Algorithm for Optimal Winner Determination in Combinatorial Auctions*

Tuomas Sandholm
sandholm@cs.cmu.edu
Computer Science Department
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213

Abstract

Combinatorial auctions, that is, auctions where bidders can bid on combinations of items, tend to lead to more efficient allocations than traditional auction mechanisms in multi-item auctions where the agents' valuations of the items are not additive. However, determining the winners so as to maximize revenue is $\mathcal{NP}$-complete. First, we analyze existing approaches for tackling this problem: exhaustive enumeration, dynamic programming, and restricting the allowable combinations. Second, we study the possibility of approximate winner determination, proving inapproximability in the general case, and discussing approximation algorithms for special cases. We then present our search algorithm for optimal winner determination. Experiments are shown on several bid distributions which we introduce. The algorithm allows combinatorial auctions to scale up to significantly larger numbers of items and bids than prior approaches to optimal winner determination by capitalizing on the fact that the space of bids is sparsely populated in practice. The algorithm does this by provably sufficient selective generation of children in the search tree, by using a secondary search for fast child generation, by using heuristics that are admissible and optimized for speed, and by preprocessing the search space in four ways. Incremental winner determination and quote computation techniques are presented.

We show that basic combinatorial auctions only allow bidders to express complementarity of items. We then introduce two fully expressive bidding languages, called XOR-bids and OR-of-XORs, with

*This material was first presented as an invited talk at the 1st International Conference on Information and Computation Economies, Charleston, SC, on 10/28/1998. A technical report version of this paper was circulated 1/28/1999 (Washington University in St. Louis, Department of Computer Science WUCS-99-01). A conference version of this paper appeared in the International Joint Conference on Artificial Intelligence (IJCAI), pp. 542–547, Stockholm, Sweden, July 31 – August 6, 1999. A brief nontechnical overview of different approaches to winner determination appeared in Decision Support Systems, 28, pp. 165–176, 2000.

which bidders can express general preferences (both complementarity and substitutability). The latter language is more concise. We show how these languages enable the use of the Vickrey-Clarke-Groves mechanism to construct a combinatorial auction where each bidder's dominant strategy is to bid truthfully. Finally, we extend our search algorithm and preprocessors to handle these languages as well as arbitrary XOR-constraints between bids.

Keywords: Auction, combinatorial auction, multi-item auction, multi-object auction, bidding with synergies, winner determination, multiagent systems.

1 Introduction

Auctions are popular, distributed and autonomy-preserving ways of allocating items (goods, resources, services, etc.) among agents. They are relatively efficient both in terms of process and outcome. They are extensively used among human bidders in a variety of task and resource allocation problems. More recently, Internet auction servers have been built that allow software agents to participate in the auctions as well [61], and some of these auction servers even have built-in support for mobile agents [47].

In an auction, the seller wants to sell the items and get the highest possible payments for them while each bidder wants to acquire the items at the lowest possible price. Auctions can be used among cooperative agents, but they also work in open systems consisting of self-interested agents. An auction can be analyzed using noncooperative game theory: what strategies are self-interested agents best off using in the auction (and therefore will use), and will a desirable social outcome—for example, efficient allocation—still follow. Auction mechanisms can be designed so that desirable social outcomes follow even though each agent acts based on self-interest.

This paper focuses on auctions with multiple distinguishable items to be allocated, but the techniques could also be used in the special case where some of the items are indistinguishable. These auctions are complex in the general case where the bidders have *preferences over bundles*, that is, a bidder's valuation for a bundle of items need not equal the sum of his valuations of the individual items in the bundle. This is often the case, for example, in electricity markets, equities trading, bandwidth auctions [34, 35], markets for trucking services [44, 43, 48], pollution right auctions, auctions for airport landing slots [40], and auctions for carrier-of-last-resort responsibilities for universal services [27]. There are several types of auction mechanisms that could be used in this setting, as the following subsections will discuss.

1.1 Sequential auction mechanisms

In a *sequential auction*, the items are auctioned one at a time [48, 5, 23]. Determining the winners in such an auction is easy because that can be done by picking the highest bidder for each item separately. However, bidding in a sequential auction is difficult if the bidders have preferences over bundles. To determine her valuation for an item, the bidder needs to estimate what items she will receive in later auctions. This requires speculation on what the others will bid in the future because that affects what items she will receive. Furthermore, what the others bid in the future depends on what they believe others will bid, etc. This counterspeculation introduces computational cost and other wasteful overhead. Moreover, in auctions with a reasonable number of items, such lookahead in the game tree is intractable, and then there is no known way to bid rationally. Bidding rationally would involve optimally trading off the computational cost of lookahead against the gains it provides, but that would again depend on how others strike that tradeoff. Furthermore, even if lookahead were computationally manageable, usually uncertainty remains about the others' bids because agents do not have exact information about each other. This often leads to inefficient allocations where bidders fail to get the combinations they want and get ones they do not.

1.2 Parallel auction mechanisms

As an alternative to sequential auctions, a parallel auction design can be used [36, 23]. In a *parallel auction* the items are open for auction simultaneously, bidders may place their bids during a certain time period, and the bids are publicly observable. This has the advantage that the others' bids partially signal to the bidder what the others' bids will end up being so the uncertainty and the need for lookahead is not as drastic as in a sequential auction. However, the same problems prevail as in sequential auctions, albeit in a mitigated form.

In parallel auctions, an additional difficulty arises: each bidder would like to wait until the end to see what the going prices will be, and to optimize her bids so as to maximize payoff given the final prices. Because every bidder would want to wait, there is a chance that no bidding would commence. As a patch to this problem, activity rules have been used [34]. Each bidder has to bid at least a certain volume (sum of her bid prices) by predefined time points in the auction, otherwise the bidder's future rights are reduced in some prespecified manner (for example, the bidder may be barred from the auction). Unfortunately, the game-theoretic equilibrium bidding strategies in such auctions are not known. It follows that the outcomes of such auctions are unknown for rational bidders.

1.3 Methods for fixing inefficient allocations

In sequential and parallel auctions, the computational cost of lookahead and counterspeculation cannot be recovered, but one can attempt to fix the inefficient allocations that stem from the uncertainties discussed above.

One such approach is to set up an aftermarket where the bidders can exchange items among themselves after the auction has closed. While this approach can undo some inefficiencies, it may not lead to an economically efficient allocation in general, and even if it does, that may take an impractically large number of exchanges among the agents [46, 3, 1].

Another approach is to allow bidders to retract their bids if they do not get the combinations that they want. For example, in the Federal Communications Commission's bandwidth auction the bidders were allowed to retract their bids [34]. In case of a retraction, the item was opened for reauction. If the new winning price was lower than the old one, the bidder that retracted the bid had to pay the difference. This guarantees that retractions do not decrease the auctioneer's payoff. However, it exposes the retracting bidder to considerable risk because at retraction time she does not know how much the retraction will end up costing her.

This risk can be mitigated by using a *leveled commitment* mechanism [51, 50], where the decommitting penalties are set up front, possibly on a per item basis. This mechanism allows the bidders to decommit but it also allows the auctioneer to decommit. A bidder may want to decommit, for example, if she did not get the combination that she wanted but only a subset of it. The auctioneer may want to decommit, for example, if he believes that he can get a higher price for the item from someone else. The leveled commitment mechanism has interesting strategic aspects: the agents do not decommit truthfully because there is a chance that the other agent will decommit, in which case the former agent is freed from the contract obligations, does not have to pay the decommitment penalty, and will collect a penalty from the latter agent. We showed that despite such strategic breach, in Nash equilibrium, the mechanism can increase the expected payoff of both parties and enable contracts which would not be individually rational to both parties via any full commitment contract [51, 45]. We also developed algorithms for computing the Nash equilibrium decommitting strategies and algorithms for optimizing the contract parameters [52]. In addition, we experimentally studied sequences of leveled commitment contracts and the associated cascade effects [2, 4].

Yet another approach would be to sell options for decommitting, where the price of the option is paid up front whether or not the option is exercised.

Each one of these methods can be used to implement bid retraction before and/or after the winning bids have been determined. While these methods can be used to try to fix inefficient allocations, it would clearly be desirable to get efficient allocations right away in the auction itself so no fixing would be necessary. Combinatorial auctions hold significant promise toward this goal.

1.4 Combinatorial auction mechanisms

Combinatorial auctions can be used to overcome the need for lookahead and the inefficiencies that stem from the related uncertainties [40, 44, 35, 43, 14]. In a combinatorial auction, there is one seller (or several sellers acting in concert) and multiple bidders.[1] The bidders may place bids on combinations of items. This allows a bidder to express complementarities between items so she does not have to speculate into an item's valuation the impact of possibly getting other, complementary items. For example, the Federal Communications Commission sees the desirability of combinatorial bidding in their bandwidth auctions, but so far combinatorial bidding has not been allowed—largely due to perceived intractability of winner determination.[2] This paper focuses on winner determination in combinatorial auctions where each bidder can bid on combinations (that is, bundles) of indivisible items, and any number of her bids can be accepted.

The rest of the paper is organized as follows. Section 2 defines the winner determination problem formally, analyzes its complexity, and discusses different approaches to attacking it. Section 3 presents our new optimal algorithm for this problem. Section 4 discusses the setup of our winner determination experiments, and Section 5 discusses the experimental results. Section 6 discusses incremental winner determination and quote computation as well as properties of quotes. Section 7 discusses other applications for the algorithm. Section 8 overviews related tree search algorithms. Section 9 discusses substitutability, introduces bidding languages to handle it, and develops an algorithm for determining winners under those languages. Section 10 presents conclusions and future research directions.

[1]Combinatorial auctions can be generalized to *combinatorial exchanges* where there are multiple sellers and multiple buyers [47, 53, 57].

[2]Also, the Commission was directed by Congress in the 1997 Balanced Budget Act to develop a combinatorial bidding system. Specifically, the Balanced Budget Act requires the Commission to " ... directly or by contract, provide for the design and conduct (for purposes of testing) of competitive bidding using a contingent combinatorial bidding system that permits prospective bidders to bid on combinations or groups of licenses in a single bid and to enter multiple alternative bids within a single bidding round." [59].

2 Winner determination in combinatorial auctions

We assume that the auctioneer determines the winners—that is, decides which bids are winning and which are losing—so as to maximize the seller's revenue. Such winner determination is easy in non-combinatorial auctions. It can be done by picking the highest bidder for each item separately. This takes $O(am)$ time where a is the number of bidders, and m is the number of items.

Unfortunately, winner determination in combinatorial auctions is hard. Let M be the set of items to be auctioned, and let $m = |M|$. Then any agent, i, can place a bid, $b_i(S) > 0$, for any combination $S \subseteq M$. We define the *length* of a bid to be the number of items in the bid.

Clearly, if several bids have been submitted on the same combination of items, for winner determination purposes we can simply keep the bid with the highest price, and the others can be discarded as irrelevant since it can never be beneficial for the seller to accept one of these inferior bids.[3] The highest bid price for a combination is

$$\bar{b}(S) = \max_{i \,\in\, \text{bidders}} b_i(S) \tag{1}$$

If agent i has not submitted a bid on combination S, we say $b_i(S) = 0$.[4] So, if no bidder has submitted a bid on combination S, then we say $\bar{b}(S) = 0$.[5]

Winner determination in a combinatorial auction is the following problem. The goal is to find a solution that maximizes the auctioneer's revenue given that each winning bidder pays the prices of her winning bids:

$$\max_{\mathcal{W} \in \mathcal{A}} \sum_{S \in \mathcal{W}} \bar{b}(S) \tag{2}$$

where $\mathcal{W}$ is a *partition*.

Definition. 2.1 *A* partition *is a set of subsets of items so that each item is included in at most one of the subsets. Formally, let $\mathcal{S} = \{S \subseteq M\}$. Then the set of partitions is*

$$\mathcal{A} = \{\mathcal{W} \subseteq \mathcal{S} | S, S' \in \mathcal{W} \Rightarrow S \cap S' = \emptyset\} \tag{3}$$

[3]Ties in the maximization can be broken in any desired way—for example, randomly or by preferring bids that are received earlier over ones received later.

[4]The assignment $b_i(S) = 0$ need not actually be carried out as long as special care is taken of the combinations that received no bids. As we show later, much of the power of our algorithm stems from not explicitly assigning a value of zero to combinations that have not received bids, and only constructing those parts of the solution space that are actually populated by bids.

[5]This corresponds to the auctioneer being able to keep items, or analogously, to settings where disposal of items is free.

Note that in a partition $\mathcal{W}$, some of the items might not be included in any one of the subsets $S \in \mathcal{W}$.

We use this notion of a partition for simplicity. It does not explicitly state who the subsets of items should go to. However, given an partition $\mathcal{W}$, it is trivial to determine which bids are winning and which are not: every combination $S \in \mathcal{W}$ is given to the bidder who placed the highest bid for S. So, a bidder can win more than one combination.

The winner determination problem can also be formulated as an integer programming problem where the decision variable $x_S = 1$ if the (highest) bid for combination S is chosen to be winning, and $x_S = 0$ if not. Formally:

$$\begin{aligned} \max_{\vec{x}} \quad & \sum_{S \in \mathcal{S}} \bar{b}(S) x_S \\ & \forall S \in \mathcal{S}:\ x_S \in \{0,1\} \text{ and } \forall i \in M:\ \sum_{S | i \in S} x_S \leq 1 \end{aligned} \tag{4}$$

The following subsections discuss different approaches to tackling the winner determination problem.

2.1 Enumeration of exhaustive partitions of items

One way to optimally solve the winner determination problem is to enumerate all *exhaustive partitions* of items.

Definition. 2.2 *An* exhaustive partition *is a partition where each item is included in exactly one subset of the partition.*

An example of the space of exhaustive partitions is presented in Figure 1. The number of exhaustive partitions grows rapidly as the number of items in the auction increases. The exact number of exhaustive partitions is

$$\sum_{q=1}^{m} Z(m,q) \tag{5}$$

where $Z(m,q)$ is the number of exhaustive partitions with q subsets, that is, the number of exhaustive partitions on level q of the graph. The quantity $Z(m,q)$—also known as the Stirling number of the second kind—is captured by the following recurrence:

$$Z(m,q) = qZ(m-1,q) + Z(m-1,q-1), \tag{6}$$

where $Z(m,m) = Z(m,1) = 1$. This recurrence can be understood by considering the addition of a new item to a setting with $m-1$ items. The first term, $qZ(m-1,q)$, counts the number of exhaustive partitions formed by adding the new item to one of the existing exhaustive partitions. There are

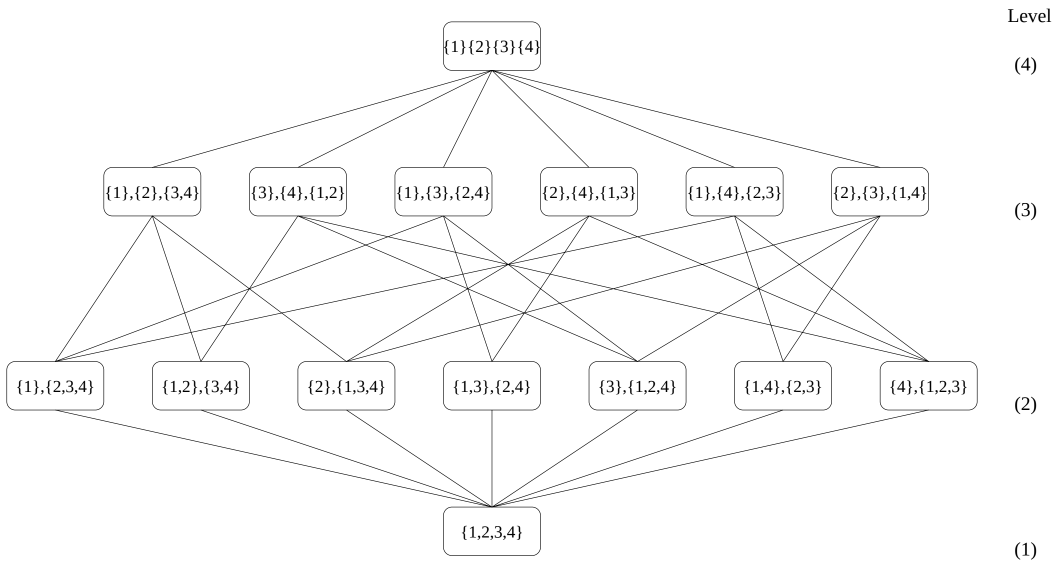

Figure 1: *Space of exhaustive partitions (in a 4-item example). Each vertex represents one exhaustive partition. Followed upward, each arc represents dividing one of the bundles within an exhaustive partition into two. Followed downward, each arc represents merging two bundles within an exhaustive partition into one.*

q choices because the existing exhaustive partitions have q subsets each. The second term, $Z(m-1, q-1)$, considers using the new item in a subset of its own, and therefore existing exhaustive partitions with only $m-1$ previous subsets are counted.

The following proposition characterizes the asymptotic complexity in closed form. The proof is presented in Appendix A.

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

The number of exhaustive partitions is so large that it is impossible to enumerate all of them unless the number of items in the auction is extremely small—below a dozen or so in practice. Therefore, any winner determination algorithm that completely enumerates the exhaustive partitions would not be viable in most settings.

2.2 Dynamic programming

Rather than enumerating the exhaustive partitions as above, the space of exhaustive partitions can be explored more efficiently using dynamic programming [42]. Based on the $\bar{b}(S)$ function, the dynamic program determines for each set S of items the highest possible revenue that can be obtained using only the items in S. The algorithm proceeds systematically from small sets to large ones. The needed optimal substructure property comes from the fact that for each set, S, the maximal revenue comes either from a single bid (with price $\bar{b}(S)$) or from the sum of the maximal revenues of *two* disjoint exhaustive subsets of S. In the algorithm below, $\mathcal{C}(S)$ corresponds to the smaller of these two subsets (or to the entire set S if that has higher revenue than the two subsets together). For each S, all possible subsets (together with that subset's complement in S) are tried. The dynamic programming algorithm is restated below.

Algorithm 2.1 (Dynamic program for winner determination)
INPUT: $\bar{b}(S)$ for all $S \subseteq A$. If no $\bar{b}(S)$ is specified in the input (no bids were received for that S), then $\bar{b}(S) = 0$.
OUTPUT: An optimal exhaustive partition $\mathcal{W}_{opt}$.

```
1. For all $x \in M$, set $\pi(x) := \bar{b}(\{x\})$, $\mathcal{C}(\{x\}) := \{x\}$

2. For $i = 2$ to $m$, do:
   For all $S \subseteq M$ such that $|S| = i$, do:

    (a) $\pi(S) := max\{\pi(S \setminus S') + \pi(S') : S' \subseteq S$ and $1 \leq |S'| \leq \frac{|S|}{2}\}$
    (b) If $\pi(S) \geq \bar{b}(S)$, then $\mathcal{C}(S) := S^*$ where $S^*$ maximizes the right
        hand side of (a)
    (c) If $\pi(S) < \bar{b}(S)$, then $\pi(S) := \bar{b}(S)$ and $\mathcal{C}(S) := S$

3. $\mathcal{W}_{opt} := \{M\}$

4. For every $S \in \mathcal{W}_{opt}$, do:
   If $\mathcal{C}(S) \neq S$, then

    (a) $\mathcal{W}_{opt} := (\mathcal{W}_{opt} \setminus \{S\}) \cup \{\mathcal{C}(S), S \setminus \mathcal{C}(S)\}$
    (b) Goto 4 and start with the new $\mathcal{W}_{opt}$
```

The time savings from dynamic programming compared to enumeration come from the fact that the revenue maximizing solutions for the subsets need not be computed over and over again, but only once. The dynamic program runs in $O(3^m)$ time [42]. This is significantly faster than enumeration. Clearly, the dynamic program also takes $\Omega(2^m)$ time because it looks at every $S \subseteq M$. Therefore, the dynamic program is still too complex to scale to large numbers of items—above about 20-30 in practice.

The dynamic program executes the same steps independent of the number of bids. This is because the algorithm generates each combination S even if no bids have been placed on S. Interpreted positively this means that the auctioneer can determine *ex ante* how long winner determination will take regardless of the number of bids that will be received. So, independent of the number of bids, dynamic programming is the algorithm of choice if the number of items is small. Interpreted negatively this means that the algorithm will scale only to a small number of items even if the number of bids is small. In Section 3 we present a search algorithm that avoids the generation of partitions that include combinations of items for which bids have not been submitted. That allows our algorithm to scale up to significantly larger numbers of items.

2.3 More compact problem representation and $\mathcal{NP}$-completeness

If no bid is received on some combination S, then those partitions $\mathcal{W}$ that include S need not be considered. The enumeration and the dynamic program discussed above do not capitalize on this observation. By capitalizing on this observation, one can restrict attention to *relevant partitions*.

Definition. 2.3 *The set of* relevant partitions *is*

$$\mathcal{A}' = \{\mathcal{W} \in \mathcal{A} | S \in \mathcal{W} \Rightarrow \text{ bid has been received on } S\} \tag{7}$$

That is, one can restrict attention to the following set of combinations of items:

$$\mathcal{S}' = \{S \subseteq M | \text{ bid has been received on } S\} \tag{8}$$

Let $n = |\mathcal{S}'|$. This is the number of relevant bids. Recall that for each combination S that has received a bid, only the highest bid is relevant; all other bids are discarded.

Now, winner determination can be formulated as the following integer program:

$$\begin{aligned} \max_{\vec{x}} \quad & \sum_{S \in \mathcal{S}'} \bar{b}(S) x_S \\ & \forall S \in \mathcal{S}' : \ x_S \in \{0,1\} \text{ and } \forall i \in M : \sum_{S | i \in S} x_S \leq 1 \end{aligned} \tag{9}$$

This integer program has fewer (or an equal number of) variables and constraints than integer program (4) because $\mathcal{S}' \subseteq \mathcal{S}$. The numbers are the same if and only if every combination has received a bid.

As discussed above, the complexity of dynamic programming is $O(3^m)$. Furthermore, $O(3^m) = O(2^{(\log_2 3)m}) = O((2^m)^{\log_2 3})$. If each combination of items has received a bid, the input includes $O(2^m)$ numbers, which means that the algorithm's running time, $O((2^m)^{\log_2 3})$, is polynomial in the size of the input. The following proposition states the general condition under which the dynamic program runs in polynomial time.

Proposition 2.2 *If the dynamic program runs in $O((n+m)^\rho)$ time for some constant $\rho > 1$, then $n \in \Omega(2^{m/\rho})$. If $n \in \Omega(2^{m/\rho})$ for some constant $\rho > 1$, then the dynamic program runs in $O(n^{\rho \log_2 3})$ time.*

Proof. Since the complexity of the dynamic program $f(n, m) \in \Omega(2^m)$, there exists a constant $c_1 > 0$ such that $f(n, m) \geq c_1 2^m$ for large m. If $f(n, m) \in O((n + m)^\rho)$, then there exists a constant $c_2 > 0$ such that $f(n, m) \leq c_2(n + m)^\rho$ for large $n + m$. Therefore, it must be the case that $c_1 2^m \leq c_2(n + m)^\rho$. Solving for n we get

$$n \geq \left[\frac{c_1}{c_2}(2^m)\right]^{1/\rho} - m \tag{10}$$

This implies $n \in \Omega(2^{m/\rho})$.

If $n \in \Omega(2^{m/\rho})$, then there exists a constant $c_3 > 0$ such that $n \geq c_3 2^{m/\rho}$ for large m. This means $m \leq \rho(\log n - \log c_3)$. Since $f(n, m) \in O(3^m)$, there exists a constant $c_4 > 0$ such that $f(n, m) \leq c_4 3^m$ for large m. Now we can substitute m to get $f(n, m) \leq c_4 3^{\rho(\log n - \log c_3)}$. Since $3^{\rho \log n} = n^{\rho \log 3}$, we get $f(n, m) \in O(n^{\rho \log_2 3})$. $\square$

However, the important question is not how complex the dynamic program is, because it executes the same steps regardless of what bids have been received. Rather, the important question is whether there exists an algorithm that runs fast in the size of the actual input, which might not include bids on all combinations. In other words, what is the complexity of problem (9)? Unfortunately, no algorithm can, in general, solve it in polynomial time in the size of the input (unless $\mathcal{P} = \mathcal{NP}$): the problem is $\mathcal{NP}$-complete [42]. Integer programming formulation (9) is the same problem as *weighted set packing* (once we view each bid as a set (of items) and the price, $\bar{b}(S)$, as the weight of the set S). $\mathcal{NP}$-completeness of winner determination then follows from the fact that weighted set packing is $\mathcal{NP}$-complete [26].

2.4 Polynomial-time approximation algorithms

One way to attempt to achieve tractability is to try to find a reasonably good relevant partition, $\mathcal{W} \in \mathcal{A}'$, instead of an optimal one. One would then like to trade off the expected cost of additional computation (cost of the computational resources and cost associated with delaying the result) against the expected improvement in the solution.

Instead of using expectations, one could try to devise an algorithm that will establish a worst case bound, that is, guarantee that the revenue from the optimal solution is no greater than some positive constant, k, times the revenue of the best solution found by the algorithm. A considerable amount of research has focused on generating such approximation algorithms that run in polynomial time in the size of the input. The relevant approximation algorithms were developed for the weighted set packing problem or the weighted independent set problem [25], but they can be used for winner determination. In this section we translate the known algorithms and inapproximability results from the theory of combinatorial algorithms into the winner determination problem.

2.4.1 General case

There has been considerable interest in taming the computational complexity of winner determination in combinatorial auctions by approximate winner determination [40, 31, 15]. The question of how close to optimal such algorithms get has been understood to be important, but has remained open. The following negative result shows that no polynomial-time algorithm can be constructed for achieving a reasonable worst case guarantee. The proof is based on a recent inapproximability result for maximum clique [22].

Proposition 2.3 (inapproximability) *For the winner determination problem (9), no polynomial-time algorithm can guarantee a bound $k \leq n^{1-\epsilon}$ for any $\epsilon > 0$ (unless $\mathcal{NP} = \mathcal{ZPP}$).*[6]

Proof. Assume for contradiction that there exists a polynomial-time approximation algorithm that establishes some bound $k \leq n^{1-\epsilon}$ for the winner determination problem. Then that algorithm could be used to solve the *weighted independent set* problem to the same k in polynomial time.[7] This is because a weighted independent set problem instance can be polynomially converted into a winner determination instance while preserving approximability. This can be done by generating one item for each edge in the graph. A bid is generated for each vertex in the graph. The bid includes exactly those items that correspond to the edges connected to the vertex.

Since the algorithm will k-approximate the weighted independent set problem in polynomial time, it will also k-approximate the independent set problem in polynomial time. A polynomial-time k-approximation algorithm for the independent set problem could directly be used to k-approximate the *maximum clique* problem in polynomial time. This is because the maximum clique problem is the independent set problem on the complement graph. But Håstad recently showed that no polynomial-time algorithm can establish a $k \leq n^{1-\epsilon}$ for any $\epsilon > 0$ for the maximum clique problem (unless $\mathcal{NP} = \mathcal{ZPP}$) [22]. Contradiction. □

[6] $\mathcal{ZPP}$ is the class of problems that can be solved in polynomial time with a randomized algorithm with zero probability of error [38].

[7] The weighted independent set problem is the problem of finding a maximally weighted collection of vertices in an undirected graph such that no two vertices are adjacent.

From a practical perspective the question of polynomial-time approximation with worst case guarantees has been answered since algorithms that come very close to the bound of the inapproximability result have been constructed. The asymptotically best algorithm establishes a bound $k \in O(n/(\log n)^2)$ [19].

One could also ask whether randomized algorithms would help in the winner determination problem. It is conceivable that randomization could provide some improvement over the $k \in O(n/(\log n)^2)$ bound. However, Proposition 2.3 applies to randomized algorithms as well, so no meaningful advantage could be gained from randomization.

A bound that depends only on the number of items, m, can be established in polynomial time. The following algorithm, originally devised by Halldórsson for weighted set packing, establishes a bound $k = 2\sqrt{m/c}$ [19]. The auctioneer can choose the value for c. As c increases, the bound improves but the running time increases. Specifically, steps 1 and 2 are $O(\binom{n}{c})$ which is $O(n^c)$, step 3 is $O(n)$, step 4 is $O(1)$, step 5 can be naively implemented to be $O(m^2n^2)$, and step 6 is $O(1)$. So, overall the algorithm is $O(\max(n^c, m^2n^2))$, which is polynomial for any given c.

Algorithm 2.2 (Greedy winner determination)
INPUT: $\mathcal{S}'$, *and* $\bar{b}(S)$ *for all* $S \in \mathcal{S}'$, *and an integer* c, $1 \le c \le m$.
OUTPUT: An approximately optimal relevant partition $\mathcal{W}_{approx} \in \mathcal{A}'$.

```
1. $\mathcal{A}'_c := \{\mathcal{W} \subseteq \mathcal{S}' : \ S, S' \in \mathcal{W} \Rightarrow S \cap S' = \emptyset, \ |\mathcal{W}| \le c\}$
2. $\mathcal{W}^*_c := \arg\max_{\mathcal{W} \in \mathcal{A}'_c} \ \sum_{S \in \mathcal{W}} \ \bar{b}(S)$
3. $\mathcal{S}'' := \{S \in \mathcal{S}' : |S| \le \sqrt{m/c}\}$
4. $t := 0$, $\mathcal{S}''_0 := \mathcal{S}''$
5. Repeat
     (a) $t := t + 1$
     (b) $X_t := \arg\max_{S \in \mathcal{S}''_{t-1}} \bar{b}(S)$
     (c) $Z_t := \{S \in \mathcal{S}''_{t-1} : X_t \cap S \ne \emptyset\}$
     (d) $\mathcal{S}''_t := \mathcal{S}''_{t-1} - Z_t$
   until $\mathcal{S}''_t = \emptyset$
6. If $\sum_{S \in \{X_1, X_2, ..., X_t\}} \ \bar{b}(S) > \sum_{S \in \mathcal{W}^*_c} \ \bar{b}(S)$
   then return $\{X_1, X_2, ..., X_t\}$
   else return $\mathcal{W}^*_c$
```

Another greedy algorithm for winner determination simply inserts bids into the allocation in largest $\frac{\bar{b}(S)}{\sqrt{|S|}}$ first order (if the bid shares items with another bid that is already in the allocation, the bid is discarded) [32]. This algorithm establishes a bound $k = \sqrt{m}$. If $c > 4$, the bound that Algorithm 2.2 establishes is better than that of this algorithm ($2\sqrt{m/c} < \sqrt{m}$). On the other hand, the computational complexity of Algorithm 2.2 quickly exceeds that of this algorithm as c grows.

If the number of items is small compared to the number of bids ($2\sqrt{m/c} < n$ or $\sqrt{m} < n$), as will probably be the case in most combinatorial auctions, then these algorithms establish a better bound than that of Proposition 2.3. The bound that Algorithm 2.2 establishes is about the best that one can obtain: Halldórsson et al. showed (using [22]) that a bound $k \le m^{1/2-\epsilon}$ cannot be established in polynomial time for any positive ϵ (unless $\mathcal{NP} = \mathcal{ZPP}$) [20]. However, the bound $k = m^{1/2-\epsilon}$ is so high that it is likely to be of limited value for auctions. Similarly, the bound $k = 2\sqrt{m/c} \ge 2$. Even a bound $k = 2$ would mean that the algorithm might only capture 50% of the available revenue. To summarize, the approach of constructing polynomial-time approximation algorithms with worst case guarantees seems futile in the general winner determination problem.

2.4.2 Special cases

While the general winner determination problem (9) is inapproximable in polynomial time, one can do somewhat better in special cases where the bids have special structure. For example, there might be some cap on the number of items per bid, or there might be a cap on the number of bids with which a bid can share items.

The desired special structure could be enforced on the bidders by restricting the allowable bids. However, that can lead to the same inefficiencies as non-combinatorial auctions because bidders may not be allowed to bid on the combinations they want. Alternatively, the auctioneer can allow general bids and use these special case algorithms if the bids happen to exhibit the desired special structure.

This section reviews the best polynomial-time algorithms for the known special cases. These algorithms were developed for the weighted independent set problem or the weighted set packing problem. Here we show how they apply to the winner determination problem:

1. If the bids have at most w items each, a bound $k = 2(w+1)/3$ can be established in $O(nw^2\Delta^w)$ time, where Δ is the number of bids that any given bid can share items with (note that $\Delta < n$) [6]. First, bids are greedily inserted into the solution on a highest-price-first basis. Then local search is used to improve the solution, and this search is terminated after a given number of steps. At each step, one new bid is inserted into the solution, and the old bids that share items with the new bid are removed from the solution. These improvements are chosen so as to maximize the ratio of the new bid's price divided by the sum of the prices of the old bids that would have to be removed.

2. Several algorithms have been developed for the case where each bid shares items with at most Δ other bids. A bound $k = \lceil(\Delta+1)/3\rceil$ can be established in linear time in the size of the input by partitioning the set of bids into $\lceil(\Delta+1)/3\rceil$ subsets such that $\Delta \leq 2$ in each subset, and then using dynamic programming to solve the weighted set packing problem in each subset [18]. Other polynomial-time algorithms for this setting establish bounds $k = \Delta/2$ [24] and $k = (\Delta+2)/3$ [21].

3. If the bids can be colored with c colors so that no two bids that share items have the same color, then a bound $k = c/2$ can be established in polynomial time [24].

4. The bids have a κ-claw if there is some bid that shares items with κ other bids which themselves do not share items. If the bids are free of κ-claws, a bound $k = \kappa - 2 + \epsilon$ can be established with local search in $n^{O(1/\epsilon)}$ time [18]. Another algorithm establishes a bound $k = (4\kappa+2)/5$ in $n^{O(\kappa)}$ time [18].

5. Let D be the largest d such that there is some subset of bids in which every bid shares items with at least d bids in that subset. Then, a bound $k = (D + 1)/2$ can be established in polynomial time [24].

Approximation algorithms for these known special cases have been improved repeatedly. There is also the possibility that probabilistic algorithms could improve upon the deterministic ones. In addition, it is possible that additional special cases with desirable approximability properties will be found. For example, while the current approximation algorithms are based on restrictions on the structure of bids and items, a new family of restrictions that will very likely lend itself to approximation stems from limitations on the prices. For example, if the function $\bar{b}(S)$ is close to additive, approximation should be easy. Unfortunately it does not seem reasonable for the auctioneer to restrict the bid prices or to discard outlier bids. Setting an upper bound could reduce the auctioneer's revenue because higher bids would not occur. Setting a lower bound above zero would disable bidders with lower valuations from bidding, and if no bidder has a valuation above the bound, no bids on those combinations would be placed. That can again reduce the auctioneer's revenue. Although forcing such special structure does therefore not make sense, the auctioneer could capitalize on special price structure if such structure happens to be present.

Put together, considerable work has been done on approximation algorithms for special cases of combinatorial problems, and these algorithms can be used for special cases of the winner determination problem. However, the worst case guarantees provided by the current algorithms are so far from optimum that they are of limited importance for auctions in practice.

2.5 Restricting the combinations to guarantee optimal winner determination in polynomial time

If even more severe restrictions apply to the bids, winner determination can be carried out optimally in polynomial time. To capitalize on this idea in practice, the restrictions would have to be imposed by the auctioneer since they—at least the currently known ones—are so severe that it is unlikely that they would hold by chance. In this section we review the bid restrictions that have been suggested for achieving polynomial winner determination:

1. If the bids have at most 2 items each, the winners can be optimally determined in $O(m^3)$ time using an algorithm for maximum-weight matching [42]. The problem is $\mathcal{NP}$-complete if the bids can have 3 items (this can be proven via a reduction from 3-set packing).

2. If the bids are of length 1, 2, or greater than m/c, then the winners can be optimally determined in $O((n_{long})^{c-1}m^3)$ time, where n_{long} is the

number of bids of length greater than m/c [42]. The $(n_{long})^{c-1}$ factor comes from exhaustively trying all combinations of $c-1$ long bids. Because each long bid uses up a large number of items, a solution can include at most $c-1$ long bids. The m^3 factor follows from solving the rest of the problem according to case 1 above. This part of the problem uses bids of length 1 and 2 on the items that were not allocated to the long bids.

3. If the allowable combinations are in a tree structure such that leaves correspond to items and bids can be submitted on any node (internal or leaf), winners can be optimally determined in $O(m^2)$ time [42]. For example, in Figure 2 left, a bid on 4 and 5 would be allowed while a bid for 5 and 6 would not. In tree structures the winners can be optimally determined by propagating information once up the tree. At every node, the best decision is to accept either a single bid for all the items in that node or the best solutions in the children of that node.

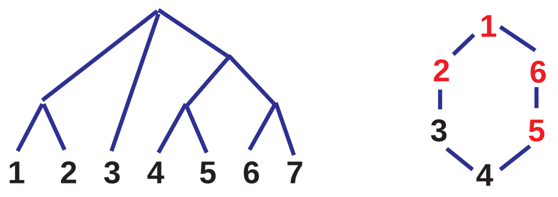

Figure 2: *Left: Allowable bids in a tree structure. Right: Interval bids.*

4. The items can be ordered and it can be required that bids are only placed on consecutive items [42]. For example, in Figure 2 right, a bid on 5, 6, 1, and 2 would be allowed while a bid on 5 and 1 would not. Without wrap-around, the winners can be optimally determined using dynamic programming. The algorithm starts from item 1, then does 1 and 2, then 1, 2, and 3, etc. The needed optimal substructure property comes from the fact that the highest revenue that can be achieved from items $1, 2, ..., h$ can be achieved either by picking a bid that has been placed on that entire combination, or by picking a bid that has been placed on the combination $g, ..., h$ and doing what was best for $1, ..., g-1$ (all choices $1 < g \leq h$ are tried). The algorithm runs in $O(m^2)$ time. Recently, Sandholm and Suri developed a faster dynamic program for this problem that runs in $O(m+n)$ time [53]. This is asymptotically the best one can hope for.

   If wrap-around is allowed, the winners can be optimally determined in $O(m^3)$ time by rerunning the $O(m^2)$ algorithm m times, each time cutting the chain at a different point [42]. Using the faster algorithm of Sandholm and Suri for each of these runs, this takes only $O(m(n+m))$ time [53].

5. One could introduce families of combinations of items where the winner determination is easy within each family, and any two combinations from different families intersect. The winner determination problem can then be solved within each family separately, and the solution from the family with the highest revenue chosen [42]. For example, one could lay the items in a rectangular grid. One family could then be columns and another family could be rows.

6. One could lay the items in a tree structure, and allow bidding on any subtree. This can be solved in $O(nm)$ time using a dynamic program [53]. The problem becomes $\mathcal{NP}$-complete already if the items are structured in a directed acyclic graph, and bids are allowed on any directed subtree.

Imposing restrictions on the bids introduces some of the same inefficiencies that are present in non-combinatorial auctions because the bidders may be barred from bidding on the combinations that they want. There is an inherent tradeoff here between computational speed and economic efficiency. Imposing certain restrictions on bids achieves provably polynomial-time winner determination but gives rise to economic inefficiencies.

3 Our optimal search algorithm

We generated another approach to optimal winner determination. It is based on highly optimized search. The motivation behind our approach is to

- allow bidding on all combinations—unlike the approach above that restricts the combinations. This is in order to avoid all of the inefficiencies that occur in non-combinatorial auctions. Recall that those auctions lead to economic inefficiency and computational burden for the bidders that need to look ahead in a game tree and counterspeculate each other.

- find the optimal solution (given enough time), unlike the approximation algorithms. This maximizes the revenue that the seller can obtain from the bids. Optimal winner determination also enables auction designs where every bidder has incentive to bid truthfully regardless of how others bid (this will be discussed later in the paper). This removes the bidder's motivation to counterspeculate and to bid strategically. If the bidders truthfully express their preferences through bids, revenue-maximizing winner determination also maximizes welfare in the system since the goods end up in the hands of the agents that value them the most.

- completely avoid loops and redundant generation of vertices, that are natural concerns for algorithms that would try to search the graph of exhaustive partitions (Figure 1) directly. Our algorithm does not search in that space directly, but instead searches in a more fruitful space.

- capitalize heavily on the sparseness of bids—unlike the dynamic program which uses the same amount of time irrespective of the number of bids. In practice the space of bids is likely to be extremely sparsely populated. For example, even if there are only 100 items to be auctioned, there are $2^{100} - 1$ combinations, and it would take longer than the life of the universe to bid on all of them even if every person in the world submitted a bid per second, and these people happened to bid on different combinations.[8] Sparseness of bids implies that the relevant partitions are a small subset of all partitions. Unlike the dynamic program, our algorithm only searches in the space of relevant partitions. It follows that the run time of our algorithm depends on the number of bids received, while in the dynamic program it does not.

3.1 Search space (SEARCH1)

We use tree search to achieve these goals. The input (after only the highest bid is kept for every combination of items for which a bid was received—all other bids are deleted) is a list of bids, one for each $S \in \mathcal{S}'$:

$$\{B_1, \ldots, B_n\} = \{(B_1.S, B_1.\bar{b}), \ldots, (B_n.S, B_n.\bar{b})\} \tag{11}$$

where $B_j.S$ is the set of items in bid j, and $B_j.\bar{b}$ is the price in bid j.

Each path in our search tree consists of a sequence of disjoint bids, that is, bids that do not share items with each other ($B_j.S \cap B_k.S = \emptyset$ for all bids j and k on the same path). So, at any point in the search, a path corresponds to a relevant partition.

Let U be the set of items that are already used on the path:

$$U = \bigcup_{j \mid B_j \text{ is on the path}} B_j.S \tag{12}$$

and let F be the set of free items:

$$F = M - U \tag{13}$$

A path ends when no bid can be added to the path. This occurs when for every bid, some of its items have already been used on the path ($\forall j, B_j.S \cap U \neq \emptyset$).

[8]However, in some settings there might be compact representations that the bidders can make that expand to a large number of bids. For example, a bidder may state that she is ready to pay \$20 for any 5 items in some set S of items. This would expand to $\binom{|S|}{5}$ bids.

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

$$g = \sum_{j | B_j \text{ is on the path}} B_j.\bar{b} \tag{14}$$

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

However, care has to be taken to correctly treat the possibility that the auctioneer may be able to keep items:

Proposition 3.1 *The auctioneer's revenue can increase if she can keep items, that is, if she does not have to allocate all items to the bidders. Keeping items increases revenue only if there is some item such that no bids (of positive price) have been submitted on that item alone.*

Proof. Consider an auction of two items: 1 and 2. Say there is no bid for item 1, a \$5 bid for item 2, and a \$3 bid for the combination of 1 and 2. Then it is more profitable (revenue \$5) for the auctioneer to keep 1 and to allocate 2 alone than it would be to allocate both 1 and 2 together (revenue \$3).

If there are bids of positive price for each item alone, the auctioneer would receive a positive revenue by allocating each item alone to a bidder (regardless of how the auctioneer allocates the other items). Therefore, the auctioneer's revenue would be strictly lower if he kept items. □

The auctioneer's possibility of keeping items can be implemented as a preprocessing step to the algorithm by placing *dummy bids* of price zero on those individual items that received no bids alone. For example in Figure 3, if item 1 had no bids on it alone and dummy bids were not used, the tree under 1 would not be generated and optimality could be lost. When dummy bids are included in the set of bids, every relevant partition $\mathcal{W} \in \mathcal{A}'$ is captured by at least one path from the root to a node (interior or leaf) in the search tree. This guarantees that the algorithm finds the optimal solution. Throughout the rest of the paper, we use this dummy bid technique. In terms of the notation, we will say that the dummy bids are already included in the list of bids $\{B_1, \ldots, B_n\}$, so the total number of bids, including the dummy bids, is n. It follows that $n \geq m$.

A naive method of constructing the search tree would include all bids (that do not include items that are already on the path) as the children of each node. Instead, the following proposition enables a significant reduction

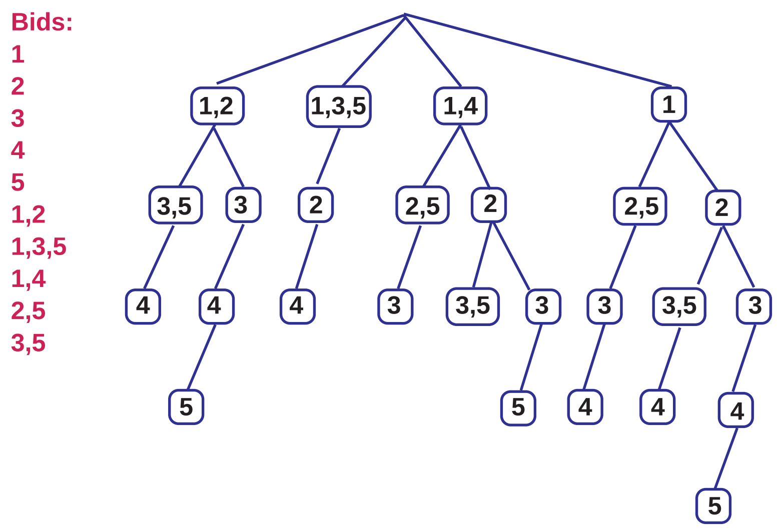

Figure 3: *SEARCH1. This example search space corresponds to the bids listed on the left. In the figure, for each bid, the items are shown but the price is not.*

of the branching factor by capitalizing on the fact that the order of the bids on a path does not matter.

Proposition 3.2 *Every relevant partition $\mathcal{W} \in \mathcal{A}'$ is represented in the tree by exactly one path*[9] *from the root to a node (interior or leaf) if the children of a node are those bids that*

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

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

$$children(\theta) = \{B \in \{B_1, \ldots, B_n\} : i^* \in B.S, B.S \cap U = \emptyset\} \tag{15}$$

Proof. We first prove that each relevant partition $\mathcal{W} \in \mathcal{A}'$ is represented by at most one path from the root to a node. The first condition of the proposition leads to the fact that a partition can only be generated in one order of bids on the path. So, for there to exist more than one path for a

[9]At any search node, an item can be allocated to a bidder via a bid, or to the auctioneer via a dummy bid, or it might not yet be allocated. Note the difference between the latter two possibilities: allocating one of the unallocated items to a dummy bid yields a new relevant partition. At any leaf of the search tree, each item is allocated either to a bidder via a bid or to the auctioneer via a dummy bid.

given partition, some bid would have to occur multiple times as a child of some node. However, the tree uses each bid as a child for a given node only once.

What remains to be proven is that each relevant partition is represented by some path from the root to a node in the tree. Assume for contradiction that some relevant partition $\mathcal{W} \in \mathcal{A}'$ is not. Then, at some point, there has to be a bid in that partition such that it is the bid with the item with the smallest index among those not on the path, but that bid is not inserted to the path. Contradiction. $\square$

Our search algorithm restricts the children according to Proposition 3.2. This can be seen, for example, at the first level of Figure 3 because all the bids considered at the first level include item 1. Figure 3 also illustrates the fact that the minimal index, i^*, does not coincide with the depth of the search tree in general.

To summarize, in the search tree, a path from the root to a node (interior or leaf) corresponds to a relevant partition. Each relevant partition $\mathcal{W} \in \mathcal{A}'$ is represented by exactly one such path. The other partitions $\mathcal{W} \in \mathcal{A} - \mathcal{A}'$ are not generated. We call this search SEARCH1.

3.1.1 Size of the tree in SEARCH1

In this section we analyze the worst case size of the tree in SEARCH1.

Proposition 3.3 *The number of leaves in SEARCH1 is no greater than $(\frac{n}{m})^m$. Also, the number of leaves in SEARCH1 is no greater than $\sum_{q=1}^{m} Z(m,q) \in O(m^m)$ (see Equations 5 and 6 for the definition of Z). Furthermore, the number of leaves in SEARCH1 is no greater than 2^n.*

The number of nodes in SEARCH1 (excluding the root) is no greater than m times the number of leaves. The number of nodes in SEARCH1 is no greater than 2^n.

Proof. We first prove that the number of leaves is no greater than $(\frac{n}{m})^m$. The depth of the tree is at most m since every node on a path uses up at least one item. Let N_i be the set of bids that include item i but no items with a smaller index than i. Let $n_i = |N_i|$. Clearly, $\forall i, j,\ N_i \cap N_j = \emptyset$, so $n_1 + n_2 + \ldots + n_m = n$. An upper bound on the number of leaves in the tree is given by $n_1 \cdot n_2 \cdot \ldots \cdot n_m$ because the branching factor at a node is at most n_{i^*}, and i^* increases strictly along every path in the tree. The maximization problem

$$\begin{aligned} \max \quad & n_1 \cdot n_2 \cdot \ldots \cdot n_m \\ \text{s.t.} \quad & n_1 + n_2 + \ldots + n_m = n \end{aligned}$$

is solved by $n_1 = n_2 = \ldots = n_m = \frac{n}{m}$. Even if n is not divisible by m, the value of the maximization is an upper bound. Therefore, the number of leaves in the tree is no greater than $(\frac{n}{m})^m$.

Now we prove that the number of leaves in SEARCH1 is no greater than $\sum_{q=1}^{m} Z(m,q) \in O(m^m)$. Since dummy bids are used, each path from the root to a leaf corresponds to a relevant exhaustive partition. Therefore the number of leaves is no greater than the number of exhaustive partitions (and is generally lower since not all exhaustive partitions are relevant). In Section 2.1 we showed that that the number of exhaustive partitions is $\sum_{q=1}^{m} Z(m,q)$. By Proposition 2.1, this is $O(m^m)$.

Next we prove that the number of nodes (and thus also the number of leaves) is no greater than 2^n. There are 2^n combinations of bids (including the one with no bids). In the search tree, each path from a root to a node corresponds to a unique combination of bids (the reverse is not true because in some combinations bids share items, so those combinations are not represented by any path in the tree). Therefore, the number of nodes is no greater than 2^n.

Because there are at most m nodes on a path (excluding the root), the number of nodes in the tree (excluding the root) is no greater than m times the number of leaves. $\square$

Proposition 3.4 *The bound $(\frac{n}{m})^m$ is always tighter (lower) than the bound 2^n.*

Proof.

$$\begin{aligned} & (\frac{n}{m})^m < 2^n \\ \Leftrightarrow \quad & m \log \frac{n}{m} < n \\ \Leftrightarrow \quad & \log \frac{n}{m} < \frac{n}{m} \end{aligned}$$

which holds for all positive numbers n and m. $\square$

The bound $(\frac{n}{m})^m$ shows that the number of leaves (and nodes) in SEARCH1 is polynomial in the number of bids even in the worst case if the number of items is fixed. On the other hand, as the number of items increases, the number of bids also increases ($n \geq m$ due to dummy bids), so in the worst case, the number of leaves (and nodes) in SEARCH1 remains exponential in the number of items m.

3.2 Fast generation of children (SEARCH2)

At any given node, θ, of the tree, SEARCH1 has to determine $children(\theta)$. In other words, it needs to find those bids that satisfy the two conditions of Proposition 3.2. A naive approach is to loop through a list of all bids at every search node, and accept a bid as a child if it includes i^*, and does not include items in U. This takes $\Theta(nm)$ time per search node because it loops through the list of bids, and for each bid it loops through the list of items.[10]

We use a more sophisticated scheme to make child generation faster. Our version of SEARCH1 uses a secondary depth-first search, SEARCH2, to quickly determine the children of a node. SEARCH2 takes place in a different space: a data structure which we call the *Bidtree*. It is a binary tree in which the bids are inserted up front as the leaves (only those parts of the tree are generated for which bids are received) (Figure 4).

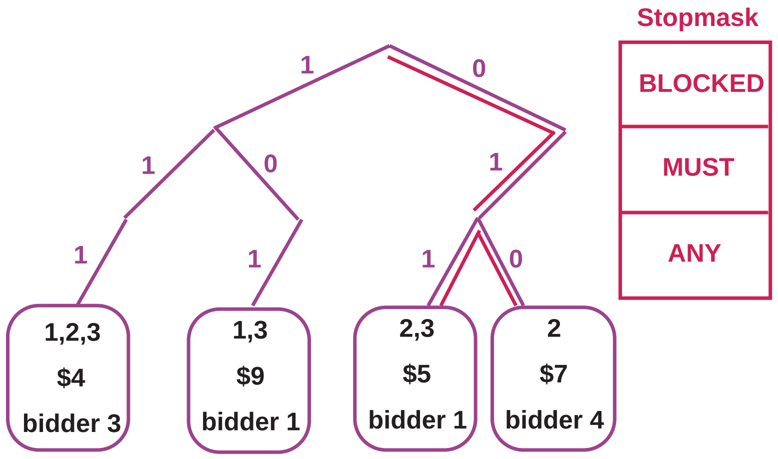

Figure 4: *Bidtree data structure. The Stopmask is used to select parts of the Bidtree where SEARCH2 is allowed to go.*

The use of a *Stopmask* differentiates the Bidtree from a classic binary tree. The Stopmask is a vector with one variable for each item, $i \in M$. Stopmask[i] can take on any one of three values: BLOCKED, MUST, or ANY. If Stopmask[i] = BLOCKED, SEARCH2 will never progress left at depth i.[11] This has the effect that those bids that include item i are pruned instantly and in place. If, instead, Stopmask[i] = MUST, then SEARCH2 cannot progress right at depth i. This has the effect that all other bids except those that include item i are pruned instantly and in place. Stopmask[i] =

[10]On a k-bit architecture, this time can be reduced by a factor of k by representing $B.S$ as a bitmask of items, and by representing U as a bitmask of items. The machine can then compute $B.S$ XOR U on the bitmasks k bits at a time. If this does not return 0, then the bid includes items in U.

[11]The root of the tree is at depth 1.

ANY corresponds to no pruning based on item i: SEARCH2 may go left or right at depth i. Figure 4 illustrates how particular values in the Stopmask prune the Bidtree.

SEARCH2 is used to generate children in SEARCH1. The basic principle is that at any given node of SEARCH1, Stopmask[i] = BLOCKED for all $i \in U$, and Stopmask[i^*] = MUST, and Stopmask[i] = ANY for all other values of i. Given these variable settings, SEARCH2 will return exactly those bids that satisfy Proposition 3.2.[12]

A naive implementation of this would set the Stopmask values anew at every node, θ, of SEARCH1, and would generate the entire list $children(\theta)$ before continuing SEARCH1 to the first child of θ. We deploy a faster method for setting the Stopmask values. We also only generate one child of θ at a time. In what follows, we present the technique in detail.

When SEARCH1 begins, Stopmask[1] = MUST, and Stopmask[i] = ANY for $i \in \{2, \ldots, m\}$. The first child of any given node, θ, of SEARCH1 is determined by a depth-first search (SEARCH2) from the top of the Bidtree until a leaf (bid) is reached. This bid becomes the node that is added to the path of SEARCH1. Every time a bid, B, is appended to the path of SEARCH1, the algorithm sets Stopmask[i] = BLOCKED for all $i \in B.S$ and Stopmask[i^*] = MUST. These MUST and BLOCKED values are changed back to ANY when backtracking a bid from the path of SEARCH1, and the MUST value is reallocated to the place in the Stopmask where it was before that bid was appended to the path. The next unexplored sibling of any child, η, of SEARCH1 is determined by continuing SEARCH2 by backtracking in the Bidtree[13] after SEARCH1 has explored the tree under η. Note that SEARCH2 never needs to backtrack above depth i^* in the Bidtree because all items with smaller indices than i^* are already used on the path of SEARCH1.

3.2.1 Ordering of children in SEARCH1 via SEARCH2

Since SEARCH2 is a depth-first search in the Bidtree, $children(\theta)$ for any given SEARCH1 node θ are generated in lexicographic order of the bid's set of items, viewed as a vector of 0's and 1's. For example, in an auction with five items, the item vector of a bid that contains items 2 and 4 would be $\langle 0, 1, 0, 1, 0 \rangle$. To demonstrate the lexicographic order, let $i^* = 2$ for the current θ, and let there be two other bids that include item 2. Let their item

[12]We implemented SEARCH2 to execute in place, that is, without memory allocation during search. That is accomplished via the observation that an open list (or a recursion stack) is not required here—unlike in most search algorithms. This is because in depth-first search where the search tree is entirely kept in memory, to decide where to go next, it suffices to know where the search focus is now, and from where it most recently came.

[13]To enable this, our algorithm stores, for each depth of SEARCH1 separately, where the search focus of SEARCH2 in the Bidtree is, and from where it most recently came.

vectors be $\langle 0,1,0,1,1\rangle$ and $\langle 0,1,1,0,0\rangle$. Now, the children of θ would be generated in the following order of item vectors:

$$\langle 0,1,1,0,0\rangle, \langle 0,1,0,1,1\rangle, \langle 0,1,0,1,0\rangle.$$

Since the children are generated—and the subtrees under them explored—one by one, this order of generating the children is also the order in which the search branches are explored. One implication of this ordering is that a bid with only one item is always the last child to be explored (Figure 3).

Alternatively, one could generate all the children of a SEARCH1 node one after the other using SEARCH2, and store them (for example, in a linked list) before exploring any of them. This would allow the children to be ordered based on additional considerations. Exploring more promising children first could improve the performance of SEARCH1 because good solutions would be found earlier. This has two main advantages. First, if SEARCH1 needs to be terminated before it has completed, a better solution will be available. Second, finding good solutions earlier allows more pruning of later search paths in SEARCH1 based on a bounding technique, discussed later in Section 3.5.

3.2.2 Complexity of SEARCH2

As discussed in the beginning of the previous section, the naive approach to generating $children(\theta)$ takes $\Theta(nm)$ time in the worst case. As desired, the use of SEARCH2 to generate $children(\theta)$ reduces this complexity, even in the worst case. For any given θ, finding the entire set $children(\theta)$, one at a time, corresponds to conducting a depth-first search (SEARCH2) in the Bidtree. The complexity of SEARCH2 is no greater than the number of edges in the Bidtree times two (to account for backtracks). The following proposition gives a tight upper bound on the number of edges in the Bidtree.

Proposition 3.5 *In a tree that has uniform depth $m+1$ (under the convention that the root is at depth 1), n leaves, and where any node has at most two children (as is the case in SEARCH2), the number of edges is at most*

$$nm - n\lfloor \log n\rfloor + 2\cdot 2^{\lfloor \log n\rfloor} - 2 \tag{16}$$

This bound is tight. Under the innocuous assumption that $m - \log n \geq c$ for some constant $c > 0$, this is

$$O(n\cdot(m-\log n)) \tag{17}$$

On the other hand, under the assumption that $m - \log n < c$, this is

$$O(n) \tag{18}$$

The proof of Proposition 3.5 is given in Appendix B.

While the complexity reduction from using SEARCH2 instead of the naive method is only moderate in the worst case, in many cases, the complexity reduction is significantly greater. As an extreme example, in the best case SEARCH2 only takes $O(1)$ time (for example, if all bids include item 1 and item 1 is already used on the path), while the complexity of the naive approach remains $\Theta(nm)$. Another good case for SEARCH2 is a wide-late-tree (Figure 20 right), where the upper part has $O(m-\log n)$ edges and the lower part has $O(2^{\log n}) = O(n)$ edges, so the overall complexity is $O(\max(m - \log n, n))$.

Also, the analysis in this section pertains to a single SEARCH1 node. Due to substantial pruning using the Stopmask, all nodes in SEARCH1 cannot suffer this worst case in child generation. Future research should further study the complexity of child generation when amortized over all SEARCH1 nodes. As will be discussed later in this paper, some of the SEARCH1 nodes will be pruned. Therefore, a sophisticated analysis of child generation would only amortize over those SEARCH1 nodes for which children are actually generated.

3.3 Anytime winner determination: Using a depth-first search strategy for SEARCH1

We first implemented the main search (SEARCH1) as depth-first search. It runs in linear space (not counting the statically allocated Bidtree data structure that uses slightly more memory as shown in Proposition 3.5). Feasible solutions are found quickly. The first relevant exhaustive partition is found at the end of the first search path (that is, at the left-most leaf). In fact, even a path from the root to an interior node could be converted into a feasible solution by having the auctioneer keep the items F that have not been allocated on the path so far.

The solution improves monotonically since our algorithm keeps track of the best solution found so far. This implements the anytime feature: if the algorithm does not complete in the desired amount of time, it can be terminated prematurely, and it guarantees a feasible solution that improves monotonically. When testing the anytime feature, it turned out that in practice most of the revenue was generated early on as desired, and there were diminishing returns to computation.

3.4 Preprocessing

Our algorithm preprocesses the bids in four ways to make the main search faster without compromising optimality. The preprocessors could also be used in conjunction with approaches to winner determination other than our search algorithm. The next subsections present the preprocessors in the order in which they are executed.

3.4.1 PRE1: Keep only the highest bid for a combination

When a bid arrives, it is inserted into the Bidtree. If a bid for the same set of items S already exists in the Bidtree (that is, the leaf that would have been created for the new bid already exists), only the bid with the higher price is kept, and the other bid is discarded. Ties can be broken randomly or, for example, in favor of the bid that was received earlier. Inserting a bid into the Bidtree is $\Theta(m)$ because insertion involves following or creating a path of length m. There are n bids to insert. So, the overall time complexity of PRE1 is $\Theta(mn)$.

3.4.2 PRE2: Remove provably noncompetitive bids

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

To determine this we search, for each bid (potential prunee), through all combinations of its disjoint subset bids. This is the same depth-first search as the main search except that it restricts the search to those bids that only include items that the prunee includes (Figure 5): BLOCKED is kept in the Stopmask for other items. This ensures that only subset bids contribute to the pruning.

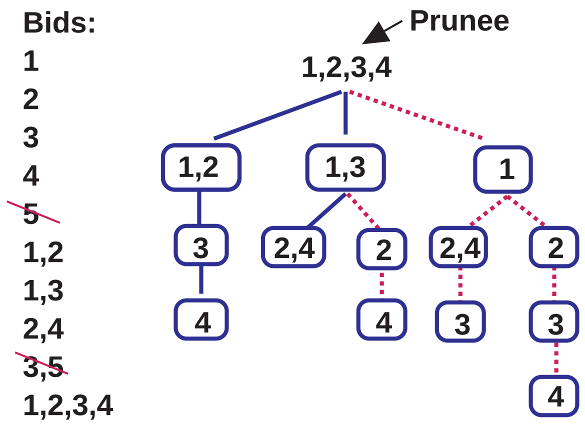

Figure 5: *A search tree generated for one prunee in PRE2. The dotted paths are not generated because pruning occurs before they are reached.*

If the prunee is determined to be noncompetitive, it is permanently discarded. It does not participate in later pruning of other bids.

Especially with bids that contain a large number of items, PRE2 can take more time than it saves in the main search. In the extreme, if some bid contains all items, the preprocessing search with that bid as the prunee is the same as the main search (except for one main search path that contains that bid only). To save preprocessing time, PRE2 is carried out partially. With such partial pruning, some of the noncompetitive bids are left unpruned. That will not compromise optimality of the main search although the main search might not be sped up as much compared to full pruning. We implemented two ways of restricting PRE2:

1. A cap Γ on the number of pruner bids that can be combined to try to prune a particular prunee bid. This limits the depth of the search in PRE2 to Γ.

2. A cap Φ on the number of items in a prunee bid. Longer bids would then not be targets of pruning. This entails a cap Φ on tree depth. It also tends to exclude wide trees because long prunees usually lead to trees with large branching factors.

With either method, PRE2 takes $O(n^{cap+2}(m - \log n))$ time, which is polynomial for a constant cap (there are n prunees, the tree for each is $O(n^{cap})$, and finding a child in the Bidtree is $O(n \cdot (m - \log n))$).[14] The latter method is usually preferable. It does not waste computation on long prunees which take a lot of preprocessing time and do not significantly increase the main search time. This is because the main search is shallow along the branches that include long bids due to the fact that each item can occur only once on a path and a long bid uses up many items. Second, if the bid prices are close to additive, the former method does not lead to pruning when a path is cut prematurely based on the cap.

3.4.3 PRE3: Decompose the set of bids into connected components

The bids are divided into sets such that no item is shared by bids from different sets. The sets are determined as follows. We define a graph where bids are vertices, and two vertices share an edge if the bids share items. We generate an adjacency list representation of the graph in $O(mn^2)$ time. Let e be the number of edges in the graph. Clearly, $e \leq \frac{n^2-n}{2}$.

We use depth-first search to generate a depth-first forest of the graph in $O(n + e)$ time [12]. Each tree in the forest is then a set with the desired

[14]This analysis makes the innocuous assumption that $m - \log n \geq c$ for some constant $c > 0$ (recall Proposition 3.5). Otherwise, the complexity of finding a child (actually, all children) is only $O(n)$.

property. PRE4 and the main search are then done in each set of bids independently, and using only items included in the bids of the set. In the presentation that follows we will denote by m' the number of items in the connected component in question, and by n' the number of bids in the connected component in question.

3.4.4 PRE4: Mark noncompetitive tuples of bids

Noncompetitive tuples of disjoint bids are marked so that they need not be considered on the same path in SEARCH1. For example, the pair of bids \$5 for items 1 and 3, and \$4 for items 2 and 5 is noncompetitive if there is a bid of \$3 for items 1 and 2, and a bid of \$7 for items 3 and 5. Formally:

Definition. 3.1 *A tuple* $\{B_i, B_j, ..., B_k\}$ *(where* $B, B' \in \{B_i, B_j, ..., B_k\} \Rightarrow B \cap B' = \emptyset$*) is* excludable based on noncompetitiveness *if there exists* $\{B_x, B_y, ..., B_z\}$ *such that*

$$\begin{aligned} \{B_i, B_j, ..., B_k\} \cap \{B_x, B_y, ..., B_z\} &= \emptyset, \text{ and} && (19) \\ B, B' \in \{B_x, B_y, ..., B_z\} &\Rightarrow B \cap B' = \emptyset, \text{ and} && (20) \\ \bigcup_{B \in \{B_x, B_y, ..., B_z\}} B.S &\subseteq \bigcup_{B \in \{B_i, B_j, ..., B_k\}} B.S, \text{ and} && (21) \\ \sum_{B \in \{B_x, B_y, ..., B_z\}} B.\bar{b} &> \sum_{B \in \{B_i, B_j, ..., B_k\}} B.\bar{b} && (22) \end{aligned}$$

Noncompetitive tuples are determined as in PRE2 except that now each prunee is a virtual bid that contains the items of the bids in the tuple ($\bigcup_{B \in \{B_i, B_j, ..., B_k\}} B.S$), and the prunee price is the sum of the prices of those bids ($\sum_{B \in \{B_i, B_j, ..., B_k\}} B.\bar{b}$). Unlike in PRE2, here the inequality for price comparison is strict so that tuples that are equally good do not all get excluded, which could compromise optimality.[15] In PRE2 this was not a concern because bids were discarded as they were determined noncompetitive.

For computational speed, we only determine the noncompetitiveness of 2-tuples, that is, pairs of bids. PRE4 is used as a partial preprocessor like PRE2, with caps Γ' or Φ' instead of Γ or Φ. PRE4 runs in $O(n'^{cap+3}(m' - \log n'))$ time. This is because considering all pairs of bids is $O(n'^2)$, the tree for each such pair is $O(n'^{cap})$, and finding a child in the Bidtree is $O(n' \cdot (m' - \log n'))$.[16] Handling 3-tuples would increase the complexity to $O(n'^{cap+4}(m' - \log n'))$, etc. Handling large tuples also slows the main search because it needs to ensure that noncompetitive tuples do not exist on the path.

[15]It would also be sound to exclude all but one of the tuples that are in a draw. In order to avoid the overhead of keeping track of such rare occurrences, we do not exclude any of the tuples that are in a draw.

[16]This analysis makes the innocuous assumption that $m' - \log n' \geq c$ for some constant $c > 0$ (recall Proposition 3.5). Otherwise, the complexity of finding a child (actually, all children) is only $O(n')$.

As a bid is appended to the path in SEARCH1, it excludes from the rest of the path those other bids that constitute a noncompetitive pair with it, and those bids that share items with it. Our algorithm determines this quickly as follows. For each bid, a list of bids to exclude is determined in PRE4. In SEARCH1, an exclusion count is kept for each bid, starting at 0. As a bid is appended to the path, the exclusion counts of those bids that it excludes are incremented. As a bid is backtracked from the path, those exclusion counts are decremented. Then, when searching for bids to append to the SEARCH1 path from the Bidtree, only bids with exclusion count 0 are accepted.[17]

3.5 Iterative deepening A* and heuristics

We sped up the main search by using an iterative deepening A* (IDA*) search strategy [28] instead of depth-first search. The search tree, use of SEARCH2 in the Bidtree to generate children of a SEARCH1 node, and the preprocessors stay the same. In practice, IDA* finds the provably optimal solution while searching a very small fraction of the entire search tree of SEARCH1 (Figure 3). The following pseudocode shows how we applied the IDA* search strategy to the winner determination problem. The function $h(F)$, discussed in detail later, gives an upper bound on how much revenue the items F that are not yet allocated on the current search path can contribute. As defined earlier in this paper, g is the sum of the prices of the bids that are on the current search path. At any search node, an upper bound on the total revenue that can be obtained by including that search node in the solution is given by $f = g + h(F)$.

Instead of using basic IDA* throughout the search, on the last IDA* iteration we keep incrementing the f-limit to equal the revenue of the best solution found so far in order to avoid futile search. In other words, once the first solution is found, our algorithm converts to branch-and-bound with the same heuristic function, h. This modification is included in the pseudocode below.

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

Global variable: f-limit

Algorithm 3.1 (IDA* for winner determination)
// Returns a set of winning bids that maximizes the sum of the bid prices

```
1. f-limit := $\infty$
2. Loop
    (a) winners, new-f := DFS-CONTOUR($M'$, $\emptyset$, 0)
    (b) if winners $\neq$ null then return winners
    (c) f-limit := min(new-f, 0.95 $\cdot$ f-limit) [18]
```

Algorithm 3.2 DFS-CONTOUR(F, *winners*, g)
// Returns a set of winning bids and a new f-cost

```
1. Compute $h(F)$ // Methods for doing this are presented later
2. If $g + h(F) <$ f-limit then return null, $g + h(F)$ // Pruning
3. IF SEARCH2 returns no children, then // End of a path reached
    (a) f-limit := g // Revert to Branch&Bound once first leaf is reached
    (b) return winners, g
4. maxRevenue := 0, bestWinners := null, next-f := 0
5. For each bid $\in$ children // Children are found using SEARCH2
    (a) solution, new-f :=
            DFS-CONTOUR(F$-$ bid.S, winners $\cup$ {bid}, g + bid.$\bar{b}$)
    (b) If solution $\neq$ null and new-f > maxRevenue, then
        i. maxRevenue := new-f
        ii. bestWinners := solution
    (c) next-f := max(next-f, new-f)
6. If bestWinners $\neq$ null then return bestWinners, maxRevenue
   else return null, next-f
```

[18]The constant 0.95 was determined experimentally from the range [0,1], as will be discussed later.

At each iteration of IDA* (except the last), f-limit gives an upper bound on solution quality (revenue). It can be used, for example, to communicate search progress to the auctioneer.

Since winner determination is a maximization problem, the heuristic function $h(F)$ should never underestimate the revenue from the items F that are not yet allocated in bids on the path because that could compromise optimality. We designed two such admissible heuristics.

Heuristic 3.1 *Take the sum over unallocated items of the item's maximal contribution. An item's maximal contribution is the price of the bid divided by the number of items in that bid, maximized over the bids to which the item belongs. Formally:*

$$h(F) = \sum_{i \in F} c(i) \quad \textit{where} \;\; c(i) = \max_{S|i \in S} \frac{\bar{b}(S)}{|S|} \tag{23}$$

Proposition 3.6 *$h(F)$ from Heuristic 3.1 gives an upper bound on how much revenue the unallocated items F can contribute.*

Proof. For any set $S \in F$, if a bid for S is determined to be winning, then each of the items in S contributes $\frac{\bar{b}(S)}{|S|}$ toward the revenue. Every item can be in only one winning bid. Therefore, the revenue contribution of any one item i can be at most $\max_{S|i \in S} \frac{\bar{b}(S)}{|S|}$. To get an upper bound on how much all the unallocated items F together can contribute, simply sum $\max_{S|i \in S} \frac{\bar{b}(S)}{|S|}$ over all $i \in F$. $\Box$

Heuristic 3.2 *Identical to Heuristic 3.1 except that accuracy is increased by recomputing $c(i)$ every time a bid is appended to the path since some other bids may now be excluded. A bid is excluded if some of its items are already used on the current path in SEARCH1, or if it constitutes a noncompetitive pair (as determined in PRE4) with some bid on the current path in SEARCH1.*

Proposition 3.7 *$h(F)$ from Heuristic 3.2 gives an upper bound on how much revenue the unallocated items F can contribute.*

Proof. Excluded bids cannot affect the revenue because they cannot be designated as winning. Therefore, $c(i)$ can be recomputed without the excluded bids, and the proof of Proposition 3.6 holds. $\Box$

We use Heuristic 3.2 with several methods for speeding it up. A tally of h is kept, and only some of the $c(i)$ values in h need to be updated when a bid is appended to the path. In PRE4 we precompute for each bid the list of items that must be updated: items included in the bid and in bids that are on the bid's exclude list. To make the update even faster, we keep for each item a list of the bids in which it belongs. The $c(i)$ value is computed by traversing that list and choosing the highest $\frac{\bar{b}(S)}{|S|}$ among the bids that have exclusion count 0. So, recomputing h takes $O(\underline{\text{m}}\text{'}\underline{\text{n}}\text{'})$ time, where $\underline{\text{m}}$' is the number of items that need to be updated, and $\underline{\text{n}}$' is the (average or greatest) number of bids in which those items belong.[19]

3.6 Time complexity of the main search (SEARCH1)

The worst case time complexity of the main search is polynomial in bids and exponential in items. This is desirable since the auctioneer can control the number of items for sale, but usually cannot control (and does not want to restrict) the number of bids received.

Specifically, by Proposition 3.3 the number of SEARCH1 nodes is $O(m' \cdot (\frac{n'}{m'})^{m'})$ even without pruning. The time per each SEARCH1 node is $O(n'm')$ because each of the per node activities (child generation, h-function updating, etc.) is $O(n'm')$. IDA* generates some nodes multiple times because all of the nodes generated in one iteration are regenerated in the next iteration. However, given that the new f-limit is lowered to the maximum f-value that was lower than the f-limit in the previous iteration (or to an even lower value), each iteration generates at least one more search node than the previous iteration. Therefore, if the number of nodes in a search tree is x, then IDA* will generate at most $x\frac{1+x}{2}$ search nodes. In practice it will generate significantly fewer because the f-limit is decreased more across iterations, as was presented in the pseudocode and as will be discussed further in the section on experimental results. Nevertheless, even in the hypothetical worst case, the overall worst case complexity remains polynomial in bids because the regeneration of nodes in IDA* only increases the complexity polynomially.

The complexity in the number of bids is complexity in the number of bids actually received. This is in contrast to approaches such as the dynamic program which executes the same steps independent of the bids that have been received.

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

4 Experimental setup

To determine the efficiency of the algorithm in practice, we ran experiments on a general-purpose uniprocessor workstation (450MHz Sun Ultra 80 with 1 Gigabyte of RAM) in C++ with four different bid distributions:

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

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

We let all the bids have the same bidder. This conservative method causes PRE1 to prune no bids. This is a good model of reality if the number of items is large and bidders place their bids on randomly chosen combinations, because then the chance that two agents bid on the same combination is small due to the number of combinations being large ($2^m - 1$). However, in some cases PRE1 is very effective. In the extreme, it prunes all of the bids except one if all bids are placed on the same combination by different bidders. When PRE1 is effective, our algorithm will run faster than suggested by the conservative experiments of this paper.

5 Experimental results

We focus on IDA* (with branch-and-bound on the last iteration after the first leaf has been reached) because it was two orders of magnitude faster than depth-first search. We lower the f-limit between iterations to 95% of the previous f-limit or to the highest value of f that was lower than the f-limit in the previous IDA* iteration, whichever is smaller (as shown in the pseudocode of the previous section). Clearly, there is never any reason to use an f-limit that is higher than the highest value of f that was lower than the f-limit in the previous IDA* iteration. The 95% criterion is used to decrease the f-limit even faster. Experimentally, this tended to be a good rate of decreasing the f-limit. If it is decreased too fast, the overall number of search nodes increases because the last iteration will lead to a large number of search nodes in DFS-CONTOUR. If it is decreased too slowly, the overall number of search nodes increases because new iterations repeat a large portion of the search from previous iterations.

For PRE2, the cap $\Phi = 8$ gave a good compromise between preprocessing time and main search time. For PRE4, $\Phi' = 5$ led to a good compromise. These values are used in the rest of the experiments. With these caps, the hard problem instances with short bids get preprocessed completely, and PRE2 and PRE4 tend to take negligible time compared to the main search because the trees under such short prunees are small. The caps only take effect in the easy cases with long bids. In the uniform distribution all bids are the same length, so PRE2 does not prune any bids because no bid is a subset of another.

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

The main search generated 35,000-40,000 nodes per second when the number of items was small (25) and the bids were short (3 items per bid). This rate decreased slightly with the number of bids, but significantly with the number of items and bid size. With the weighted random distribution with 400 items and 2,000 bids, the search generated only 12 nodes per second. However, the algorithm solved these cases easily because the search paths were short and the heuristic focused the search well. Long bids make the heuristic and exclusion checking slower but the search tree shallower which makes them easier for our algorithm than short bids overall. This observation is further supported by the results below. Each point in each graph represents an average over 15 problem instances. The execution times presented include all preprocessing times. For the problem sizes that we consider, previous optimal winner determination approaches (exhaustive enumeration and dynamic programming) cannot be used at all in practice because they would take prohibitively long, as was discussed earlier in this paper.

5.1 Basic experiments on the different bid distributions

Problem instances drawn from the random distribution tended to be easy (Figure 6). The reason is that the search was shallow because the bids were long. Each bid uses up a large number of items, so there can only be a small number of bids on a search path.

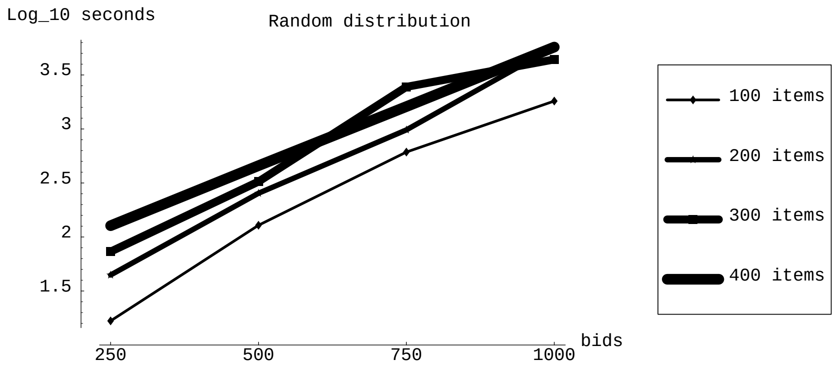

Figure 6: *Execution time on the random distribution. The run times for any given point (number of items, number of bids) varied by more than an order of magnitude. Despite the averaging over 15 instances, the points (300 items, 750 bids) and (200 items, 1000 bids) are unusually high because those points happened to encounter unusually hard instances.*

The weighted random distribution was even easier (Figure 7). In both the weighted and the unweighted case, the curves are sublinear meaning that execution time is polynomial in bids. This is less clear in the unweighted case. Later in this paper we extend the experiments to a higher ratio of bids to items. Then the sublinearity becomes apparent even in the unweighted case.

Comparing Figures 7 and 8, one can see that on the weighted random distribution, the bulk of the execution time was spent preprocessing rather than searching. This could be avoided by turning off the preprocessors for that distribution. However, to keep the experiments honest and the algorithm (and its parameterization) distribution independent, we used the same preprocessor caps, $\Phi = 8$ and $\Phi' = 5$, for all of the distributions. Future research could focus on automatically identifying how the caps should be set based on the problem instance. On all of the other distributions than weighted random, the preprocessing time was a small fraction of the search time. The plots for execution time and search time were almost identical, so we only present the plots for execution time.

The uniform distribution was harder (Figure 9). The bids were shorter so the search was deeper. Figure 10 shows the complexity decrease as the

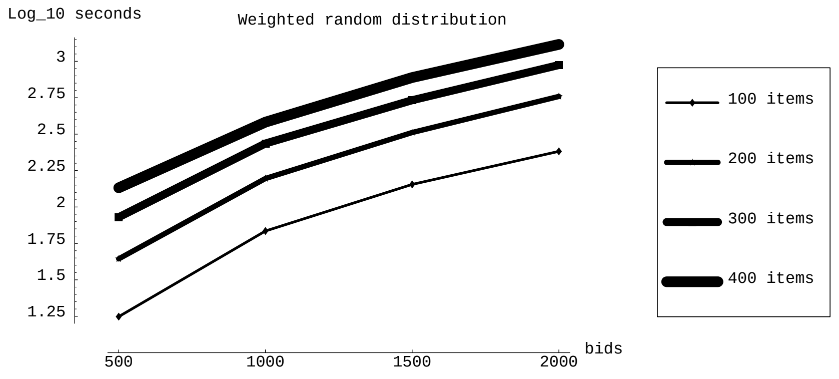

Figure 7: *Execution time (preprocessing time plus search time) on the weighted random distribution.*

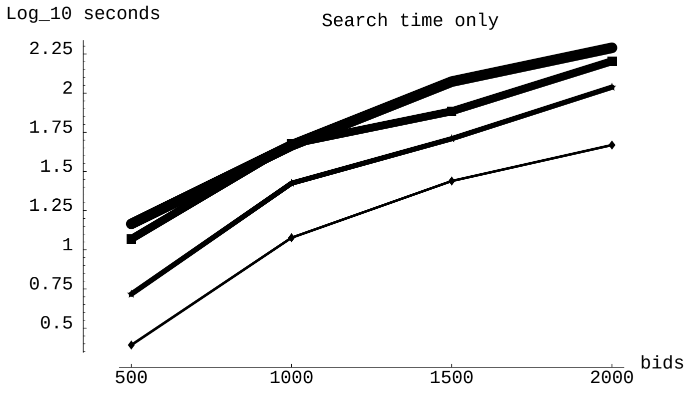

Figure 8: *Search time on the weighted random distribution.*

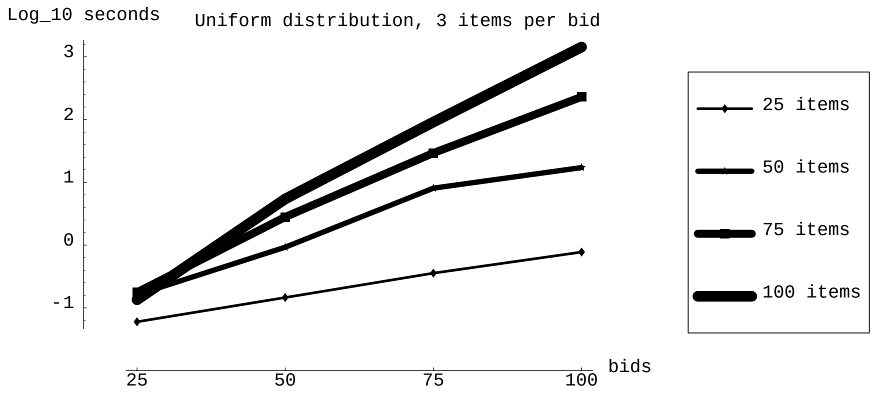

Figure 9: *Execution time on the uniform distribution.*

bids get longer, that is, the search gets shallower.

The decay distribution was also hard (Figure 11). Complexity first in-

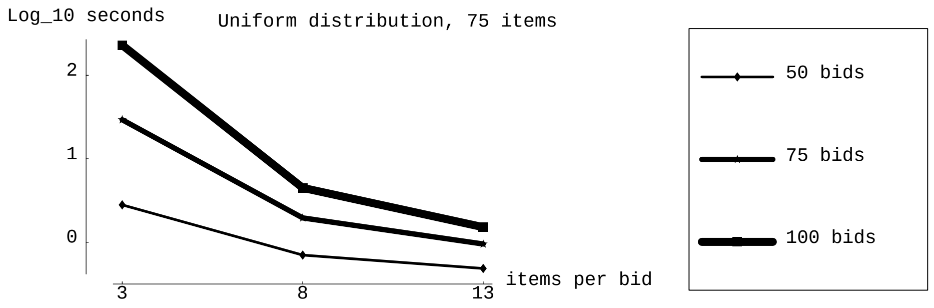

Figure 10: *Execution time on the uniform distribution when the number of items per bid varies.*

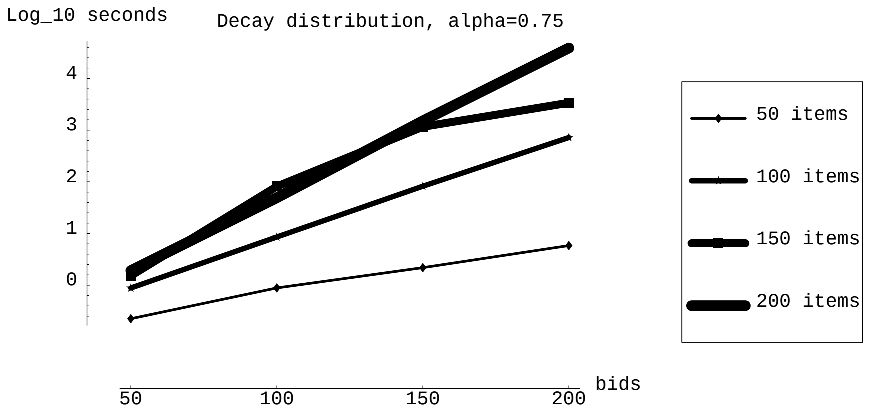

Figure 11: *Execution time on the decay distribution. Despite the averaging over 15 instances, the points (150 items, 100 bids) and (150 items, 150 bids) are unusually high because those points happened to encounter unusually hard instances.*

creases in α, and then decreases (Figure 12). Left of the maximum, PRE3 decomposes the problem leading to small, fast searches. The hardness peak moves slightly left as the number of bids grows because the decomposition becomes less successful. Right of the maximum, all bids are in the same set. The complexity then decreases with α because longer bids lead to shallower search. In all the other graphs for the decay distribution, we fix α at the hardest value, 0.75.

5.2 Scaling up to larger numbers of bids

In practice, the auctioneer usually can control the number of items that are being sold in a combinatorial auction, but cannot control the number of bids

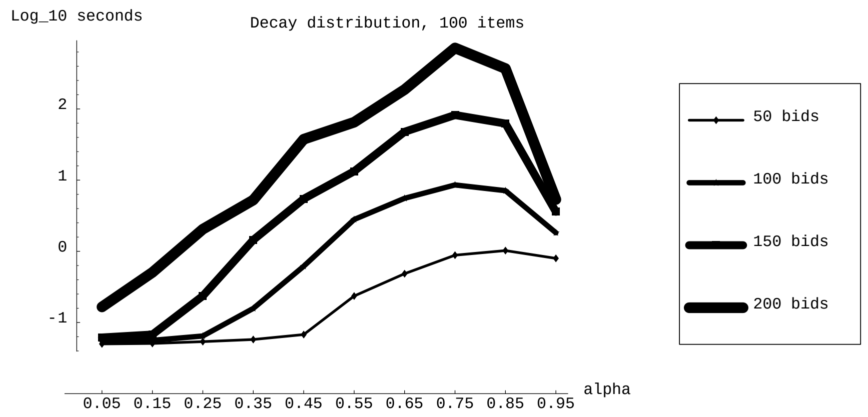

Figure 12: *Execution time on the decay distribution when α varies.*

that end up being submitted. Therefore, it is particularly important that the winner determination algorithm scales up well to large numbers of bids. In this section we test how our algorithm scales to large numbers of bids when we fix the number of items ($m = 50$).

Figure 13 shows that on the random and weighted random distributions, the algorithm scales very well to large numbers of bids. The complexity is clearly polynomial in bids, and once the number of bids reaches a certain level, additional bids increase the complexity only slightly.

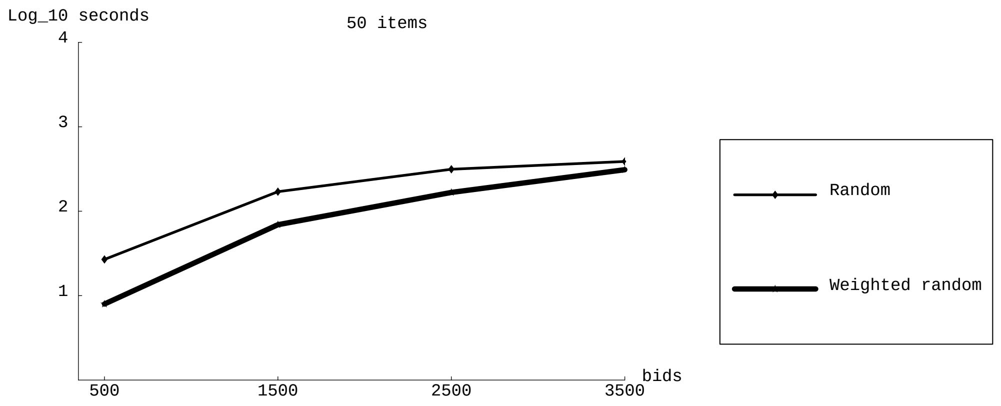

Figure 13: *Execution time on the random and weighted random distributions.*

The uniform and decay distributions were again harder (Figure 14). However, even on these distribution, the curves are sublinear, meaning that execution time is polynomial in bids.

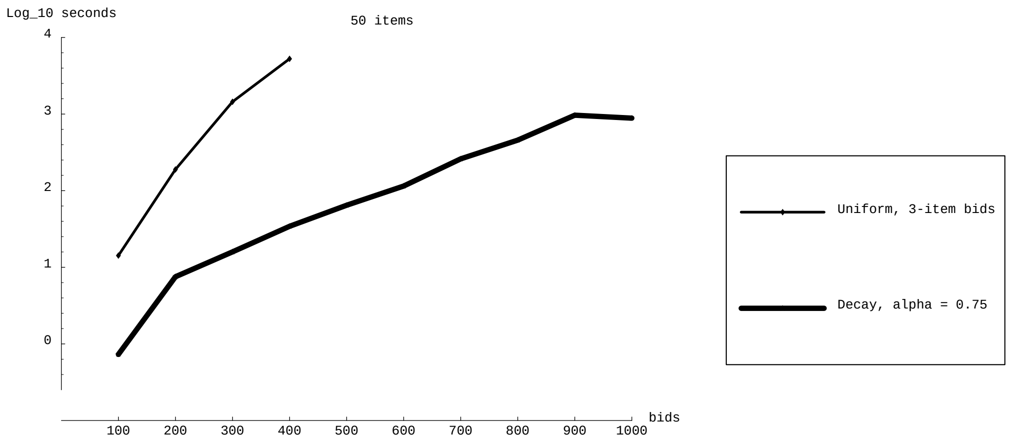

Figure 14: *Execution time on the uniform and decay distributions.*

5.3 Varying the number of items while fixing the number of bids

We also studied the complexity in the number of items by fixing the number of bids. Figure 15 shows that on the random and weighted random distributions, the algorithm scales very well to large numbers of items. The curves are sublinear, meaning that execution time is polynomial in items.

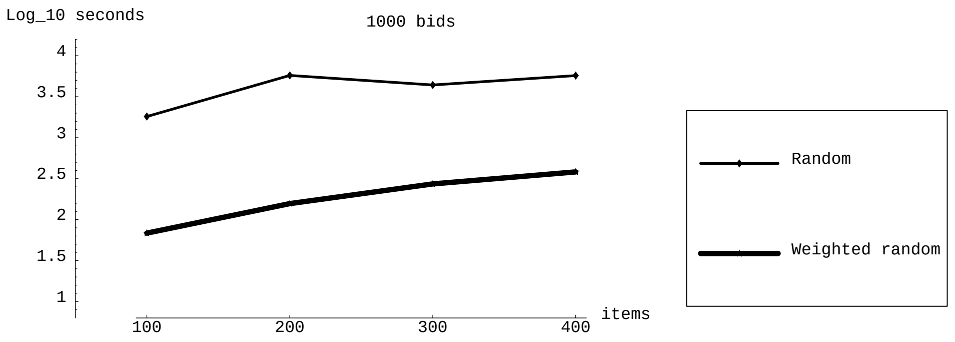

Figure 15: *Execution time on the random and weighted random distributions for a varying number of items.*

Figures 16 and 17 show that the uniform and decay distributions were again harder. Execution time increases rapidly as the number of items increases. However, even on these distributions, the curves are sublinear, meaning that execution time is polynomial in items.

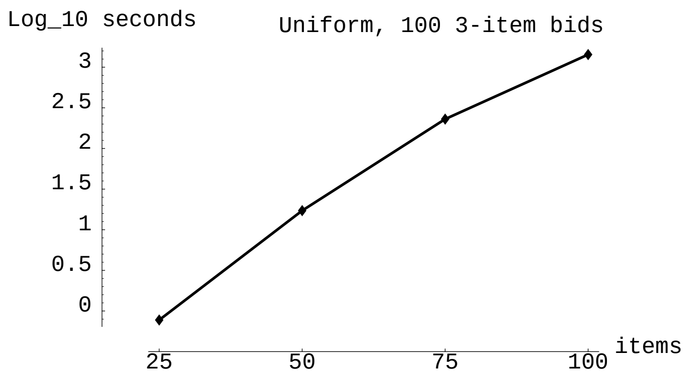

Figure 16: *Execution time on the uniform distribution for a varying number of items.*

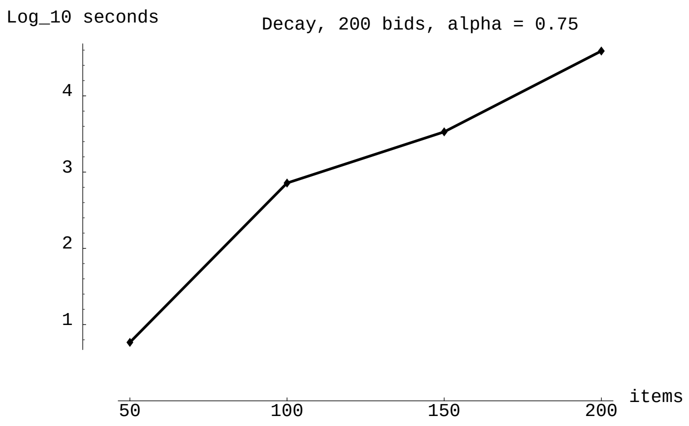

Figure 17: *Execution time on the decay distribution for a varying number of items.*

5.4 Fixing the ratio of bids to items

From a practical perspective it seems reasonable that the number of bids grows as the number of items grows. Therefore, an additional experiment was called for where neither the number of items nor the number of bids was fixed. In this experiment we fixed the ratio of bids to items ($\frac{n}{m} = 10$).

Figure 18 presents the execution times for the random and weighted random distributions. The curves are sublinear, which means that the execution time is polynomial in bids and in items.

Figure 19 presents the execution times for the uniform and decay distributions. These were again significantly harder than the random and weighted random distributions.

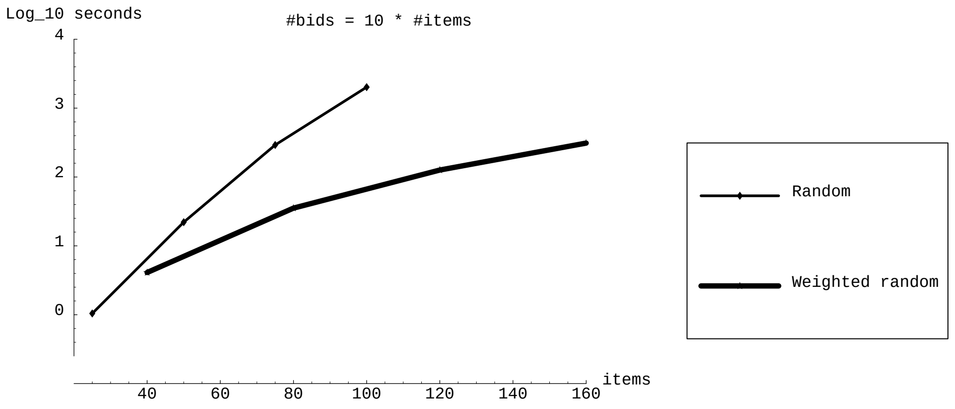

Figure 18: *Execution time on the random and weighted random distributions.*

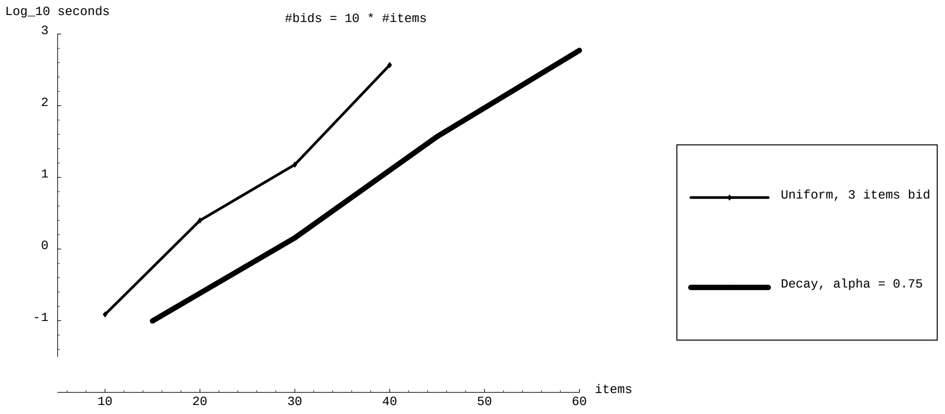

Figure 19: *Execution time on the uniform and decay distributions.*

6 Incremental computations

In many auction settings it is important to provide results while the auction is still open. Motivated by this need, we designed incremental methods for computing results. The following subsections discuss incremental winner determination and incremental quote computation.

6.1 Incremental winner determination

In some auction settings it is important to be able to compute optimal solutions in the interim after some bids have been received while others are still to be received. This allows the bidders to be told who the current winners would be if no further bids were to be received. This information can be naively produced by running the preprocessors and the main search from scratch every time a new bid arrives. However, the amount of redundant computation can be reduced by using the following observations.

1. Say that a new bid $B = (B.S, B.\bar{b})$ has arrived since the winner determination was last carried out. If PRE1 or PRE2 prune the new bid, that bid will not affect the solution. It follows that the solution that was optimal before the new bid arrived remains optimal, so PRE3, PRE4, and the main search need not be executed again.

   If the new bid is noncompetitive, it can be permanently discarded. Other bids that arrive later on cannot make it competitive. Neither can this bid prune later bids that would not be pruned anyway.

2. If PRE1 and PRE2 do not prune the new bid B, then the winners can be determined as follows. Let r^*_{old} be the old optimal revenue without the new bid B. Now, conduct the winner determination with the items $M - B.S$, and with bids that do not include any items in $B.S$. Call the new optimal revenue $r^*_{M-B.S}$. If $r^*_{M-B.S} + B.S > r^*_{old}$, then the optimal revenue is $r^*_{M-B.S} + B.S$ and the optimal solution includes the new bid B and the winning bids from this new winner determination. Otherwise, the old revenue r^*_{old} and the corresponding bids are optimal.

   The winner determination with items $M - B.S$ can be done in one IDA* iteration by setting the f-limit to $r^*_{old} - B.S$. If IDA* returns no solution, then the old optimal solution is best. On the other hand, if IDA* returns a solution, then that solution—together with bid B—is best.

   That single IDA* iteration can involve unnecessary search if the price of the new bid, $B.\bar{b}$, is high relative to the prices of the old bids, because in such scenarios new solutions that are better than the old solution can be found even while using a higher f-limit, leading to more pruning in the search. One can reduce the number of nodes generated in the last iteration by starting IDA* with a higher f-limit, and then lowering it between iterations if a solution is not found. There is never a need to lower it below $r^*_{old} - B.S$. One can start lowering the f-limit from some upper bound on $r^*_{M-B.S}$. An upper bound, UB, $UB \geq r^*_{M-B.S}$, is given by the revenue of the old optimum minus the highest revenue that could have been obtained from non-overlapping old bids whose items are contained in the new bid. Formally,

   $$UB = r^*_{old} - r^*_{old}(B.S) \tag{24}$$

   where

   $$r^*_{old}(B.S) = \max_{\{\mathcal{T} \mid \forall T_i, T_j \in \mathcal{T},\ T_i, T_j \subseteq B.S \text{ and } T_i \cap T_j = \emptyset\}} \sum_{T \in \mathcal{T}} \bar{b}(T) \tag{25}$$

   where $\bar{b}(T)$ is computed using the old bids only.

3. Incremental winner determination can be used even if several bids have arrived since the most recent winner determination.

   Observation 1. from above can be extended to this setting. Both the old bids and new bids are used to prune new bids in PRE1 and PRE2. To determine whether the old solution is optimal, only new bids are pruned. If some of them do not get pruned, so the solution needs to be recomputed, then old bids are also targets of pruning. An old bid that was not pruned before might now be prunable in light of the new bids.

   Observation 2. from above can also be extended to this setting. If observation 1. fails, so PRE3, PRE4, and the main search do need to be executed, the old optimal revenue, r^*_{old}, provides a lower bound on the new optimal revenue, r^*_{new}. This lower bound can be used as the initial f-limit in the main search so that IDA* completes in one iteration. If IDA* returns a solution with that f-limit, then a new optimal solution has been found that includes some of the new bids. Otherwise, the old optimal solution remains best.

   That single IDA* iteration can involve unnecessary search if the prices of the new bids are high relative to the prices of the old bids, because in such a scenario a higher f-limit (leading to more pruning) can suffice for finding a new optimal solution that is better than the old optimal solution. One can reduce the number of search nodes in the last iteration by starting IDA* with a higher f-limit, and then lowering it between iterations if a solution is not found. There is never need to lower it below r^*_{old} because $r^*_{new} \geq r^*_{old}$. One can start lowering the f-limit from some upper bound on r^*_{new}. An upper bound, UB_S, $UB_S \geq r^*_{new}$, is given by

$$UB_S = \max[r^*_{old}, r^*_{old} + r^*(S) - r^*_{old}(S)] \tag{26}$$

   where

$$S = \bigcup_{i \,\in\, \text{new bids}} B_i.S, \tag{27}$$

   $r^*(S)$ is the maximal revenue that can be obtained from the items S using the old and the new bids, and $r^*_{old}(S)$ is the maximal revenue that can be obtained from the items S using the old bids only. These revenues can be determined using the preprocessing and search techniques of this paper, and if $|S| < |M|$, computing them is faster than carrying out the overall winner determination.

6.2 Incremental quote computation

In some settings it is also important to be able to tell a prospective bidder how much she would have to bid on any given combination, S, so that she would become the winner of S given the other bids so far. For example, a bidder may post the following query: "How much would I have to bid to obtain the set of items $S = \{2, 3, 7\}$ (assuming that no further bids will be submitted) ?". We developed several methods for computing such quotes:

1. **Exact quote.** Let r^* be the revenue of the current optimal solution. Now, remove the items S and all bids that include items in S, and determine the winners again optimally. Call the new revenue $r^*_{reduced}$. Now, an exact quote on S is given by $r^* - r^*_{reduced}$.

2. **Quote bounds.** An upper bound on a quote ("If you bid \$x, you will certainly get S (assuming that no further bids will be submitted)") can be computed as an upper bound on (or exact value) r^* minus a lower bound on (or exact value) $r^*_{reduced}$. Similarly, a lower bound on a quote ("If you bid only \$y, you will certainly not get S (assuming that no further bids will be submitted)") can be computed as a lower bound on (or exact value) r^* minus an upper bound on (or exact value) $r^*_{reduced}$.

   A lower bound can be computed using an approximation algorithm (such as those discussed earlier in this paper), or by running the anytime search algorithm of this paper for some amount of time. An earlier solution that was obtained without some of the newest bids can also be used as a lower bound. This allows winner determination to run only occasionally while still supporting approximate quotes in the meantime. One can also compute several of these lower bounds and provide the highest one.

   An upper bound can be computed using a linear programming relaxation of the problem (discussed further in Section 8), or by using the h-function value directly as the upper bound. Alternatively, the f-limit in IDA* can be used as an upper bound. Since the f-limit does not provide an upper bound on the last IDA* iteration, and one does not know in advance whether the current iteration will be the last, one can use the highest value of f that was lower than the f-limit in the previous iteration. As yet another alternative, one can use the incrementally computed upper bound, UB_S, from the previous subsection. This allows winner determination to run only occasionally while still supporting approximate quotes in the meantime. One can also compute several of these upper bounds and provide the lowest one.

3. **Binary search for a quote and quote bounds.** A hypothetical bid is placed on combination S. Binary search is carried out on the price of that new bid, running the preprocessors and the main search algorithm for each price that the binary search generates for the new bid. Observations 1. and 2. from the previous subsection can be directly used to speed up the execution within each such price point. If the price is very low, the preprocessors can quickly determine that

the hypothetical bid is not competitive. This provides a lower bound on the quote and let's the binary search proceed further. If the price is very high, the search will quickly find solutions that include the new bid and give revenue that exceeds the revenue that was obtainable without that bid. When this occurs, the search can be interrupted because an upper bound on the quote has been established, and the binary search can proceed further.

6.2.1 Properties of quotes

Quotes (even exact quotes) in combinatorial auctions have some interesting properties. First, the quote on a combination is not necessarily the sum of the quotes on individual items:

Proposition 6.1 *Quotes in a combinatorial auction are not always additive.*

Proof. Consider an auction of two items: 1 and 2. Say that three bids have been submitted: \$4 for item 1, \$1 for item 2, and \$6 for the combination $\{1, 2\}$. Now, the quote for item 1 is \$5, the quote for item 2 is \$2, and the quote for the combination $\{1, 2\}$ is \$6. But $\$5 + \$2 \neq \$6$. □

This means that providing quotes on individual items does not provide enough information for a bidder who prefers combinations.

In usual single-item auctions, the quote can increase (or stay the same) as a new bid arrives. New bids cannot reduce the quote. However, this monotonicity ceases to hold in combinatorial auctions:

Proposition 6.2 *As a bid is submitted on some combination (or individual item) in a combinatorial auction, the quotes on other combinations (and individual items) might increase or decrease.*

Proof. Consider the example from the proof of Proposition 6.1. Say that a new bid of \$7 is submitted on combination $\{1, 2\}$. Now, the quote on item 1 increases to \$6, and the quote on item 2 increases to \$3.

If, instead, a bid of \$4.5 is submitted for item 1, then the quote on item 2 decreases to \$1.5. □

Finally, if a bid is currently a winner, then the quote on the set of items of that bid is simply the price of that bid.

7 Other applications for the algorithms

The algorithms for winner determination can be directly used to solve weighted set packing, weighted independent set, and weighted maximum clique because they are in fact analogous problems. In addition they can be used for coalition structure generation in characteristic function games. The characteristic function assigns a value to each potential coalition of agents. The objective is to partition the agents exhaustively into disjoint coalitions so as to maximize the sum of the values of those coalitions. The agents correspond to items, coalitions to combinations, and coalition values to bid prices. This problem is analogous to the winner determination problem if the coalitions have nonnegative values. This can usually be achieved by normalizing if it does not hold up front. However, coalition structure generation differs from winner determination in two major ways.

First, the values of most coalitions are usually nonzero so the space of "bids" is densely populated. That reduces the usefulness of algorithms that capitalize on the sparseness of bids. This speaks in favor of using the dynamic program. However, that will not scale up past small numbers of agents either.

Second, in coalition structure generation the values of coalitions might not be known to any party, and only values of complete coalition structures might be known, or they become known during a search process. Sandholm et al. [49] recently proved inapproximability results and devised an anytime approximation algorithm with worst case guarantees for this case which is harder than the case where the values of coalitions are observed.

8 Related tree search algorithms

A significant amount of work in the field of operations research has focused on problems that are analogous to the combinatorial auction winner determination problem, particularly weighted set packing and weighted independent set. The approximation approaches were discussed in detail earlier in this paper. In addition, considerable research has been done on exact algorithms for these problems. These algorithms typically use a branch-and-bound search strategy where an h-function value (that is, an upper bound on how much revenue the yet unallocated items can contribute) is computed from the linear programming problem that is obtained by taking the winner determination problem (with the items and bids that are still usable given the bids that are already on the search path) and changing the constraints $x_S \in \{0, 1\}$ to $x_S \leq 1$. This linear program at each search node can be solved in polynomial time in the size of the input (which itself is $\Theta(nm)$) using interior point methods, or fast on average using the simplex method [11, 60]. The upper bound that the linear programming relaxation provides can be tightened (lowered) by introducing *cuts*. These are additional constraints that do not affect the solution of the integer program, but which constrain the linear programming polytope. Major research questions in these algorithms include inventing new sound cuts, constructing fast algorithms for constructing cuts, as well as experimenting on the tradeoff between the time spent on generating cuts and the time saved due to the reduction in search tree size that stems from the tighter upper bounds. This is a vast literature (see, for example, [16, 33, 39]). There are several modern books on this approach [11, 60], and good overviews of applying this approach to combinatorial auctions have been written very recently [13, 37]. Generally this approach leads to a smaller number of search nodes than our approach (due to tighter upper bounds), but spends more time per search node. Which is faster overall depends on the problem instance distribution. Instead of viewing these two approaches as mutually exclusive, they can be viewed as complementary. A hybrid algorithm could use techniques both from our approach (the preprocessors, fast child generation methods, the combination of IDA* and branch-and-bound that we use, the h-function that is fast to compute, etc.) and the traditional integer programming approach (using a linear programming relaxation to provide upper bounds, introducing cuts, etc.). For example, a hybrid algorithm could use the slow but relatively accurate linear programming relaxation only occasionally, while using our fast h-function to provide upper bounds for most search nodes. Also, it is always sound to compute both types of upper bounds, and to use the lowest of them.

After this paper was circulated as a technical report, and while it was under review for Artificial Intelligence, a very similar algorithm, called CASS, was published by Fujishima et al. [15]. In fact, they present two algorithms, VSA and CASS. Here we focus on CASS because VSA does not generally find the optimal solution. Despite significant similarities, CASS differs from our algorithm in certain ways. CASS uses a branch-and-bound search strategy while we use IDA* combined with branch-and-bound. Neither search strategy always dominates the other. IDA* can lead to fewer search nodes because the f-limit allows pruning of parts of the search space that branch-and-bound would search. On the other hand, branch-and-bound can lead to fewer search nodes because IDA* searches the nodes that are close to the root multiple times.

CASS uses a basic $O(mn)$ child generation technique while we developed a more elaborate method for $O(n \cdot (m - \log n))$ child generation. CASS uses *binning* to improve the average case speed of child generation. Specifically, CASS keeps m lists of bids. In each list are all those bids that include item i, $i \in \{1, \ldots, m\}$. Then, to find the children of a node, the algorithm only needs to loop through one of these list: the one where $i = i^*$ (recall that i^* is the smallest index among the items that are not on the path yet). This binning technique has also been used by Garfinkel and Nemhauser [16].

CASS uses a rough approximation of our h-function, but one which supports computing the h-function values in advance rather than during search. CASS also uses caching while we do not. Caching could be directly incorporated into our algorithm as well.

CASS relabels the items before the search so that promising bids are tried early in the search. This relabeling could be directly incorporated into our algorithm as well. Experimentally it proved to be very effective in CASS. CASS also uses child ordering heuristics (which determine the order in which children of a node are considered). Incorporating child ordering heuristics into our algorithm is not as straightforward because our incremental child generation using the Bidtree imposes a particular order of the children as was discussed in Section 3.2.1. However, child ordering heuristics could be incorporated into our algorithm by generating all of the children of a node in a batch using the Bidtree, and then ordering the children before the search proceeds to any of them. Finally, CASS does not use as sophisticated preprocessing as we do. Our preprocessors could be used in conjunction with CASS as well.

9 Substitutability, XOR-bids, and OR-of-XORs bidding

The methods for winner determination that were reviewed early in this paper, and most other work on combinatorial auctions (see for example [42, 14]), are based on a setting where each bidder can bid on combinations of items, any number of a bidder's bids can be accepted. This works well when bids are superadditive: $\bar{b}(S \cup S') \geq \bar{b}(S) + \bar{b}(S')$. However, when some of the bids are not superadditive, this can lead to problems. For example, what happens if agent 1 bids $b_1(\{1\}) = \$5$, $b_1(\{2\}) = \$4$, and $b_1(\{1,2\}) = \$7$, and there are no other bidders? The auctioneer could allocate items 1 and 2 to agent 1 separately, and that agent's bid for the combination would value at $\$5 + \$4 = \$9$ instead of $7. So, the current techniques focus on capturing synergies (complementarities) among items. In practice, local substitutability (subadditivity of the bid prices) can occur as well. As a simple example, when bidding for a landing slot for an airplane, the bidder is willing to take any one of a host of slots, but does not want more than one.

We address this issue in our Internet auction house prototype which is part of our electronic commerce server called *eMediator* (see `http://www.cs.cmu.edu/~amem/eMediator`). We developed a bidding language where the bidders can submit *XOR-bids*, that is, bids on combinations such that only one of the bids can get accepted. This allows the bidders to express general preferences with both complementarity and substitutability (see also [47] and [40]). In other words, the bidder can express any value function $v : 2^m \to \Re$.[20] For example, a bidder in a 4-item auction may submit the following input to the auctioneer:

$$(\{1\}, \$4) \text{ XOR } (\{2\}, \$4) \text{ XOR } (\{3\}, \$2) \text{ XOR } (\{4\}, \$2) \text{ XOR}$$

[20]By default, the bid for the empty set of items is 0.

$(\{1,2\},\$8)$ XOR $(\{1,3\},\$6)$ XOR $(\{1,4\},\$6)$ XOR
$(\{2,3\},\$6)$ XOR $(\{2,4\},\$6)$ XOR $(\{3,4\},\$3)$ XOR
$(\{1,2,3\},\$10)$ XOR $(\{1,2,4\},\$10)$ XOR $(\{1,3,4\},\$7)$ XOR
$(\{2,3,4\},\$7)$ XOR $(\{1,2,3,4\},\$11)$

Such fully expressive bidding languages have several advantages. First, each bidder can express her preferences rather than having to approximate them with an utterance in some less expressive bidding language. This extra expressiveness generally also increases the economic efficiency of the allocation because the winner determination algorithm takes as input the fully expressive bids rather than approximations. Second, a fully expressive bidding languages enables the auctioneer to extract truthful valuation revelations as the bids. With a fully expressive bidding language, bidding truthfully can be made incentive compatible (a dominant strategy) by using the *Vickrey-Clarke-Groves mechanism* [58, 7, 17]. This means that each bidder is motivated to bid truthfully regardless of how others bid. This renders counterspeculation unnecessary. The Vickrey-Clarke-Groves mechanism can be applied to the combinatorial auction setting as follows, assuming that a fully expressive bidding language is being used. Winning bids are determined so as to maximize the auctioneer's revenue under the constraint that each item can be allocated to at most one bid. The amount that an agent needs to pay is the sum of the others' winning bids had the agent not submitted any bids, minus the sum of the others' winning bids in the actual optimal allocation. Therefore, the winner determination problem has to be solved once overall, and once per winning agent without any of that agent's bids. This makes fast winner determination even more crucial. Note that, for example, just removing one winning bid at a time would not constitute an incentive compatible mechanism. Incentive compatibility can also be lost if either winner determination or price determination is done approximately.

While XOR-bids are fully expressive, representing one's preferences in that language often leads to large numbers of bids that are all combined with XOR. To maintain full expressiveness, but at the same time to make the representation more concise, we propose another bidding language which we call *OR-of-XORs*. This is also supported in *eMediator*. In this language, a set of bids can be combined with XOR, forming and *XOR-disjunct*. These XOR-disjuncts can then be combined with non-exclusive ORs to represent independence. For example, a bidder who wants to submit the same offer as in the example above, can do so by submitting the following more concise input to the auctioneer:

$[(\{1\},\$4)]$
OR
$[(\{2\},\$4)]$

OR
$[(\{3\}, \$2)\ \text{XOR}\ (\{4\}, \$2)\ \text{XOR}\ (\{3, 4\}, \$3)]$

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

Since the traditional winner determination problem with no XORs is a special case of winner determination in the OR-of-XORs bidding language, the $\mathcal{NP}$-completeness and inapproximability results apply to winner determination in the OR-of-XORs bidding language as well.[21]

9.1 Winner determination with XORs

Our winner determination algorithm can be easily adapted to handle XOR-constraints between bids, be it in the simple XOR-bidding language, in the OR-of-XORs language, or in a language where the bidder can place XOR-constraints arbitrarily between bids. If two bids are in the same XOR-disjunct, and one of these bids is appended to the search path in SEARCH1, then the other one should be excluded from being appended to that search path. As discussed earlier in this paper, exclusion also occurs if two bids share items, or if two bids constitute a noncompetitive pair (as determined in PRE4). Therefore, as was discussed, an exclusion mechanism is already part of the algorithm. For each bid, a list of bids to exclude is kept. Then, when a bid is appended to the search path in SEARCH1, the exclusion counts of all the bids that this bid excludes are incremented (and decremented once this bid is backtracked from the search path). At any point of the search, only bids with exclusion count 0 are considered for appending to the search path.

Now, handling XORs in the algorithm is easy. Whenever two bids are in the same XOR-disjunct, they are simply added to each others' exclusion lists, and the algorithm will handle XORs correctly. XOR-constraints reduce the size of the search space, so the search can be faster with such constraints than without.

9.2 Preprocessing with XORs

The preprocessors do not work directly with XOR-constraints. In this section we uncover what the issues are, and design modifications to the preprocessors

[21] After this paper was circulated as a technical report, and while it was under review for Artificial Intelligence, another fully expressive input method was proposed [15]. In that approach, whenever bids are to be mutually exclusive, a dummy item is created that is included in those bids. Therefore, the bids cannot ever occur together in a solution because they share an item.

so that they correctly handle XOR-constraints.

9.2.1 First preprocessor (PRE1)

Consider a 2-item auction where bidder one has submitted

$$[(\{1\}, \$4) \text{ XOR } (\{2\}, \$3)]$$

and bidder two has submitted

$$[(\{1\}, \$2)]$$

Now, the naive application of PRE1 would use the bid $(\{1\}, \$4)$ to permanently remove the bid $(\{1\}, \$2)$, and the maximal revenue would then be \$4. However, by accepting the bids $(\{2\}, \$3)$ and $(\{1\}, \$2)$, the auctioneer can achieve a revenue of \$5. In other words, PRE1 discarded a bid when it should not have.

PRE1 can be easily modified to correctly handle XOR-constraints between bids. This is achieved by having PRE1 discard bids less aggressively. First, a bid can be used to prune another bid (for the same set of items) *if the bids are in the same XOR-disjunct.*[22] Second, a bid can be used to prune a bid (for the same set of items) from another XOR-disjunct *if the former bid is alone in its XOR-disjunct.*

9.2.2 Second preprocessor (PRE2)

Consider a 4-item auction where bidder one has submitted

$$[(\{1,2,3,4\}, \$10)]$$

and bidder two has submitted

$$[(\{1,3\}, \$4) \text{ XOR } (\{2,4\}, \$7)]$$

Now, the naive application of PRE2 would use the bids $(\{1,3\}, \$4)$ and $(\{2,4\}, \$7)$ to prune the bid $(\{1,2,3,4\}, \$10)$. However, after that pruning the maximal revenue will be only \$7 while it should be \$10. In other words, PRE2 discarded a bid when it should not have.

PRE2 can be easily modified to correctly handle XOR-constraints between bids. This is achieved by having PRE2 discard bids less aggressively. First, when pruning a bid *within one XOR-disjunct, only allow one bid to prune another* (when the former bid's item set is a subset of the latter bid's, and the former bid's price is higher or equal to the latter bid's price) rather

[22]This is likely to be of limited value in practice since there seems to be no reason for having multiple bids for the same set of items in one XOR-disjunct.

than allowing a set of bids to prune a bid as PRE2 does without XOR-constraints. This is because the bids in the set cannot actually occur together because they are combined with XOR. Second, bids can be used to prune a bid from a different XOR-disjunct, *given that each of the former bids is alone in its XOR-disjunct*.

9.2.3 Third preprocessor (PRE3)

Consider a 2-item auction where a bidder has submitted

$$[(\{1\}, \$4) \text{ XOR } (\{2\}, \$3)]$$

Now, the naive application of PRE3 would decompose the problem into two connected components, each with one bid. Then, the winner determination would run independently on each connected component with a total revenue of \$7, while the actual maximal revenue is only \$4. In other words, PRE3 decomposed the problem when it should not have.

PRE3 can be easily modified to correctly handle XOR-constraints between bids. This is achieved by having PRE3 decompose the problem less aggressively. Specifically, in the graph that PRE3 constructs for the purpose of identifying connected components, *an extra edge is added between every pair of bids that are in the same XOR-disjunct*. Then, as before, the connected components are identified in the graph, and PRE4 as well as the main search are carried out on each connected component separately.

9.2.4 Fourth preprocessor (PRE4)

Consider an auction where bidder one has submitted

$$[(\{1, 3\}, \$5)],$$

bidder two has submitted

$$[(\{2, 5\}, \$4)],$$

and bidder three has submitted

$$[(\{1, 2\}, \$3) \text{ XOR } (\{3, 5\}, \$7)]$$

Now, the naive application of PRE4 would use the bids $(\{1, 2\}, \$3)$ and $(\{3, 5\}, \$7)$ to mark the pair of bids $(\{1, 3\}, \$5)$ and $(\{2, 5\}, \$4)$ noncompetitive together. This would lead to a maximal revenue of only \$7, while the actual maximal revenue is \$9.

PRE4 can be easily modified to correctly handle XOR-constraints between bids. This is achieved by having PRE4 mark tuples of bids noncompetitive less aggressively. First, if bids are in the same XOR-disjunct, they never need to be marked noncompetitive together because the XOR-constraint already ensures that they will never be on the same search path in SEARCH1. Second, bids can be used to mark bids from different XOR-disjuncts noncompetitive together, *given that each of the former bids is alone in its XOR-disjunct*.

9.3 Incremental winner determination and quotes with XORs

The incremental techniques for determining winners and computing quotes, discussed in Section 6, can easily be used even when XOR-constraints between bids are allowed. The only modification to the incremental techniques is that the main search and the preprocessors have to be adjusted as discussed in the previous subsections.

10 Conclusions and future research

Combinatorial auctions, that is, auctions where bidders can bid on combinations of items, tend to lead to more efficient allocations than traditional auction mechanisms in multi-item auctions where the agents' valuations of the items are not additive. This is because the users can express complementarities in their bids, and the winner determination algorithm will take these into account. This paper tackled winner determination in combinatorial auctions where each bidder can bid on unrestricted combinations of items. In such settings, determining the winners so as to maximize the auctioneer's revenue is $\mathcal{NP}$-complete.

The space of partitions of items was first explicated, and closed form asymptotic upper and lower bounds on the number of exhaustive partitions were derived. That number is so large that enumeration will only work with a very small number of items. Dynamic programming avoids some of the redundancy, but it does not scale beyond auctions with a small number of items because it generates the entire search space independent of what bids have actually been received. We showed under what conditions the running time scales polynomially.

The approach of compromising optimality to achieve polynomial-time winner determination is futile if one is interested in worst case approximation guarantees: using the recently proven inapproximability of maximum clique, we proved that the winner determination problem is inapproximable. If the combinations are restricted, somewhat better guarantees can be established by known approximation algorithms for the weighted independent set problem and the weighted set packing problem, but the guarantees remain so weak that they are unlikely to be useful in the domain of auctions in practice. The main hope for practical approximations for special cases lies in new forms of special structure—especially on bid prices.

By imposing severe restrictions on the allowable combinations, optimal winner determination can be guaranteed in polynomial time. However, these restrictions introduce some of the same economic inefficiencies that are present in non-combinatorial auctions.

To tackle the limitations of the existing approaches to winner determination, we developed a search algorithm for optimal winner determination. Via experiments on several different bid distributions that we introduced, we showed that our algorithm significantly enlarges the envelope of inputs for which combinatorial auctions with optimal winner determination are computationally feasible. The highly optimized algorithm achieves this mainly by capitalizing on the fact that the space of bids is often sparsely populated in practice. Unlike dynamic programming, it generates only the populated parts of the search space. The algorithmic methods to implement this include provably sufficient selective generation of children in the search tree, a secondary search to find children quickly, and heuristics that are admissible and optimized for speed. The algorithm also preprocesses the search space by keeping only the highest bid for each combination, by removing bids that are provably noncompetitive (this is determined via search), by decomposing the problem into independent parts, and by marking noncompetitive tuples of bids (this is again determined via search).

The IDA* search algorithm, used in our main search, can easily be distributed across multiple computers for additional speed [41]. The burden of search could also be imposed on the bidders by giving each bidder a portion of the search space to explore. This introduces two risks. First, a bidder may only search her portion partially so as to save computational effort. Such free-riding can be desirable to her since the other bidders are likely to find good solutions anyway—assuming that they themselves do not free-ride. Second, a bidder may report a suboptimal solution if that solution leads to a higher payoff for her. To prevent these problems, the auctioneer can randomly select one or more bidders after they have reported the best solutions that they found, and re-search their portions. If, in some portion, the auctioneer finds a better solution than the reported one, the bidder gets caught of fraudulent searching, and a penalty can be imposed. If the penalty is high enough compared to the cost of computation and compared to the maximal gain from reporting insincerely, and if the probability of getting checked upon is sufficiently high, each bidder is motivated to search her portion truthfully.

The algorithms for winner determination can be directly used to solve weighted set packing, weighted independent set, and weighted maximum clique because they are in fact the analogous problem. Coalition structure generation in characteristic function games is also a very similar problem, but it often has differing features as we discussed.

We presented techniques for incremental winner determination and quote computation. We also showed that quotes are nonadditive, and that a new bid may increase *or decrease* the quotes on other combinations.

We showed that basic combinatorial auctions only allow bidders to express complementarity of items. We then proposed a bidding language called XOR-bids. It is fully expressive so the bidders can express both complementarity and substitutability. We discussed how this bidding language enables the use of the Vickrey-Clarke-Groves mechanism to construct a combinatorial auction where every bidder's dominant strategy is to bid truthfully rather than strategically. We then introduced another fully expressive bidding language called OR-of-XORs. It has all the advantages of XOR-bids, and is more concise. Finally, we extended our search algorithm and preprocessors to handle both of these languages and languages where XOR-constraints can be arbitrarily inserted between bids.

Our search algorithm can easily be extended to settings where the auctioneer cannot keep items. This is accomplished by not using dummy bids and by considering a solution feasible (at any search node) if and only if all of the items are allocated to bidders. In such settings, some bids with negative (and zero) prices may have to be accepted.

In our current work we are developing even faster winner determination algorithms for combinatorial auctions [53, 56], and algorithms for clearing combinatorial reverse auctions and exchanges [47, 53, 57]. We are also studying the impact of side constraints and non-price attributes on the complexity of clearing combinatorial markets [55]. In the direction of more restricted market designs, we are studying the complexity of clearing markets where the goods that are for sale are identical, and the bidders may use different bidding languages such as supply/demand curves [54]. Finally, we are studying selective bid elicitation in combinatorial auctions [10, 9] and are designing optimal bidding agents that may not know their own valuations for bundles *ex ante*, but rather have to compute them [44, 48, 30, 29].

11 Acknowledgment

I thank Kate Larson who significantly contributed to the proof of Proposition 2.1. I also thank two anonymous reviewers for their helpful comments.

References

[1] Martin Andersson and Tuomas Sandholm. Contract type sequencing for reallocative negotiation. In *Proceedings of the Twentieth International Conference on Distributed Computing Systems*, Taipei, Taiwan, April 2000.

[2] Martin R Andersson and Tuomas Sandholm. Leveled commitment contracting among myopic individually rational agents. In *Proceedings of the Third International Conference on Multi-Agent Systems (ICMAS)*, pages 26–33, Paris, France, July 1998.

[3] Martin R Andersson and Tuomas Sandholm. Time-quality tradeoffs in reallocative negotiation with combinatorial contract types. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 3–10, Orlando, FL, 1999.

[4] Martin R Andersson and Tuomas Sandholm. Leveled commitment contracts with myopic and strategic agents. *Journal of Economic Dynamics and Control*, 25:615–640, 2001. Special Issue on Agent-Based Computational Economics. Early version: National Conference on Artificial Intelligence (AAAI), p. 38–45, Madison, WI, 1998.

[5] Craig Boutilier, Moises Goldszmidt, and Bikash Sabata. Sequential auctions for the allocation of resources with complementarities. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 527–534, Stockholm, Sweden, August 1999.

[6] Barun Chandra and Magnús M. Halldórsson. Greedy local search and weighted set packing approximation. In *10th Annual SIAM-ACM Symposium on Discrete Algorithms (SODA)*, pages 169–176, January 1999.

[7] E H Clarke. Multipart pricing of public goods. *Public Choice*, 11:17–33, 1971.

[8] L Comtet. *Advanced Combinatorics*. D. Reidel Pub. Co., 1974.

[9] Wolfram Conen and Tuomas Sandholm. Minimal preference elicitation in combinatorial auctions. In *IJCAI-2001 Workshop on Economic Agents, Models, and Mechanisms*, pages 71–80, Seattle, WA, August 2001.

[10] Wolfram Conen and Tuomas Sandholm. Preference elicitation in combinatorial auctions: Extended abstract. In *Proceedings of the ACM Conference on Electronic Commerce (ACM-EC)*, Tampa, FL, October 2001.

[11] William J Cook, William H Cunningham, William R Pulleyblank, and Alexander Schrijver. *Combinatorial Optimization*. John Wiley & Sons, 1998.

[12] Thomas H Cormen, Charles E Leiserson, and Ronald L Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[13] Sven de Vries and Rakesh Vohra. Combinatorial auctions: A survey. Draft, August 28th, 2000.

[14] C DeMartini, A Kwasnica, J Ledyard, and D Porter. A new and improved design for multi-object iterative auctions. Technical Report 1054, California Institute of Technology, Social Science, November 1998.

[15] Yuzo Fujishima, Kevin Leyton-Brown, and Yoav Shoham. Taming the computational complexity of combinatorial auctions: Optimal and approximate approaches. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 548–553, Stockholm, Sweden, August 1999.

[16] R Garfinkel and G Nemhauser. The set partitioning problem: Set covering with equality constraints. *Operations Research*, 17(5):848–856, 1969.

[17] Theodore Groves. Incentives in teams. *Econometrica*, 41:617–631, 1973.

[18] Magnús M. Halldórsson. Approximations of independent sets in graphs. In K. Jansen and J. Rolim, editors, *The First International Workshop on Approximation Algorithms for Combinatorial Optimization Problems (APPROX)*, pages 1–14, Aalborg, Denmark, July 1998. Springer LNCS 1444.

[19] Magnús M. Halldórsson. Approximations of weighted independent set and hereditary subset problems. *Journal of Graph Algorithms and Applications*, 4(1):1–16, 2000. Early versions appeared in Computing and Combinatorics, Proceedings of the 5th Annual International Conference (COCOON), Tokyo, Japan, 1999, and in Lecture Notes in Computer Science, Vol. 1627, Springer, Berlin, 1999, pp. 261–270.

[20] Magnús M. Halldórsson, Jan Kratochvíl, and Jan Arne Telle. Independent sets with domination constraints. *Discrete Applied Mathematics*, 1998. Also appeared in Proceedings of the 25th International Conference on Automata, Languages, and Programming (ICALP), Aalborg, Denmark, July 1998. Springer Lecture Notes in Computer Science 1443.

[21] Magnús M. Halldórsson and H C Lau. Low-degree graph partitioning via local search with applications to constraint satisfaction, max cut, and 3-coloring. *Journal of Graph Algorithms and Applications*, 1(3):1–13, 1997.

[22] Johan Håstad. Clique is hard to approximate within $n^{1-\epsilon}$. *Acta Mathematica*, 182:105–142, 1999.

[23] D B Hausch. Multi-object auctions: Sequential vs. simultaneous sales. *Management Science*, 32:1599–1610, 1986.

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

[25] Dorit S. Hochbaum. *Approximation algorithms for NP-hard problems.* PWS Publishing Company, 1997.

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

[27] Frank Kelly and Richard Steinberg. A combinatorial auction with multiple winners for universal services. *Management Science*, 46(4):586–596, April 2000.

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

[29] Kate Larson and Tuomas Sandholm. Computationally limited agents in auctions. In *AGENTS-01 Workshop of Agents for B2B*, pages 27–34, Montreal, Canada, May 2001.

[30] Kate Larson and Tuomas Sandholm. Costly valuation computation in auctions: Deliberation equilibrium. In *Theoretical Aspects of Reasoning about Knowledge (TARK)*, pages 169–182, Siena, Italy, July 2001.

[31] John Ledyard. Personal communications at the National Science Foundation Workshop on Research Priorities in Electronic Commerce, 1998. Austin, TX, September 10-12th.

[32] Daniel Lehmann, Lidian Ita O'Callaghan, and Yoav Shoham. Truth revelation in rapid, approximately efficient combinatorial auctions. In *Proceedings of the ACM Conference on Electronic Commerce (ACM-EC)*, pages 96–102, Denver, CO, November 1999.

[33] E Loukakis and C Tsouros. An algorithm for the maximum internally stable set in a weighted graph. *Intern. J. Computer Math.*, 13:117–129, 1983.

[34] R Preston McAfee and John McMillan. Analyzing the airwaves auction. *Journal of Economic Perspectives*, 10(1):159–175, 1996.

[35] John McMillan. Selling spectrum rights. *Journal of Economic Perspectives*, 8(3):145–162, 1994.

[36] Paul Milgrom. Putting auction theory to work: The simultaneous ascending auction. Technical report, Stanford University, Department of Economics, 1997. Revised 4/21/1999.

[37] Noam Nisan. Bidding and allocation in combinatorial auctions. In *Proceedings of the ACM Conference on Electronic Commerce (ACM-EC)*, pages 1–12, Minneapolis, MN, 2000.

[38] Christos H Papadimitriou. *Computational Complexity*. Addison-Wesley, 1995.

[39] Panos M Pardalos and Nisha Desai. An algorithm for finding a maximum weighted independent set in an arbitrary graph. *Intern. J. Computer Math.*, 38:163–175, 1991.

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

[41] Alexander Reinefeld and Volker Schnecke. AIDA* - asynchronous parallel IDA*. In *10th Canadian Conf. on AI*, pages 295–302, Banff, Canada, 1994.

[42] Michael H Rothkopf, Aleksandar Pekeč, and Ronald M Harstad. Computationally manageable combinatorial auctions. *Management Science*, 44(8):1131–1147, 1998.

[43] Tuomas Sandholm. A strategy for decreasing the total transportation costs among area-distributed transportation centers. In *Nordic Operations Analysis in Cooperation (NOAS): OR in Business*, Turku School of Economics, Finland, 1991.

[44] Tuomas Sandholm. An implementation of the contract net protocol based on marginal cost calculations. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 256–262, Washington, D.C., July 1993.

[45] Tuomas Sandholm. *Negotiation among Self-Interested Computationally Limited Agents*. PhD thesis, University of Massachusetts, Amherst, 1996. Available at http:// www.cs.cmu.edu/ ~sandholm/ dissertation.ps.

[46] Tuomas. Sandholm. Contract types for satisficing task allocation: I theoretical results. In *AAAI Spring Symposium Series: Satisficing Models*, pages 68–75, Stanford University, CA, March 1998.

[47] Tuomas Sandholm. eMediator: A next generation electronic commerce server. In *Proceedings of the Fourth International Conference on Autonomous Agents (AGENTS)*, pages 73–96, Barcelona, Spain, June 2000. Early version appeared in the AAAI-99 Workshop on AI in Electronic

Commerce, Orlando, FL, pp. 46–55, July 1999, and as a Washington University, St. Louis, Dept. of Computer Science technical report WU-CS-99-02, Jan. 1999.

[48] Tuomas Sandholm. Issues in computational Vickrey auctions. *International Journal of Electronic Commerce*, 4(3):107–129, 2000. Special Issue on Applying Intelligent Agents for Electronic Commerce. A short, early version appeared at the Second International Conference on Multi–Agent Systems (ICMAS), pages 299–306, 1996.

[49] Tuomas Sandholm, Kate S Larson, Martin R Andersson, Onn Shehory, and Fernando Tohmé. Coalition structure generation with worst case guarantees. *Artificial Intelligence*, 111(1–2):209–238, 1999. Early version appeared at the National Conference on Artificial Intelligence (AAAI), pages 46–53, 1998.

[50] Tuomas Sandholm and Victor R Lesser. Issues in automated negotiation and electronic commerce: Extending the contract net framework. In *Proceedings of the First International Conference on Multi-Agent Systems (ICMAS)*, pages 328–335, San Francisco, CA, June 1995. Reprinted in *Readings in Agents*, Huhns and Singh, eds., pp. 66–73, 1997.

[51] Tuomas Sandholm and Victor R Lesser. Leveled commitment contracts and strategic breach. *Games and Economic Behavior*, 35:212–270, 2001. Special issue on AI and Economics. Short early version appeared as *Advantages of a Leveled Commitment Contracting Protocol* in the proceedings of the National Conference on Artificial Intelligence (AAAI), pp. 126–133, Portland, OR, 1996. Extended version: University of Massachusetts at Amherst, Computer Science Department technical report 95-72.

[52] Tuomas Sandholm, Sandeep Sikka, and Samphel Norden. Algorithms for optimizing leveled commitment contracts. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 535–540, Stockholm, Sweden, 1999. Extended version: Washington University, Department of Computer Science technical report WUCS-99-04.

[53] Tuomas Sandholm and Subhash Suri. Improved algorithms for optimal winner determination in combinatorial auctions and generalizations. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 90–97, Austin, TX, 2000.

[54] Tuomas Sandholm and Subhash Suri. Market clearability. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1145–1151, Seattle, WA, 2001.

[55] Tuomas Sandholm and Subhash Suri. Side constraints and non-price attributes in markets. In *IJCAI-2001 Workshop on Distributed Constraint Reasoning*, pages 55–61, Seattle, WA, 2001.

[56] Tuomas Sandholm, Subhash Suri, Andrew Gilpin, and David Levine. CABOB: A fast optimal algorithm for combinatorial auctions. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1102–1108, Seattle, WA, 2001.

[57] Tuomas Sandholm, Subhash Suri, Andrew Gilpin, and David Levine. Winner determination in combinatorial auction generalizations. In *AGENTS-01 Workshop on Agent-Based Approaches to B2B*, pages 35–41, Montreal, Canada, May 2001.

[58] W Vickrey. Counterspeculation, auctions, and competitive sealed tenders. *Journal of Finance*, 16:8–37, 1961.

[59] Wireless Telecommunications Bureau, Federal Communications Commission. Wireless telecommunications action WT 98-35, September 1998.

[60] Laurence A Wolsey. *Integer Programming.* John Wiley & Sons, 1998.

[61] Peter R Wurman, Michael P Wellman, and William E Walsh. The Michigan Internet AuctionBot: A configurable auction server for human and software agents. In *Proceedings of the Second International Conference on Autonomous Agents (AGENTS)*, pages 301–308, Minneapolis/St. Paul, MN, May 1998.

A Proof of Proposition 2.1

Proof. We first prove that the number of exhaustive partitions is $O(m^m)$. Let there be m items in the auction. Let there be a set of *locations* where combinations of items can form, at most one combination per location. Let the number of locations be m. Let any state of all m locations together be a candidate for an exhaustive partition. This allows for any exhaustive partition to form since an exhaustive partition can have at most m combinations. Now, say that the items get placed in locations one item at a time. Each item could be placed in m different locations, and there are m items to place. Therefore, the number of candidates is m^m. Thus, the number of exhaustive partitions is $O(m^m)$.[23]

[23] The number of candidates, m^m, overestimates the number of exhaustive partitions. Some exhaustive partitions are counted multiple times because for any given combination in the exhaustive partition, the exhaustive partition is counted once for each location where the combination can form (although it should be counted only once).

What remains to be proven is that the number of exhaustive partitions is $\omega(m^{m/2})$. We use the following lemma in the proof.

Lemma A.1 *Let a and b be positive integers. Then $\lceil \frac{a}{b} \rceil! \leq (\frac{a}{b})^{\frac{a}{b}}$ for $\frac{a}{b} \geq 2$.*

Proof.

$$\begin{aligned}
\frac{\lceil \frac{a}{b} \rceil!}{(\frac{a}{b})^{\frac{a}{b}}} \leq \frac{\frac{a}{b}+1}{\frac{a}{b}} \cdot \frac{\frac{a}{b}}{\frac{a}{b}} \cdot \frac{\frac{a}{b}-1}{\frac{a}{b}} \cdots \frac{2}{\frac{a}{b}} &\leq \frac{\frac{a}{b}+1}{\frac{a}{b}} \cdot \frac{\frac{a}{b}-1}{\frac{a}{b}} \\
&= (1+\frac{b}{a})(1-\frac{b}{a}) \\
&= 1-(\frac{b}{a})^2 \\
&\leq 1
\end{aligned}$$

Therefore $\lceil \frac{a}{b} \rceil! \leq (\frac{a}{b})^{\frac{a}{b}}$. $\square$

One way to count the number of exhaustive partitions is to use *Bell numbers.* The Bell number β_m is equal to the number of exhaustive partitions that can be generated with m items. There are several ways of calculating Bell numbers, including Dobinski's formula [8]:

$$\beta_m = \frac{1}{e} \sum_{i=0}^{\infty} \frac{i^m}{i!}$$

To show that the number of exhaustive partitions is $\omega(m^{m/2})$ it suffices to show that

$$\lim_{m \to \infty} \frac{\beta_m}{m^{m/2}} = \infty$$

Since each term in the series of Dobinski's formula is positive, it suffices to take one term, $\frac{i^m}{i!}$ and show that

$$\lim_{m \to \infty} \frac{\frac{i^m}{i!}}{m^{m/2}} = \infty$$

Set $i = \lceil \frac{m}{b} \rceil$ for some constant b, where $b > 2$. Then the expression we are interested in is

$$\begin{aligned}
\frac{(\lceil \frac{m}{b} \rceil)^m}{(\lceil \frac{m}{b} \rceil)! m^{m/2}} &\geq \frac{(\frac{m}{b})^m}{(\lceil \frac{m}{b} \rceil)! m^{m/2}} \\
&\geq \frac{(\frac{m}{b})^m}{(\frac{m}{b})^{m/b} m^{m/2}} \\
&= \frac{m^{\frac{m(b-2)}{2b}}}{b^{\frac{m(b-1)}{b}}}
\end{aligned}$$

We now calculate

$$\lim_{m\to\infty} \frac{m^{\frac{m(b-2)}{2b}}}{b^{\frac{m(b-1)}{b}}}$$

Since the natural logarithm and exponential functions are continuous, we can calculate the limit as follows. Let

$$\begin{aligned}
\phi(m) &= \ln \frac{m^{\frac{m(b-2)}{2b}}}{b^{\frac{m(b-1)}{b}}} \\
&= \frac{m(b-2)}{2b} \ln m - \frac{m(b-1)}{b} \ln b \\
&= \frac{m}{b}[\frac{b-2}{2} \ln m - (b-1)\ln b]
\end{aligned}$$

Then

$$\begin{aligned}
\lim_{m\to\infty} \frac{m^{\frac{m(b-2)}{2b}}}{b^{\frac{m(b-1)}{b}}} &= \lim_{m\to\infty} e^{\phi(m)} \\
&= \lim_{m\to\infty} e^{\frac{m}{b}} e^{\frac{b-2}{2}\ln m-(b-1)\ln b} \\
&= \lim_{m\to\infty} e^{\frac{m}{b}} \lim_{m\to\infty} e^{\frac{b-2}{2}\ln m-(b-1)\ln b} \\
&= \infty \cdot \infty \\
&= \infty
\end{aligned}$$

Thus, we have shown that $\beta_m \in \omega(m^{\frac{m}{2}})$. □

B Proof of Proposition 3.5

Proof. Throughout the proof, when we use the term "tree", we mean a tree that has n leaves, all at depth $m+1$, and has nodes with 1 or 2 children each. Because the number of leaves in the tree is n, the number of binary nodes (nodes with two children) has to be $n-1$. The rest of the nodes have to be unary (nodes with one child each). The claim is that a worst case (case with the largest number of edges given n) can be constructed by having all of the binary nodes as close to the root as possible. Specifically, depths 1 to $\lfloor \log n \rfloor$ of the tree consist of binary nodes only, and the remaining binary nodes, if any, are at depth $\lfloor \log n \rfloor + 1$. We will call any such tree a *wide-early-tree.* Figure 20 (left) illustrates a wide-early-tree where n is a power of 2 so there are no binary nodes at depth $\lfloor \log n \rfloor + 1$. For any given n, all wide-early-trees (which differ only based on how the binary nodes are distributed at depth $\lfloor \log n \rfloor + 1$) have the same number of edges. Assume, to contradict the claim, that there is some tree T that has n leaves, all at depth $m+1$,

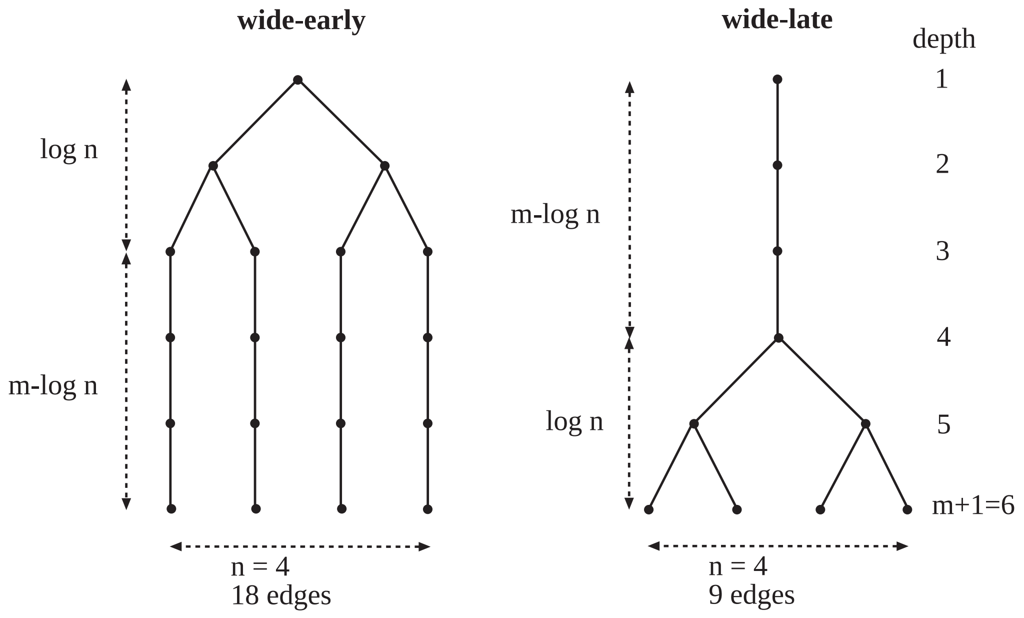

Figure 20: *Left: A wide-early-tree. Right: A wide-late-tree.*

and T has $n-1$ binary nodes with the rest of the nodes being unary, and T has a larger number of edges than a wide-early-tree. Now, since T is not wide-early, it must have a unary node, θ_1, at some depth p (say, without loss of generality, that the right child of θ_1 is empty), and a binary node, θ_2, at some depth q, $p < q$. But now we can perform the following transformation on the tree (Figure 21). Make the right child of θ_1 consist of a path of $q-p$ new edges, followed by what used to be the right subtree of θ_2. Make the right child of θ_2 empty. The resulting overall tree has n leaves, all at depth $m+1$, and $n-1$ of the nodes are binary with the rest being unary, *but the tree has a larger number of edges than T.* Contradiction. This proves that any wide-early-tree is indeed a worst case.

What remains to be proven is the number of edges in a wide-early tree with n leaves, all of which are at depth $m+1$, when each node has at most two children.

Case 1: n is a power of 2 (Figure 20 left). In this case, there is a complete binary tree at the top of the tree. The number of leaves of that binary tree is n. The number of nodes in a complete binary tree is twice the number of leaves minus one, that is, $2n-1$ in our case. The number of edges in a tree is the number of nodes minus one, that is, $2n-2$ in our case.

Below the binary tree there are n branches of length $m-\log n$, amounting to $n \cdot (m-\log n)$ edges.

So, the total number of edges is $2n-2+n\cdot(m-\log n) = nm - n\log n + 2n - 2$.

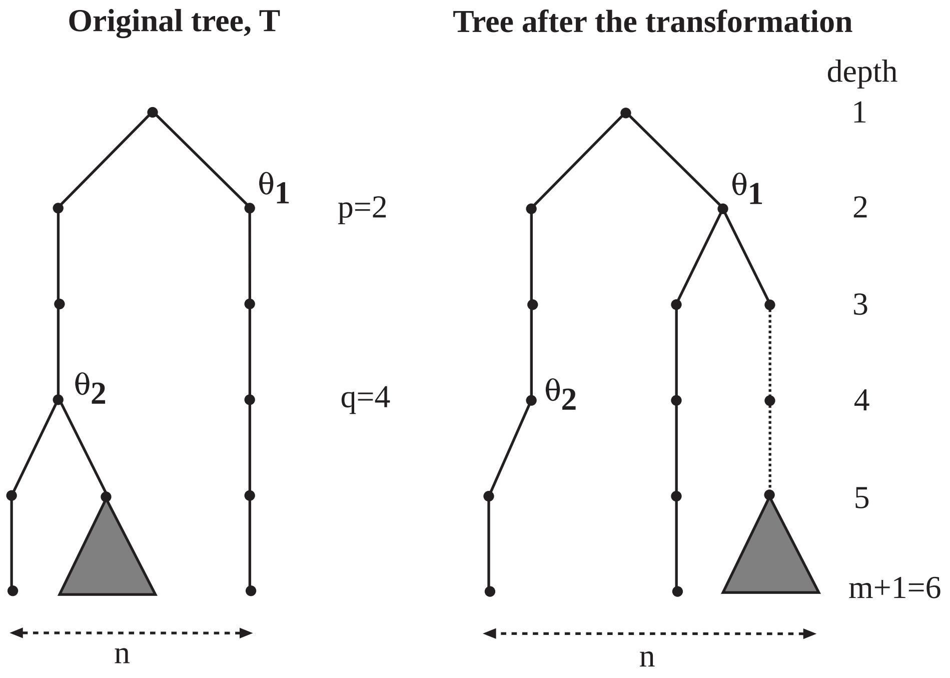

Figure 21: *The transformation used in the proof.*

Case 2: n is not a power of 2 (Figure 22). In this case, the top part of a tree is a complete binary tree. It has $2^{\lfloor \log n \rfloor}$ leaves. The number of nodes in complete a binary tree is twice the number of leaves minus one, that is, $2 \cdot 2^{\lfloor \log n \rfloor} - 1$ in our case. The number of edges in a tree is the number of nodes minus one, that is, $2 \cdot 2^{\lfloor \log n \rfloor} - 2$ in our case.

The middle part includes both unary and binary nodes. Since there are n leaves in the entire tree, and all the nodes below the middle part are unary, the middle part must have exactly n leaves. Since the depth of the middle part is one, the number of edges equals the number of leaves, that is, there are n edges in the middle part.

The bottom part includes n branches of length $m - \lfloor \log n \rfloor - 1$, amounting to $n \cdot (m - \lfloor \log n \rfloor - 1)$ edges.

So, the total number of edges in the tree is $2 \cdot 2^{\lfloor \log n \rfloor} - 2 + n + n \cdot (m - \lfloor \log n \rfloor - 1) = nm - n\lfloor \log n \rfloor + 2 \cdot 2^{\lfloor \log n \rfloor} - 2$. Note that when n is a power of 2, this collapses to the number of edges in case 1 above. Therefore, this case covers case 1 as a special case.

Since the edge counts are exact, the bound in the proposition is tight.

What remains to be proven is the asymptotic complexity. First,

$$\begin{array}{ll} & nm - n\lfloor \log n \rfloor + 2 \cdot 2^{\lfloor \log n \rfloor} - 2 \\ \leq & nm - n \cdot (\log n - 1) + 2n - 2 \\ \in & O(n \cdot (m - \log n + 3)) \end{array}$$

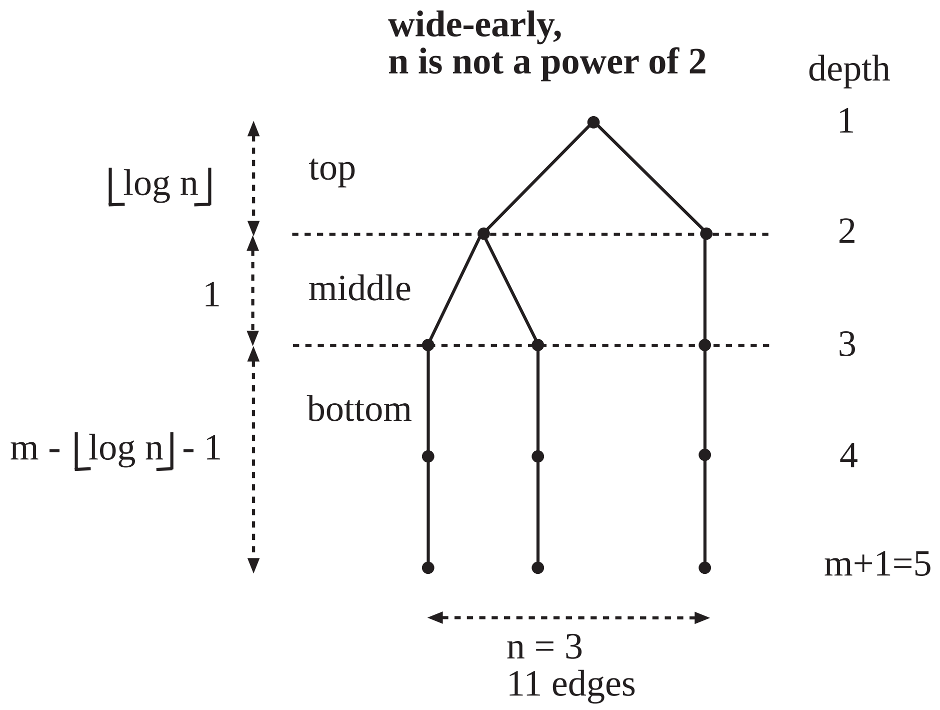

Figure 22: *A wide-early-tree where n is not a power of 2.*

The analysis splits into two case based on the relative magnitude of m and n.

Case A: $m - \log n \geq c$**.** In this case, $n \cdot (m - \log n + 3) \leq (1 + \frac{3}{c}) n \cdot (m - \log n) \in O(n \cdot (m - \log n))$.

Case B: $m - \log n < c$**.** Note that always $m - \log n \geq 0$ because $n \leq 2^m$. In this case, $n \cdot (m - \log n + 3) \leq n \cdot (c + 3) \in O(n)$. $\square$