On Polynomial-Time Preference Elicitation with Value Queries

Martin A. Zinkevich
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
maz@cs.cmu.edu

Avrim Blum
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213
avrim@cs.cmu.edu

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

ABSTRACT

Preference elicitation — the process of asking queries to determine parties' preferences — is a key part of many problems in electronic commerce. For example, a shopping agent needs to know a user's preferences in order to correctly act on her behalf, and preference elicitation can help an auctioneer in a combinatorial auction determine how to best allocate a given set of items to a given set of bidders. Unfortunately, in the worst case, preference elicitation can require an exponential number of queries even to determine an approximately optimal allocation. In this paper we study natural special cases of preferences for which elicitation can be done in polynomial time via value queries. The cases we consider all have the property that the preferences (or approximations to them) can be described in a polynomial number of bits, but the issue here is whether they can be elicited using the natural (limited) language of value queries. We make a connection to computational learning theory where the similar problem of *exact learning with membership queries* has a long history. In particular, we consider preferences that can be written as *read-once formulas* over a set of gates motivated by a shopping application, as well as a class of preferences we call *Toolbox DNF*, motivated by a type of combinatorial auction. We show that in each case, preference elicitation can be done in polynomial time. We also consider the computational problem of allocating items given the parties' preferences, and show that in certain cases it can be done in polynomial time and in other cases it is NP-complete. Given two bidders with Toolbox-DNF preferences, we show that allocation can be solved via network flow. If parties have read-once formula preferences, then allocation is NP-hard even with just two bidders, but if one of the two parties is additive (e.g., a shopping agent purchasing items individually and then bundling them to give to the user), the allocation problem is solvable in polynomial time.

EC'03, June 9–12, 2003, San Diego, California, USA.

Categories and Subject Descriptors

F.2.0 [**Analysis of Algorithms and Problem Complexity**]: General; J.4 [**Social and Behavioral Sciences**]: Economics; I.2.6 [**Artificial Intelligence**]: Learning

General Terms

Algorithms Economics Theory

Keywords

combinatorial auctions, preference elicitation, learning

1. INTRODUCTION

In most auctions where multiple (say, n) distinct items are being sold, a bidder's valuation for the items is not additive. Rather, the bidder's preferences generally exhibit *complementarity* (a bundle of items is worth more than the sum of its parts—e.g., a flight to Hawaii and a hotel room in Hawaii) and *substitutability* (a bundle is worth less than the sum of its parts—e.g., a flight to Hawaii and a flight to the Bahamas on the same day). *Combinatorial auctions*, where agents can submit bids on *bundles* of items, are economically efficient auction mechanisms for this setting. The computational problem of determining the winners, given the bids, is a hard optimization problem that has recently received significant research attention.

An equally important problem, which has received less attention, is that of obtaining enough preference information from the bidders so that there is a basis for allocating the items (usually the objective is to maximize the sum of the bidders' valuation functions). There are $2^n - 1$ bundles, and each agent may need to bid on all of them to fully express its preferences. This can be undesirable because there are exponentially many bundles to evaluate, and furthermore determining one's valuation for any given bundle can be computationally or cognitively expensive. So in practice, when the number of items for sale is even moderate, the bidders will not bid on all bundles. Instead, they may wastefully bid on bundles that they will not win, and they may suffer reduced economic efficiency by failing to bid on bundles they would have won.

Recently, an approach has been proposed where the auctioneer is enhanced by *elicitor* software that incrementally elicits the bidders' preferences by querying them (based on the preference information elicited so far) [6, 8, 7]. In the worst case, an exponential amount of communication is required to allocate the items even approximately optimally, if the bidders can have general preferences [10]. (This holds even when bidders can dispose of extra items for free, that is, their valuation functions are monotone.) However, experimentally, preference elicitation appears to help quite a bit [9]. Nonetheless, the amount of querying can be prohibitively large when the bidders have general (monotone) preferences.[1]

An analogous issue arises with shopping agents. Consider the following scenario. Alice goes to her software agent and asks it to help her purchase a vacation. In order to act on her behalf, the agent first needs to find out Alice's preferences (how much is a trip to Hawaii worth compared to a trip to the Bahamas, does it substantially increase the value to her if she can get some entertainment booked in advance, etc.). Then, after scouring the Internet, the agent needs to solve the computational problem of deciding on the best vacation package—the one that maximizes Alice's valuation minus the cost of the trip. In this scenario, there is no auctioneer. Rather, the elicitor is the buyer's helper. Again, the amount of querying can be prohibitively large when the buyer has general (monotone) preferences.

In this paper we study natural classes of preferences that we show can be elicited in polynomial time with value queries, and yet are rich enough to express complementarity and substitutability. We consider a setting in which there is a universe of n items, and a user has different valuations over different subsets of these items. The auctioneer (or software agent) can get information about these preferences using *value queries*: proposing a bundle and asking the user for her valuation of that bundle. This is the same as the notion of a *membership query* in computational learning theory.

The restrictions that we place on the preferences will imply that the preferences *can* be compactly represented. This means that preference elicitation would be *easy* if the elicitor was allowed to ask "give me a computer program for (an approximation to) your preferences". But, in practice, a human will not be able to express her preferences by uttering a formula that completely captures them. Value queries are a standard and much more natural communication protocol.

We begin by arguing that *read-once formulas* over a certain set of gates (defined below) can express many natural preference functions. We then show that if the user's preferences *can* be expressed as a read-once formula of this form, then preference elicitation can be done in polynomial time using value queries (this builds upon work of Angluin et al. [2] and Bshouty et al. [4, 5] for learning over classes of gates motivated from learning theory).[2]

More precisely, we assume the bidder's preferences can be described as a read-once formula over $\{0,1\}^n$ (an input $x \in \{0,1\}^n$ corresponds to the bundle containing the items i such that $x_i = 1$) using gates $\{\mathrm{SUM}, \mathrm{MAX}, \mathrm{ALL}\}$, with real-valued multipliers on the inputs. A read-once formula is a function that can be represented as a tree ("read-once" means that even the initial inputs may only have out-degree 1). A SUM node sums the values of its inputs; a MAX node takes the maximum value of its inputs; an ALL node sums its inputs *unless* one of the inputs is zero, in which case the output is 0. For convenience, we will sometimes view inputs as $x \in \{0,1\}^n$ and sometimes as $x \subseteq \{1,\ldots,n\}$. Each input i also has a positive real-valued multiplier v_i (representing the intrinsic value of that item). For example, a legal function on 3 inputs might be $\mathrm{ALL}(2x_1, \mathrm{MAX}(10x_2, 4x_3))$, which gives value 12 to the input 110, 6 to the input 101, and 0 to the input 011.

Read-once formulas of this type allow for many natural preferences. For example, suppose items are flights and hotel rooms in different locations (e.g., input $x_{i,j,0}$ represents the ith flight to location j, and $x_{i,j,1}$ represents the ith hotel room in location j) and we want to take just one trip. Then for each location j we could compute $\mathrm{ALL}(\mathrm{MAX}\{v_{i,j,0}x_{i,j,0}\}_i, \mathrm{MAX}\{v_{i,j,1}x_{i,j,1}\}_i)$, and then at the root of the tree we would take a MAX over the different destinations.[3]

We then generalize our results by considering a broader class of gates. Let MAX_k output the sum of the k highest inputs, and $\mathrm{ATLEAST}_k$ output the sum of its inputs if there are at least k positive inputs, and 0 otherwise. Finally, we consider $\mathrm{GENERAL}_{k,l}$, a parameterized gate capable of representing all the above types of gates. We show that read-once preferences including all of these gates can be elicited in a polynomial number of queries. We also give positive results for the setting where preferences are *approximately* read-once formulas with MAX and SUM gates.

In addition to the elicitation problem, we study the computational *allocation problem* of determining how the items should be divided among agents with read-once preferences. We show that once the preference function f is known, the problem of finding the subset of items x that maximizes $f(x) - g(x)$, where g is a *linear* cost function (or equivalently, maximizing $f(x) + g(\{1,\ldots,n\} - x)$, where g is a linear valuation function), can be done in polynomial time. This is natural for the case of a shopping agent that buys items individually on the web for a user with valuation function f. However, if g is a general read-once formula, then we show this optimization is NP-complete.

Finally, we consider the class of preferences that can be expressed as monotone polynomials. E.g., $f(x) = ax_1x_2 + bx_2x_3x_4 + cx_3x_4$. We call this class *Toolbox DNF* because it captures settings where each agent has a set of tasks to accomplish (one per term in the polynomial), each task requiring a specific set of tools (the variables in the term) and each having its own value (the coefficient on that term). For example, the tools may be medical patents, and producing each medicine requires a specific set of patents. The value of a set of items to the agent is the sum of the values of the tasks that the agent can accomplish with those items. We show that *Toolbox DNF* preferences can be elicited in a polynomial number of value queries, and that given two agents with *Toolbox DNF* preferences, the items can be optimally allocated in polynomial time using network flow.

More broadly, in the combinatorial auctions literature, the issue of preference elicitation is often put as "we know the

[1] Ascending combinatorial auctions [3, 11, 12] can be viewed as a special case of the preference elicitation framework where the queries are of the form: "Given these prices on items (and possibly also on bundles), which bundle would you prefer the most?".

[2] The reason for so much work on read-once formulas is that Angluin [1] shows that reconstructing read-twice functions from membership (value) queries alone can require an exponential number of queries, even if the function is simply an OR of ANDs (a monotone DNF).

[3] Notice that the multiplicative values at the leaves may not be uniquely specified: if there is only one flight and one hotel in a particular destination, and their combined value is \$1000, then this can be arbitrarily split into values $v, 1000-v$ for the flight and hotel in the formula.

problem is easy if preferences are linear, and hard if preferences are arbitrary. What if preferences are *somewhat* linear?" Our answer is that if one defines "somewhat linear" as read-once (with certain types of gates) or as Toolbox DNF, then preference elicitation with value queries is easy, while still allowing preferences that exhibit interesting behavior.

2. ELICITING READ-ONCE FORMULAS OVER {SUM, MAX, ALL}

Let us define a read-once formula to be *canonical* if no internal node in the tree has a child with the same label. It is not hard to see that for the node functions $\{\text{SUM}, \text{MAX}, \text{ALL}\}$ we can assume without loss of generality that the formula is canonical. In particular, for these gates, if a node and its child are of the same type we can just merge them. We begin, as a warmup, with the easier problem of eliciting when there are no ALL gates.

LEMMA 1. *One can elicit read-once formulas over* $\{\text{SUM}, \text{MAX}\}$ *using* $O(n^2)$ *value queries.*

PROOF. Let $S = \{1, \ldots, n\}$. We will ask two sets of questions:

1. For every $a \in S$, what is $f(\{a\})$? This is n questions.
2. For every pair $a, b \in S$ what is $f(\{a, b\})$? This is $n(n-1)/2$ questions.

Notice that an item $a \in S$ is in the tree if and only if $f(\{a\}) > 0$. Let us call the set of items in the tree T. Second, notice that if the least common ancestor (LCA) of two inputs a and b is a SUM node, then $f(\{a, b\}) = f(\{a\}) + f(\{b\})$, whereas if the LCA is a MAX node, then $f(\{a, b\}) = \text{MAX}(f(\{a\}), f(\{b\}))$. We can therefore use the answers to our queries to construct a graph G with the vertices representing the items in T, and an edge between two vertices if their least common ancestor is a MAX gate.

We now determine the structure of the tree root-down. Notice that if the root node of the formula is a SUM gate, then there will be no edges in G between any two vertices in different subtrees of the root and the graph will be disconnected. On the other hand, if the root node is a MAX gate, then every item in the first subtree will be connected to every item in the other subtrees. Since there are at least two subtrees, this means that if the root is a MAX, then the graph *is* connected.

So, if the graph is disconnected, we place a SUM gate at the root, partition the items into subtrees according to the connected components of the graph, and then recursively solve to discover the structure of each subtree. Formally, we are using the fact that if f_i is the ith subtree and S_i are the items in subtree f_i, then for any $S' \subseteq S_i$, $f(S') = f_i(S')$. On the other hand, if the graph is connected, we place a MAX gate at the root, partition the items into subtrees according to the connected components of the *complement* of the graph, and then recursively solve for the subtrees. Here we are using the fact that in the complement of the graph, there is an edge between two nodes if and only if the lowest common ancestor is a SUM node, so we can use the same argument as before.

Finally, we set the leaf multipliers v_i to the values given by the n unary queries asked in step 1. $\square$

We now proceed to our first main theorem.

THEOREM 1. *One can elicit read-once formulas over* $\{\text{SUM}, \text{MAX}, \text{ALL}\}$ *gates using* $O(n^2)$ *value queries.*

The high-level outline of the proof is as follows. First, notice that if one thinks of a value greater than zero as being *true* and a value equal to zero being *false*, then MAX and SUM act as OR, and ALL acts as AND. We can now apply an algorithm of Angluin et al. [2] that exactly learns (elicits) read-once formulas of AND and OR gates using membership (value) queries to determine the AND/OR structure. Next we expand each OR gate back into a tree of MAX and SUM using the algorithm of Lemma 1. One complication is that to apply that technique here we need to deal with two issues: (1) we only get to observe the output of the full tree, not the specific subtree, and (2) we can only directly manipulate the inputs to the full tree, not the inputs to the specific subtree. Finally, we need to find a consistent set of value multipliers for the inputs.

DEFINITION 1. *Given a read-once formula f consisting of* SUM, MAX, *and* ALL *gates, we define the* ***Boolean image*** *of f to be the function g where for all sets of items S, $g(S)$ is true if and only if $f(S) > 0$.*

As noted above, the Boolean image of a read-once formula f over $\{\text{SUM}, \text{MAX}, \text{ALL}\}$ is equivalent to a read-once monotone Boolean formula with an AND gate wherever f has an ALL gate, and an OR gate wherever f has a MAX or SUM gate. This direct mapping may produce a non-canonical tree (because of a SUM node beneath a MAX node, for instance). The canonical Boolean image is the tree in which all subtrees of OR nodes have been merged together.

DEFINITION 2. *Given a canonical real-valued read-once formula f and its canonical Boolean image g, define* $\text{bool}(u)$ *for a node u in f to be its associated node in g. For a given node v in g, let $r(v)$ be the highest node in* $\text{bool}^{-1}(v)$ *(the one closest to the root of f).*

THEOREM 2 (ANGLUIN ET AL. [2]). *There is an algorithm that exactly identifies any monotone read-once formula over {OR, AND} using $O(n^2)$ queries.*

We can use Theorem 2 to elicit the canonical Boolean image of f, but we now need to expand each OR node back into a subtree of {MAX, SUM}. The next two lemmas show us how to do this. Observe that the only test required to determine the structure of a {MAX, SUM} tree is a test of equality of value between two sets. Lemma 2 will describe how to perform such a test when the function one is interested in is not the root. Lemma 3 shows how to treat the inputs to this OR node like items to elicit the label of their least common ancestor.

LEMMA 2. *Suppose one has g, the canonical Boolean image of f, and v is a node of g. Let $u = r(v)$. Then, for two sets of items S_a and S_b, one can determine if $u(S_a) = u(S_b)$ in two queries.*

PROOF. Suppose that S' is the set of items that are descendents of v, and $Z = S - S'$ is all other items. Without loss of generality, we can assume $S_a, S_b \subseteq S'$. Notice that it is possible $f(S_a) = f(S_b)$ but $u(S_a) \neq u(S_b)$ if v has an ancestor labelled AND. It is also possible that

$f(S_a \bigcup Z) = f(S_b \bigcup Z)$ but $u(S_a) \neq u(S_b)$ if u had a MAX gate as a parent.

Thus, we construct a set R that is the items $x \in Z$ such that the LCA of v and x is an AND node. This is identical to the set of all items in Z that have an ALL node as a LCA with u. This means that for every ALL node that is an ancestor of u, its children that are not ancestors of u have positive output. Also, for every MAX node that is an ancestor of u, its children that are not ancestors of u have zero value. Thus, if c is the sum of the values of the children of the ancestors of u on input R, then $u(S_a)+c = f(S_a \bigcup R)$ or $u(S_a) = f(S_a \bigcup R) = 0$. Thus, $u(S_a) = u(S_b)$ if and only if $f(S_a \bigcup R) = f(S_b \bigcup R)$. □

LEMMA 3. *Suppose one has g, the canonical Boolean image of f, and suppose v_1 and v_2 are siblings in g with an OR parent. Then one can determine whether the LCA of $r(v_1)$ and $r(v_2)$ is a* MAX *or* SUM *in four queries.*

PROOF. Let v_3 be the parent of v_1 and v_2, and let $u_i = r(v_i)$. Observe that no node on the path between u_1 and u_3 is an ALL gate, and no node on the path between u_2 and u_3 is an ALL gate. Also, the LCA of u_1 and u_2 is a descendent of u_3. Suppose that S_i is the set of items that are descendants of of v_i. The LCA of u_1 and u_2 is a MAX node if and only if $u_3(S_1 \bigcup S_2) = u_3(S_1)$ or $u_3(S_1 \bigcup S_2) = u_3(S_2)$. Using Lemma 2, this can be tested using four queries.[4] □

LEMMA 4. *Suppose one has g, the canonical Boolean image of f, and an internal node v in g with k children. Then, one can determine the subtree corresponding to* $\mathrm{bool}^{-1}(v)$ *in $2k(k-1)$ queries, and how the leaves of that subtree map to the children of v.*

PROOF. If v is an AND gate, then $\mathrm{bool}^{-1}(v)$ is an ALL gate. So, we simply construct one ALL gate with children that are leaves labelled with the children of v.

If v is an OR gate, define $u = r(v)$. Define $v_1, \ldots, v_k$ to be the children of v, and define $u_i = r(v_i)$. Using Lemma 3, we can determine whether the LCA of each pair u_i, u_j is a MAX or SUM in 4 queries for a total of $4k(k-1)/2 = 2k(k-1)$ queries. We can now apply the graph decomposition technique from the proof of Lemma 1 to compute the whole subtree. □

LEMMA 5. *Given the structure of a read-once formula f over* $\{\mathrm{MAX}, \mathrm{SUM}, \mathrm{ALL}\}$*, one can determine a consistent set of values (multipliers) on the leaves using at most $3n$ value queries, where n is the number of items.*

The proof is in Appendix A.

Proof (of Theorem 1): The overall algorithm works as follows. Apply the algorithm from [2] to get the Boolean image of g. Use Lemma 4 to find the fine structure of f. Observe that overall all of the internal nodes have n total children, so the total number of queries is less than $2n(n-1)$. Then use Lemma 5 to find the weights. □

3. ELICITING READ-ONCE PREFERENCES OVER MORE GENERAL OPERATORS

We can also elicit preferences with more general gates that we call $\mathrm{ATLEAST}_k$, MAX_k, and $\mathrm{GENERAL}_{k,l}$. An $\mathrm{ATLEAST}_k$ node returns the sum of its inputs if it receives a positive value on at least k inputs. Otherwise, it returns zero. This is a generalization of the ALL node. A MAX_k node returns the sum of the k highest-valued inputs. A $\mathrm{GENERAL}_{k,l}$ gate returns the sum of the l highest-valued inputs if and only if at least k inputs are positive, otherwise it returns zero. We restrict k to be less than or equal to l. Every read-once gate discussed in this paper is a specific instance of a $\mathrm{GENERAL}_{k,l}$ gate.

For instance, imagine that on a vacation to the Bahamas, Alice wanted entertainment. If she got to go out on at least three nights, then the trip would be worthwhile. Otherwise, she would rather stay home. Each night, she takes the maximum valued entertainment option. Then there is an $\mathrm{ATLEAST}_3$ node combining all of the different nights.

In a different situation, imagine that Joe wants a more relaxing vacation in Hawaii, where he does not want to go out more than three nights. In this case, a MAX_3 gate will be useful. For each night, he chooses the best possible entertainment given to him. Then, he takes the best three nights of entertainment.

Finally, imagine that Maggie wants a moderately active vacation, and is interested in going to Paris for a week, and wants at least three but no more than four nights of entertainment. Then a $\mathrm{GENERAL}_{3,4}$ gate will describe her preferences.

THEOREM 3. *A* read-once $\mathrm{GENERAL}_{k,l}$ *function can be learned in polynomial time.*

The proof sketch is in Appendix C.

4. ALLOCATION WITH READ-ONCE PREFERENCES

Suppose we have two parties with preference functions f and g and we want to maximize social welfare, that is, find the allocation $(A, S-A)$ that maximizes $f(A) + g(S-A)$. In this section we show that if one of these functions (say, g) is additive (that is, the value of a bundle is the sum of the values of the items in the bundle) and the other (say, f) is read-once, then we can find the optimal allocation in polynomial time. However, if both f and g are read-once formulas—even containing just ALL, MAX, and SUM gates—then the allocation problem is NP-hard.

Another way to think of maximizing social welfare when g is linear is to think of g as a "cost" function, and we are maximizing $f(A) - g(A)$. For example, we are a software agent and want to find the set of items A that maximizes the difference between the value of A to our user and the cost of A, when items are each purchased separately.

4.1 Read-once valuation function and additive valuation function

THEOREM 4. *Given one party with a known* MAX-SUM-ALL *read-once valuation, and another party with a known additive valuation, there exists a polynomial-time algorithm for maximizing their joint welfare.*

PROOF. The idea is that we recursively learn two things about each node u of f: what subset S' of the items that are descendents of u maximizes $u(S') - g(S')$, and what subset S'' of the items that are descendents of u maximizes $u(S'') - g(S'')$ given the restriction that u is positive.

[4] In reality, one can use less than four queries per test.

Observe that this is trivial if u is a leaf. If not, assume that $u_1, \ldots, u_k$ are the children of u, and that S_i are the items that are descendents of u_i. Define $S'_i \subseteq S_i$ to be the items that maximize $u_i(S'_i) - g(S'_i)$, and $S''_i \subseteq S_i$ to be the items that maximize $u_i(S''_i) - g(S''_i)$ given that $u_i(S''_i)$ is positive.

If u has an ALL label, then $S'' = \bigcup_{i=1}^{k} S''_i$. If $u(S'') - g(S'') > 0$, then $S' = S''$. Otherwise, $S' = \emptyset$. This justifies the need for maintaining both sets.

If u has a SUM label, then $S' = \bigcup_{i=1}^{k} S'_i$. Define j to be the index of the child of u that loses the least from being positive. More formally,

$$j = \operatorname{argmin}_{i \in \{1 \ldots k\}} \left(u_i(S'_i) - g(S'_i)\right) - \left(u_i(S''_i) - g(S''_i)\right).$$

Then we find that $S'' = S''_j \cup \left(\bigcup_{i \neq j} S'_j\right)$.

If u has a max label, then define a to be the index of the best S'_i, or more formally $a = \operatorname{argmax}_{i \in \{1 \ldots k\}} u(S'_i) - g(S'_i)$. Then $S' = S'_a$. Define b to be the index of the best S''_i, or more formally $b = \operatorname{argmax}_{i \in \{1 \ldots k\}} u(S''_i) - g(S''_i)$. Then $S'' = S''_b$.

Each of these sets can be found in polynomial time, so the runtime is polynomial. □

THEOREM 5. *Given one party with a known* GENERAL$_{k,l}$ *read-once valuation, and another party with a known additive valuation, there exists a polynomial-time algorithm for maximizing their joint welfare.*

The proof is similar to that above and is deferred to the full version of the paper.

4.2 NP-hardness of allocation among two parties

THEOREM 6. *Given two agents with known valuations which are* MAX-SUM-ALL *read-once functions, it is NP-hard to find an allocation whose welfare is more than $\frac{1}{2}$ of the welfare of the optimal allocation.*

The proof is in Appendix D.

Notice that achieving welfare of *at least* half the optimal is easy because we can simply give all the items to the agent who values the total the most. This shows that it is NP-hard to do any better.

5. LEARNING PREFERENCES THAT ARE *ALMOST* READ-ONCE

In this section, we will consider the setting where a person's valuation function is a δ-approximation of a read-once function: given that the person's valuation function is f, there exists a read-once function f' such that for all sets S, we have $|f(S) - f'(S)| < \delta$. This algorithm works only for the case of read-once formulas over $\{\text{MAX}, \text{SUM}\}$.

THEOREM 7. *Given black-box access to f, a δ-approximation of a read-once function consisting of* MAX *and* SUM *nodes, a function g can be learned in $n(n-1)/2$ queries such that for any set of items S', we have $|g(S') - f(S')| < 6\delta|S'| + \delta$.*

PROOF. Define f' to be the read-once function such that $|f - f'| < \delta$, and let $v_a = f'(\{a\})$. The key behind this algorithm is that we will throw away all items a where $v_a < 4\delta$, because they will interfere with our LCA test. In order to achieve this, we throw away a if $f(\{a\}) < 5\delta$. Observe that if $v_a \geq 6\delta$, then a will not be thrown away.

We now argue that our test for LCA is correct if all the items a have a value $v_a \geq 4\delta$ or more. Consider the following test for the LCA of a and b: a and b have a max node as an LCA if and only if $f(\{a,b\}) \leq f(\{a\}) + 2\delta$ or $f(\{a,b\}) \leq f(\{b\}) + 2\delta$.

Assume that the LCA is a max node. Then $f'(\{a,b\}) = f'(\{a\})$ or $f'(\{a,b\}) = f'(\{b\})$. Without loss of generality, assume that $f'(\{a,b\}) = f'(\{a\})$. Then:

$$\begin{aligned} f(\{a,b\}) &< f'(\{a,b\}) + \delta \\ f(\{a,b\}) &< f'(\{a\}) + \delta \\ f(\{a,b\}) &< (f(\{a\}) + \delta) + \delta \\ f(\{a,b\}) &< f(\{a\}) + 2\delta \end{aligned}$$

Assume that the LCA is a SUM node. Then $f'(\{a,b\}) = f'(\{a\}) + f'(\{b\}) > f'(\{a\}) + 4\delta$. This implies that $f(\{a,b\}) > f(\{a\}) + 2\delta$. Similarly, $f(\{a,b\}) > f(\{b\}) + 2\delta$.

Given this LCA test, we can learn the structure of the tree as in Lemma 1. We associate the value $f(\{a\})$ to the leaf with item a, even though this value might be off by almost δ. Now, observe that if we throw away items not in S', then this does not affect the difference between $f'(S')$ and $g(S')$. Also, incorrectly calculating the value of an item not in S' does not affect this difference. Now, for each item in S', we can get an error at most less than 6δ, because we threw out an item of value just below 6δ. Also, there is a difference of less than δ between f' and f. □

6. TOOLBOX DNF

We now consider another natural class of preferences that we call *Toolbox DNF*, that can be elicited in polynomial time via value queries. In Toolbox DNF, each bidder has an explicit list of m bundles $S_1, \ldots, S_m$ called *minterms*, with values $v_1, \ldots, v_m$ respectively. The value given to a generic set S' is assumed to be the *sum* of values of the S_i contained in S'. That is,

$$v(S') = \sum_{S_i \subseteq S'} v_i.$$

These preferences are natural if, for example, the items are tools or capabilities and there are m tasks to perform that each require some subset of tools. If task i has value v_i and requires set of tools S_i, then these preferences represent the value that can be attained by any given collection of tools. We show that Toolbox-DNF can be elicited in time polynomial in the number of items and the number of minterms.

THEOREM 8. *Toolbox DNF can be elicited using $O(mn)$ value queries.*

PROOF. We will find the minterms one at a time in an iterative fashion. We can clearly test for the presence of *some* minterm by simply testing if $v(S) > 0$. We can then repeatedly remove elements to find a minimal positive set S_1 in n queries. That is, $v(S_1) > 0$ but for all $x \in S_1$, $v(S_1 - \{x\}) = 0$. This will be our first minterm, and we set $v_1 = v(S_1)$.

At a generic point in time, we will have found minterms $S_1, \ldots, S_i$ with associated values $v_1, \ldots, v_i$. Let v^i be the Toolbox-DNF given by the terms found so far; that is,

$$v^i(S') = \sum_{S_j \subseteq S' : j \leq i} v_j.$$

Define $\bar{v}^i = v - v^i$. Observe that $\bar{v}^i$ is also a toolbox function, namely, $\bar{v}^i(S') = \sum_{S_j \subseteq S' : j > i} v_j$. Also, we can "query" $\bar{v}^i$ at any set S' by querying $v(S')$ and subtracting $v^i(S')$. Thus, we can find a minterm of $\bar{v}^i$ using the same procedure as above, and by our observation, this will be a new minterm of v. We simply continue this process until at some point we find that $\bar{v}^m(S) = 0$ and we are done. □

THEOREM 9. *Given two players with Toolbox-DNF preferences having m_1 and m_2 minterms respectively, optimal allocation can be done in time polynomial in n and $m_1 + m_2$.*

PROOF. Construct a node-weighted bipartite graph with one node on the left for each of the first player's minterms, and one node on the right for each of the second player's minterms. Give node i a weight v_i (where v_i is defined with respect to the associated player). An edge is drawn between a node on the left and a node on the right if the associated minterms share any items.

Notice that any independent set I on this graph represents a legal allocation. The first player gets all items in minterms associated with nodes in I on the left, and the second player gets all items in minterms associated with nodes in I on the right. This is legal because that if an item is in a node on the left and a node on the right, then there is an edge between these nodes, and only one of them would be in I. If remaining items are thrown away, then the total welfare of this allocation is equal to the total weight of independent set I.

Similarly, any legal allocation corresponds to an independent set of the same total weight. If the allocation gives set A_1 to player 1, and A_2 to player 2, then just pick the nodes on the left whose bundles are in A_1, and the nodes on the right whose bundles are in A_2.

Thus, the allocation problem reduces to finding a maximum weight independent set in a bipartite graph, which can be solved via maximum flow. Specifically, it is a standard result that the complement of I, which is a minimum-weight vertex cover, corresponds directly to the minimum cut in an associated flow-network. □

7. CONCLUSIONS AND FUTURE WORK

Elicitation of real-valued preferences is a key capability in many allocation problems, especially in electronic commerce. When multiple items are to be allocated, preference elicitation is difficult because the parties' preferences over items are generally not additive due to complementarity and substitutability. In this paper we studied the elicitation of real-valued preferences over subsets of items, showing that *read-once* preferences over a natural set of gates (a restriction that is still powerful enough to capture complementarity and substitutability), and toolbox DNF preferences, can be elicited in a polynomial number of value queries. Such elicitation can be used by a shopping agent to elicit its user's preferences. It can also be used to elicit bidders' preferences in a combinatorial auction (answering the value queries truthfully can be made an *ex post equilibrium* by using Clarke pricing [6]—assuming the agent has read-once preferences). We also showed that if the party's preferences are close to read-once, then a good approximation of the preferences can be elicited quickly.

We also studied the complexity of the computational problem of allocating the items given the parties' preferences. We showed that this is NP-hard even with just two parties with read-once valuations. However, in the natural setting where only one of the parties has a read-once valuation and the other has an additive valuation function, the allocation problem is solvable in polynomial time.

There are several interesting avenues for future research along these lines. For one, in these multi-item allocation problems, what is the limit on the generality of preferences that can be elicited in polynomial time via value queries? Second, when there are multiple parties, one agent's preferences can be used to decide what information needs to be elicited from another party in order to determine an optimal (or approximately optimal) allocation of items. This has been the driving motivation in the work on preference elicitation in combinatorial auctions [6, 8, 7, 9], but our work in this paper did not yet capitalize on this extra power. In the future, it would be interesting to harness this power, together with the possibilities that preference restrictions open, to design effective goal-driven preference elicitation algorithms.

Acknowledgements

We would like to thank Jeff Jackson for a number of helpful conversations. This material is based upon work supported under NSF grants CCR-9732705, CCR-0085982, CAREER Award IRI-9703122, IIS-9800994, ITR IIS-0081246, ITR IIS-0121678, and an NSF Graduate Research Fellowship. Any opinion, findings, conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

[1] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1988.

[2] D. Angluin, L. Hellerstein, and M. Karpinski. Learning read-once formulas with queries. In *Journal of the ACM*, volume 40, pages 185–210, 1993.

[3] S. Bikhchandani, S. de Vries, J. Schummer, and R. V. Vohra. Linear programming and Vickrey auctions, 2001. Draft.

[4] N. Bshouty, T. Hancock, L. Hellerstein, and M. Karpinski. An algorithm to learn read-once threshold formulas, and transformations between learning models. *Computational Complexity*, 4:37–61, 1994.

[5] N. H. Bshouty, T. R. Hancock, and L. Hellerstein. Learning arithmetic read-once formulas. *SIAM Journal on Computing*, 24(4):706–735, 1995.

[6] W. Conen and T. Sandholm. Preference elicitation in combinatorial auctions: Extended abstract. In *Proceedings of the ACM Conference on Electronic Commerce (ACM-EC)*, pages 256–259, Tampa, FL, Oct. 2001. A more detailed description of the algorithmic aspects appeared in the IJCAI-2001 Workshop on Economic Agents, Models, and Mechanisms, pp. 71–80.

[7] W. Conen and T. Sandholm. Differential-revelation VCG mechanisms for combinatorial auctions. In *AAMAS-02 workshop on Agent-Mediated Electronic Commerce (AMEC)*, Bologna, Italy, 2002.

[8] W. Conen and T. Sandholm. Partial-revelation VCG mechanism for combinatorial auctions. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 367–372, Edmonton, Canada, 2002.
[9] B. Hudson and T. Sandholm. Effectiveness of preference elicitation in combinatorial auctions. In *AAMAS-02 workshop on Agent-Mediated Electronic Commerce (AMEC)*, Bologna, Italy, 2002. Extended version: Carnegie Mellon University, Computer Science Department, CMU-CS-02-124, March. Also: Stanford Institute for Theoretical Economics workshop (SITE-02).
[10] N. Nisan and I. Segal. The communication complexity of efficient allocation problems, 2002. Draft. Second version March 5th.
[11] D. C. Parkes and L. Ungar. Iterative combinatorial auctions: Theory and practice. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, pages 74–81, Austin, TX, Aug. 2000.
[12] P. R. Wurman and M. P. Wellman. AkBA: A progressive, anonymous-price combinatorial auction. In *Proceedings of the ACM Conference on Electronic Commerce (ACM-EC)*, pages 21–29, Minneapolis, MN, Oct. 2000.

APPENDIX

A. LEARNING THE WEIGHTS

There are some situations where there are multiple consistent values for the leaves of the function f. For example, if a plane ticket and hotel are fed into an ALL gate, and together they are worth \$1000, then any pair of values $(x, 1000-x)$ are consistent for the leaves. Our algorithm will guarantee to learn *some* consistent set of values.

We will use a recursive technique to learn the tree. At each node, we will either learn the function f' at that node, or a function f'' such that there exists some $c \in \mathbb{R}$ such that for all S'', if $f'(S'') > 0$ then $f'(S'') = f''(S'')+c$, otherwise $f''(S'') = 0$. We call such an f'' a *shift* of function f', even though the shift is only on the inputs where $f' > 0$. In the case that we learn a shift f'', the function that we learn will be "shiftable" in the sense that it will be easy to make an additive increase or decrease of the value of the function on all positive inputs by recursively shifting the values of the subfunctions. This will be done as a postprocessing step. An example of a function that is not shiftable is a SUM node with two leaves as children.

For easier recursive application, we will allow leaves to have positive and negative values, and find a function where each node computes a real value and a boolean value. If the boolean value is false, the real value will be zero. Leaves will return *true* and their associated real value if the item is in the input, *false* and zero otherwise. SUM nodes will return the OR of the boolean values of their children and the sum of their real values. MAX nodes will return the OR of their children and the highest real value of a *true* child if this is true, otherwise zero. This means that if one child has a negative real value and is *true*, and one child has a zero real value and is *false*, MAX will return the negative value. ALL will return the AND of its children and the sum of its children's real values. The result of the tree is the real value returned by the root. Observe that this is identical to the original if the values of the leaves are all positive.

DEFINITION 3. *We define a node to be shiftable using a recursive definition:*

1. *A leaf is shiftable.*
2. *A* SUM *node is never shiftable.*
3. *A* MAX *node is shiftable if all of its children are shiftable.*
4. *An* ALL *node is shiftable if at least one of its children is shiftable.*

We define a tree to be shiftable if its root is shiftable, and a function to be shiftable if the representative tree is shiftable.

LEMMA 6. *Suppose that f is a read-once function that is shiftable. We can construct a new read-once function f' such that $f' = f + c$ for all inputs where f returns a true value.*

PROOF. We perform this construction recursively, only modifying the values of the leaves. For a leaf node, it is sufficient to add c to its value. This is the base case.

If the tree is not a leaf, the read-once function has a MAX or ALL label at the root. Then the root cannot be a SUM node.

Assume the root is a MAX node. In this case, we know that the value of the tree, if it is *true*, will be the value of one of the subfunctions $f_1, \ldots, f_k$. Observe that all subfunctions are shiftable. Thus, by the inductive hypothesis, we can change each subfunction such that $f'_i = f_i + c$.

Assume the root is an ALL node. In this case, we know that the value of the tree, if it is *true*, will be the value of the sum of the subfunctions $f_1, \ldots, f_k$. Now, there must exist some subfunction f_i that is shiftable. Therefore, we change the value of this subfunction such that $f'_i = f_i + c$, and leave the remaining subfunctions unchanged. □

LEMMA 7. *Given the structure of a read-once function f and a node N, suppose that f' is the true function associated with N. One can determine values for the leaves that are descendants of N such that the resulting function f'' is a shift of f', using at most $(3/2)n$ queries, where n is the number of nodes that are descendants of N. If f' is not shiftable, then we guarantee that $f'' = f'$.*

PROOF. We prove this inductively on the depth of the tree. This holds for a leaf. Now let us consider trees of larger depth. Define S' to be the items that are descendants of N. We begin by showing we can construct a set R similar to that of Lemma 2. Define R to be the set of all items that have an ALL gate as a LCA with N and are not descendants of N.

Thus, in an argument similar to Lemma 2, there exists a $c \in \mathbb{R}$ such that for all $S'' \subseteq S'$, $f(S'' \bigcup R) = f'(S'') + c$.

Now, define $N_1, \ldots, N_k$ to be the children of N. Define f'_i to be the function associated with each N_i, and S_i to be the items that are descendants of N_i. Recursively for each subtree, we can discover a f_i such that there exists a $c_i \in \mathbb{R}$ where for all S, $f_i(S) = f'_i(S) + c_i$. In order to obtain these functions, we required $(3/2)(n-k)$ queries.

Suppose that N has a MAX label. Then if one of the subtrees, say i, is not shiftable, then $f_i(S_i) = f'_i(S_i) = f'(S_i)$, so $f_i(S_i) + c = f(S_i \bigcup R)$. Thus, with one query we can discover c. Then, for each subtree f_j that is shiftable, we can find c_j. It is true that $f'_j(S_j) + c = f(S_j \bigcup R)$, and so $c_j = f_j(S_j) - f'_j(S_j) = f_j(S_j) - (f(S_j \bigcup R) - c)$. Thus, we can find a function equal to f'_j when N_j is true by finding

$f_j'' = f_j - c_j$. This requires overall less than or equal to k queries.

However, if N has a MAX label and all the subtrees are shiftable, then the tree is shiftable. However, we can still construct a set of functions f_i'' such that for all i, for all $S'' \subseteq S'$ such that N_i is true, $f_i''(S'') = f_i'(S'') + c$. This can be done by defining $c_i' = f_i(S_i) - f(S_i \bigcup R)$, and then $f_i'' = f_i - c_i'$. Observe that $f_i'' = f_i - (f_i - (f_i' + c)) = f_i' + c$ when the input makes N_i true.

Suppose that N has an ALL label. Then we directly make $f'' = \text{ALL}_{i=1}^k f_i'$. N is *true* if and only if all N_i are true, and in this case $\sum_{i=1}^k f_i' = \sum_{i=1}^k (f_i + c_i) = (\sum_{i=1}^k f_i) + (\sum_{i=1}^k c_i)$. Thus, if all of the subtrees of N_i are not shiftable, then $f'' = f'$.

The most complex case is when N has a SUM label. In this case, we can learn the value of each subtree exactly. We measure $f(S_1 \bigcup S_2 \bigcup R)$, $f(S_1 \bigcup R)$, and $f(S_2 \bigcup R)$. Observe that

$$\begin{aligned} f(S_1 \bigcup S_2 \bigcup R) - f(S_2 \bigcup R) &= f'(S_1 \bigcup S_2) - f'(S_2) \\ &= f_1'(S_1). \end{aligned}$$

If $k > 2$, then for each $i > 2$ we measure $f(S_1 \bigcup S_i \bigcup R)$. Thus, for each $1 \leq i \leq k$ such that the ith subtree is shiftable, we can calculate $c_i' = f_i(S_i) - f_i'(S_i)$, and construct $f_i'' = f_i - c_i'$. This requires at most $k + 1 \leq 3k/2$ queries. □

Proof (of Lemma 5): One can elicit values such that there exists a c such that if the root is true, $f'(S') = f(S') + c$ using $(3/2)(m-1)$ queries, where m is the number of nodes in the tree (which can be no more than twice the number of items). If f is not shiftable, we are done. If f is shiftable, one can use a single query on the set of all items S to find $c = f'(S) - f(S)$. Then one can compute $f' - c$. □

B. LEARNING A MAX_L TREE

Before we begin, we assume the tree is canonical, that no MAX_1 node v has a MAX_1 child v', because the children of v' can be made children of v. Also, no node is labeled MAX_0, because such nodes and their descendants can be removed from the tree without affecting the output.

We begin with a large number of definitions, grouped for ease of reference. S', S'' are sets of items and v, v' are nodes.

- Define r to be the root node.
- Define $T = \{a \in S : f(a) > 0\}$.
- Define $L(v)$ to be l if v is labeled MAX_l.
- Define $C(v)$ to be the children of v.
- Define $D(v)$ to be the items that are descendants of v.
- Define $DP(v) = \{D(v') : v' \in C(v)\}$.
- Define $C(v, S') = \{v' : v' \in C(v), D(v') \cap S' \neq \emptyset\}$.
- A set S' is **dependent** if there exists some $a \in S'$ such that $f(S') = f(S' \backslash \{a\})$.
- A set is **independent** if it is not dependent.
- A set S' is a **minimal dependent set** if it is dependent and there exists no dependent proper subset $S'' \subset S'$.
- A set S' is **pairwise independent** if no subset of S' of size 2 is dependent.
- A node v is a **witness** for a set S' if $L(v) < |C(v, S')|$.
- A set S' **represents** v if $L(v) + 1 = |S'| = |C(v, S')| + 1$.
- Given a graph G which has the items in T as nodes, G is **legal** if for all $a, b \in T$ that are connected, $LCA(a, b) \neq r$.
- Given a graph G of T, define $R(G)$ to be the collection of sets such that for each set $S' \in R(G)$, S' contains exactly one item from each connected component in G.

We will attempt to find $DP(r)$ and $L(r)$ using a polynomial number of value queries[5]. We will do this by constructing a legal graph and then adding edges to it. The intermediate outcome of the algorithm depends on $L(r)$:

- $L(r) = 1$: we will quickly find that $L(r) = 1$ and be able to use an applicable trick.
- $L(r) = |C(r)|$: $DP(r)$ will be the connected components of the graph.
- $1 < L(r) < C(r)$. We will eventually find a set S' that represents r, and use it to find $DP(r)$.

We will not need to know $L(r)$ in advance, we will find it in the course of the algorithm. Before we begin the algorithm, we will prove some lemmas about witnesses, representatives, and dependent sets.

Fact 1. *1. A set S' is dependent if and only if it has a witness.*

2. If S' is dependent and $S' \subseteq S''$, then S'' is dependent.

3. If S' is independent and $S' \supseteq S''$, then S'' is independent.

4. A minimal dependent set S' represents some node v.

5. A set S' that represents some node v is a minimal dependent set.

6. A set $\{a, b\} \subseteq T$ is dependent if and only if $L(LCA(a, b)) = 1$.

7. The subset of a pairwise independent set is pairwise independent.

8. If S' is pairwise independent, then it has no witnesses v where $L(v) = 1$.

Lemma 8. *We can determine if $L(r) = 1$. If $L(r) = 1$, we can find $DP(r)$. If $L(r) > 1$, we can find a legal graph G such that for all $S' \in R(G)$, S' is pairwise independent.*

Proof. We construct a graph with nodes in T, connecting two items $\{a, b\}$ if and only if they are a dependent set. By an argument similar to that of Lemma 1, if G is connected, then the root is a MAX_1 node. In this case, the connected components of the complement of G is $DP(r)$, and $L(r) = 1$.

If G is not connected, then the root is not a MAX_1 node. Thus, $L(r) > 1$. For all edges (a, b) in the graph G, $L(LCA(a, b)) = 1$, and therefore $LCA(a, b) \neq r$.

Now, consider an arbitrary $S' \in R(G)$. If $\{a, b\} \subseteq S'$, then there is no edge between a and b. Thus, $\{a, b\}$ is independent, and S' is pairwise independent. □

Now we will be constructing more legal graphs by adding edges.

[5]In the remaining discussion, we will just use the phrases "we can" or "one can" to indicate one can find some set, partition, or value with a polynomial numbe of value queries.

FACT 2. *If all $S' \in R(G)$ are pairwise independent, and G' has all of the edges of G, then all $S' \in R(G)$ are pairwise independent.*

Now, we proceed to the primary reason that dependent sets are useful.

FACT 3. *Given a set S'', one can find a minimal dependent set S' such that $S' \subseteq S''$.*

LEMMA 9. *Given a pairwise independent set S'' a minimal dependent set $S' \subseteq S''$ that represents some unknown node v, and an element $a \in S''$, one can determine if $a \in D(v)$. Specifically, $a \in D(v)$ if and only if there exists some $b \in S'$ such that $S' \backslash \{b\} \cup \{a\}$ is dependent.*

COROLLARY 1. *One can find $D(v) \cap S''$.*

COROLLARY 2. *Given a legal graph G, a pairwise independent set $S'' \in R(G)$, and a minimal dependent set $S' \subseteq S''$, one can find if S' represents r.*

PROOF. If $a \in D(v)$, then either:

- For all $b \in S'$, $LCA(a,b) = v$. Then for all $b \in S'$, $(S \backslash \{b\}) \cup \{a\}$ represents v and is dependent.
- There exists some $b \in S'$ where $LCA(a,b) \neq v$. In this case, $(S \backslash \{b\}) \cup \{a\}$ represents v and is dependent.

If $a \notin D(v)$, then take an arbitrary $b \in S'$, and define $S''' = S' \backslash \{b\}$. We will prove that $S''' \cup \{a\}$ does not have a witness.

First, observe that $L(v) = |C(v,S')| - 1 = |C(v,S''')| = |C(v,S''' \cup \{a\})|$. Observe that S''' is independent. For all $v' \neq v$, $C(v',S''') \leq 1$. Thus, $C(v',S''' \cup \{a\}) \leq 2$. However, since $S''' \cup \{a\} \subseteq S''$ is pairwise independent, it has no witness where $L(v') = 1$, so it cannot have any witness. □

LEMMA 10. *Given a pairwise independent set S'', a minimal dependent set $S' \subseteq S''$, an element $a \in D(v) \cap (S'' \backslash S')$, an element $b \in S'$, then $LCA(a,b) \neq v$ if and only if $(S' \backslash \{b\}) \cap \{a\}$ is dependent and for all $c \in S' \backslash \{b\}$, $(S' \backslash \{c\}) \cap \{a\}$ is independent.*

COROLLARY 3. *Given a legal graph G, a pairwise independent set $S'' \in R(G)$, and a minimal dependent set $S' \subseteq S''$ that represents v, one can find $DP(v)$.*

The proof is similar to that above.

LEMMA 11. *We can learn MAX_l tree using a polynomial number of value queries.*

The algorithm is as follows:

1. Begin with a legal graph G such that all $S' \in R(G)$ are pairwise independent.
2. Choose an $S'' \in R(G)$.
3. If S'' is independent, then $L(r) = |C(r)|$, and the components of the graph G are $DP(r)$.
4. If S'' is dependent, then find a minimal dependent set $S' \subseteq S''$.
5. If S' represents the root, then $L(r) = |S'| - 1$, and one can find $DP(r)$.
6. If S' does not represent the root, then for all pairs $a, b \in S'$, add an edge (a,b) in G. Continue from step 2.

This algorithm will terminate because $|S'| > 1$, and therefore we are always adding edges to the graph.

C. PROOF SKETCH FOR $\mathrm{GENERAL}_{K,L}$ PREFERENCES

Observe that for the boolean image of a $\mathrm{GENERAL}_{k,l}$ node is a $THRESH_k$ node, a node that is true if at least k inputs are true, and false otherwise. So, the boolean image of a read-once $\mathrm{GENERAL}_{k,l}$ function is a read-once $THRESH_k$ function. An algorithm to learn such functions is discussed in [4]. Like learning a $\{\mathrm{SUM}, \mathrm{MAX}, \mathrm{ALL}\}$ function, when we are learning a $\mathrm{GENERAL}_{k,l}$ function, we can begin by learning the boolean image.

$THRESH_1$ nodes represent one or many $\mathrm{GENERAL}_{1,l}$ nodes, just like OR nodes can represent one or many MAX or SUM nodes. Also, $THRESH_k$ nodes, when $k > 1$, represents exactly one $\mathrm{GENERAL}_{k,l}$ node for some $l \geq k$.

We can calculate the output of a specific node to within an additive factor, by forming a set like in Lemma 2. Using equality tests, one can learn the MAX_l structure corresponding to a $THRESH_1$ node. Thus, with an equality test, we can learn the structure of the original function.

As before, the weights can be computed in polynomial time. We will recursively learn a function for each node that may be off by a fixed constant value on sets where the original function is positive. Again, we can think about shifting subfunctions as necessary. We define the concept of a shiftable node recursively.

1. A leaf is shiftable.
2. Suppose an internal node is labeled $\mathrm{GENERAL}_{k,l}$ and has m children.
   (a) If $k = l < m$ and all the children are shiftable, the node is shiftable.
   (b) If $k = l = m$ and there is a shiftable child, the node is shiftable.
   (c) Otherwise, the node is not shiftable.

The tree is learned recursively from the bottom. One can learn a shiftable tree within a constant factor on all positive inputs and a tree that is not shiftable exactly.

We can recursively learn the function for each node, perhaps being off by a constant value on the positive values. As before, if a node is not shiftable, we can learn the function exactly. If $k < l$, then it is analogous to a SUM gate. If $k = l = m$, then it is analogous to the ALL gate. If $k = l < m$, then it is loosely analogous to the MAX gate. This is the hardest case where we solve a simple system of linear equalities in order to determine how to much to shift each subfunction.

D. PROVING APPROXIMATE NP-HARDNESS

Suppose we begin with a SAT instance, which we will translate into an instance of this problem. The maximum global welfare will be 2 if the instance is satisfiable, and 1 otherwise. Thus, in order to get more than 1/2 of the optimal welfare, we must solve an NP-complete problem. Let $\{C_1, \ldots C_k\}$ denote the clauses of the given formula, and let $X = \{x_1, \ldots, x_n\}$ denote the variables.

The plan is that Agent I's valuation will be 1 if the allocation represents an assignment to the variables and 0 otherwise. Agent II's valuation can be equal to 1 if the allocation represents a satisfying assignment, and will be 0 if it does not represent a satisfying assignment. If the allocation does not represent an assignment, the value is between zero and 1 inclusive.

A variable x_i will be represented by two sets of items: $P_i = \{p_i^1, \ldots, p_i^k\}$ and $N_i = \{n_i^1, \ldots, n_i^k\}$. If Agent I has all the items in set N_i, then x_i is "positive". If Agent I has all the items in set P_i, then the variable is "negative". If every variable is either positive or negative, then the allocation is called "legal". The function:

$$f_i = \mathrm{MAX}\left(\mathrm{ALL}_{j=1}^{k} p_i^j, \mathrm{ALL}_{j=1}^{k} n_i^j\right)$$

is k if x_i is positive or negative, zero otherwise. Agent I's valuation is:

$$v_I = \frac{1}{nk}\mathrm{ALL}_{i=1}^{n} f_i$$

Observe that $v_I = 1$ if each variable is positive or negative, and $v_I = 0$ otherwise.

Now, we will use the set D_j to represent the clause C_j. D_j is defined as follows: p_i^j is in D_j if and only if x_i occurs positively in C_j. n_i^j is in D_j if and only if x_i occurs negatively in C_j. So, given that the allocation is legal, observe that:

$$\max_{a \in D_j} a$$

is 1 only if there is a variable occurring positively in j which is not negative according to the allocation, or a variable occurring negatively in j which is not positive according to the allocation. Also, this function is never more than 1. The valuation of Agent II is:

$$v_{II} = \frac{1}{k}\mathrm{ALL}_{j=1}^{k}\mathrm{MAX}\{a : a \in D_j\}.$$

This function is never greater than 1. If the allocation is legal but does not correspond to a satisfying assignment, the value is zero.

Suppose that there is a satisfying assignment. Then for each variable x_i, if it is positive give all p_i^j to Agent II and all n_i^j to Agent I, and if it is negative give all n_i^j to Agent II and all p_i^j to Agent I. This allocation has a value 2.

Thus, if the assignment is not legal, the first agent has value 0 and the total value is less than or equal to 1. If the assignment is legal but not a satisfying assignment, the total value is 1. The optimal global welfare is 2 if there is a satisfying assignment, and 1 otherwise.