An Efficient Approximate Allocation Algorithm for Combinatorial Auctions

Edo Zurel * and Noam Nisan †

Institute of Computer Science,
The Hebrew University of Jerusalem

ABSTRACT

We propose a heuristic for allocation in combinatorial auctions. We first run an *approximation* algorithm on the linear programming relaxation of the combinatorial auction. We then run a sequence of greedy algorithms, starting with the order on the bids determined by the approximate linear program and continuing in a hill-climbing fashion using local improvements in the order of bids. We have implemented the algorithm and have tested it on the complete corpus of instances provided by Vohra and de Vries as well as on instances drawn from the distributions of Leyton-Brown, Pearson, and Shoham. Our algorithm typically runs two to three orders of magnitude faster than the reported running times of Vohra and de Vries, while achieving an average approximation error of less than 1%. This algorithm can provide, in less than a minute of CPU time, excellent solutions for problems with over 1000 items and 10,000 bids. We thus believe that combinatorial auctions for most purposes face no practical computational hurdles.

1. INTRODUCTION

In recent years we have seen a surge of interest in, so called, combinatorial auctions. These are auctions in which a multitude of non-identical items are sold concurrently. The combinatorial nature of the auction comes from allowing bidders to place bids on combinations of items and not just on single items. Thus, for example, a bidder may offer $10 for the combination of a left sock and a right sock, but make no offer at all for only a single sock. Similarly, a bidder may make an offer of $10 for a blue shirt or for a red shirt, but not be willing to pay more than $10 even if he gets both shirts. In general, a combinatorial auction allows bidders to express complementarities – where the bid for a combination of items is worth more than the sum of the separate items – or substitutabilities – where the bid for a combination of items is less than the sum of the separate items. Such combinatorial auctions have been suggested for a host of auction situations such as those for spectrum licenses, pollution permits, landing slots, computational resources, and others. See [14] for a survey.

When auctioning multiple related items, combinatorial auctions are desirable as they allow bidders to express their true valuation of combinations, and thus should lead to more economically efficient allocations (as well as, likely, higher seller revenue). However, several problems exist before combinatorial auctions can be used in a large scale. One of the major problems is the computational difficulty of determining an optimal allocation for a given set of bids. This problem is usually formalized as a packing problem and is known to be NP-complete to solve, or even to approximate [12]. The problem has received much attention lately [14, 3, 7, 13, 1, 4, 11, 10], with three approaches usually taken: heuristics to improve the running time of finding the optimal solution, heuristics to improve the solution quality of efficient algorithms, or special cases that can be optimally solved efficiently.

This paper follows the second approach: we present a very fast heuristic that finds allocations that are close to optimal. Our algorithm efficiently solves problems that are large enough and provides allocations that are good enough, as to suggest to us that the computational difficulty of allocation is not an issue in most practical combinatorial auctions! Obviously, more experimentation with real combinatorial auctions should be done before this last statement can be fully verified.

Our heuristic is based on the following simple strategy: First, run the linear-programming relaxation of the packing problem; then, use these results to re-order the bids; and, finally, run a greedy algorithm. Similar heuristics are common in combinatorial optimization, and in the context of combinatorial auctions it was suggested in [7]. Our algorithm has two novel elements that improve both its running time and its approximation error: instead of solving the linear program we only *approximate it*, and instead of performing a single greedy algorithm we attempt *local improvements in the ordering of the bids.*

Since solving a linear program may be quite costly for large input sizes, we only approximate the solution. We take advantage of the fact that combinatorial auctions fall into the category of, so called, "positive linear programs", or "fractional packing problems" that can be approximated efficiently. Our approximation algorithm finds a solution

*E-mail:zurel@cs.huji.ac.il.

†E-mail:noam@cs.huji.ac.il. Supported by grants from the Israeli Ministry of Science and the Israeli Academy of Sciences.

EC'01, October 14-17, 2001, Tampa, Florida, USA.

for the linear programming relaxation of the combinatorial auction that is, provably, within a factor of $1+\epsilon$ from the optimal one. This approximate result is sufficient for our purposes since we are only using it to determine an order on the bids to be used by the second stage. Optimality of the LP solution neither guarantees, nor is required, for the quality of solution produced by the second stage.

Instead of running a single greedy algorithm based on the resulting order, we perform a hill-climbing process where we repeatedly improve the solution quality by making a single change in the *order* of the bids, and run a greedy algorithm on the modified order.

We shortly present the main ideas of each of these phases, leaving the details to section 2.

1.1 The Approximate Positive Linear Program Algorithm

In [2, 9, 6] primal-dual approximation algorithms were suggested for linear problems where all the coefficients are non-negative. Variants of this type of algorithms were used to obtain fast asymptotical sequential running times [9], as well as parallel algorithms [6] and a distributed implementation [2]. The main idea in these algorithms is a certain relaxation process where at each stage a primal variable is increased and the dual variables are updated by an amount that is exponential in the amount that they are "covered". We present our algorithm, which uses the same idea, in terms of a price-adjustment process.

Assume that there are n goods and m input bids, each offering a value v_j for a bundle of goods $S_j \subseteq 1...n$ [1]. We will be maintaining a (scaled) price p_i for each good. The key concept is the definition of a (scaled) fractional allocation to a bid j at prices $p_1...p_n$ to be, $A_j(p_1...p_n) = exp(-\sum_{i\in S_j} \frac{p_i}{v_j})$. Thus if the offered price is small compared to the item prices, $\sum_{i\in S_j} p_i >> v_j$, the allocation is nearly zero, while if it is large, $\sum_{i\in S_j} p_i << v_j$, the allocation is nearly 1. The total demand for an item i at prices $p_1...p_n$ is the sum of the allocations for it: $D_i(p_1...p_n) = \sum_{j|i\in S_j} A_j(p_1...p_n)$.

The algorithm repeatedly picks the item i with highest demand $D_i(p_1...p_n)$ at the current prices, and increases its price by a small (carefully chosen) quantity δ. It then updates the values of A_j and D_i and repeats. Feasible solutions to the primal and dual problems can be obtained at each stage by scaling the vector of prices and the vector of fractional allocations.

We prove that the values these primal and dual feasible solutions approach each other rapidly, thus yielding an approximate solution for the linear program.

The worst case asymptotic running time of our algorithm ($O(n^3 m \log(m/\epsilon)/\epsilon^3)$) is rather similar to the asymptotic running times of previous algorithms for approximate positive linear programs. However, our algorithm has two advantages that make it especially appropriate for implementation. First, we argue theoretically that on "normal" inputs it will actually run in time $O(nm \log(m/\epsilon)/\epsilon^2)$ (which, for fixed ϵ, is nearly linear in the size of a matrix needed in order to represent the input.) Second, the algorithm is very simple, and all the "hidden constants" are small. To the best of our knowledge we are the first to implement an approximation algorithm for positive linear programs, and indeed we can report significantly faster running times than commercial exact solvers of linear programs. We believe that our algorithm for approximate positive linear programs is of independent interest.

[1] As observed in [4, 7] this does not lose generality, since more complicated bids may be expressed in this fashion.

1.2 Hill-climbing Using a Greedy Algorithms

The approximate linear programming phase provides a fractional allocation $0 \leq A_i \leq 1$ for every bid, and an item price p_j for every good. The basic intuition from [7] is that we should attempt to allocate bids with larger A_i before bids with lower A_i and, for those bids with $A_i = 0$, attempt allocating bids with larger $v_j / \sum_{i\in S_j} p_j$ first. Thus the greedy algorithm first orders the bids according to this ranking. It then greedily goes over the bids according to this order, satisfying in turn each bid that does not request any item that was previously allocated.

This greedy algorithm usually performs reasonably well in experiments, but not very well – its distance from optimal is usually in the range of 5%-50%. We thus attempt improving the result using local improvements. The key idea here is to define locality in terms of the *order*: a local improvement is a modification of the order of the bids via moving a single bid to first place and then re-running the greedy algorithm on the modified order. The modified order is maintained if the solution of the greedy algorithm has improved, and another local improvement is attempted, until no more local improvements are possible. We have found, in experiments, that only a very small number of local improvements are usually needed. In order to maintain simplicity, we have not attempted fancier variants of hill-climbing such as simulated annealing.

1.3 Experimental Results

We have performed a significant amount of experiments on a C implementation of the algorithm (available on the web at `http://www.cs.huji.ac.il/~zurel`). We have used the complete data set of [14] which is available online to evaluate our algorithms. This data set contains 2240[2] problems with numbers of items ranging between 25 to 400 and number of bids ranging between 50 to 2000, drawn from five different distributions. All our experiments were done on an "old PC" (Pentium II 450 MHz with 128MB RAM running Linux kernel 2.2.5). We compared our results to the optimal results and to the running times reported in [14] which were obtained using commercial (CPLEX) integer programming software running on a rather strong computer (SGI Origin 200 with four MIPS R10000 processors at 225 MHz each with 1GB RAM), and which compared favorably with other programs for combinatorial auctions.

The bottom line is given in the following two sets of results showing the quality of the solution and the running times of our algorithm on the complete data set of [14]. Each point in each of these graphs represents the average taken over all instances from a specific distribution and a specific problem size in the data set. We see that the solution quality is usually less than 0.5% away from optimal, where for no distribution and problem size it is more than 4% away (and on none of the 2240 instances tested, more than 14% from optimal). The running times are shown as a fraction of the running times of CPLEX reported by [14],

[2] The set actually contains 2560 problems, but [14] have only provided the optimal solutions for 2240 of them.

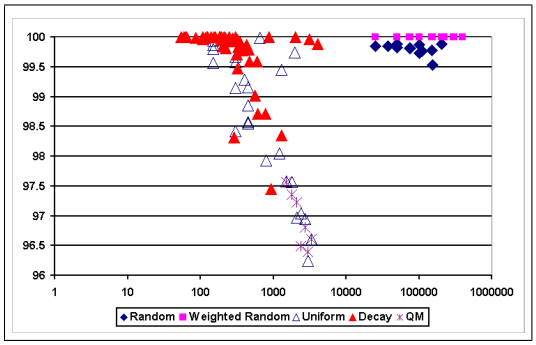


Figure 1: Approximation accuracy vs. problem size

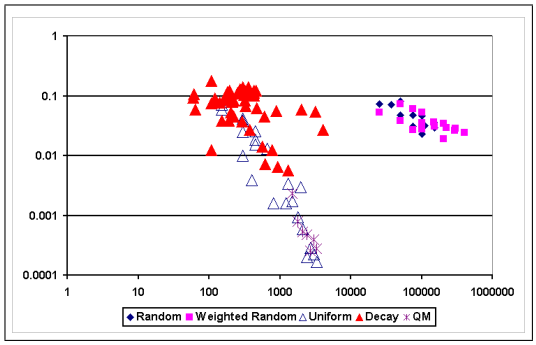


Figure 2: Speedup over CPLEX vs. problem size

and are usually between one and four orders of magnitude faster. (Probably another order of magnitude is implied for the algorithm itself since the reported times of CPLEX are on a faster computer.) For the larger problems in the data set this means reducing the time from about half an hour to seconds. A more detailed analysis and breakdown of the results appears in section 3.

We have performed a significant amount of experiments to evaluate the different aspects of the performance of our algorithm. We present the bottom line of these experiments in the form of a FAQ, with details appearing in section 3.

Q: How well does the algorithm perform on larger problem sizes and on other distributions?
A: We have also tried our algorithm on instances drawn from the distributions of [5] for problem sizes as large as 32000 goods and 128000 bids. The running times remain very good – under one minute for the largest problem which had a total of over 700,000 items in bids. In fact, for some of the larger problems, solving the problem was much faster than generating the input using the code supplied by [5]. As [5] do not provide the optimal solutions, we could not evaluate the solution quality. Estimates for the approximation error were obtained using the dual solution of the LP, providing an *upper-bound* to the gap of about 2.85% on the average, with less than 1% on the larger problems. Based on our experience with the distributions of [14], even these small numbers are likely serious over-estimates of approximation error.

Q: Does the algorithm function well even when the "integrality gap" is large?
A: A major issue in combinatorial optimization is the "integrality gap" : how far away the fractional optimum is from the integer one. The distributions of [14] exhibit integrality gaps as small as zero and as large as 40%. The algorithm tracks the optimal solution well, and runs quickly even in the cases of a large integrality gap.

Q: Is approximate LP really faster than usual LP algorithms?
A: Yes, getting an approximation to within 10% is about one to two orders of magnitude faster than the LP Solver in MATLAB, with the gap getting wider for larger problems. Furthermore, the actual running times do behave like $O(mn\log(mn))$, suggesting that the speedup should further increase for yet larger problems.

Q: How is the running time of the Approximate LP affected by the approximation error?
A: It seems that the reasonable limits of the approximate LP algorithm are about 0.1%-1% approximation error, corresponding to the theoretical quadratic dependence on ϵ.

Q: How does the approximation error in the first phase affect the final solution quality?
A: Not much. The final solution quality is hardly affected by reducing the error in the first phase from 20% to 1%.

Q: Does the hill-climbing phase reduce the error significantly?
A: The error is usually reduced by a factor of 5 – 20, depending on distribution. For one distribution ("weighted random") the average error dropped from 44% to 0%. A single local modification already provides a large improvement.

Q: If allocation isn't the problem with combinatorial auctions, what is?
A: The communication cost of sending the auction inputs over a T-1 (1Mb) line is comparable to the cost of solving them. For certain distributions, the communication cost is 3 times larger than the solution cost. See section 3.6 for details. The communication costs of combinatorial auctions are further analyzed in [8].

Q: Can the solution quality be further improved?
A: We have also tried a variant where the hill climb attempts 2 order changes and ran it on the distributions that yielded the lowest accuracy. The solution error decreased from 4% to 2%, and while the running time of the hill climb stage was increased by a factor of $O(n)$, it still maintained a speedup of two order of magnitudes over CPLEX for the large problem instances. See section 3.4 for details.

2. THE ALGORITHM

Input: The input of a combinatorial auction over items $1..n$ consists of m bids, where each bid is described by an offer $v_j \geq 0$ and the set of items $S_j \subseteq 1..n$.

Output: The output is a collection W of winning bids with highest possible value $\sum_{j\in W} v_j$ that are pairwise disjoint (i.e. for all $j \neq j' \in W$ we have $S_j \cap S_{j'} = \emptyset$).

Our algorithm proceeds in two phases: in the first phase a linear programming relaxation is solved approximately and in the second phase a hill climbing process is run using greedy algorithms. We present these two phases in the next two subsections.

2.1 The Approximate Positive Linear Programming phase

The input to this phase includes besides the user inputs described above, an allowed approximation error ϵ. This phase provides an ϵ-error approximation to the following linear problem, that is presented in both its primal and dual forms:

The Primal LP Problem:	**The Dual LP Problem:**
Find: $p_1...p_n$	**Find:** $A_1...A_m$
Minimizing: $\sum_i p_i$	**Maximizing:** $\sum_j A_j v_j$
Subject to:	**Subject to:**
• $\forall i = 1...n$: $p_i \geq 0$	• $\forall j = 1...m$: $A_j \geq 0$
• $\forall j = 1...m$: $\sum_{i \in S_j} p_i \geq v_j$	• $\forall i = 1...n$: $\sum_{j \mid i \in S_j} A_j \leq 1$

Basic Linear Programming theory implies that any feasible primal has value no less than any feasible dual, $\sum_i p_i \geq \sum_j A_j v_j$, with equality achieved for the optimal solutions. The phase ends when the values are within ϵ from each other.

Phase Output: $p_1...p_n$ and $A_1...A_m$ such that $\sum_i p_i \leq (1+\epsilon) \cdot \sum_j A_j v_j$.

The algorithm is very simple:

Approx. Positive Linear Programming Algorithm

Initialize: $\forall i\ p_i \leftarrow 0$, $\forall j\ \alpha_j \leftarrow 0$, $\forall j\ A_j \leftarrow 1$

Repeat:

1. $\forall i\ D_i \leftarrow \sum_{j \mid i \in S_j} A_j$.
2. find b such that $D_b = \max_i D_i$.
3. $p_b \leftarrow p_b + \delta_b$ (see below for choice of δ).
4. $\forall j$ such that $b \in S_j$: $\alpha_j \leftarrow \alpha_j + \frac{\delta_b}{v_j}$, $A_j \leftarrow e^{-\alpha_j}$.
5. $\alpha \leftarrow \min_j \alpha_j$.

Until $\frac{\sum_i p_i}{\alpha} \leq (1+\epsilon)\frac{\sum_j A_j v_j}{D_b}$

Output: Scaled prices $\forall i\ \ \overline{p}_i = \frac{p_i}{\alpha}$ and scaled allocations $\forall j\ \ \overline{A}_j = \frac{A_j}{D_b}$.

Choice of δ: $\delta_b = \epsilon \frac{\sum_{j \mid b \in S_j} A_j}{\sum_{j \mid b \in S_j} \frac{A_j}{v_j}}$

The correctness of this algorithm is given by the following theorem.

THEOREM 2.1. *The algorithm produces solutions with values that are within ϵ of each other. Its running time is bounded by $O\left(\frac{R|I|\log(m/\epsilon)}{\epsilon^2}\right)$. Here $R = \tau^*/min_j v_j$, where τ^* is the optimal solution value, and $|I|$ is the total input size.*

Let us take a moment to analyze the running time. The easiest way to interpret R is as the product of the number of winning bids by the ratio of the average winning bid to the smallest bid. For "normal" auctions we would expect the ratio to be constant and the number of winning bids to be equal to n divided by the average bid size. In the worst case, the ratio can be bounded by $O(n/\epsilon)$ by throwing away very small bids and the number of winning bids is bounded by n. In the "normal case", we get a running time of $O(mn\log(m/\epsilon)/\epsilon^2)$ (since the input size is equal to m times the average bid size).

The proof of the theorem is given in the appendix.

2.2 The Hill Climbing phase

The input to this phase is the set of bids, (S_j, v_j), as well as the output of the approximate linear programming phase: the scaled prices $\overline{p}_1...\overline{p}_n$ and the scaled fractional allocations $\overline{A}_1...\overline{A}_m$. The algorithm is simple:

Hill Climbing Algorithm:

1. Initialize $\pi \leftarrow InitialOrder(\overline{p}_1...\overline{p}_n, \overline{A}_1...\overline{A}_m)$.
2. Repeat $\pi \leftarrow LocallyImprove(\pi)$ Until no improvement is made.
3. Return $Greedy(\pi)$.

InitialOrder($\overline{p}_1...\overline{p}_n$, $\overline{A}_1...\overline{A}_m$)

Order the bids by decreasing value of $\overline{A}_j$; bids with $\overline{A}_j = 0$ are ordered with decreasing value of $v_j / \sum_{i \in S_j} \overline{p}_i$.

Greedy(π):

1. $W \leftarrow \emptyset$, $AllocatedItems \leftarrow \emptyset$
2. For $j = 1...m$ (where bids' order is π), if $S_j \cap AllocatedItems = \emptyset$ then
   - $W \leftarrow W \cup \{j\}$
   - $AllocatedItems \leftarrow AllocatedItems \cup S_j$
3. Return the set W, with value $\sum_{j \in W} v_j$.

LocallyImprove(π):

For $j = 1...m$ (Where bids' order is π)

1. $\pi' \leftarrow \pi$, with element j moved to first place
2. If value of $Greedy(\pi') >$ value of $Greedy(\pi)$ then return π'.

This algorithm can not be guaranteed to provide an optimal result, of course, but clearly returns a feasible allocation:

PROPOSITION 2.2. *Hill Climbing always returns a feasible allocation.*

In our implementation we have put an upper bound of n on the number of times that the loop inside the Local Improvement routine may execute. This upper bound significantly reduces the time, and in experiments does not seem to noticeably degrade the solution quality. A similar upper bound may be put on the number of local improvements allowed in the main loop, although experimentation has showed that even without an upper bound the number of improvements is very small. Both of these bounds do not affect the feasibility of the solution.

The worst case running time of the hill climbing phase is given by the running time of a single greedy pass times the product of the two upper bounds chosen. In worst case, the greedy algorithm takes $O(|I|)$ time, where $|I|$ is the total input size. Typically, however, and somewhat surprisingly, a greedy pass will only take $O(m+n)$ time: If the order π is good then we expect a constant fraction of the first bids to be winning, where "first bids" are the bids encountered until at least a constant fraction of the items are allocated. Once a constant fraction of items are allocated, non-winning bids are expected to be detected as such in constant time per bid (since we expect one of the first items in the bid to already be allocated). Thus the time for the "first bids" should be $O(n)$ and the time for the rest $O(m)$. In our implementation we put an upper bound of n on the loop inside the Local Improvement routine. Since the number of local improvements is usually constant, we get that the total running time of this stage is "normally" $O(mn)$. Experimental results shown in section 3 confirm this argument.

3. EXPERIMENTAL RESULTS

Most of our experimental result are on the corpus of instances provided by Vohra and de Vries [14]. This corpus uses five different distributions, where each distribution contains a sequence of problem sizes, varying both the number of items as well as the number of bids. The problem size is defined as the total number of items in bids (i.e. the sum of the bid sizes), which is different from the product of the number of items and number of bids. The results in this section are usually reported for each distribution separately, with the graphs drawn according to the problem size. In some cases we use a simple indexing of the parameters values of [14], indexed by increasing problem size, and in other cases we use the problem size itself. Each data point in the graphs below represents the average taken over all instances from a given distribution and problem size. Unless specified otherwise, the approximation error ϵ of the linear programming phase was chosen to be 20%. The following list gives a short description of the five distributions; further details can be found in [14].

Distributions of Vohra and de Vries [14]

1. **Random:** For each bid, pick the number of items randomly from 1..m. Randomly choose that many items without replacement. Pick the bid offers randomly from [0,1].
   The number of items varies from 100 to 400 and the number of bids varies from 500 to 1000.

2. **Weighted Random:** For each bid, pick the number of items randomly from 1..m. Randomly choose that many items without replacement. Pick the bid offers randomly from [0,number of items in bid].
   The number of items varies from 100 to 400 and the number of bids varies from 500 to 2000.

3. **Uniform:** For each bid, pick a constant number of items randomly from 1..n. Randomly choose that many items without replacement. Pick the bid offers randomly from [0,1].
   The number of items varies from 25 to 100, the number of bids varies from 50 to 1100, and the bid size is either 3,8, or 11.

4. **Decay:** For each bid, give it one random item. Then repeatedly add a new random item with probability α until an item isn't added or the bid contains all n items. Pick the bid offers randomly from [0, number of items in bid].
   The number of items varies from 50 to 200, the number of bids varies from 50 to 200, and the probability ranges from 5% to 95%.

5. **Quadratic Model:** Each object k is assigned a value V_k which is common to all bidders. For each bidder j choose a subset M_j of objects at random, and a parameter μ_j. M_j represents the group of objects which are viewed by bidder j as being complementary to each other, and μ_j indicates the strength of the complementarities for this bidder. The value of bidder j of the subset S of objects is $\sum_{k\in S} V_k + \mu_j \sum_{k,q\in S\cap M_j} V_k V_q$.
   The number of items is 100, and the number of bids varies from 500 to 1100, while the bid size is fixed at 3. v_k, μ_j, and M_j are not given by [14].

3.1 Solution Quality

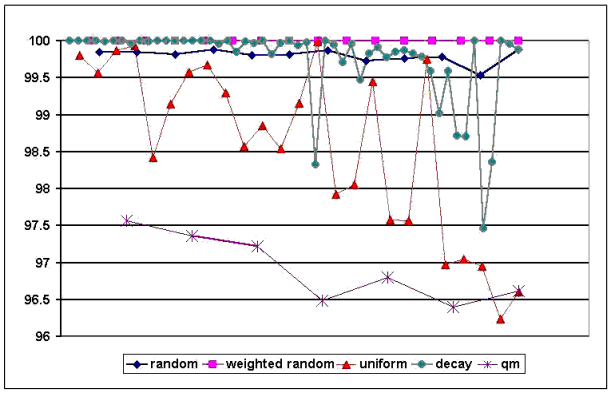


Figure 3: Solution Quality by problem size

Figure 3 shows the solution quality of ALPH (Approximate LP followed by Hill Climb) as compared to the optimal solution reported by [14], for each of the five distributions. The x axis is the problem number where the problems are ordered by size, and the y axis is the percentage of the ALPH solution out of the optimal solution. As can be seen the approximation error depends on the distribution, is always quite low, and for some distributions very low.

Comment: the problem sizes of different distributions are not in the same scale in this figure. The same information, but with all distributions on the same scale of problem size, appears in figure 1.

3.2 Running Time

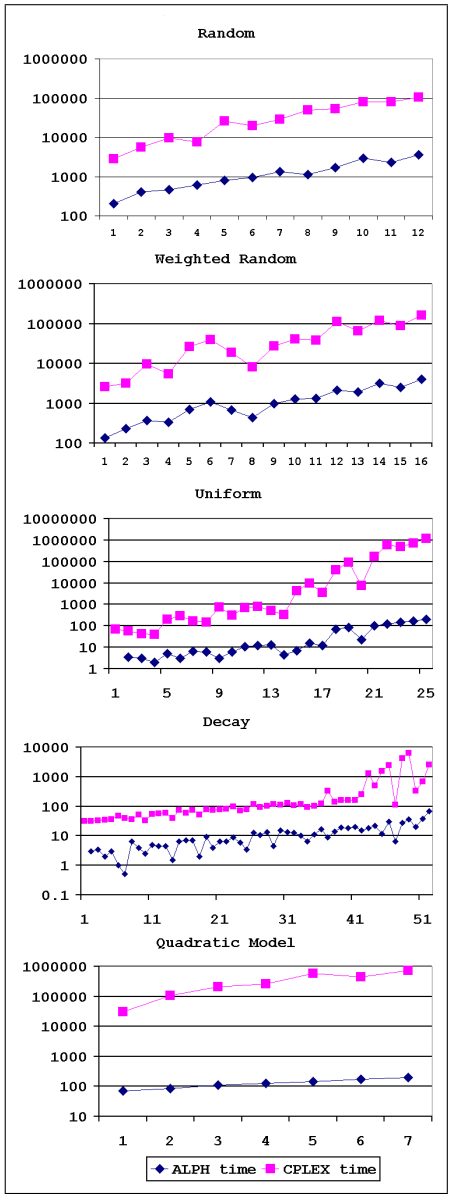


Figure 4: ALPH and CPLEX running times for each problem in distribution ordered by problem size

In figure 4, the X axis has the problem number, ordered by the size of the input, and the Y axis is time (in milliseconds) on a logarithmic scale. It's clear that in random distribution the running times of the ALPH (Approximate LP followed by Hill Climbing) are one order of magnitude better than CPLEX. In weighted random distribution ALPH is 14 to 50 times faster than CPLEX, and the gap increases with problem size. In uniform distribution ALPH is usually at least two orders of magnitude faster than CPLEX, where the gap increases with problem size up to 4 orders of magnitude. In decay distribution ALPH is usually more than 8 times faster than CPLEX. In quadratic model distribution ALPH is three orders of magnitude faster than CPLEX (in fact about 2000 times faster).

3.3 Solution Quality and the Integrality Gap

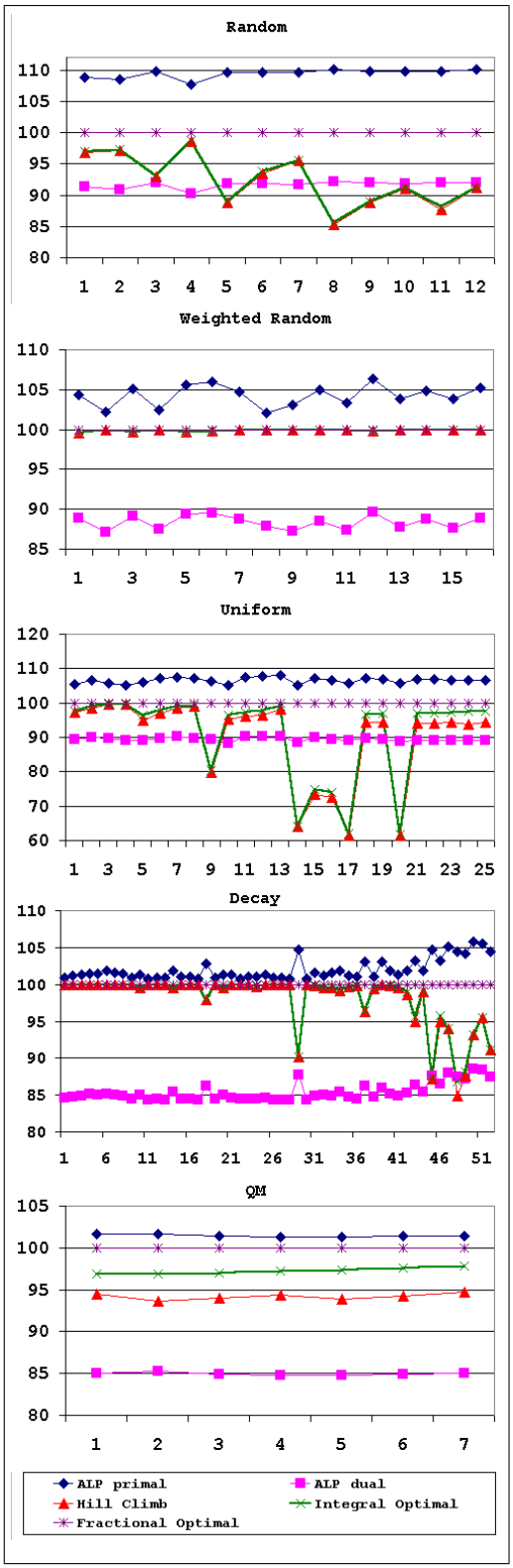
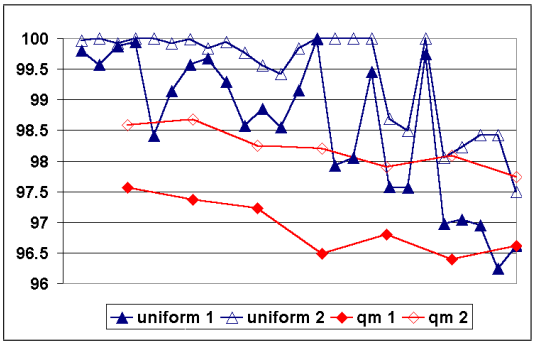


Figure 5: ALP primal, ALP dual, optimal and hill climb solutions as percent of optimal fractional solution for each distribution against problem size

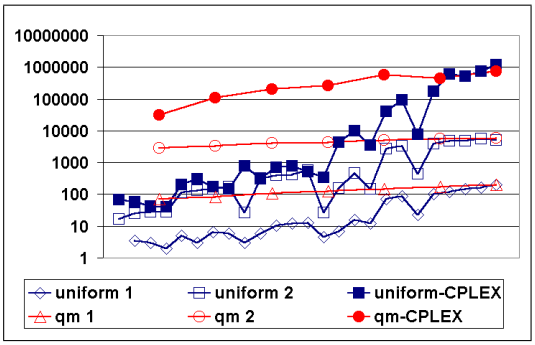
It is well known that the gap between an optimal fractional allocation (the solution of the LP) and an optimal integer allocation is a critical factor in combinatorial optimization. The charts in figure 5 compare the following four values to the value of the optimal fractional solution: the optimal integer solution, the solution found by the ALPH algorithm, as well as the primal and dual solutions found by the approximate linear programming stage. Each figure describes the problems of a single distribution, where the x axis is the problem number in the order of the problems as defined by the problem input size, and the y axis is percent (of the value of the optimal fractional solution for that problem instance). As expected, the ALP primal value is always above 100%, and the ALP dual is always below it at a constant distance from each other (ϵ was 20% for all problem instances here). It seems that usually (but not always) the primal solution is closer to optimal than the dual.

The integrality gap varies widely among the distributions and the different problems sizes. Notice that the solution provided by the ALPH algorithm is close to the optimal integral solution even when the gap is large. One can observe that in random, weighted random, and decay distributions, the Hill Climb value and Integral Optimal value are almost completely aligned. In uniform distribution, it's clear that for small problem size the error is quite small, but it increases for the large problem sizes. In the quadratic model distribution, the error is consistent in size (around 4%).

3.4 Further Improvements on Hill Climbing



(a) Solution Quality Comparison



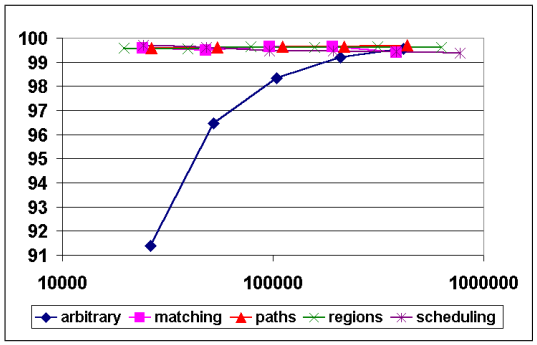
(b) ALPH and CPLEX time (in ms)

Figure 6: 1-jump and 2-jump hill climb for uniform and quadratic model distributions

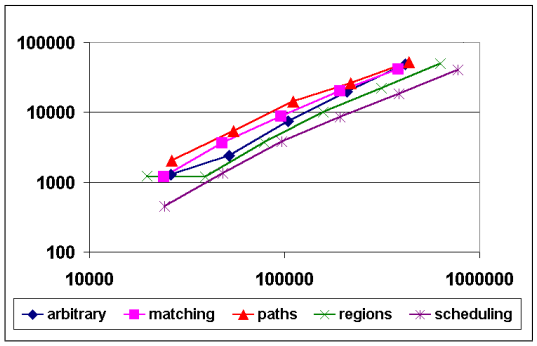
Our results indicate that for large instances of the uniform distribution and all instance of the quadratic model distribution, hill climbing is significantly less effective than for the other distributions with an average error of around 4%. On the other hand, it's clear that for these problems we achieve a huge speedup (3 orders of magnitude) over CPLEX solution time. Therefore it is logical to attempt to spend some more time in the hill climb stage in order to achieve a higher accuracy.

Recall that the *LocallyImprove* method in the Hill Climbing Algorithm simply tries to move element j to the first place, where j was limited to $1...n$. We tried a simple variant of this, where **two** elements, $i = 1...n$, and $j = 2...i+3$ are moved to the first and second position in the order accordingly. This resulted in an improvement of about 2% over the results of the regular hill climbing, with a time cost of about 5 seconds. In general, the two element variation costs $O(n)$ times more than the regular hill climbing phase, so the time for normal run is $O(n^2 m)$ instead of $O(nm)$. Figure 6(a) plots the accuracy of the different schemes (percent of the optimal integral value) against different problem sizes. 1 indicates regular hill climbing, and 2 indicates the two element variation. Figure 6(b) plots the time cost of the different schemes on a logarithmic scale against different problem sizes.

3.5 Results on Other Distributions



(a) Approximation accuracy (percent of upper bound)



(b) ALPH time (in ms)

Figure 7: ALPH performance by problem size on Leyton-Brown et. al. distributions

We have performed extensive experiments with the engines provided by Leyton-Brown, Pearson, and Shoham [5] for generating combinatorial auction problem instances. We have tested our algorithm on several problems generated from each of the five main distributions there: **arbitrary, matching, paths, regions, and scheduling**. For details

about these distributions see [5]. For each of the five distributions we generated 10 groups of problems with input size varying from 100 to 1000000 items in bids. Each of these groups consists of 10 problem instances, and overall 520 problems were generated. Note that these engines create a problem input, but do not provide an optimal solution value for it. For small instances (up to 5000 items in bids) we used a branch and bound algorithm to calculate the exact optimal solution (see table 1 for details on the performance of ALPH on problems containing 200 bids).

For the larger instances, we used the primal solution value of an ALP (run with $\epsilon = 1\%$) as an upper bound for the actual optimal solution. Figure 7(a) shows that an accuracy of over 99% is achieved for most problem instances, with the exception of small instances of the arbitrary distribution. The running times reported in figure 7(b) were obtained using $\epsilon = 20\%$ for the ALP stage. The chart plots running time in milliseconds on a logarithmic scale against the problem size on a logarithmic scale. Its clear from the chart that the algorithm solves under a minute even for problem sizes of close to a million items in bids.

Two facts stand out: our algorithm runs quickly, and the approximation error is always small and is tiny for large problems. Extrapolating from the results of section 3.3, we may assume that even these small upper bounds for the approximation error are a serious over-estimate and that the real error is even smaller.

Comment: The distributions of [5] were constructed to mimic "real" bids in combinatorial auctions. The tiny gap between the integer solution and the fractional solution in these distributions suggests one of two explanations: either the distributions simply missed the "hard cases" or real combinatorial auctions *will* usually exhibit a very tiny gap between the integer allocation and the fractional one.

Table 1: ALPH solution quality of 200-bid problems from Leyton-Brown et. al. distributions

Distribution	Items	Bids	Avg. Bid Size	gap from LP upper bound	gap from optimal
arbitrary	136.9	202.3	13.64	29.2%	4.0%
matching	74.0	200.0	3.00	0.7%	0.2%
paths	58.0	202.0	3.50	0.5%	0.0%
regions	58.2	202.4	9.39	2.8%	0.5%
scheduling	33.9	202.8	5.29	3.0%	0.3%

3.6 Communication and Solution of Combinatorial Auctions

Table 2 depicts the time cost of communicating and solving (using ALPH) the largest problem in each of the distributions in both Vohra and de Vries and Leyton-Brown, Pearson, and Shoham. It's clear from the table that if communication is done over a T-1 line (1Mb), then the cost of communication of the auction problem is comparable to the cost of solving it. In fact, the communication time is 3 times larger than the solution time in the weighted random and uniform distributions.

Table 2: Time Comparison of problem communication and solution

Type of Distribution	Items	Bids	Size (KB)	Time to Solve (sec)	Time to send over 1Mb (sec)
random	400	1000	752	4.03	5.88
w. random	400	2000	1484	3.87	11.59
uniform	100	1100	24	0.06	0.19
decay	100	200	14	0.16	0.11
q. model	100	1100	23	0.17	0.18
arbitrary	6032	32001	1697	21.20	13.26
matching	32365	128001	2727	22.60	21.30
paths	32782	128000	2933	23.65	22.91
regions	11889	64000	2489	12.56	19.45
scheduling	9855	128011	3587	20.66	28.02

3.7 Approximate Positive Linear Programming Algorithm

3.7.1 *Speed relative to MATLAB*

The ALP algorithm may itself be compared to other LP-solvers. We have chosen to compare it to the LP-solver supplied in MATLAB, although it should be noted that the output of the ALP is NOT the optimal solution to the LP but only an approximation to it. Figure 8 plots the ratio of

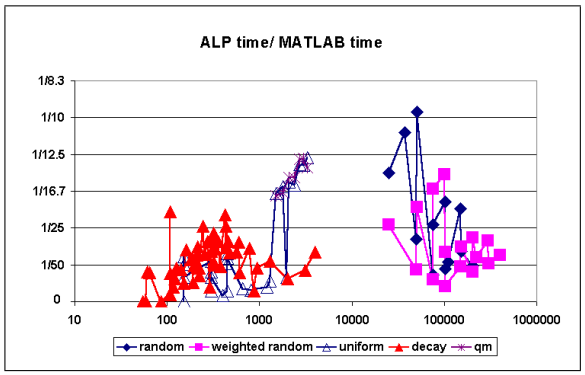


Figure 8: ALP time versus MATLAB time

ALP time (for error factor of 20%) to MATLAB time against the input size. It's clear that ALP is at least 10 times faster than MATLAB for all input sizes in all distributions. Also, it's clear that for problems with size larger than 200000, ALP is at least 30 times faster.

3.7.2 *Approximation error as a function of number of rounds*

Figure 9 describes the ratio of the value of ALP primal solution to the value of the ALP dual solution, as a function of the number of iterations run in the ALP. The figure describes the run of a single typical instance of a problem from the uniform distribution in [14]. The y axis is therefor $\frac{\sum_i p_i/\alpha}{\sum_j A_j v_j / D_b} - 1$ while the x axis is the number of iterations divided by the number of bids in the problem. Note that according to theorem 2.1, the relationship between the number of iterations and the error ratio is quadratic. The figure contains a regression line which tests this on the actual data,

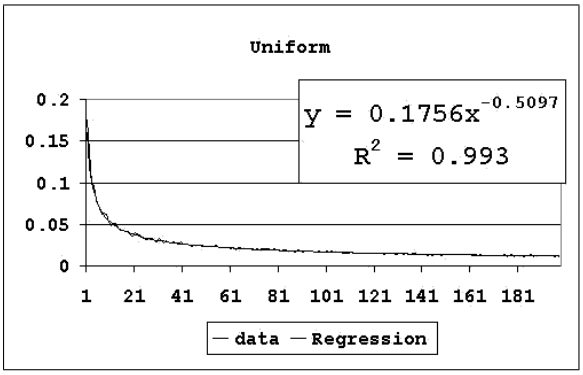


Figure 9: Ratio of primal to dual solutions against (num of iterations)/(num of bids)

and indeed the exponent is very close to expected $\left(-\frac{1}{2}\right)$. We have tested this on the other distributions, with similar results.

3.7.3 *Running Time Asymptotics*

As previously mentioned, we expect that in most cases, for fixed ϵ, the running time of ALP is $O(nm \ln m)$. We've tested this on the problems in [14] by plotting the ratio (ALP time)/$(n \cdot m)$ on Y axis versus the matrix problem size $(n \cdot m)$ on X axis. Figure 10 contains this chart, with X axis on a logarithmic scale. In this scale we would expect to see a linear dependence of (time)/$(n \cdot m)$ in $\log(n \cdot m)$, which despite the noise seems to be the case for each distribution. Note that the running time behavior of Random and Weighted Random distributions are different from the others. This is due to the bid size, which is small in Uniform, Decay, and Quadratic Model, while in Random and Weighted Random it is $O(n)$.

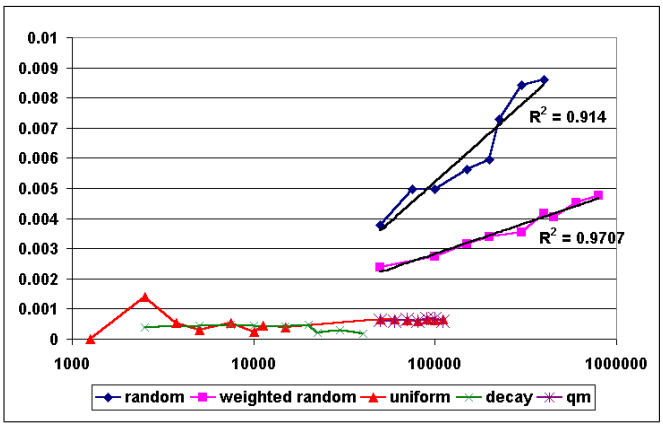
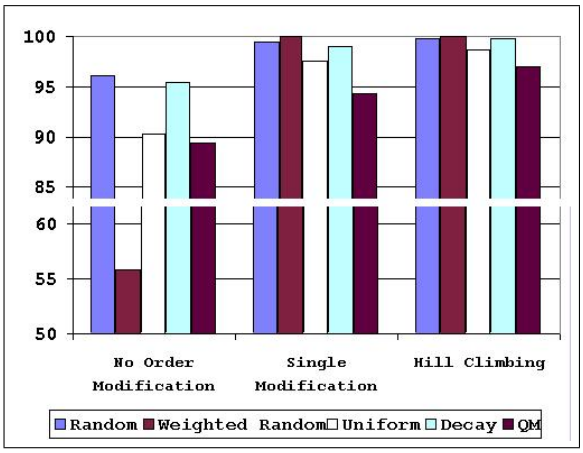


Figure 10: $\frac{ALPtime}{m \cdot n}$ **vs.** $m \cdot n$

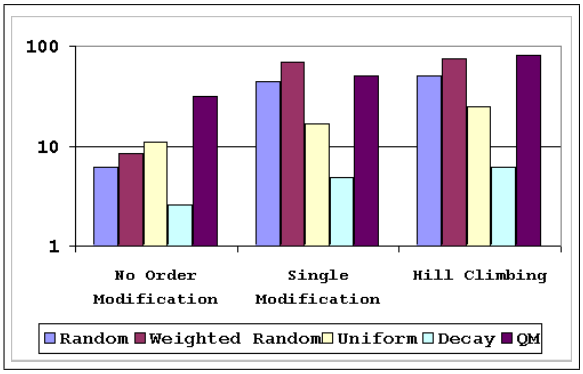
3.8 Hill climbing

3.8.1 *Effect of Local Improvement*

Solution quality: Figure 11(a) describes the solution quality per distribution for each of the allocation decision mechanisms discussed : No Order Modification (regular greedy algorithm), Single Modification (single local improvement), and Hill Climbing. The y axis is simply the accuracy (in percent) of the solution as compared to the optimal integral solution. The figure shows that there is a major improvement from a single greedy algorithm to the case of a single local improvement, and some further improvement for full hill climbing.



(a) Quality Comparison



(b) Time Comparison

Figure 11: Comparison of Greedy, Single Modification, and Hill Climbing

Running time cost: Figure 11(b) gives the average time cost per distribution for each of the allocation decision mechanisms. The y axis is milliseconds on a logarithmic scale. The hill climb must cost at least as much as the single modification, since the last phase of the hill climb (where no further improvements occur) is the same as a single modification run. However, one can deduce from figure 11(b) that most of the improvements occur quite early in the phase from the fact that the overall cost of the hill climb is almost identical to the cost of the single modification.

3.8.2 *Solution Quality as a Function of ALP Approximation Error*

Figure 12 shows the accuracy of 10 instances (2 from each distribution) of the hill climbing phase, when the ALP phase was solved using $\epsilon = 20\%$, 5%, and 1%. There is no clear relationship between ϵ of ALP stage, and the overall accuracy achieved by the hill climb algorithm.

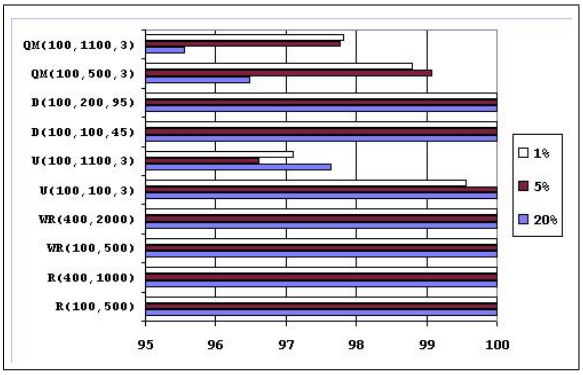


Figure 12: Accuracy as function of ϵ

3.8.3 *Running Time Asymptotics*

As previously mentioned, we expect that in most cases, the running time of the hill climbing phase is $O(nm)$. We've tested this on the problems in [14] by plotting the ratio Hill Climbing time/$n \cdot m$ on Y axis versus $n \cdot m$ on X axis. Figure 13 shows that for all distributions, this ratio fluctuates around a fixed constant, as expected. Running a linear regression on the running time data yields a tiny coefficient with absolute value smaller than 5E-9 for all distributions confirming the expected constant relationship between the hill climb time and $n \cdot m$.

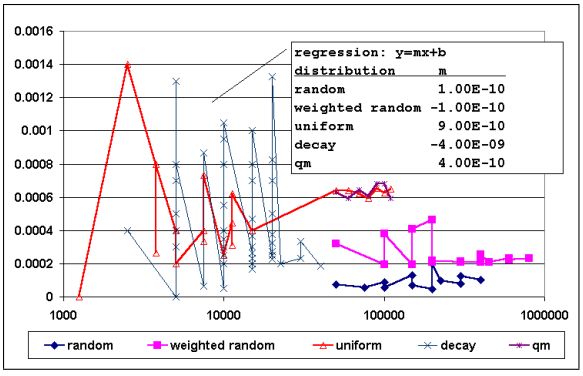


Figure 13: $\frac{HillClimbtime}{n \cdot m}$ against $(n \cdot m)$ on a logarithmic scale

4. CONCLUSION

We have implemented an allocation algorithm for combinatorial auctions that very rapidly obtains allocations that are very close to optimal. We have tested it on many types of problem instances and have found that the solution quality is usually within 1% of optimal, and in no case more than 4% from optimal. The algorithm is fast enough to run in less than a minute on problems with thousands of items and tens of thousands of bids.

While more experimentation with instances coming from real combinatorial auctions is clearly needed, we believe that our algorithm is good enough to solve most practical combinatorial auctions. It seems that the real computational hurdles in combinatorial auctions are thus not the allocation problem, but rather the complexity of determining, expressing, and communicating the bids to the auctioneer.

5. REFERENCES

[1] A. Andersson, M. Tenhunen, and F. Ygge. Integer programming for combinatorial auction winner determination. In *ICMAS*, 2000.

[2] Y. Bartal, J. Byers, and D. Raz. Global Optimization Using Local Information with Applications to Flow Control. In *Proc. of the 38th Ann. IEEE Symp. on Foundations of Computer Science*, 1997.

[3] Y. Fujishima, K. Leyton-Brown, and Y. Shoham. Taming the computational complexity of combinatorial auctions: Optimal and approximate approaches. In *Proceedings of IJCAI'99*, Stockholm, Sweden, July 1999. Morgan Kaufmann.

[4] D. Lehmann, L. I. O'Callaghan, and Y. Shoham. Truth revelation in rapid, approximately efficient combinatorial auctions. In *1st ACM conference on electronic commerce*, 1999.

[5] K. Leyton-Brown, M. Pearson, and Y. Shoham. Towards a Universal Test Suite for Combinatorial Auction Algorithms. *EC*, 2000.

[6] M. Luby and N. Nisan. A Parallel Approximation Algorithm for Positive Linear Programming. In *Proc. Of 25th ACM Symposium on Theory of Computing*, pp. 448-457, 1993.

[7] N. Nisan. Bidding and Allocation in Combinatorial Auctions. In *EC*, 2000.

[8] N. Nisan. The Communication Complexity of Combinatorial Auctions. Available from `http://www.cs.huji.ac.il/~noam/ccaucecon.pdf`, 2001.

[9] S. Plotkin, D. Shmoys, and E. Tardos. Fast Approximation Algorithms for Fractional Packing and Covering Problems. *Stanford Technical Report No. STAN-CS-92-1419*, 1992.

[10] M. H. Rothkhof, A. Pekeč, and R. M. Harstad. Computationally Manageable Combinatorial Auctions. *Management Science*, 44(8):1131–1147, 1998.

[11] T. Sandholm. An algorithm for optimal winner determination in combinatorial auctions. In *IJCAI-99*, 1999.

[12] T. W. Sandholm. Limitations of the vickrey auction in computational multiagent systems. In *Proceedings of the Second International Conference on Multiagent Systems (ICMAS-96)*, pages 299–306, Keihanna Plaza, Kyoto, Japan, December 1996.

[13] M. Tennenholtz. Some tractable combinatorial auctions. In *Proceedings of AAAI*,2000.

[14] R. Vohra and S. de Vries. Combinatorial auctions: A survey, 2000.

APPENDIX

Recall that the main theorem is: *The approximate positive linear programming algorithm produces solutions with values that are within ϵ of each other. Its running time is bounded by $O\left(\frac{R|I|\log(m/\epsilon)}{\epsilon^2}\right)$. Here $R = \tau^*/min_j v_j$, where τ^* is the optimal solution value, and $|I|$ is the total input size.*
To prove this theorem, we'll start with some notations:

Notation: The (un-scaled) primal value at iteration t is denoted by $\phi^t = \sum_i p_i^t$, where p_i^t are the variable values at iteration t.

Notation: The (un-scaled) dual value at iteration t is denoted by $\sigma^t = \sum_j A_j^t v_j$ where A_j^t are the allocation variables at iteration t.

Notation: τ^* is the value of the optimal fractional solution.

PROPOSITION A.1. *For all iterations t, $\overline{p}_1^t..\overline{p}_n^t$ is a primal feasible solution.*

PROOF. Recall that α is the minimum α_j, and

$$\alpha_j = \sum_{i \in S_j} \frac{p_i}{v_j}$$

Obviously all $\overline{p}_i^t \geq 0$, so it remains to prove that $\forall j \ \sum_{i \in S_j} \overline{p}_i \geq v_j$. But

$$\forall j \sum_{i \in S_j} \overline{p}_i = \sum_{i \in S_j} \frac{p_i}{\alpha} \geq \sum_{i \in S_j} \frac{p_i}{\alpha_j} = \sum_{i \in S_j} \frac{p_i}{\sum_{i \in S_j} \frac{p_i}{v_j}} = v_j$$

PROPOSITION A.2. *For all iterations t, $\overline{A}_1^t..\overline{A}_m^t$ is a dual feasible solution.*

PROOF. Obviously all $\overline{A}^t \geq 0$ so it remains to prove that $\forall i \ \ \sum_{j|i \in S_j} \overline{A}_j \leq 1$ (i.e. no item is allocated more than once). Recall that $D_b > 0$ is the maximum D_i and

$$\forall i D_i = \sum_{j|i \in S_j} A_j$$

But

$$\forall i \sum_{j|i \in S_j} \overline{A}_j = \sum_{j|i \in S_j} \frac{A_j}{D_b} \leq \sum_{j|i \in S_j} \frac{A_j}{D_i} = \sum_{j|i \in S_j} \frac{A_j}{\sum_{j|i \in S_j} A_j} = 1$$

COROLLARY A.3. *For all iterations t if $\alpha > 0$ then $\frac{\sigma^t}{D_b} \leq \tau^* \leq \frac{\phi^t}{\alpha}$.*

PROOF. This follows immediately from propositions A.1 and A.2, since any primal feasible solution will have a value greater or equal to the optimal solution, which in turn will have a value greater or equal to any dual feasible solution. □

THEOREM A.4. *For $0.5 > \epsilon > 0$, after $T = O\left(\frac{R \log(m/\epsilon)}{\epsilon^2}\right)$ iterations we have that $\frac{\phi^T}{\alpha} \leq (1+\epsilon)\tau^*$, where $R = \tau^*/min_j v_j$.*

In order prove the theorem, we'll first prove some helpful lemmas.

LEMMA A.5. *For all i we have that $\left(1 - \frac{\epsilon}{2}\right) \sum_{j|i \in S_j} A_j = \sum_{j|i \in S_j} \left(1 - \frac{\delta_i}{2v_j}\right) A_j$.*

PROOF. This is a direct corollary from the choice of δ as $\delta_i = \epsilon \frac{\sum_{j|i \in S_j} A_j}{\sum_{j|i \in S_j} \frac{A_j}{v_j}}$, which implies that $\frac{\delta_i}{2} \sum_{j|i \in S_j} \frac{A_j}{v_j} = \frac{\epsilon}{2} \sum_{j|i \in S_j} A_j$. We thus have

$$\sum_{j|i \in S_j} \frac{A_j \delta_i}{2v_j} = \sum_{j|i \in S_j} A_j - \left(1 - \frac{\epsilon}{2}\right) \sum_{j|i \in S_j} A_j$$

$$\Rightarrow \left(1 - \frac{\epsilon}{2}\right) \sum_{j|i \in S_j} A_j = \sum_{j|i \in S_j} A_j - \sum_{j|i \in S_j} \frac{A_j \delta_i}{2v_j}$$

$$\Rightarrow \left(1 - \frac{\epsilon}{2}\right) \sum_{j|i \in S_j} A_j = \sum_{j|i \in S_j} A_j \left(1 - \frac{\delta_i}{2v_j}\right)$$

□

FACT A.6. $\forall x: \ \ e^{-x} \leq 1 - x + \frac{x^2}{2} = 1 - x\left(1 - \frac{x}{2}\right)$

LEMMA A.7. $\sigma^{t+1} \leq \sigma^t - \delta_b^{t+1}\left(1 - \frac{\epsilon}{2}\right) \sum_{b \in S_j} A_j^t$

PROOF. In iteration $t+1$, the price of chosen item b is increased by δ_b^{t+1}. $\forall j | b \in S_j$ the value A_j is updated and so, using fact A.6, we have that

$$\forall j | b \notin S_j : \ A_j^{t+1} = A_j^t,$$

$$\forall j | b \in S_j : A_j^{t+1} = A_j^t e^{-\frac{\delta^{t+1}}{v_j}} \leq A_j^t \left(1 - \frac{\delta^{t+1}}{v_j}\left(1 - \frac{\delta^{t+1}}{2v_j}\right)\right).$$

But,

$$\sigma^{t+1} = \sum A_j^{t+1} v_j = \sum_{j|b \in S_j} A_j^{t+1} v_j + \sum_{j|b \notin S_j} A_j^{t+1} v_j \leq$$

$$\leq \sum_{j|b \in S_j} A_j^t v_j \left(1 - \frac{\delta^{t+1}}{v_j}\left(1 - \frac{\delta^{t+1}}{2v_j}\right)\right) + \sum_{j|b \notin S_j} A_j^t v_j \leq$$

$$\leq \sigma^t - \sum_{j|b \in S_j} A_j^t \delta^{t+1} \left(1 - \frac{\delta^{t+1}}{2v_j}\right).$$

So, using A.5, we get

$$\sigma^{t+1} \leq \sigma^t - \delta_b^{t+1}\left(1 - \frac{\epsilon}{2}\right) \sum_{b \in S_j} A_j^t.$$

□

LEMMA A.8. $\sigma^{t+1} \leq \sigma^t \left(1 - \frac{\delta^{t+1}}{\tau^*}\left(1 - \frac{\epsilon}{2}\right)\right)$

PROOF. Recall that $\forall i D_i = \sum_{j|i \in S_j} A_j$, so using A.7 we substitute for D_b and get

$$\sigma^{t+1} \leq \sigma^t - \delta_b^{t+1}\left(1 - \frac{\epsilon}{2}\right) D_b \leq \sigma^t \left(1 - \delta_b^{t+1} \frac{D_b}{\sigma^t}\left(1 - \frac{\epsilon}{2}\right)\right)$$

But A.3 implies that $\frac{\sigma^t}{D_b} \leq \tau^*$, and thus

$$\sigma^{t+1} \leq \sigma^t \left(1 - \frac{\delta^{t+1}}{\tau^*}\left(1 - \frac{\epsilon}{2}\right)\right).$$

□

LEMMA A.9. $\alpha \geq -\ln\left(\frac{\sigma^t}{\min_j v_j}\right)$

PROOF.

$$\sigma^t = \sum A_j^t v_j = \sum e^{-\alpha_j} v_j \geq \min_j v_j \sum e^{-\alpha_j} \geq \min_j v_j e^{-\alpha}$$

We thus have that $e^{-\alpha} \leq \frac{\sigma^t}{\min_j v_j}$, and thus
$\alpha \geq -\ln\left(\frac{\sigma^t}{\min_j v_j}\right)$. □

FACT A.10. $\forall x: \ -\ln(1-x) \geq x$

LEMMA A.11. *For all iterations T,*
$\alpha \geq \frac{1}{\tau^*} \sum_{t \in 1..T} \delta^t \left(1 - \frac{\epsilon}{2}\right) - \ln\left(\frac{\sigma^0}{\min_j v_j}\right)$.

PROOF. We know from A.8 that

$$\sigma^{t+1} \leq \sigma^t \left(1 - \frac{\delta^{t+1}}{\tau^*}\left(1 - \frac{\epsilon}{2}\right)\right)$$

So, by induction on t, we have

$$\sigma^{t+1} \leq \sigma^0 \prod_{t=1..T} \left(1 - \frac{\delta^t}{\tau^*}\left(1 - \frac{\epsilon}{2}\right)\right).$$

Using A.9, we get

$$\alpha \geq -\ln\left(\frac{\sigma^0}{\min_j v_j}\right) - \sum_{t=1..T} \ln\left(1 - \frac{\delta^t}{\tau^*}\left(1 - \frac{\epsilon}{2}\right)\right).$$

Using A.10 we get

$$\alpha \geq -\ln\left(\frac{\sigma^0}{\min_j v_j}\right) + \sum_{t=1..T} \left(\frac{\delta^t}{\tau^*}\left(1 - \frac{\epsilon}{2}\right)\right) \geq$$

$$\geq \frac{1}{\tau^*} \sum_{t=1..T} \delta^t \left(1 - \frac{\epsilon}{2}\right) - \ln\left(\frac{\sigma^0}{\min_j v_j}\right).$$

□

LEMMA A.12. *After T iterations, we can bound from above the primal solution of ALP by*

$$\frac{\phi^t}{\alpha} \leq \tau^* \left[\frac{\sum_{t=1..T} \delta^t}{\left(1 - \frac{\epsilon}{2}\right)\sum_{t=1..T} \delta^t - \tau^* \ln\left(\frac{\sigma^0}{\min_j v_j}\right)}\right]$$

PROOF. According to the ALP algorithm, at stage t some p_i grew by δ^t, so

$$\frac{\phi^T}{\alpha} = \frac{\sum_i p_i^T}{\alpha} = \frac{\sum_{t=1..T} \delta^t}{\alpha}$$

Using A.11, we thus get:

$$\frac{\phi^t}{\alpha} \leq \tau^* \left[\frac{\sum_{t=1..T} \delta^t}{\left(1 - \frac{\epsilon}{2}\right)\sum_{t=1..T} \delta^t - \tau^* \ln\left(\frac{\sigma^0}{\min_j v_j}\right)}\right]$$

□

FACT A.13. *For $0 < \epsilon < 0.5$, we have that $\frac{3}{3-2\epsilon} \leq 1 + \epsilon$.*

We are finally ready to complete the proof of A.4.

PROOF. We start with the following claim.

CLAIM A.14. *For $T \geq \frac{6\tau^*}{\min_j v_j \epsilon^2} \ln\left(\frac{\sigma^0}{\min_j v_j}\right)$ we have that $\frac{\phi^T}{\alpha} \leq \tau^*(1+\epsilon)$.*

Before proving the claim, let us see why it proves the theorem. The ratio of the average offer and minimal offer is given by: $\frac{\frac{1}{m}\sum_j v_j}{\min_j v_j}$, and may be assumed to be bounded from above by m/ϵ (since very small bids can be thrown out). At the start of ALP, all allocations are set to 1 so: $\sigma^0 = \sum_j v_j$ and thus $T = \frac{6\tau^*}{\min_j v_j \epsilon^2} \ln\left(\frac{\sigma^0}{\min_j v_j}\right) = O\left(\frac{R \ln(m/\epsilon)}{\epsilon^2}\right)$.

Now for the proof of claim A.14. Using Lemma A.12, we only need to show that

$$\left[\frac{\sum_{t=1..T} \delta^t}{\left(1 - \frac{\epsilon}{2}\right)\sum_{t=1..T} \delta^t - \tau^* \ln\left(\frac{\sigma^0}{\min_j v_j}\right)}\right] \leq (1+\epsilon).$$

Denote: $a = \sum_{t=1..T} \delta^t$, and $b = \tau^* \ln\left(\frac{\sigma^0}{\min_j v_j}\right)$. So we wish to see when $\frac{a}{\left(1-\frac{\epsilon}{2}\right)a - b} \leq (1+\epsilon)$. For $a = \frac{6b}{\epsilon}$, we have

$$\frac{a}{\left(1 - \frac{\epsilon}{2}\right)a - b} = \frac{\frac{6b}{\epsilon}}{\left(1 - \frac{\epsilon}{2}\right)\frac{6b}{\epsilon} - b} = \frac{\frac{6}{\epsilon}}{\frac{6}{\epsilon} - 4} = \frac{3}{3 - 2\epsilon} \leq 1 + \epsilon.$$

Where the last inequality uses fact A.13 and is what we wanted. Now translating back, this means that we get to $1+\epsilon$ when

$$a = \sum_{t=1..T} \delta^t = \frac{6b}{\epsilon} = \frac{6\tau^*}{\epsilon} \ln\left(\frac{\sigma^0}{\min_j v_j}\right)$$

Recall that in the ALP algorithm,

$$\delta_i = \epsilon \frac{\sum_{j|i\in S_j} A_j}{\sum_{j|i\in S_j} \frac{A_j}{v_j}} \geq \epsilon \min_j v_j \frac{\sum_{j|i\in S_j} A_j}{\sum_{j|i\in S_j} A_j} = \epsilon \min_j v_j$$

$$\Rightarrow T\epsilon \min_j v_j \leq \sum_{t=1..T} \delta^t = \frac{6\tau^*}{\epsilon} \ln\left(\frac{\sigma^0}{\min_j v_j}\right)$$

$$\Rightarrow T \leq \frac{6\tau^*}{\min_j v_j \epsilon^2} \ln\left(\frac{\sigma^0}{\min_j v_j}\right).$$

□

To complete the proof of the main theorem we just need to analyze the running time of each iteration.

LEMMA A.15. *Each iteration can be implemented in time $O(|I|)$ where $|I|$ is the input size (i.e. the sum of all bid sizes).*

PROOF. In a straight forward implementation of each step, the running time is dominated by the time needed to update all values of D_i, which takes $O(1)$ operations per each item i in each set S_j. □

Comment: A theoretically faster implementation can be obtained by updating the D_i's as the A_j's are modified, and storing the D values and the A values in heaps.