Multi-Attribute Exchange Market: Search for Optimal Matches*

Eugene Fink
Language Technologies
Carnegie Mellon University
Pittsburgh, PA 15213
e.fink@cs.cmu.edu
www.cs.cmu.edu/∼eugene

Jianli Gong
Computer Science and Engineering
University of South Florida
Tampa, FL 33620
jlgong@csee.usf.edu

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

Abstract – *We describe an exchange system for trading complex goods, such as used cars or nonstandard financial securities. The system allows traders to represent their buy and sell orders by multiple attributes, provide complex price constraints, and specify preferences among acceptable trades. We describe a technique for fast identification of most preferable matches between buy and sell orders, which maximize trader satisfaction.*

Keywords: E-commerce, exchange markets, indexing and retrieval, best-first search.

1 Introduction

The growth of the Internet has led to the development of electronic markets, which include bulletin boards, auctions, and exchanges. Electronic bulletin boards help buyers and sellers find each other; however, they often require customers to invest time into reading multiple ads, and many buyers prefer on-line auctions, such as eBay (www.ebay.com). Auctions have their own problems, including significant computational costs, lack of liquidity, and asymmetry between buyers and sellers. Exchange-based markets support fast-paced trading and ensure symmetry between buyers and sellers, but 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 must indicate a specific stock. For most goods, the description of desirable trades is more complex, and an exchange market for consumer goods should allow complex constraints in specifications of buy and sell orders. For instance, a car buyer needs to specify a model, options, color, and other features of a desirable vehicle.

Economists and computer scientists have long realized the importance of auctions and exchanges, and studied a variety of trading models. The related computer science research has led to successful Internet auctions, such as eBay (www.ebay.com) and Yahoo Auctions (auctions.yahoo.com), as well as on-line exchanges, such as Island (www.island.com) and NexTrade (www.nextrade.org). Researchers have also developed efficient systems for combinatorial auctions, which allow the trading of sets of goods rather than individual items [Lavi and Nisan, 2000; Nisan, 2000; Sandholm, 2000; Sandholm and Suri, 2001]. In particular, Rothkopf *et al.* [1998] gave a detailed analysis of combinatorial auctions and described semantics of combinatorial bids that allowed fast matching. Gonen and Lehmann [2000, 2001] studied branch-and-bound heuristics for processing combinatorial bids and integrated them with linear programming. Andersson *et al.* [2000] compared the main techniques for combinatorial auctions and proposed an integer-programming representation that allowed richer bid semantics. Computer scientists have also studied exchange markets; in particular, Wurman *et al.* [1998] built a general-purpose system for auctions and exchanges, Sandholm and Suri [2000] developed an exchange for combinatorial orders, and Kalagnanam *et al.* [2000] investigated techniques for placing orders with complex constraints.

A recent project at the University of South Florida has been aimed at developing an exchange for complex goods. Johnson [2001] and Hu [2002] have defined related trading semantics and constructed indexing structures for fast identification of matches between buyers and sellers [Fink *et al.*, 2004]. We have continued this work and built a system that identifies the most preferable matches, which maximize trader satisfaction. We define a market for complex goods (Section 2), describe a technique for identifying most preferable matches (Section 3), and show how its performance depends on the market size (Section 4).

2 General exchange model

We describe a general model of trading complex goods, and illustrate it with an example 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 the desired vehicle and a maximal acceptable price; for instance, she may indicate that she wants a red Mustang, made after 2001, with at most 20,000 miles, and she is willing to pay \$19,000. Similarly, a seller can place a *sell order;* for instance, a dealer may offer a brand-new Mustang of any color for \$18,000. An exchange system must generate transactions that satisfy both buyers and sellers; in the car example, it must determine that a brand-new red Mustang for \$18,500 satisfies both the buyer and dealer.

Attributes. A specific market includes a certain set of items that can be bought and sold, defined by a list of attributes. As a simplified example, we describe a car by four attributes: model, color, year, and mileage. An attribute may be a set of explicitly listed values, such as car model; an interval of integers, such as year; or an interval of real values, such as mileage.

When a trader makes a purchase or sale, she has to specify a set of acceptable values for each attribute. 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 $I_1 \times I_2 \times ... \times I_n$. 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 2001, and it should have at most 20,000 miles; then, the item set is $I = \{\texttt{Mustang}, \texttt{Camaro}\} \times \{\texttt{red}, \texttt{white}\} \times [2002..2004] \times [0..20{,}000]$. A trader can use specific values or ranges for each attribute; for instance, she can specify a year as 2004 or as a range from 2002 to 2004. She can also specify a list of several values or ranges; for example, she can specify colors as $\{\texttt{red}, \texttt{white}\}$, and years as $\{[1901..1950], [2002..2004]\}$.

A trader can also define an item set I as the union of several Cartesian products. For example, if she needs either a used red Mustang or a new red Camaro, she can specify the set $I = \{\texttt{Mustang}\} \times \{\texttt{red}\} \times [2002..2004] \times [0..20{,}000] \ \cup \ \{\texttt{Camaro}\} \times \{\texttt{red}\} \times \{2004\} \times [0..200]$.

Price functions. A trader should specify a limit on the acceptable price; for instance, a buyer may be willing to pay \$18,500 for a Mustang, but only \$17,500 for a Camaro, and offer an extra \$500 if a car is red. Formally, a price limit is a real-valued function defined on the set I; for each item $i \in I$, it gives a certain limit $Price(i)$. If a price function is a constant, it is specified by a numeric value; else, it is a C++ procedure that inputs an item and outputs the corresponding limit.

Order sizes. If a trader wants to buy or sell several identical items, she can include their number in the order, which is called an *order size*. For instance, if a dealer is selling ten identical cars, the size of her order is ten. To summarize, an order includes three elements:

- Item set, $I = I1_1 \times ... \times I1_n \cup ... \cup Ik_1 \times ... \times Ik_n$.
- Price function, $Price\colon I \to \mathbf{R}$.
- Order size, *Size*.

FILL-PRICE($Price_b, Price_s, i$)
The function inputs the prices of a buy and sell order, and an item i that matches both orders.

If $Price_b(i) \geq Price_s(i)$,
    then return $(Price_b(i) + Price_s(i))/2$
Else, return NONE (no acceptable price)

Figure 1: Finding the fill price for two matching orders.

Fills. When a buy order matches a sell order, the respective parties can complete a trade; we use the term *fill* to refer to the traded items and their price. We define a fill by a specific item i, its price p, and the number of purchased items, denoted *size*. If $(I_b, Price_b, Size_b)$ is a buy order, and $(I_s, Price_s, Size_s)$ is a matching sell order, then a fill satisfies the following conditions:

- $i \in I_b \cap I_s$.
- $Price_s(i) \leq p \leq Price_b(i)$.
- $size = \min(Size_b, Size_s)$.

If the buyer's price limit is larger than the seller's limit, we split the difference between the buyer and seller; in Figure 1, we give a function that finds the price of a fill.

Quality functions. Buyers and sellers may have preferences among acceptable trades, which depend on a specific item i and its price p; for instance, a buyer may prefer a Mustang for \$18,000 to a Camaro for \$17,000. We represent preferences by a real-valued function $Qual(i, p)$, encoded by a C++ procedure, which 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 trading i_2 at p_2. 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 $p_1 \leq p_2$, then, for every item i, $Qual_b(i, p_1) \geq Qual_b(i, p_2)$.

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

We do not require a trader to specify a quality function for each order; by default, the quality 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:* $Qual_s(i, p) = (p - Price(i))/Price(i)$.

Monotonic attributes. The value of goods may monotonically depend on some of their numeric attributes; for example, the quality of a car decreases with an increase in mileage. When an attribute has this property, we say that it is *monotonically decreasing.* To formalize this concept, suppose that a market has n attributes, consider the mth attribute, and assume that it is either integer or real-valued. We denote attribute values of a given item by $i_1, ..., i_m, ..., i_n$, and a transaction price by p. The mth attribute is monotonically decreasing if all price and quality functions satisfy the following constraints:

- *Price monotonicity:*
If *Price* is price for a buy or sell order, and $i_m \leq i'_m$, then, for every two items $(i_1,.., i_{m-1}, i_m, i_{m+1},.., i_n)$ and $(i_1, .., i_{m-1}, i'_m, i_{m+1}, .., i_n)$, we have $\mathit{Price}(i_1, .., i_m, .., i_n) \geq \mathit{Price}(i_1, .., i'_m, .., i_n)$.

- *Buy monotonicity:*
If Qual_b is quality for a buy order, and $i_m \leq i'_m$, then, for every two items $(i_1,.., i_{m-1}, i_m, i_{m+1},.., i_n)$ and $(i_1, .., i_{m-1}, i'_m, i_{m+1}, .., i_n)$, and every price p, $\mathit{Qual}_b(i_1, .., i_m, .., i_n, p) \geq \mathit{Qual}_b(i_1, .., i'_m, .., i_n, p)$.

- *Sell monotonicity:*
If Qual_s is quality for a sell order, and $i_m \leq i'_m$, then, for every two items $(i_1,.., i_{m-1}, i_m, i_{m+1},.., i_n)$ and $(i_1, .., i_{m-1}, i'_m, i_{m+1}, .., i_n)$, and every price p, $\mathit{Qual}_s(i_1, .., i_m, .., i_n, p) \leq \mathit{Qual}_s(i_1, .., i'_m, .., i_n, p)$.

Similarly, if the quality of goods grows with an increase in an attribute value, we say that the attribute is *monotonically increasing;* for example, the quality of a car increases with the year of production.

3 Search for matches

The exchange system includes a central structure for indexing of orders with fully specified items. 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, the system adds it to an unordered list of *nonindex orders.* In Figure 2, we show the system's main loop, which alternates between processing new orders and identifying matches for old nonindex orders. When the system receives a new order, it immediately searches for matching index orders. If there are no matches, and the new order is an index order, then the system adds it to the indexing structure. If it gets a nonindex order and does not find a fill, it adds the order to the list of nonindex orders. After processing all new orders, the system tries to fill old nonindex orders; for each nonindex order, it identifies matching index orders.

Indexing structure. The indexing structure includes two identical trees: one is for buy orders, and the other is for sell orders. In Figure 3, we show a tree for sell

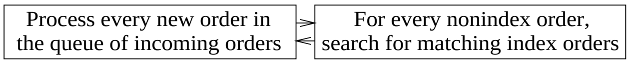


Figure 2: Main loop of the system.

orders; its depth 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 depth 1 divide the orders by the second attribute, and each node at depth 2 corresponds to specific values of the first two attributes. In general, a node at depth $(i - 1)$ divides orders by the values of the ith attribute, and each node at depth i corresponds to all orders with specific values of the first i attributes. Every leaf includes orders with identical items, 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.

The nodes include summary data that help to retrieve matching orders. Every node contains the following data about the orders in the corresponding subtree:

- Minimal and maximal price of orders in the subtree.
- Min and max value for each monotonic attribute.
- Time of the latest addition of an order to the subtree.

The system also 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.

Best-first search. The system identifies optimal matches for an order by best-first search in an indexing tree. It uses a node's summary data to estimate the quality of matches in the node's subtree; at each step, it processes the node with the highest quality estimate. We present the notation for the order and node structures in Figure 4, and give pseudocode for the best-first search in Figures 5–8.

The system computes a quality estimate for a node only if all branching in the node's subtree is on monotonic attributes; a node with this property is called *monotonic.* For example, node 6 in Figure 3 is monotonic; the branching in its subtree is on year and mileage, which are monotonic attributes. On the other hand, node 2 is not monotonic because its subtree includes branching on color. In Figure 5, we give a function that inputs a monotonic node and constructs the best possible item that may be present in the node's subtree based on the summary data. To estimate the node's quality, the system computes the quality of this item traded at the best possible price from the summary data. For example, consider node 6 in Figure 3; all orders in its subtree include white Camaros, and the summary data show that the best year is 2004, and the

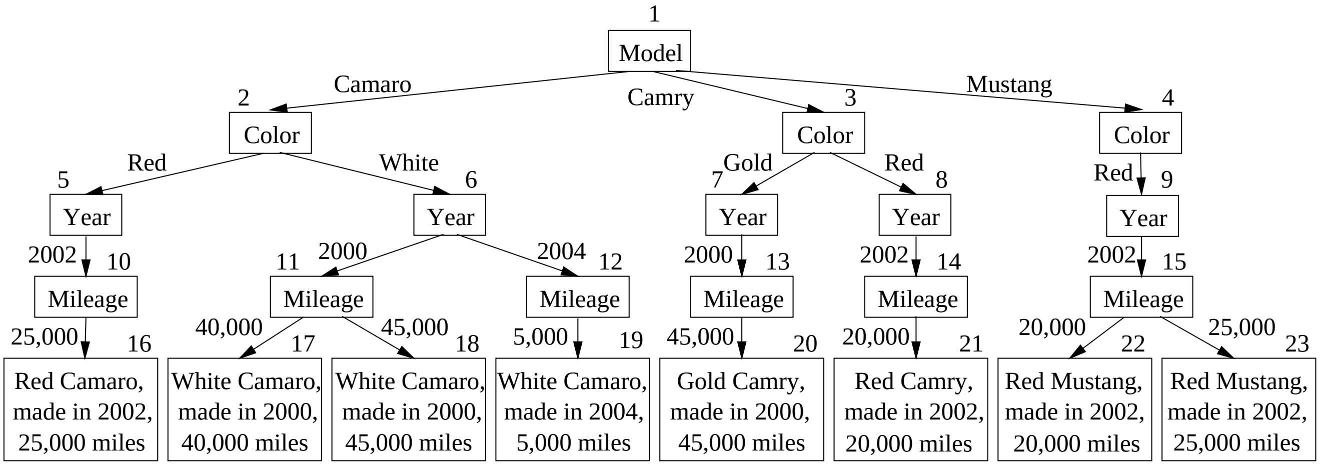


Figure 3: Example of an indexing tree for a used-car market.

Elements of the order structure:

Price[*order*]	price function
Qual[*order*]	quality function
Size[*order*]	order size
Place-Time[*order*]	time of placing the order
Search-Time[*order*]	time of the last search for matches

Elements of the indexing-tree node structure:

Min-Price[*node*]	min price of orders in *node*'s subtree
Max-Price[*node*]	max price of orders in *node*'s subtree
Depth[*node*]	depth of *node* in the indexing tree
Product-Num[*node*]	number of the matching Cartesian product in a given item set
Quality[*node*]	for nonleaf node, the quality estimate; for leaf node, the quality of the best-price unprocessed order

Additional elements of the leaf-node structure:

Item[*node*]	item in *node*'s orders
Current-Order[*node*]	best-price unprocessed order in *node*

Figure 4: Notation for the main elements of the structures that represent orders and nodes of an indexing tree. Note that the leaf-node structure includes the five elements of the node and two additional elements. We use this notation in the pseudocode in Figures 5–8.

best mileage is 5,000. If we suppose that the best order price in this subtree is \$14,000, then the quality estimate is $Qual(\texttt{Camaro}, \texttt{white}, 2004, 5{,}000, \$14{,}000)$.

The retrieval of matches for a given order involves two steps. First, the system finds all smallest-depth monotonic nodes that match the given order (Figure 6); for example, if a buyer is looking for a Camaro or Mustang made after 2001, and the tree of sell orders is as shown in Figure 3, then the system retrieves nodes 5, 6, and 9. Second, it finds the best matching orders in the subtrees of the selected nodes; we give a procedure for this step in Figure 7, and its two main subroutines in Figure 8. The procedure arranges the nodes into a priority queue by their quality estimates; at each iteration, it processes the highest-quality node. If the node is not a leaf, the procedure identifies its children that match the given order and adds them to the priority queue. If the node is a leaf, it determines the best-price matching order in the leaf and completes the respective trade. The search terminates when the system fills the given order or runs out of matches.

4 Performance

We describe experiments with a used-car market and corporate-bond market, on a 2-GHz Pentium computer with 1-Gigabyte memory. A more detailed report of these experiments is available in Gong's [2002] and Hershberger's [2003] masters theses. First, we have considered a used-car market that includes all models offered by AutoNation (www.autonation.com), described by eight attributes: transmission (2 values), number of doors (3 values), interior color (7 values), exterior color (52 values), model (257 values), year (104 values), option package (1,024 values), and mileage (500,000 values). Second, we have experimented with corporate bonds, described by two attributes: issuing company (5,000 values) and maturity date (2,550 values).

We have varied the number of orders in the market from four to 300,000, which have included an equal number of buy and sell orders. We have also varied the *matching density,* which is the mean percentage of sell orders that match a buy order; in other words, it is the probability that a randomly selected buy order matches a randomly chosen sell order. We have experimented with three matching-density values: 0.001, 0.01, and 1.

For each setting of control variables, we have measured the time of one pass through the system's main loop (Figure 2), which includes processing new orders and matching old orders. We have also measured the processing speed, that is, the average number of orders processed per second; if the system gets more orders per

BEST-ITEM($node$)
The function inputs a monotonic node of an indexing tree.

For m from 1 to $Depth[node]$:
  Set i_m to the mth-attribute value on the path from the root to $node$
For m from $Depth[node] + 1$ to n:
  Set i_m to the best value of the mth attribute in $node$'s summary data
Return $(i_1, ..., i_n)$

Figure 5: Construction of the best possible item. The function inputs a monotonic node and generates the best item that may be present in the subtree rooted at this node.

MATCHING-NODES($order$, $root$)
The procedure inputs an order and the root of an indexing tree.
We denote the order's item set by $I1_1 \times ... \times I1_n \cup ... \cup Ik_1 \times ... \times Ik_n$.

Initialize an empty set of matching monotonic nodes, denoted $nodes$
For l from 1 to k, call PRODUCT-NODES($Il_1 \times ... \times Il_n$, $Search\text{-}Time[order]$, $root$, $nodes$)
Return $nodes$

PRODUCT-NODES($Il_1 \times ... \times Il_n$, $Search\text{-}Time$, $node$, $nodes$)
The subroutine inputs a Cartesian product $Il_1 \times ... \times Il_n$, the previous-search time, a node of the indexing tree, and a set of monotonic nodes. It finds the matching monotonic nodes in the subtree rooted at the given node, and adds them to the set of monotonic nodes.

If $Search\text{-}Time$ is larger than $node$'s time of the last order addition, then terminate
If $node$ is monotonic:
  $Product\text{-}Num[node] := l$
  Add $node$ to $nodes$
If $node$ is not monotonic:
  Identify all children of $node$ that match $Il_{Depth[node]+1}$
  For each matching $child$, call PRODUCT-NODES($Il_1 \times ... \times Il_n$, $Search\text{-}Time$, $child$, $nodes$)

Figure 6: Retrieval of matching monotonic nodes. The procedure identifies the smallest-depth monotonic nodes that match the item set of a given order. The PRODUCT-NODES subroutine uses depth-first search to retrieve the matching monotonic nodes for one Cartesian product.

second, the number of unprocessed orders keeps growing, and the system eventually has to reject some of them. We show the results in Figure 9; the system processes 200 to 20,000 orders per second in the used-car market, and 500 to 50,000 orders per second in the corporate-bond market.

5 Concluding remarks

The reported work is a step toward the development of exchange markets for complex nonstandard goods. We have represented complex goods by multiple attributes, allowed price and quality functions in the description of orders, and developed a system for identifying highest-quality matches between buy and sell orders. The system supports markets with up to 300,000 orders on a 2-GHz computer with 1-Gigabyte memory; it keeps all orders in the main memory, and its scalability is limited by the available memory.

The related open problems include richer semantics for order description, advanced indexing structures, and improved scalability. We plan to develop semantics for trading different types of goods in the same market, and for buying and selling sets of goods rather than individual items; for example, a customer may need to buy a car and trailer together, in the same market. We are also working on more effective data structures, which will allow finding matches between nonindex orders.

Acknowledgments

We are grateful to Hong Tang and Jenny Hu, who helped to prepare, format, and proofread this article. We are also grateful to Savvas Nikiforou for his extensive 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 PowerLoom Corporation and by the National Science Foundation grant No. EIA-0130768.

NODE-MATCHES(*order*, *nodes*)

The procedure inputs an order and matching monotonic nodes of an indexing tree.

Initialize an empty priority queue of matching nodes, denoted *queue*,
  which prioritizes the nodes by their quality estimates
For each *node* in *nodes*, call NODE-PRIORITY(*order*, *node*, *queue*)
While *queue* is nonempty:
  Set *node* to the highest-priority node in *queue*, and remove it from *queue*
  If *node* is a leaf:
    $match := Current\text{-}Order[node]$
    Set $Current\text{-}Order[node]$ to the next order among *node*'s orders, sorted by price
    Complete the trade between *order* and *match*
    If $Size[order] = Size[match]$, then remove *order* and *match* from the market and terminate
    If $Size[order] < Size[match]$:
      $Size[match] := Size[match] - Size[order]$
      Remove *order* from the market and terminate
    If $Size[order] > Size[match]$:
      $Size[order] := Size[order] - Size[match]$
      Remove *match* from the market, and call LEAF-PRIORITY(*order*, *node*, *queue*)
  If *node* is not a leaf:
    $l := Product\text{-}Num[node]$
    Identify all children of *node* that match $Il_{Depth[node]+1}$
    For each matching *child*:
      If *child* is a leaf:
        Set $Current\text{-}Order[child]$ to the first order among *child*'s orders, sorted by price
        Call LEAF-PRIORITY(*order*, *child*, *queue*)
      If *child* is not a leaf:
        $Product\text{-}Num[child] := l$
        Call NODE-PRIORITY(*order*, *child*, *queue*)
Set $Search\text{-}Time[order]$ to the current time

Figure 7: Retrieval of matching orders. The procedure finds the best matches for a given order and completes the corresponding trades. It uses the NODE-PRIORITY and LEAF-PRIORITY subroutines given in Figure 8.

NODE-PRIORITY(*order*, *node*, *queue*)

The subroutine inputs the given order, a matching monotonic node, and the priority queue of nodes.
If the order may have matches in the node's subtree, then the node is added to the priority queue.

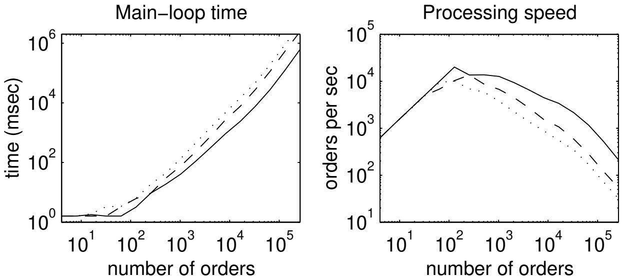
$i :=$ BEST-ITEM(*node*)
If *order* is a buy order, then $p :=$ FILL-PRICE($Price[order], Min\text{-}Price[node], i$)
Else, $p :=$ FILL-PRICE($Max\text{-}Price[node], Price[order], i$)
If $p =$ NONE, then terminate
$Quality[node] := Qual[order](i, p)$
Add *node* to *queue*, prioritized by *Quality*

LEAF-PRIORITY(*order*, *leaf*, *queue*)

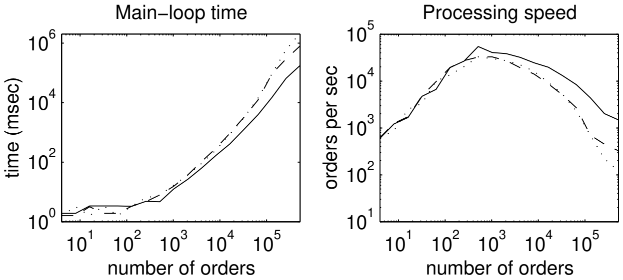
The subroutine inputs the given order, a matching leaf, and the priority queue of leaves. If the order's price matches the price of the leaf's best-price unprocessed order, then the leaf is added to the queue.

$match := Current\text{-}Order[leaf]$
If $match =$ NONE, then terminate (no more orders in *leaf*)
If *order* is a buy order, then $p :=$ FILL-PRICE($Price[order], Price[match], Item[leaf]$)
Else, $p :=$ FILL-PRICE($Price[match], Price[order], Item[leaf]$)
If $p =$ NONE, then terminate
$Quality[leaf] := Qual[order](Item[leaf], p)$
Add *leaf* to *queue*, prioritized by *Quality*

Figure 8: Subroutines of the NODE-MATCHES procedure, given in Figure 7. The NODE-PRIORITY subroutine adds a nonleaf node to the priority queue, arranged by quality estimates; the LEAF-PRIORITY subroutine adds a leaf node to the same queue. These subroutines use the FILL-PRICE function (Figure 1) and BEST-ITEM function (Figure 5).



(a) Used-car market.



(b) Corporate-bond market.

Figure 9: Performance in the used-car market and corporate-bond market. We give the results for matching density of 0.001 (solid lines), 0.01 (dashed lines), and 1 (dotted lines).

References

[Andersson *et al.*, 2000] 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.

[Fink *et al.*, 2004] Eugene Fink, Joshua Marc Johnson, and Jenny Ying Hu. Exchange market for complex goods: Theory and experiments. *Netnomics*, 6(1):21–42, 2004.

[Gonen and Lehmann, 2000] 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.

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

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

[Hershberger, 2003] Jonathan Wade Hershberger. Exchanges for complex commodities: Toward a general-purpose system for on-line trading. Master's thesis, Department of Computer Science and Engineering, University of South Florida, 2003.

[Hu, 2002] 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.

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

[Kalagnanam *et al.*, 2000] 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.

[Lavi and Nisan, 2000] 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.

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

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

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

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

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

[Wurman *et al.*, 1998] Peter R. Wurman, Michael P. Wellman, and William E. Walsh. The Michigan Internet AuctionBot: A configurable auction server for human and software agents. In *Proceedings of the Second International Conference on Autonomous Agents*, pages 301–308, 1998.