Restricted Truthful Combinatorial Auction Mechanisms

Andrei Lapets Alex Levin

May 14, 2007

1 Introduction

In work on truthful approximation mechanisms for restricted combinatorial auctions [MN02], Mu'alem and Nisan characterized truthfulness for multi-unit combinatorial auctions with single-minded, single-value bidders. To accomplish this, they defined several useful concepts, including monotonicity and bitonicity of allocation algorithms, critical value payment functions, and alignment of boolean conditions with allocation algorithms. They then defined a small language of combinators for constructing truthful mechanisms, and characterized the conditions under which these combinators could be combined to form truthful allocation mechanisms.

We review the relevant definitions and results in the work done by Mu'alem and Nisan. We then define a language for algorithms which, given a vector of real-value bids, must allocate a single item based on those bids. We use the results in Mu'alem and Nisan to define, with the help of standard programming language design and implementation techniques, algorithms for automatically generating critical value functions from algorithm definitions, as well as for checking whether an algorithm is a monotonic allocation algorithm. We use these algorithms to define a type system for this language which is sound with respect to the space of monotonic allocation algorithms. That is, any algorithm which can be type-checked is necessarily a weakly monotonic allocation algorithm for every bidder. Finally, we provide additional analysis and a few extensions of the definitions and results in Nisan and Mu'alem, which may prove useful for future work in which similar language techniques can be employed in automatically checking properties such as bitonicity.

2 Definitions and Necessary Prerequisites

We restrict ourselves to single-item, single-minded, single-value mechanisms, and review what the results in Mu'alem and Nisan say about this restricted set of combinatorial auctions. We assume there is some fixed number of bidders, and denote by $v_i \in \mathbb{R}^+$ the bid made by bidder i. As usual, let $v_{-i} = \langle v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_n \rangle$.

Definition 1 (Definition 3 from [MN02]). An allocation algorithm A is monotone when for any bidder j and any vector of bids v_{-j}, if v_j leads to an allocation for j, then any $v'_j \geq v_j$ also leads to an allocation for j.

Lemma 2 (Lemma 1 from [MN02]). *For any monotone allocation algorithm, for any v_{-j} there exists a single critical value $\theta_j(v_{-j})$ such that for all $v_j > \theta_j(v_{-j})$, bidding v_j will lead to the item being allocated to bidder j, and for all $v_j < \theta_j(v_{-j})$, bidding v_j will lead to the item not being allocated to bidder j.*

They then have the necessary tools to characterize allocation mechanisms which are truthful.

Theorem 3 (Theorem 1 from [MN02]). *If a mechanism is such that every bidder not allocated an item pays nothing, it is truthful if and only if its allocation algorithm is monotone and its payment scheme is based on critical value.*

3 Language for Monotonic Allocation Algorithms

We define a simple typed functional programming language for describing single item allocation algorithms. The general idea is similar to the one in the work of Nisan and Mu'alem [MN02], in which the authors describe a more coarse language of high-level combinators for constructing truthful mechanisms. The main difference between their language and ours is that our language has a finer collection of constructs (that is, statements) which correspond more closely to the kinds of constructs found in popular programming languages (such as calls to subroutines, loops, and assignment of values to variables).

In our implementation, the user defines allocation algorithms by typing definitions of allocation functions into an interactive environment. All allocation algorithms can expect a finite vector of bids as input, where each bid is denoted v_i for some some agent i.

3.1 Abstract Syntax and Normalization

While the syntax of our language is relatively rich, we use several common syntactic normalization techniques to simplify the input into a more restricted grammar of normal forms. This is done before any operations are performed on the syntax tree, and it simplifies the equational reasoning which is necessary during type checking. We present the abstract syntax which is used before normalization takes place:

$$
\begin{array}{rrcl}
 & i & \in & \mathbb{N} \\
 & x & \in & Vars \\
 & r & \in & \mathbb{R} \uplus \infty \\
\text{primitives} & p & ::= & r \mid v_i \\
\text{expressions} & e & ::= & p \mid \texttt{alloc}_i \mid \texttt{if } c \texttt{ then } e_1 \texttt{ else } e_2 \mid \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \\
\text{conditions} & c & ::= & e_1 \geq e_2 \mid e_1 \leq e_2 \mid e_1 > e_2 \mid e_1 < e_2
\end{array}
$$

Note that "for" loops are not currently included in the abstract syntax definition, as they are currently sugar and are automatically unrolled (which means that the starting and terminating counter values must be constants). Complex conditions can be re-arranged into simpler normal forms. For example, consider an `if` statement whose condition contains another `if` statement as seen below:

```
if (e1 ≥ (if c then e1'' else e2'')) then
    e1'
else
    e2'
```

It is easy to transform this expression into the following, equivalent expression:

```
let b1 = e1' in
let b2 = e2' in
if c then
    (if (e1 ≥ e1'') then b1 else b2)
else
    (if (e1 ≥ e2'') then b1 else b2)
```

Effectively, we push the c inside the `if` statement in the top-level condition up the expression tree. The purpose of b_1 and b_2 is to prevent duplication of the entire body for each of the two branches in the original `if` statement. In practice, the increase in size of the expression tree from this transformation is not problematic, though it obviously can make the expression tree more difficult to read. Variants of the above transformation can be applied repeatedly to code which contains `if` statements with complex conditions to simplify all conditions to the form $p_1 \geq p_2$ where p_i are primitives as defined by the abstract syntax. We now present below the abstract syntax for the normal form, along with the abstract syntax for types and critical values.

$$
\begin{array}{rrcl}
 & i & \in & \mathbb{N} \\
 & x & \in & Vars \\
 & r & \in & \mathbb{R} \uplus \infty \\
\text{primitives} & p & ::= & r \mid v_i \\
\text{expressions} & e & ::= & p \mid \texttt{alloc}_i \mid \texttt{if } c \texttt{ then } e_1 \texttt{ else } e_2 \mid \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \\
\text{conditions} & c & ::= & p_1 \geq p_2 \\
\text{critical values} & \theta & ::= & e \\
 & \psi & ::= & e \mid \overline{e} \mid \bullet \\
\text{types} & \tau & ::= & \texttt{Allocation}\langle \theta_1, \ldots, \theta_n \rangle \mid \texttt{Value} \mid \texttt{Condition}\langle \psi_1, \ldots, \psi_n \rangle
\end{array}
$$

The type checker and interpreter need only concern themselves with syntax trees which are instances of the above grammar. Any critical value θ is simply an expression, and a type can either denote an allocation algorithms with a vector of critical value functions, a value, or a condition expression with specialized critical value functions. These specialized critical value functions ψ will be explained in the type inference section below.

3.2 Type Inference Rules

We now provide the inductive inference rules for the type system. By convention, we introduce a variable environment Γ which maps variables to expressions. The type inference algorithm simply applies these rules inductively to a normalized expression tree, starting from the leaves and going up. Base case rules can always be applied to expression tree leaves unconditionally, while inductive rules can only be applied if the premises above the line are satisfied. If at any point in this process, none of the rules can be applied, the type inference algorithm returns an error.

$$
\textsc{Alloc}\ \frac{}{\Gamma \vdash \texttt{alloc}_i : \texttt{Allocation}\langle \infty, \ldots, \infty, \theta_i = 0, \infty, \ldots, \infty \rangle}
$$

$$
\textsc{Value}\ \frac{}{\Gamma \vdash p : \texttt{Value}} \qquad \textsc{Var}\ \frac{}{\Gamma, x : \tau \vdash x : \tau} \qquad \textsc{Let}\ \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma,\ x : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 : \tau_2}
$$

$$
\textsc{If-Value}\ \frac{\Gamma \vdash e_1 : \texttt{Value} \quad \Gamma \vdash e_2 : \texttt{Value}}{\Gamma \vdash \texttt{if } c \texttt{ then } e_1 \texttt{ else } e_2 : \texttt{Value}}
$$

$$\textsc{If-Alloc}\ \frac{\begin{array}{c}\forall i,\ \theta_i'' = U(\psi_i, \theta_i, \theta_i') \\ \Gamma \vdash c : \texttt{Condition}\langle \psi_1, \ldots, \psi_n \rangle \\ \Gamma \vdash e_1 : \texttt{Allocation}\langle \theta_1, \ldots, \theta_n \rangle \\ \Gamma \vdash e_2 : \texttt{Allocation}\langle \theta_1', \ldots, \theta_n' \rangle\end{array}}{\Gamma \vdash \texttt{if}\ \ c\ \ \texttt{then}\ \ e_1\ \ \texttt{else}\ \ e_2 : \texttt{Allocation}\langle \theta_1'', \ldots, \theta_n'' \rangle}$$

$$\textsc{Cond-VV}\ \frac{}{\Gamma \vdash v_i \geq v_j : \texttt{Condition}\langle \bullet, \ldots, \bullet, \psi_i = v_j, \bullet, \ldots, \bullet, \psi_j = \overline{v_i}, \bullet, \ldots, \bullet \rangle}$$

$$\textsc{Cond-VR}\ \frac{}{\Gamma \vdash v_i \geq r : \texttt{Condition}\langle \bullet, \ldots, \bullet, \psi_i = r, \bullet, \ldots, \bullet \rangle}$$

$$\textsc{Cond-RV}\ \frac{}{\Gamma \vdash r \geq v_j : \texttt{Condition}\langle \bullet, \ldots, \bullet, \psi_j = \overline{r}, \bullet, \ldots, \bullet \rangle}$$

The base cases, the `let` rule, and the `if` rule for values are all straightforward. Note the types in the three COND rules. The ψ function for any bidder k whose bid does not appear in the condition is a placeholder $\bullet$, which indicates that this condition does not depend on v_k. For any bidder i that can turn the condition from false to true by raising its bid (for example, if bidder i raises v_i in the COND-VV rule above v_j, the condition becomes true), the ψ function is the threshold point for v_i at which the condition becomes true. Similarly, for any bidder j that can turn the condition from true to false by raising its bid (for example, if bidder j raises v_j in the COND-VV), the ψ function is the threshold point for v_j at which the condition becomes false, denoted by $\overline{v_j}$.

The most interesting rule is the IF-ALLOC rule. Both branches of the `if` statement must be allocation mechanisms themselves, with associated vectors of critical value functions. However, an additional constraint must be met, and it is encapsulated in the unification function U, which checks for alignment between the condition and the two branch allocation algorithms. If the check succeeds, U produces a new critical value function for each bidder. Note that U can be applied to each bidder's functions individually, making its implementation much more feasible. Below, we provide the algorithm for U:

Algorithm: $U(\psi_i, \theta_i, \theta_i') :=$
  if $(\theta_i \geq \psi_i) \wedge (\theta_i' \geq \psi_i)$ **then**
    return (θ_i)
  else if $(\theta_i < \psi_i)$ **then**
    return $(\texttt{if}\ \ (\psi_i \geq \theta_i')\ \ \texttt{then}\ \ \theta_i'\ \ \texttt{else}\ \ \psi_i)$
  else
    we cannot establish either possibility, fail
  end if

By symmetry of the `if` statement, we can define $U(\overline{\psi_i}, \theta_i, \theta_i') := U(\psi_i, \theta_i', \theta_i)$.

We must explicitly perform the checks in both the first and second cases in the algorithm, as it is possible that one direction will be proven by our implementation of the ordering relation on critical value functions, while the other direction will not. If the critical value functions are constant, both checks are trivial. However, if they are arbitrary expressions, this requires additional machinery. Fortunately, we can address this problem by using a data structure for representing very simple congruence closures corresponding to constraints on bids. Data structures for congruence closures are used in many settings, such as in the *Simplify* theorem prover for program checking [DNS03]. For an example of how this might be done, consider the condition

$$c = (\texttt{if}\ \ (v_1 \geq v_2)\ \ \texttt{then}\ \ v_1\ \ \texttt{else}\ \ v_2) \geq v_1.$$

This condition is certainly always true, as the left-hand side takes the maximum of the two values. We can automatically verify this by generating a context of constraints for each branch of the `if` statement. First, we note that the left branch will only be evaluated if $v_1 \geq v_2$, so we add $v_1 \geq v_2$ to a context and try the left branch of the `if`, checking whether $v_1 \geq v_1$ under that context, which is trivially the case. Now, we add $\neg(v_1 \geq v_2) = v_2 > v_1$ to the context and try the right branch, checking whether $v_2 \geq v_1$. We know that $v_2 > v_1$ from our context, so clearly $v_2 \geq v_1$. We have show that c is true regardless of which branch of the `if` is evaluated, so we have thus shown that c must be true.

Lemma 4. *If ψ_i, θ_i, and θ_i' are independent of the bid v_i, and $U(\psi_i, \theta_i, \theta_i')$ succeeds, then the result it returns is also independent of the bid v_i.*

Proof. This is immediate from the fact that the possible return values are θ_i and $\texttt{if}\ \ (\psi_i \geq \theta_i')\ \ \texttt{then}\ \ \theta_i'\ \ \texttt{else}\ \ \psi_i$. □

Thus, if the type inference algorithm succeeds for some expression tree, U will have built up the bidders' critical value functions for that allocation algorithm.

4 Proof of Soundness

It remains to show that the inference rules defined for our language are sound, where soundness is defined with respect to the space of truthful allocation algorithms. We will then show that we have a type system which allows for automatic inductive verification of the truthfulness of any well-typed program.

Theorem 5 (Soundness). *Any expression tree constructed according to the above inference rules (that is, any well-typed expression tree) whose type is $\tau = \texttt{Allocation}\langle\theta_1, \ldots, \theta_n\rangle$ represents a monotonic allocation mechanism when the price function is defined by the critical value functions in τ.*

Proof. We proceed by induction over the structure of the type derivation. Notice that for the base case (ALLOC), the critical value function is defined to be zero for the bidder to whom the item is always allocated, and infinity for all other bidders. This mechanism satisfies the weak monotonicity condition, and truthful bidding is a weakly dominant strategy for all bidders.

The (VALUE) and (IF-VALUE) cases are irrelevant, as we are assuming that the expression's type indicates that it is an allocation. The (LET) and (VAR) cases are trivial as well,

Only the (IF-ALLOC) case remains. We know from our inductive hypothesis that e_1 and e_2 are monotonic and that their type vectors hold the critical value functions for the bidders. If $U(\psi_i, \theta_i, \theta_i')$ succeeds for some i, we know that one of two possibilities must hold.

One possibility is that $\theta_i \geq \psi_i$, in which case it must also be that $\theta_i' \geq \psi_i$. This latter fact makes it clear that bidder i will not get the item for any $v_i < \psi_i$. The fact that $\theta_i \geq \psi_i$ means that i will not get the item for any $\psi_i \leq v_i < \theta_i$. However, for $v_i \geq \theta_i \geq \psi_i$, bidder i will be allocated the item. Thus, θ_i, the result of U, is indeed the critical value in this case.

The other possibility is that $\theta_i < \psi_i$, which means the bidder would always be allocated the item for any $v_i \geq \psi_i$. The bidder would also be allocated the item for any $\theta_i' \leq v_i < \psi_i$, so the critical value is the minimum of θ_i' and ψ_i, which is exactly what U returns.

Since e_1 and e_2 are both monotonic, `if` c `then` e_1 `else` e_2) must also be monotonic in both of these cases. Since the critical values returned by U are indeed the algorithm's critical values, we again know by Theorem 3 that the mechanism this expression represents is truthful. □

It is important to observe that the type inference rules enforce monotonicity locally, which is restrictive, and means that the type system is not complete. To see this, note that we require that every subexpression in the inductively constructed tree represents an allocation algorithm which is monotonic and has a critical value function. Now consider the following counterexample. Say b_2 is an algorithm which allocates to bidder 1 when $5 < v_1 < 10$. Now, say $b_1 = \texttt{alloc}_1$ and c is a condition such that $\psi_1 = 7$. It is clear that (`if` c `then` b_1 `else` b_2) is a monotonic allocation algorithm, and that the critical value function for bidder 1 is 5. However, b_2 would obviously be rejected by the type inference algorithm, and thus, so would (`if` c `then` b_1 `else` b_2).

One approach that would allow more truthful mechanisms to type check might involve first generating a more complex set of critical *intervals* in the value space, then checking for monotonicity violations from the top down. Such an approach would require a more rich type system, however, as it would be necessary to represent and reason about collections of functions defining intervals. However, similar techniques could be applied to this more complex case.

5 Further Directions

Further work is needed to determine whether it is possible to provide enough information in the type vectors to determine whether or not an expression represents an allocation algorithm which is bitonic with respect to some function. Nonetheless, we consider some extensions to the numerous constructions of truthful mechanisms that the article [MN02] considers, with an eye toward constructing expressions that can automatically be verified once more is understood about how bitonicity checking.

5.1 Taking Maxima of Mechanisms

The authors of [MN02] introduce a MAX operator, whereby they run two truthful algorithms on a given input, and choose the outcome of the one that would have given the higher social welfare. If A_1 and A_2 are two allocation rules, $\mathrm{MAX}(A_1, A_2)$ will not necessarily be truthful. However, in [MN02], the authors determine a condition on A_1 and A_2 that will make sure that truthfulness holds. This condition is bitonicity.

Definition 6 (Definition 5 from [MN02]). Let A be a monotone allocation rule. We say that A is *bitonic* if for every bidder j and any v_{-j} (i.e. the values of the other bidders) the welfare $w(A(v))$ is a non-increasing function of v_j for $v_j < \theta_j$ and a non-decreasing function of v_j for $v_j \geq \theta_j$. Here θ_j is the critical value for the jth bidder, which depends on v_{-j} (we do not explicitly write this dependence for notational simplicity).

Notice that bitonicity is defined with respect to the social welfare function w. We extend the definition to an arbitrary function $f : D \to \mathbb{R}$, where D is the set of decisions. The function f can be the revenue generated by the mechanism, or the number of bidders that get allocated.

Definition 7. Let A be a monotone allocation rule, and $f : D \to \mathbb{R}$ be a real-valued function on the set of decisions. We say that A is *bitonic with respect to* f (or f-bitonic) if for every bidder j and any v_{-j}, we have that $f(A(v))$ is a non-increasing function of v_j for $v_j < \theta_j$ and a non-decreasing function of v_j for $v_j \geq \theta_j$.

We can similarly define a MAX operator with respect to an arbitrary function f.

Definition 8. If A_1 and A_2 are two allocation rules, then $\mathrm{MAX}_f(A_1, A_2)$, the maximum of A_1 and A_2 with respect to f, allocates as follows. On bid vector v, the mechanism simulates A_1 and A_2 and takes outcome $A_i(v)$ which maximizes $f(A_i(v))$.

Remark 9. *Throughout this section, we implicitly conflate the bidders' values v and their bids b. However, this poses no problem, since we will be constructing truthful mechanisms, where a dominant strategy is for the bidders to bid their values.*

Notice that if an allocation rule A is bitonic with respect to functions f and g, then it is bitonic with respect to fg.

The fundamental observation is that if A_1 and A_2 are both f-bitonic allocation rules, then $\text{MAX}_f(A_1, A_2)$ is a monotonic and bitonic algorithm. This allows us to inductively construct truthful mechanisms, and in more cases than the Mu'alem and Nisan.

Theorem 3 of [MN02] is basically the statement of the fact for $f = w$ (the social welfare function). However, the proof therein does not anywhere use the fact that we are maximizing with respect to the social welfare function. Instead, bitonicity is used to get a handle on when the mechanism will allocate via A_1 or A_2. The arguments about critical points (hence monotonicity) and bitonicity thus go through unchanged for an arbitrary function f.

We give some examples to illustrate these kinds of constructions.

Example 10. Suppose that we are allocating m identical items and bidder i wants m_i items. Suppose also that the mechanism designer knows the m_i (this is a special case of the *known single-minded* setting).

Let r be any monotonic ranking of the values, e.g. ranking by value v_i or by density $\bar{v}_i = v_i/m_i$. In other words, by raising the value of v_i, the bidder does no worse in terms of ranking. Suppose that $v_1, v_2, \cdots, v_n$ are all ordered according to r. For $s \leq n$, let $G_r(s)$ be the mechanism that allocates to bidders $1, 2, \ldots, s$ their desired bundle at the critical price. So, for the density ranking, all bidders would have to pay a price per item equal to v_{s+1}/m_{s+1}. We do not worry about feasibility at this point; it does not matter that some of the $G_r(s)$ might be allocating more than m items. This issue will be handled by the feasibility function.

It is easy to see that each $G_r(s)$ is monotonic. If one of the first s players raises his bid, he will remain among the top s, since the ranking rule is monotonic. Let f be the feasibility function, which is 1 if the allocation is feasible and 0 otherwise. Let g be the function that counts the number of bidders that have been allocated. We claim that each $G_r(s)$ is bitonic with respect to f and g. If a bidder who loses in $G_r(s)$ raises his bid to another losing bid, this in no way affects who gets allocated by the algorithm. Therefore, the feasibility and number of winners stays the same. Furthermore, a winning bidder who raises his bid also does not affect the two functions.

Therefore, $\text{MAX}_{fg}\{G(s) : 1 \leq s \leq n\}$ is a truthful (and fg bitonic) allocation algorithm. A moment's thought shows that it allocates to the same bidders as the greedy allocation algorithm in this setting.

We will return to the greedy algorithm for the known single-minded setting for multiple types of goods later on.

Example 11. We are in the known single-minded case with different goods. There can be multiple copies of each good. Suppose that bidder ith desired bundle is S_i. We consider the Exhaustive search algorithm for at most k bids, Exst-k. This algorithm basically looks for a set $J \subset [n] := \{1, \ldots, n\}$ of bidders to satisfy such that the allocation is feasible and the total social welfare is as large as possible.

Let $G(J)$ be the algorithm that allocates to bidders in J and does not allocate to the others (again, the feasibility issue will be handled by including a feasibility function f). Clearly

$G(J)$ is a truthful mechanism. (Notice that the critical value for bidders in J is 0 while the critical value for bidders not in J is ∞.) Furthermore, it is immediate that A_J is bitonic with respect to f (feasibility) and w (social welfare).

Therefore we immediately see that the Exsc-k algorithm is truthful, as we can recast it in the form $\text{MAX}_{fw}\{G(J) : J \subset [n], |J| \leq k\}$.

Example 12. Let us consider the digital goods setting. Here, we have an unlimited quantity of a good that we can distribute to those bidders who are willing to pay enough.

For example, consider the mechanism ProfitExtract_R, which we have encountered in [SW06]. Suppose that there are n bidders, with bid values $v_1, \ldots, v_n$, and say that R is independent of all the bids. Let z_i represent the ith highest bid. Consider the allocation rule A_i, which allocates to the i highest bidders if and only if $iz_i \geq R$; if it allocates, it charges a price of R/i to each winning bidder. Then,

$$\text{ProfitExtract}_R = \text{MAX}_N\{A_i : i \in [n]\} \tag{1}$$

where N is the number of items that are actually allocated. Clearly each A_i is monotonic. Furthermore, it is bitonic with respect to N. Indeed, if a losing bidder raises his value to another losing bid, he will not get allocated, and the number of items allocated will stay the same. A bidder who does get allocated can raise his bid, and still be allocated, hence the number of items allocated will stay the same.

Let us slightly modify this situation. Suppose we can still allocate an unlimited number of goods, but now, bidder i wants to have m_i units and his value for having them is v_i. He has no additional value for being allocated more than m_i units, and no value at all for fewer than m_i units. Furthermore, suppose that m_i is known to the bidder. Let $\bar{v}_i = v_i/m_i$ be the price per unit he is willing to pay. The profit extraction algorithm is modified in a natural way. We rank the bids by price per unit, i.e. we reorder such that $\bar{v}_1 \geq \bar{v}_2 \geq \cdots \geq \bar{v}_n$. To extract a profit of R, we seek the largest number of bidders such that, if we allocate s items, each bidder's value per item is at least R/s. Let A_i be the algorithm that, as before, tries to allocate to the i highest bidders at a price per item equal to $R/s(i)$, where $s(i)$ is the total number of items these i highest bidders desire. It allocates if all those bidders have price per item above $R/s(i)$, and charges $R/s(i)$.

Again, it is clear that each A_i is monotonic, and N-bitonic as well. This means that $\text{ProfitExtract}_R = \text{MAX}\{A_i : i \in [n]\}$ is monotonic.

Example 13. We now return to the general greedy algorithm in the known single-minded case (namely, there could be numerous different items, and numerous units of each). This algorithm proceeds as follows. It orders the bids by some monotone ranking r (so, without loss of generality, we can say that $r(v_1) \leq r(v_2) \leq \cdots \leq r(v_n)$ (having $r(v_i)$ be lower is better). Then, proceeding to each bidder in turn, it sees whether it can allocate its desired bundle (given the goods that are still unallocated), and allocates if it can.

Let J be a subset of bidders, and let $G(J)$ be the algorithm that allocates to all the bidders in J their desired bundles (again, we do not worry about feasibility at this point).

Note that this algorithm tries to allocated to the most highly-ranked bidders it can. (Again, it is confusing, but note that bidder ranked first is the most highly ranked). This leads us to put a lexicographical order on the J's. Let $J, J' \subset [n]$, and let $i_1 < i_2 < \cdots < i_r$ be the *rankings* of the bidders in J, and analogously for $j_1 < j_2 < \cdots < j_{r'}$ and J'. Then we say that J is ranked better than J' and write $J \succ J'$ in the following case. Suppose that $i_1 = j_1, i_2 = j_2, \ldots, i_\nu = j_\nu$. If $\nu < r$, we say that $J \succ J'$ if $i_{\nu+1} < j_{\nu+1}$. Or, when $\nu = r$, we say $J \succ J'$ if $r > r'$ (in this case, J simply represents an allocation to more bidders).

Consider a particular $G(J)$. A losing bidder who raises his bid cannot affect the outcome. Furthermore, it is not hard to see that if a winning bidder raises his bid, we will get an allocation J' such that $J' \succeq J$. Therefore, if f is the feasibility function, and g is any function that is monotonically increasing with respect to $\prec$ (i.e. the ranking of the winning bidders), we see that $\text{MAX}_{fg}\{G(J) : J \subset [n]\}$ allocates just as the greedy algorithm, and is monotonic.

We have therefore seen that taking a maximum of allocation algorithms with respect to functions other than that social welfare can be a useful technique for producing truthful functions. An appealing goal would be to use such a construct to produce mechanisms that have good revenue properties. Namely, one would like to take the maximum with respect to profit of several mechanisms. For example, one would like to take the profit-maximum over the partial greedy allocation mechanisms $G_r(s)$ of Example 10.

Unfortunately, bitonicity with respect to revenue is a tricky property to achieve. Indeed, if we try to take the profit-maximum of the $G_r(s)$, then, by the pricing structure, by raising the highest losing bid to something that still loses, one would increase the revenue of the mechanism, which violates profit-bitonicity. And indeed, the profit-maximum of the $G_r(s)$ is not truthful. For example, suppose we have 2 goods to allocate and 3 bidders, with $v_1 = 10, v_2 = 8$, and $v_3 = 3$ as their values. Each bidder wants just one unit. Allocating just to bidder 1 would give the mechanism a revenue of 8, while allocating to bidders 1 and 2 would assure a revenue of 6. So, the mechanism would allocate just to bidder 1. However, suppose bidder 2 misreports his value to be $v_2' = 5$. Then, allocating just to bidder 1 generates a profit of 5 for the mechanism, while allocating to bidders 1 and 2 gives a profit of 6. Hence, agent 2 will now be allocated for a price of 3, which is lower than its value. Hence, the misreport gives agent 2 a strictly better outcome.

The trickiness of achieving profit bitonicity partially helps explain why the authors of [SW06] have to do something more complicated to achieve profit competitiveness, namely splitting up the bids and profit extraction. Notice that the price charged by ProfitExtract_R is independent of the losing bids.

5.2 Revenue-Competitive Digital Goods Auctions with Multiple Desired Items

Let us consider the case where we have an unlimited quantity of a given good to allocate, and we have n bidders. Suppose that bidder i wants m_i copies of the good.

For a bid vector $\boldsymbol{b}$, we denote by $\mathcal{F}(\boldsymbol{b})$ the optimal omniscient single-price revenue, gotten by allocated the desired bundle to a subset of the bidders at a price per item equal to the lowest price per item among the winning bids. We let $\mathcal{F}^{(2)}(\boldsymbol{b})$ be the optimal omniscient second-price revenue among allocations where at least two bidders win.

The paper [SW06] proves that in general one cannot hope to compete with $\mathcal{F}$ (for adversarial bid inputs), but one can compete with $\mathcal{F}^{(2)}$, provided one is willing to use a randomized mechanism. One such randomized mechanism is the RSPE mechanism, which randomly breaks up the bid vector $\boldsymbol{b}$ into two subsets $\boldsymbol{b}'$ and $\boldsymbol{b}''$, and runs $\textsc{ProfitExtract}_{\mathcal{F}(\boldsymbol{b}')}(\boldsymbol{b}'')$ and $\textsc{ProfitExtract}_{\mathcal{F}(\boldsymbol{b}'')}(\boldsymbol{b}')$.

We run the same algorithm in our case, first assigning the bidders into one of two groups and trying to extract profit from each of the groups (the profit extraction function should be the one appropriate for this setting). Because $\mathcal{F}(\boldsymbol{b}')$ is independent of $\boldsymbol{b}''$ and vice-versa, Example 12 shows us that this algorithm is truthful.

We now attempt to analyze the competitiveness of this mechanism. Notice that if $\mathcal{F}(\boldsymbol{b}') \leq \mathcal{F}(\boldsymbol{b}'')$, then we will be able to extract a profit of $\mathcal{F}(\boldsymbol{b}')$ from $\boldsymbol{b}''$ and vice-versa. Therefore, at the very least, we will be able to attain a revenue of $\min(\mathcal{F}(\boldsymbol{b}'), \mathcal{F}(\boldsymbol{b}''))$.

Suppose now that in the omniscient mechanism $\mathcal{F}^{(2)}$, we have that r bidders winning, and s items get allocated. Suppose, after reordering, that the winning bidders are bidders 1 through r, and that the minimum price per bid among them is p. Then we have $\mathcal{F}^{(2)}(\boldsymbol{b}) = ps$. Assume now that we have broken $\boldsymbol{b}$ up into two bid vectors $\boldsymbol{b}'$ and $\boldsymbol{b}''$. Say that there are r' of the winning bidders for $\mathcal{F}^{(2)}$ among $\boldsymbol{b}'$, representing s' items allocated, and define r'' and s'' similarly. We then have that $\mathcal{F}(\boldsymbol{b}') \geq ps'$ and $\mathcal{F}(\boldsymbol{b}'') \geq ps''$. Then,

$$\frac{\min(\mathcal{F}(\boldsymbol{b}'), \mathcal{F}(\boldsymbol{b}''))}{\mathcal{F}^{(2)}(\boldsymbol{b})} \leq \frac{\min(ps', ps'')}{ps} = \frac{\min(s', s'')}{s}.$$

Notice that the dependence on p has disappeared. It follows that the inverse of the competitive ratio is equal to

$$\frac{\mathbb{E}[R]}{\mathcal{F}^{(2)}(\boldsymbol{b})} = \frac{1}{s} \sum_{i=1}^{s-1} \min(i, s-i) N_{m_1,\ldots,m_r}(i) 2^{-r},$$

where $N_{m_1,\ldots,m_r}$ represents the number of ways of obtaining i as a sum of a subcollection of the $m_1, \ldots, m_r$.

We have not been able to analyze the competitiveness in general, because the $N_{m_1,\ldots,m_r}$ function may be difficult to bound. We expect, however, that it will be bad. Indeed, recall

that we have to randomize this algorithm because if an adversary knew what $\boldsymbol{b}'$ and $\boldsymbol{b}''$ were, it could construct a bid vector with very high bids in $\boldsymbol{b}'$ and very low bids in $\boldsymbol{b}''$ or vice-versa. From here, it is easy to see that the adversary could make the algorithm compete arbitrarily badly with $\mathcal{F}^{(2)}$.

In the multiple unit case, the adversary can simulate this situation by having all bidders but one desire only one item with low values. and one bidder having a very high value for many units. Then, having this bidder in $\boldsymbol{b}'$ will resemble the situation of many high bidders there, and low bidders in $\boldsymbol{b}''$.

Our job is therefore to quantify just how badly the algorithm competes as a function of the m_i. Again, for arbitrary m_i, it may be tricky to bound this, and we have not been able to achieve a general result. Instead, we focus on the case where the bidders all desire either 1 or μ units. Suppose that α of the bidders who desire μ items and β of the bidders who desire 1 item win. We then have

$$N_{m_1,\ldots,m_r}(i) = \sum_{j=0}^{\alpha} \binom{\alpha}{j} \binom{\beta}{i-\mu j}.$$

Using this formula, we are able to numerically compute the maximum competitive ratio (i.e. worst competitiveness) for a particular value of μ over a range of α and β. In our tests, a minimum was achieved at $\alpha = \beta = 1$. This makes sense. Having $\alpha > 1$ will make it possible that players desiring μ items will be split up by the random partitioning, counteracting the effect of having some players desiring many items, allowing us to extract more revenue. Furthermore, the maximum competitive ratio $2(\mu+1)$ (in particular, for $\mu = 1$, we obtain 4, as in the paper).

It should be possible to actually get a proof for this, rather than an argument by numeric simulation (regardless of how convincing the simulation is). This should not be difficult.

5.3 Potential Areas of Future Research

As we have seen, the MAX constructs provide promising ways of constructing truthful allocation rules from building blocks. Because bitonicity is such a crucial component of the construction, it would be very desirable to have automatic proofs of bitonicity, at least in the "easy" cases, where the argument basically says that changing a bid cannot affect the allocation. The ranking-based algorithms, like the $G(J)$ seem amenable to automation.

Of course, in order to fully exploit the construct, it would be necessary to have a way of automatically turning an allocation rule into an expression maximizing building block allocation rules. This seems extremely difficult. Indeed, even going the other way, i.e. from a MAX construction to an algorithm is tricky. While this can be done naively, it loses a lot in terms of efficiency. For example, in the case of the greedy algorithms, if we unwind the MAX construction naively, we end up having to do a great deal of unnecessary recomputation.

Nevertheless, as we have seen, disparate algorithms can have the same building blocks when we consider them as maximizations. This is evident in the examples of Exsc-k and the general greedy algorithm. The maximizations are just over different functions. Therefore, if we are somehow able to automate proofs of bitonicity in several cases over a large class of functions, proofs of truthfulness of distinct algorithms would follow.

References

[DNS03] D. Detlefs, G. Nelson, and J. Saxe. Simplify: A theorem prover for program checking, 2003.

[MN02] A. Mu'alem and N. Nisan. Truthful approximation mechanisms for restricted combinatorial auctions, 2002. In AAAI (poster), 2002. also presented at Dagstuhl workshop on Electronic Market Design.

[SW06] Andrew Goldberg; Anna Karlin; Mike Saks; and Andrew Wright. Competitive auctions. *Games and Economic Behavior*, 2006.