Fast-Paced Trading of Multi-Attribute Goods

Eugene Fink
Computer Science and Eng.
University of South Florida
Tampa, Florida 33620
eugene@csee.usf.edu
www.csee.usf.edu/∼eugene

Josh Johnson
Electronic Arts, Tiburon
2301 Lucien Way, Suite 395
Maitland, Florida 32751
joshjohnson@cfl.rr.com

John Hershberger
Computer Science and Eng.
University of South Florida
Tampa, Florida 33620
jhershbe@csee.usf.edu
www.csee.usf.edu/∼jhershbe

Abstract – *We present an exchange system for trading complex nonstandard goods, such as used cars. We explain the representation of desirable purchases and sales, describe data structures for fast identification of matches between buyers and sellers, and give experiments on the system's performance.*

Keywords: E-commerce, exchange markets.

1 Introduction

The Internet has opened opportunities for efficient on-line trading, and researchers have developed various automated auctions, which have become a popular means for on-line trading; however, auctions have several drawbacks, including lack of liquidity and asymmetry between buyers and sellers. Traditional exchange markets do not have these drawbacks, but they require rigid standardization of tradable items. For example, the New York Stock Exchange allows trading of about 3,000 stocks, and a buyer or seller has to indicate a specific stock.

For most goods, the description of a desirable trade is more complex; for instance, a car buyer needs to specify a model, options, color, and other features of a desirable vehicle. She may also want to include preferences; for example, she may indicate that a red car is preferable to a white car. An exchange for nonstandard goods should allow complex constraints in specifications of buy and sell orders, support fast-paced trading for markets with millions of orders, and include optimization techniques that maximize the traders' satisfaction.

We have defined trading semantics for complex goods, and developed an exchange system that supports markets with up to 300,000 orders [5, 8, 9, 10]. We begin with a review of previous work on auctions and exchanges (Section 2). We then give a formal model of a market for complex goods (Section 3), and describe data structures and algorithms for identifying matches between buy and sell orders (Sections 4 and 5). Finally, we show how the system's performance depends on the market size and order complexity (Section 6).

2 Previous work

We review related work on combinatorial auctions, representation of complex goods, and exchanges for nonstandard goods.

Combinatorial auctions. Researchers have developed several efficient systems for combinatorial auctions, which allow buying and selling sets of goods rather than individual items.

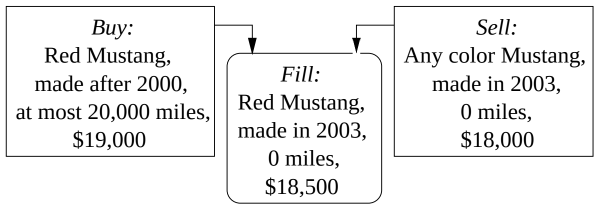
Rothkopf, Pekeč, and Harstad gave a detailed analysis of combinatorial auctions and described semantics of combinatorial bids that allowed fast matching [15]. Nisan discussed alternative semantics, formalized the problem of searching for optimal and near-optimal matches, and proposed a linear-programming solution [12, 14].

Sandholm and his colleagues developed combinatorial auctions that supported markets with several thousand bids [16, 17, 19]. Gonen and Lehmann studied branch-and-bound heuristics for processing combinatorial bids and integrated them with linear programming [6, 7]. Andersson, Tenhunen, and Ygge compared the main techniques for combinatorial auctions and proposed an integer-programming representation that allowed richer bid semantics [1].

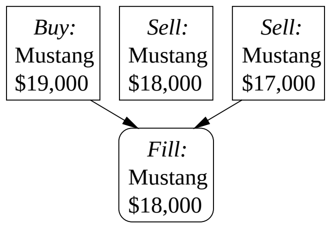
Although the developed systems can efficiently process several thousand bids, their running time is superlinear in the number of bids, and they do not scale to larger markets.

Advanced semantics. Several researchers have studied techniques for specifying the dependency of item price on the number and quality of items. They have also investigated techniques for processing "flexible" bids, specified by hard and soft constraints.

Che analyzed auctions that allowed negotiating not only the price, but also the quality of goods [3]. A bid in these auctions was a function that specified a desired trade-off between price and quality. Cripps and Ireland considered a similar setting and suggested several strategies for bidding on price and quality [4]. Bichler discussed a market that allowed negotiations on any attributes of goods [2].



(a) Matching orders and the resulting trade.



(b) Selection among alternative matches.

Figure 1: Examples of trades in a used-car market.

Sandholm and Suri studied combinatorial auctions that allowed bulk discounts [18]; that is, they enabled a bidder to specify a dependency between item price and transaction size. Lehmann, Lehmann, and Nisan also considered the dependency of price on transaction size, showed that the problem of finding the best matches was NP-hard, and developed a greedy approximation algorithm [13].

This initial work leaves many open problems, including the use of complex constraints with general preference functions, symmetric treatment of buyers and sellers, and design of efficient matching algorithms for advanced semantics.

Exchanges. Auction researchers have traditionally viewed exchanges as a variety of auction markets, called continuous double auctions. Wurman, Walsh, and Wellman proposed a theory of exchange markets and implemented a general-purpose system for auctions and exchanges [20]. Sandholm and Suri developed an exchange for combinatorial orders, but it could not support markets with more than one thousand orders [17]. Kalagnanam, Davenport, and Lee investigated techniques for placing orders with complex constraints and identifying matches between them, which scaled to a few thousand orders [11].

The related open problems include development of a scalable exchange system for large combinatorial markets, as well as support for orders with complex constraints.

3 Multi-attribute market

We present a formal model for describing desirable purchases and sales, which allows the use of hard and soft constraints.

Example. We consider an exchange for trading new and used cars. To simplify this example, we assume that a trader can describe a car by four attributes: model, color, year, and mileage. A prospective buyer can place a *buy order,* which includes a description of a desired vehicle and a maximal acceptable price; for instance, she may indicate that she wants a red Mustang, made after 2000, with at most 20,000 miles, and she is willing to pay $19,000. Similarly, a seller can place a *sell order;* for example, a dealer may offer a brand-new Mustang of any color for $18,000.

An exchange system must search for matches between buy and sell orders, and generate corresponding *fills,* that is, transactions that satisfy both buyers and sellers (Figure 1a). If the system finds several matches for an order, it should choose the match with the best price; for instance, the buy order in Figure 1(b) should trade with the cheaper of the two sell orders.

Attributes. A specific market includes a certain set of items that can be bought and sold, defined by a list of attributes and possible values of each attribute. When a trader places an order, she has to specify a set of acceptable values for each attribute, which is called an *attribute set.* She specifies some set I_1 of values for the first attribute, some set I_2 for the second attribute, and so on. The resulting set I of acceptable items is the Cartesian product $I = I_1 \times I_2 \times \ldots$. For example, suppose that a car buyer is looking for a Mustang or Camaro, the acceptable colors are red and white, the car should be made after 2000, and it should have at most 20,000 miles; then, the item set is $I = \{\texttt{Mustang}, \texttt{Camaro}\} \times \{\texttt{red}, \texttt{white}\} \times [2001..2003] \times [0..20{,}000]$.

A trader can also define an item set I as the union of several Cartesian products. For instance, if she wants to buy either a used red Mustang or a new red Camaro, she can specify the set $I = \{\texttt{Mustang}\} \times \{\texttt{red}\} \times [2001..2003] \times [0..20{,}000] \cup \{\texttt{Camaro}\} \times \{\texttt{red}\} \times \{2003\} \times [0..200]$.

Note that the Cartesian-product representation is a simplification based on the assumption that all items have the same attributes. Some markets do not satisfy this assumption; for instance, if we trade cars and bicycles on the same market, we need two lists of attributes.

Attribute sets. A trader can use specific values or ranges for each attribute; note that ranges work only for numeric attributes, such as year and mileage. A market specification may also include certain *standard sets* of values, such as "all sports cars" and "all American cars," and a trader can use them in her orders. We must include all standard sets in the market description, and traders cannot define new sets. We specify a standard set by a list of values or numeric ranges; for example, a set of American cars is a list of models: $\{\texttt{Camaro}, \texttt{Mustang}, \ldots\}$.

A trader can also use intersections and unions in the specification of attribute sets. For instance, suppose that a car buyer is interested in Mustangs, Camaros, and Japanese sports cars. Suppose further that we have defined a standard set of all Japanese cars, and another standard set of all sports cars. Then, the buyer can represent the desired set of models as $\{$`Mustang`, `Camaro`$\} \cup ($`Japanese-Cars` $\cap$ `Sports-Cars`$)$.

To summarize, an attribute set is one of the following:

- Specific value, such as Mustang or 2001.
- Range of values, such as [2001..2003].
- Standard set of values, such as all Japanese cars.
- Intersection of several attribute sets.
- Union of several attribute sets.

Price and quality. A trader should specify a limit on the acceptable price; for instance, a car buyer may be willing to pay \$19,000 for a Mustang, but only \$18,000 for a Camaro. This price limit is a real-valued function defined on the set I, encoded by a C++ procedure; for each item $i \in I$, it gives a certain limit $Price(i)$.

A trader may also specify preferences among acceptable transactions, which depend on an item i and its price p; for instance, a car buyer may indicate that a Mustang for \$18,000 is better than a Camaro for \$17,000. We represent preferences by a real-valued function $Qual(i,p)$ that assigns a numeric quality to each pair of an item and price. Larger values correspond to better transactions; that is, if $Qual(i_1,p_1) > Qual(i_2,p_2)$, then trading i_1 at price p_1 is better than i_2 at p_2. The encoding of a quality function is a C++ procedure, which inputs an item description and price, and outputs a numeric quality value. Each trader can use her own quality functions and specify different functions for different orders. Note that buyers look for low prices, whereas sellers prefer to get as much money as possible, which means that quality functions must be monotonic on price:

- *Buy monotonicity:*
  If $Qual_b$ is quality for a buy order, and $p_1 \le p_2$, then, for every item i, $Qual_b(i,p_1) \ge Qual_b(i,p_2)$.
- *Sell monotonicity:*
  If $Qual_s$ is quality for a sell order, and $p_1 \le p_2$, then, for every item i, $Qual_s(i,p_1) \le Qual_s(i,p_2)$.

We do not require a trader to specify a quality function for each order; by default, quality is defined through price. This default function is the difference between the price limit and actual price, divided by the price limit:

- *For buy orders:* $Qual_b(i,p) = (Price(i) - p)/Price(i)$.
- *For sell orders:* $Qual_s(i,p) = (p - Price(i))/Price(i)$.

Order sizes. If a trader wants to buy or sell several identical items, she can include their number in the order specification. We assume that an order size is a natural number, thus enforcing discretization of continuous goods, such as orange juice. The trader can also specify a minimal acceptable size of a transaction, and indicate that a transaction size must be divisible by a certain number, called a *size step.*

To summarize, an order includes six elements:

```
FILL-SIZE($Max_b, Min_b, Step_b, Max_s, Min_s, Step_s$)
The algorithm inputs the size specification of a buy order,
$Max_b$, $Min_b$, and $Step_b$, along with the size specification
of a matching sell order, $Max_s$, $Min_s$, and $Step_s$.

Let $step$ be the least common multiple of $Step_b$ and $Step_s$
$size := \lfloor \min(Max_b, Max_s)/step \rfloor \cdot step$
If $size \ge \max(Min_b, Min_s)$, then return $size$
Else, return NONE (no acceptable size)
```

Figure 2: Computing the fill size for two matching orders.

- Item set, I.
- Price function, $Price\colon I \to \mathbf{R}$.
- Quality function, $Qual\colon I \times \mathbf{R} \to \mathbf{R}$.
- Overall order size, Max.
- Minimal acceptable size, Min.
- Size step, $Step$.

Fills. When a buy order matches a sell order, the corresponding parties can complete a trade; we use the term *fill* to refer to the traded items and their price (Figure 1). We define a fill by a specific item i, its price p, and the number of traded items, denoted *size.* If $(I_b, Price_b, Max_b, Min_b, Step_b)$ is a buy order, and $(I_s, Price_s, Max_s, Min_s, Step_s)$ is a matching sell order, then a fill must satisfy the following conditions:

- $i \in I_b \cap I_s$.
- $Price_s(i) \le p \le Price_b(i)$.
- $\max(Min_b, Min_s) \le size \le \min(Max_b, Max_s)$.
- $size$ is divisible by $Step_b$ and $Step_s$.

If the buyer's price limit is larger than the seller's limit, we split the price difference between the buyer and seller, which means that $p = (Price_b(i) + Price_s(i))/2$. Furthermore, we assume that the buyer and seller are interested in trading at the maximal size, or as close to the maximal size as possible; thus, the fill has the largest possible size (Figure 2).

After getting a fill, the trader may keep the initial order, reduce its size, or remove the order; the default option is the size reduction. If the reduced size is zero, the system removes the order from the market. If the size remains positive but drops below the minimal acceptable size Min, the order is also removed.

4 Indexing structure

The exchange system includes a central structure for indexing of orders with fully specified items, which do not have ranges, standard sets, intersections, or unions. If we can put an order into this structure, we call it an *index order.* If an order includes a set of items, rather than a fully specified item, it is added to an unordered list of *nonindex orders.* This indexing scheme allows fast retrieval of index orders that match a given order; however, the system does not identify matches between two nonindex orders.

Indexing tree. The indexing structure consists of two

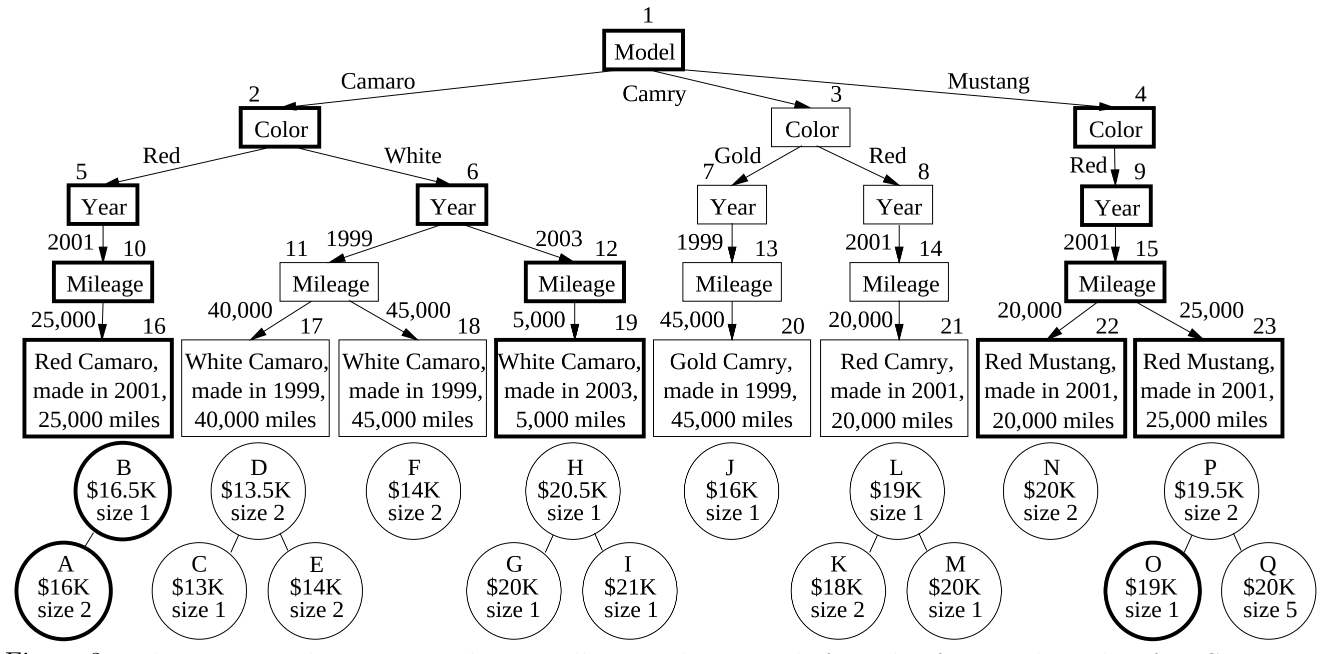


Figure 3: Indexing tree with seventeen orders. We illustrate the retrieval of matches for an order to buy four Camaros or Mustangs made after 2000. We show the matching nodes by thick boxes, and the retrieved orders by thick circles.

identical trees: one is for buy orders, and the other is for sell orders. In Figure 3, we show a tree for sell orders; its height equals the number of attributes, and each level corresponds to one of the attributes. The root node encodes the first attribute, and its children represent different values of this attribute. The nodes at the second level divide the orders by the second attribute, and each node at the third level corresponds to specific values of the first two attributes. In general, a node at level i divides orders by the values of the ith attribute, and each node at level $(i+1)$ corresponds to all orders with specific values of the first i attributes. If some items are not currently on sale, the tree does not include the corresponding nodes.

Every nonleaf node includes a red-black tree that allows fast retrieval of its children with given values; for example, the root node in Figure 3 includes a red-black tree that indexes its children by model values.

Standard sets. If a market includes standard sets of values, such as "all sports cars" and "all American cars," traders can use them in their orders. For every attribute, the system maintains a central table of standard sets, which consists of two parts (Figure 4a). The first part includes a sorted list of values for every set; it allows determining whether a given value belongs to a specific set, by the binary search in the corresponding list. The second part includes all values that belong to at least one set; for each value, we store a sorted list of sets that contain it.

Every node of an indexing tree also includes a table of standard sets; for example, the root node in Figure 3 contains a table of sets for the first attribute (Figure 4b). Every set in the table includes a list of pointers to its elements in the node's red-black tree; for instance, `American-Cars` points to `Camaro` and `Mustang`.

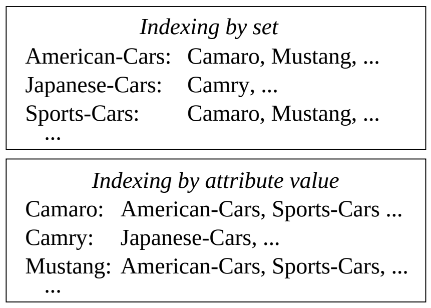
Basic operations. The nonleaf-node structure (Figure 4b) supports fast addition and deletion of children, as well as fast retrieval operations:

- Retrieval of a child with a given attribute value.
- Retrieval of all children in a given standard set.
- Retrieval of all children in a given range.

The system can also retrieve all children that belong to an intersection or union of several attribute sets. Given an intersection, the system identifies the matching children for one of its elements, and then prunes the children that do not belong to the other elements. Given a union, the system identifies matching children for each of its elements, and then generates the union of the resulting sets of children.

Leaf nodes. A leaf node includes orders with identical items, which are sorted by price, from the best to the worst; that is, the system sorts buy orders from the highest to the lowest price limit, and sell orders from the lowest to the highest price. We use a red-black tree to maintain this sorting, which allows fast insertion and deletion of orders.

Time stamps. The system keeps track of the "age" of each order, and uses it to avoid repetitive search for matches among the same index orders. Every order has two time stamps; the first is the time of placing the order, and the second is the time of the last search for matches. Furthermore, for each node of the indexing tree, the system keeps the time of the last addition of an order to the corresponding subtree. If the system repeats the search for matches for some order, it skips



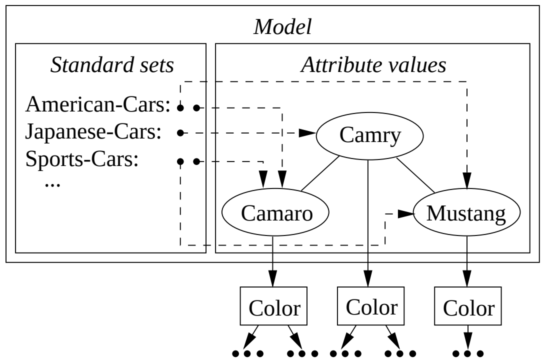
Indexing by set

American-Cars:	Camaro, Mustang, ...
Japanese-Cars:	Camry, ...
Sports-Cars:	Camaro, Mustang, ...
...	

Indexing by attribute value

Camaro:	American-Cars, Sports-Cars ...
Camry:	Japanese-Cars, ...
Mustang:	American-Cars, Sports-Cars, ...
...	

(a) Central table of standard sets.



(b) Standard sets in a node of the indexing tree.

Figure 4: Standard sets of values. The system includes a central table of sets (a), and every node in the indexing tree includes a table of sets for the respective attribute (b).

the subtrees that have no new orders (see Section 5).

Adding and deleting an order. When a trader places an index order, the system adds it to the corresponding leaf; for example, if a dealer places an order to sell a red Camaro, made in 2001, with 25,000 miles, the system adds it to node 16 in Figure 3. If the leaf is not in the tree, the system adds the appropriate new branch. After adding an order, it updates the time stamps of the ancestor nodes; for instance, if it adds a new order to node 16, then it updates the time stamps of nodes 1, 2, 5, 10, and 16.

When the system fills an index order, or a trader cancels her old order, the system removes the order from the corresponding leaf. If the leaf does not include other orders, the system deletes it from the tree; if the deleted node is the only leaf in some subtree, the system removes this subtree. For example, the deletion of order J in Figure 3 leads to the removal of nodes 7, 13, and 20.

5 Search for matches

The system alternates between processing new orders and finding matches for old nonindex orders (Figure 5). When it receives a new order, it immediately identifies matching index orders. After processing all new orders, it tries to fill old nonindex orders; for each nonindex order, it identifies matching index orders.

We give a two-step algorithm that identifies matches for a given order; it first finds the indexing-tree leaves that match the item set of the given order, and then selects the highest-quality matches in these leaves. In Figure 6, we present the notation for the order and leaf-node structures used by the matching algorithm; in Figures 7 and 8, we give pseudocode for the two main steps of the algorithm.

Matching leaves. The algorithm in Figure 7 retrieves matching leaves for the item set of a given order. Recall that we represent the item set by a union of Cartesian products.

The DFS subroutine finds all matches for one Cartesian product using depth-first search in the indexing tree; it identifies all children of the root that match the first element of the Cartesian product, and then recursively processes the respective subtrees. For example, suppose that a buyer is looking for a Mustang or Camaro made after 2000, with any color and mileage, and the tree of sell orders is as shown in Figure 3. The subroutine determines that nodes 2 and 4 match the model, and processes the two respective subtrees. It identifies three matching nodes for the second attribute, three nodes for the third attribute, and finally four matching leaves; we show these nodes by thick boxes.

If the system already tried to find matches for a given order during the previous execution of the main loop, it skips the subtrees that have not been modified since the previous search. If the order includes a union of several Cartesian products, the system calls the DFS subroutine for each product.

Best matches. After the system identifies matching leaves, it selects the best matching orders in these leaves, according to the quality function of the given order. In Figure 8, we give an algorithm that identifies the highest-quality matches and completes the respective trades. It arranges the leaves in a priority queue by the quality of the best unprocessed match in a leaf. At each step, it processes the best available match; it terminates after it fills the given order or runs out of matches.

For example, consider the tree in Figure 3, and suppose that a buyer places an order for four Mustangs or Camaros made after 2000. We suppose further that she uses the default quality function, which depends only on price. The system first retrieves order A, with price \$16,000 and size 2, then order B with price \$16,500, and finally order O with price \$19,000; we show these orders by thick circles.

6 Experiments

We describe experiments with artificial market data and with a used-car market. A more detailed report of these experiments is available in Johnson's masters thesis [10]. We have run the system on a 2-GHz Pentium computer with one-gigabyte memory.

We have implemented an artificial market setup that allows control over the number of orders, number of at-

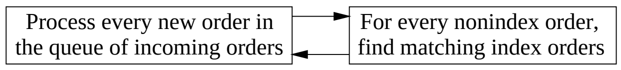

Process every new order in the queue of incoming orders
For every nonindex order, find matching index orders

Figure 5: Main loop of the system.

Elements of the order structure:

Price[*order*]	price function
Qual[*order*]	quality function
Max[*order*]	overall order size
Min[*order*]	minimal acceptable size
Step[*order*]	size step
Place-Time[*order*]	time of placing the order
Search-Time[*order*]	time of the last search for matches

Elements of the leaf-node structure:

Item[*leaf*]	item in the leaf's orders
Order[*leaf*]	best-price unprocessed order in the leaf
Quality[*leaf*]	quality of the best-price unprocessed order

Figure 6: Notation for the main elements of the structures that represent an order and a leaf node. We use this notation in the matching-algorithm pseudocode in Figures 7 and 8.

```
MATCHING-LEAVES(order, root)
The algorithm inputs an order and the root of an indexing tree.
It returns the leaves that match the item set of the order.

Initialize an empty set of matching leaves, denoted leaves
For each Cartesian product $I_1 \times I_2 \times ...$ in order's item set:
    Call DFS($I_1 \times I_2 \times ...$, Search-Time[order], root, 1, leaves)
Return leaves
```

```
DFS($I_1 \times I_2 \times ...$, Search-Time, node, k, leaves)
The subroutine inputs a Cartesian product $I_1 \times I_2 \times ...$,
the previous-search time, a node of the indexing tree,
the node's depth in the tree, and a set of leaves.
It finds the matching leaves of the subtree rooted at
the given node, and adds them to the set of leaves.

If Search-Time is larger than node's last-addition time,
    then terminate
If node is a leaf, then add node to leaves
If node is not a leaf, then:
    Identify all children of node that match $I_k$
    For each matching child:
        Call DFS($I_1 \times I_2 \times ...$, Search-Time, child, $k+1$, leaves)
```

Figure 7: Retrieval of matching leaves. The algorithm identifies the leaves of an indexing tree that match the item set of a given order. The DFS subroutine uses depth-first search to retrieve matching leaves for one Cartesian product.

```
BEST-MATCHES(order, leaves)
The algorithm inputs a given order and matching leaves.
It identifies the best matches for the order in these leaves.

Initialize an empty priority queue of matching leaves,
    denoted queue, which prioritizes the leaves by the
    quality of the best-price unprocessed order
For each leaf in leaves:
    Set Order[leaf] to the first order among leaf's orders,
        sorted by price
    Call LEAF-PRIORITY(order, leaf, queue)
While Max[order] $\geq$ Min[order] and queue is nonempty:
    Set leaf to the highest-priority leaf in queue,
        and remove it from queue
    match := Order[leaf]
    Set Order[leaf] to the next order among leaf's orders,
        sorted by price
    Call TRADE(order, match)
    Call LEAF-PRIORITY(order, leaf, queue)
If Max[order] < Min[order],
    then remove order from the market
Else, set Search-Time[order] to the current time
```

```
LEAF-PRIORITY(order, leaf, queue)
The subroutine inputs the given order, a matching leaf, and the
priority queue of leaves. If the order's price matches the price of
the leaf's best-price unprocessed order, then the leaf is added
to the queue.

match := Order[leaf]
If match = NONE, then terminate (no more orders in leaf)
i = Item[leaf]
If the price of order does not match the price of match for i,
    then terminate
Else, $p := (Price[order](i) + Price[match](i))/2$
Quality[leaf] := Qual[order](Item[leaf], p)
Add leaf to queue, prioritized by Quality
```

```
TRADE(order, match)
The subroutine inputs the given order and the highest-quality
order with matching item and price. If the sizes of these two
orders match, it completes the trade between them.

If Search-Time[order] > Place-Time[match], then terminate
size := FILL-SIZE(Max[order], Min[order], Step[order],
                  Max[match], Min[match], Step[match])
If size = NONE, then terminate
Complete the trade between order and match
Max[order] := Max[order] − size
Max[match] := Max[match] − size
If Max[match] < Min[match],
    then remove match from the market
```

Figure 8: Retrieval of matching orders. The algorithm finds the best matches for a given order and completes the corresponding trades. The LEAF-PRIORITY subroutine adds a given leaf to the priority queue, arranged by the quality of a leaf's best-price unprocessed match. The TRADE subroutine completes the trade between the given order and the best available match.

tributes in an item description, and number of values per attribute. We have varied the number of orders from one to 300,000, which is the maximal possible number for one-gigabyte memory. We have randomly generated these orders, which include an equal number of buy and sell orders. We have considered markets with one, three, ten, thirty, and one hundred attributes, and we have varied the number of values per attribute from two to 1,000.

For each setting of the control variables, we have measured the main-loop time and throughput. The *main-loop time* is the time of one pass through the system's main loop (Figure 5), which includes processing new orders and matching old nonindex orders. The *throughput* is the maximal acceptable rate of placing new orders; if the system gets more orders per second, the number of unprocessed orders keeps growing and eventually leads to an overflow.

We give the dependency of the system's performance on the control variables in Figures 9–11; the scales of all graphs are logarithmic. In Figure 9, we show how the performance changes with the number of orders. The main-loop time is approximately linear in the number of orders. The throughput in small markets grows with the number of orders; it reaches a maximum when the market grows to about two hundred orders, and slightly decreases with further increase in the market size. In Figure 10, we give the dependency of the performance on the number of attributes. The main-loop time is super-linear in the number of attributes, whereas the throughput is in inverse proportion to the same super-linear function. In Figure 11, we show that the main-loop time grows sub-linearly with the number of values per attribute, and the throughput slightly decreases with an increase in the number of values.

We have also experimented with a used-car market described by eight attributes: transmission, number of doors, interior color, exterior color, year, model, option package, and mileage. In Figure 12, we show the dependency of the performance on the number of orders. The system scales to markets with 300,000 orders, and it processes 500 to 5,000 new orders per second.

7 Concluding remarks

The modern economy includes a variety of marketplaces, and the Internet has led to the development of new efficient markets. Computer scientists have studied algorithms for various auctions and standardized exchanges, but they have done little work on exchange markets for complex nonstandard goods. The reported work is a step toward the development of automated exchanges for nonstandard goods. We have built an exchange system that allows constraints and preference functions in the description of orders, supports markets with up to 300,000 orders, and processes hundreds of orders per second.

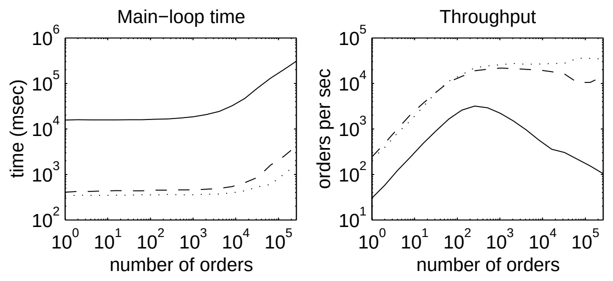


Figure 9: Dependency of the performance on the *number of orders.* We consider three different settings of the control variables, which correspond to dotted, dashed, and solid lines. The dotted lines show experiments with one attribute and two values per attribute. The dashed lines are for three attributes and sixteen values per attribute. The solid lines are for ten attributes and 1,000 values per attribute.

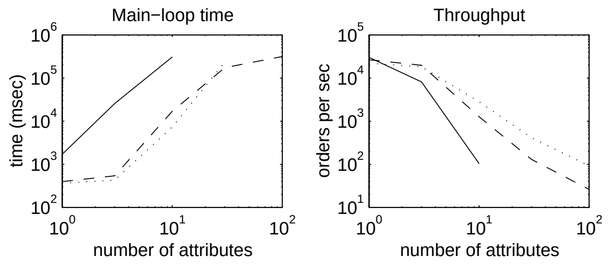


Figure 10: Dependency of the performance on the *number of attributes.* The dotted lines show experiments with two values per attribute and 300 orders. The dashed lines are for sixteen values per attribute and 10,000 orders. The solid lines are for 1,000 values per attribute and 300,000 orders. We do not plot solid lines for thirty and one hundred attributes, because we have not been able to run the corresponding experiments, which would require more than one-gigabyte main memory.

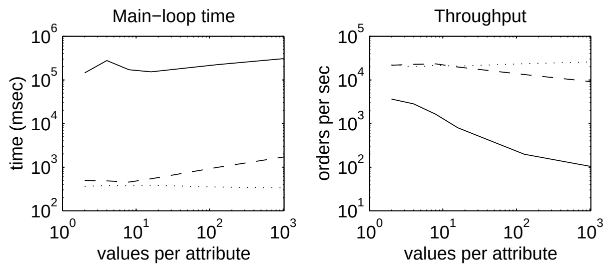


Figure 11: Dependency of the performance on the *number of values per attribute.* The dotted lines show experiments with one attribute and 300 orders. The dashed lines are for three attributes and 10,000 orders. The solid lines are for ten attributes and 300,000 orders.

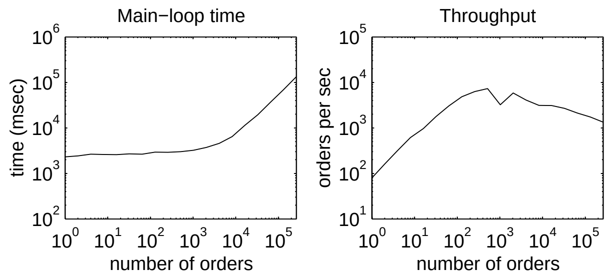


Figure 12: Dependency of the performance on the number of orders for the used-car market with eight attributes.

Acknowledgments. We are grateful to Hong Tang and Jianli Gong for their help in preparing this article, and to Savvas Nikiforou for his help with software and hardware installations. We thank Ganesh Mani, Dwight Dietrich, Steve Fischetti, Michael Foster, and Alex Gurevich for their feedback and help in understanding real-world exchanges. This work has been partially sponsored by the DYNAMiX Technologies Corporation and by the National Science Foundation grant No. EIA-0130768.

References

[1] Arne Andersson, Mattias Tenhunen, and Fredrik Ygge. Integer programming for combinatorial auction winner determination. In *Proceedings of the Fourth International Conference on Multi-Agent Systems*, pages 39–46, 2000.

[2] Martin Bichler. An experimental analysis of multi-attribute auctions. *Decision Support Systems*, 29(3):249–268, 2000.

[3] Yeon-Koo Che. Design competition through multi-dimensional auctions. RAND *Journal of Economics*, 24(4):668–680, 1993.

[4] Martin Cripps and Norman Ireland. The design of auctions and tenders with quality thresholds: The symmetric case. *Economic Journal*, 104(423):316–326, 1994.

[5] Eugene Fink, Joshua Marc Johnson, and Jonathan Hershberger. Multi-attribute exchange market: Theory and experiments. In *Proceedings of the Sixteenth Canadian Conference on Artificial Intelligence*, pages 603–610, 2003.

[6] Rica Gonen and Daniel Lehmann. Optimal solutions for multi-unit combinatorial auctions: Branch and bound heuristics. In *Proceedings of the Second* ACM *Conference on Electronic Commerce*, pages 13–20, 2000.

[7] Rica Gonen and Daniel Lehmann. Linear programming helps solving large multi-unit combinatorial auctions. In *Proceedings of the Electronic Market Design Workshop*, 2001.

[8] Jianli Gong. Exchanges for complex commodities: Search for optimal matches. Master's thesis, Department of Computer Science and Engineering, University of South Florida, 2002.

[9] Jenny Ying Hu. Exchanges for complex commodities: Representation and indexing of orders. Master's thesis, Department of Computer Science and Engineering, University of South Florida, 2002.

[10] Joshua Marc Johnson. Exchanges for complex commodities: Theory and experiments. Master's thesis, Department of Computer Science and Engineering, University of South Florida, 2001.

[11] Jayant R. Kalagnanam, Andrew J. Davenport, and Ho S. Lee. Computational aspects of clearing continuous call double auctions with assignment constraints and indivisible demand. Technical Report RC21660(97613), IBM, 2000.

[12] Ran Lavi and Noam Nisan. Competitive analysis of incentive compatible on-line auctions. In *Proceedings of the Second* ACM *Conference on Electronic Commerce*, pages 233–241, 2000.

[13] Benny Lehmann, Daniel Lehmann, and Noam Nisan. Combinatorial auctions with decreasing marginal utilities. In *Proceedings of the Third* ACM *Conference on Electronic Commerce*, pages 18–28, 2001.

[14] Noam Nisan. Bidding and allocation in combinatorial auctions. In *Proceedings of the Second* ACM *Conference on Electronic Commerce*, pages 1–12, 2000.

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

[16] Tuomas W. Sandholm. Approach to winner determination in combinatorial auctions. *Decision Support Systems*, 28(1–2):165–176, 2000.

[17] Tuomas W. Sandholm and Subhash Suri. Improved algorithms for optimal winner determination in combinatorial auctions and generalizations. In *Proceesings of the Seventeenth National Conference on Artificial Intelligence*, pages 90–97, 2000.

[18] Tuomas W. Sandholm and Subhash Suri. Market clearability. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 1145–1151, 2001.

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

[20] Peter R. Wurman, William E. Walsh, and Michael P. Wellman. Flexible double auctions for electronic commerce: Theory and implementation. *Decision Support Systems*, 24(1):17–27, 1998.