certain limited-use CPUs, which are faster than those used for our other tests). It is difficult to determine how much of the remaining performance advantage of CPLEX is due to the high quality of this commercial software product; it is worth noting that, when profiled, over 90% of our serial solver’s CPU time is devoted to calculating a new bound using an LP solver. Since our open-source LP solver is known to be much slower than CPLEX, our system could likely be made competitive with the CPLEX solution simply by incorporating the CPLEX LP solver once appropriate licensing issues are resolved.

traditional partitioning methods, and also provides better scalability to additional processors. In short, while the WDP problem remains *NP*-hard, we can provide optimal solutions to larger problem instances by incorporating additional processors via nagging.

The results presented are also interesting because they are somewhat inconsistent. Clearly, problem domain matters: some domains, like the CATS scheduling domain, seem intrinsically more amenable to nagging, while other domains, like the sunflower domain, are notably less so (not that partitioning does any better; it performs even more poorly than nagging in the sunflower domain, with $\psi = 1.00$ for $P = 2$). Two reasons may account for nagging's poor performance. First, the problem transformation function may provide inadequate shuffling, so that ''good'' bounds are not found quickly with high enough probability. However, even if the problem transformation function were at fault, we would expect to occasionally succeed in producing some speedup: yet the empirical results obtained ($\sigma_{\max} = 1.03$) fail to support this conjecture.[7] A second, and more compelling, rationale is that the bounding function may not be ''tight'' enough to provide pruning opportunities in problem domains that favor many near-optimal solutions. In these environments, it is less likely for bounds found by a (relatively weak) bounding function to exceed the best solution found so far (which may be quite close in value to the optimal solution). We note that prior work on this topic relied exclusively on empirical evaluations within the CATS domains, where linear programming does indeed appear to provide a ''tight'' bounding function [9]. In our experiments, we note that the sunflower problems behave quite differently, with much less pruning occurring, than problems from the CATS domains. Interestingly, it is precisely the sunflower domain where the CASS system, which does not rely on a linear programming bounding function, outperforms our serial solver, suggesting that one avenue for future study might be to merge additional bounding functions into our solver.

[7] Put another way, we would also expect to see ϕ and ψ (or at least $\sigma_{\max}$) increase as P increases, but this is not consistent with empirical data collected in the course of our study.

There are several additional avenues for future study. While nagging has been successfully applied to many different search algorithms and problem domains, including A* search, $\alpha\beta$ minimax search, Davis-Loveland-Putnam 3SAT problems, automated theorem proving, and Bayesian learning, each new application yields greater insight in the method and its impact on performance [6, 12, 24, 27]. Efforts are now underway to apply nagging to multi-mode resource-constrained project scheduling and other, similar, problems. Improvements to the NICE distributed infrastructure are also being studied, including a port of nagging to grid-based environments [16]. For CA/WDP, we are currently studying the effect of configuration and design decisions, including, *e.g.*, the branching factor of the NICE hierarchy, whether or not a nagger should abandon its search when it finds a better solution, and mixing nagging and partitioning.

The essence of nagging is to play multiple processors against each other in the hope of finding a problem transformation where better bounds are found faster. In this paper, we have shown that this is a viable strategy for performance improvement when solving combinatorial auction winner determination problems: in short, if you are seeking optimal solutions and you have multiple processors available, these processors can be successfully and usefully employed via nagging to produce the optimal solution in reduced time.

Acknowledgements

Support for this research was provided in part by the National Science Foundation through grant ITR/ACI0218491.

References

[1] A. Andersson, M. Tenhunen, and F. Ygge, "Integer Programming for Combinatorial Auction Winner Determination," *Proceedings of the Fourth International Conference on MultiAgent Systems (ICMAS 2000)*, Institute of Electrical and Electronics Engineers Computer Society (2000), p. 39.

[2] A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM Users Guide and Reference Manual*, Oak Ridge National Laboratory, Oak Ridge, TN (May 1994).

[3] A. Bonaccorsi, B. Codenotti, N. Dimitri, M. Leoncini, G. Resta, and P. Santi, "Generating Realistic Data Sets for Combinatorial Auctions," *Institute of Electrical and Electronics Engineers International Conference on E-Commerce* (June 2003), pp. 331-338.

[4] S. de Vries and R. Vohra, "Combinatorial Auctions: A Survey," Technical Report 1296, Center for Mathematical Studies in Economics and Management Science, Northwestern University, Evanston, IL (2000).

[5] P.J. Fleming and J.J. Wallace, "How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results," *Communications of the Association for Computing Machinery* **29**:3 (March 1986), pp. 219-221.

[6] S. Forman and A.M. Segre, "NAGSAT: A Randomized, Complete, Parallel Solver for 3SAT," *Fifth International Symposium on the Theory and Applications of Satisfiability Testing* (May 2002), pp. 236-243.

[7] Y. Fujishima, K. Leyton-Brown, and Y. Shoham, "Taming the Computational Complexity of Combinatorial Auctions: Optimal and Approximate Approaches," *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI99)* (1999), pp. 548-553.

[8] R. Gonen and D. Lehmann, "Optimal Solutions for Multi-Unit Combinatorial Auctions: Branch and Bound Heuristics," *Proceedings of the 2nd Association for Computing Machinery Conference on Electronic Commerce (EC00)* (2000), pp. 13-20.

[9] R. Gonen and D. Lehmann, "Linear Programming Helps Solving Large Multi-Unit Combinatorial Auctions," *Proceedings of the Electronic Market Design Workshop* (2001).

[10] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A High-Performance, Portable Implementation of the MPI Message-Passing Interface Standard," *Parallel Computing* **22**:6 (1996), pp. 789-828.

[11] H.H. Hoos and C. Boutilier, "Solving Combinatorial Auctions Using Stochastic Local Search," *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence* (2000), pp. 22-29.

[12] W. Lam and A.M. Segre, "A Parallel Learning Algorithm for Bayesian Inference Networks," *IEEE Transactions on Knowledge and Data Engineering* **14**:1 (January/February 2002), pp. 93-105.

[13] D. Lehmann, L.I. O'Callaghan, and Y. Shoham, "Truth Revelation in Rapid, Approximately Efficient Combinatorial Auctions," CS-TN-99-88, Stanford University, Palo Alto, CA (1999).

[14] K. Leyton-Brown, M. Pearson, and Y. Shoham, "Towards a Universal Test Suite for Combinatorial Auction Algorithms," *Proceedings of the 2nd Association for Computing Machinery Conference on Electronic Commerce (EC00)* (2000), pp. 66-76.

[15] K. Leyton-Brown, Y. Shoham, and M. Tennenholtz, "An Algorithm for Multi-Unit Combinatorial Auctions," *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence* (2000), pp. 56-61.

[16] Y. Liu, A.M. Segre, and S. Wang, "A High Throughput Approach to Combinatorial Search on Grids," *Proceedings of the 15th IEEE International Symposium on High Performance Distributed Computing* (June 2006), pp. 351-352.

[17] N. Nilsson, *Problem-Solving Methods in Artificial Intelligence*, McGraw Hill, New York, NY (1971).

[18] N. Nisan, "Bidding and Allocation in Combinatorial Auctions," *Proceedings of the 2nd Association for Computing Machinery Conference on Electronic Commerce (EC00)* (2000), pp. 1-12.

[19] S.J. Rassenti, V.L. Smith, and R.L. Bulfin, "A Combinatorial Auction Mechanism for Airport Time Slot Allocation," *Bell Journal of Economics* **13**:2 (1982), pp. 402-417.

[20] M.H. Rothkopf, A. Pekec, and R.M. Harstad, "Computationally Manageable Combinatorial Auctions," Technical Report 95-09, DIMACS, Rutgers, NJ (1995).

[21] T. Sandholm, "An Algorithm for Optimal Winner Determination in Combinatorial Auctions," *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI99)* (1999), pp. 542-547.

[22] T. Sandholm and S. Suri, "BOB: Improved Winner Determination in Combinatorial Auctions and Generalizations," *Artificial Intelligence* **145**:1-2, North Holland (2003), pp. 33-58.

[23] A.M. Segre, C.P. Elkan, and A. Russell, "A Critical Look at Experimental Evaluations of EBL," *Machine Learning* **6**:2, Kluwer Academic (March 1991), pp. 183-196.

[24] A.M. Segre, S. Forman, G. Resta, and A. Wildenberg, "Nagging: A Scalable, Fault-Tolerant, Distributed Search Paradigm," *Artificial Intelligence* **140**:1-2, North Holland (September 2002), pp. 71-106.

[25] A.M. Segre and D.B. Sturgill, "Using Hundreds of Workstations to Solve First-Order Logic Problems," *Proceedings of the Twelfth National Conference on Artificial Intelligence* (July 1994), pp. 187-192.

[26] J.E. Smith, "Characterizing Computer Performance with a Single Number," *Communications of the Association for Computing Machinery* **32**:10 (October 1988), pp. 1202-1206.

[27] D.B. Sturgill and A.M. Segre, "Nagging: A Distributed Adversarial Search-Pruning Technique Applied to First-Order Logic," *Journal of Automated Reasoning* **19**:3, Kluwer Academic (December 1997), pp. 347-376.

[28] D.B. Sturgill and A.M. Segre, "A Novel Asynchronous Parallelization Scheme for First-Order Logic," *Proceedings of the Twelfth Conference on Automated Deduction* (June 1994), pp. 484-498.

[29] F. Wilcoxon, "Individual Comparisons by Ranking Methods," *Biometrics* **1** (1945), pp. 80-83.

[30] E. Zurel and N. Nisan, "An Efficient Approximate Allocation Algorithm for Combinatorial Auctions," *Proceedings of the 3rd Association for Computing Machinery Conference on Electronic Commerce (EC01)* (2001), pp. 125-136.