Office of Graduate Studies
University of South Florida
Tampa, Florida

CERTIFICATE OF APPROVAL

This is to certify that the thesis of

JIANLI GONG

in the graduate degree program of
Computer Science
was approved on August 6, 2002
for the Master of Science in Computer Science degree

Examining Committee:

Major Professor: Eugene Fink, Ph.D.

Member: Dmitry B. Goldgof, Ph.D.

Member: Sudeep Sarkar, Ph.D.

Committee Verification:

Associate Dean

EXCHANGES FOR COMPLEX COMMODITIES:

SEARCH FOR OPTIMAL MATCHES

by

JIANLI GONG

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

August 6, 2002

Major Professor: Eugene Fink, Ph.D.

Acknowledgements

I gratefully acknowledge the help of Eugene Fink, who has supervised my thesis work. I am grateful to Dmitry Goldgof and Sudeep Sarkar for their valuable comments and suggestions. I also appreciate the help of my fellow graduate students: Josh Johnson, Jenny Hu, Savvas Nikiforou, and Hong Tang.

I have received invaluable support from my wife, Yian Li, and my daughter, Chao Gong, who have provided encouragement through my studies. My parents, Xixin Gong and Yumin Yian, have also given me never ending support.

Table of Contents

List of Figures ii

Abstract iv

Chapter 1 Introduction 1

Chapter 2 Previous work 3

Chapter 3 Motivating example 10

Chapter 4 General exchange model 12
- 4.1 Orders 12
- 4.2 Quality functions 13
- 4.3 Order sizes 14
- 4.4 Market attributes 15
- 4.5 Fills 16

Chapter 5 Order representation 19

Chapter 6 Indexing structure 22
- 6.1 Architecture 22
- 6.2 Indexing trees 24
- 6.3 Basic tree operations 26

Chapter 7 Search for matches 32
- 7.1 Depth-first search 32
- 7.2 Best-first search 36

Chapter 8 Performance 41
- 8.1 Artificial markets 41
- 8.2 Real markets 50

Chapter 9 Concluding remarks 64

References 65

Chapter A Artificial markets 76
- A.1 Processing time 76
- A.2 Matching time 81
- A.3 Maximal throughput 86

List of Figures

3.1 Matching orders and the resulting trade 10
3.2 Choosing the match with the best price 11
3.3 Example of order sizes 11

4.1 Computing the fill size for two matching orders 18

6.1 Main data structures 22
6.2 Example of index and nonindex orders 23
6.3 Top-level loop of the matcher engine 23
6.4 Processing of a new order 24
6.5 Search for index orders that match a given order 25
6.6 Indexing tree with seventeen orders 27
6.7 Adding orders to an indexing tree 29
6.8 Updating the summary data after addition of an order 30
6.9 Updating the summary data after deletion of an order 31

7.1 Retrieval of matching leaves 33
7.2 Retrieval of matching orders 34
7.3 Example of retrieval 35
7.4 Retrieval of the highest-quality matches 38
7.5 Retrieval of matches from a leaf node 39
7.6 Adding matching nodes to a priority queue 39
7.7 Calculating the quality estimate for a node 40

8.1 Dependency of the matching time on the number of orders 44
8.2 Dependency of the throughput on the number of orders 45
8.3 Dependency of the matching time on the number of attributes 46
8.4 Dependency of the throughput on the number of attributes 47
8.5 Dependency of the matching time on the matching density 48
8.6 Dependency of the throughput on the matching density 49
8.7 Performance in the car market for matching density of 0.001 and 0.01 . 51
8.8 Performance in the car market for matching density of 0.1 and 1 52
8.9 Dependency of the matching time on the number of orders 53
8.10 Dependency of the throughput on the number of orders 54
8.11 Dependency of the matching time on the matching density 55
8.12 Dependency of the throughput on the matching density 56
8.13 Performance in the commercial-paper market for density of 0.001 and 0.01 58
8.14 Performance in the commercial-paper market for density of 0.1 and 1 . . 59
8.15 Dependency of the matching time on the number of orders 60

8.16 Dependency of the throughput on the number of orders 61
8.17 Dependency of the matching time on the matching density 62
8.18 Dependency of the throughput on the matching density 63

A.1 Time of processing the new orders for matching density of 0.001. 77
A.2 Time of processing the new orders for matching density of 0.01. 78
A.3 Time of processing the new orders for matching density of 0.1. 79
A.4 Time of processing the new orders for matching density of 1. 80
A.5 Time of matching the pending orders for matching density of 0.001. . . . 82
A.6 Time of matching the pending orders for matching density of 0.01. 83
A.7 Time of matching the pending orders for matching density of 0.1. 84
A.8 Time of matching the pending orders for matching density of 1. 85
A.9 Maximal number of orders per second for matching density of 0.001. . . . 87
A.10 Maximal number of orders per second for matching density of 0.01. . . . 88
A.11 Maximal number of orders per second for matching density of 0.1. 89
A.12 Maximal number of orders per second for matching density of 1. 90

EXCHANGES FOR COMPLEX COMMODITIES:

SEARCH FOR OPTIMAL MATCHES

by

Jianli Gong

An Abstract

of a thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Date of Approval:
August 6, 2002

Major Professor: Eugene Fink, Ph.D.

The Internet has opened opportunities for efficient on-line markets, which currently include bulletin boards and auctions, as well as exchanges for standardized commodities. A recent project at the University of South Florida has been aimed at the development of an on-line exchange for complex nonstandard commodities, such as used cars and collectible stamps.

We have investigated the use of objective functions in the complex-commodity exchange, which allow a trader to specify price constraints and preferences among potential trades. We explain the representation of orders with price and preference constraints, and describe a technique for fast identification of most preferable trades, which maximize the satisfaction of market participants. We give an empirical evaluation of the implemented system, which shows that the system scales to markets with 300,000 orders and usually processes 100 to 1,000 new orders per second.

Abstract Approved: ______________________________

Major Professor: Eugene Fink, Ph.D.
Assistant Professor, Department of Computer Science

Date Approved: ______________________________

Chapter 1

Introduction

The growth of the Internet has led to the development of electronic markets, and economists expect that it will play an increasingly important rule in both wholesale and retail transactions [Feldman, 2000]; the Internet marketplaces include bulletin boards, auctions, and exchanges [Klein, 1997; Turban, 1997; Wrigley, 1997; Bakos, 2001].

Electronic bulletin boards vary from sale catalogs to newsgroup postings, which help buyers and sellers to find each other; however, they often require a customer to invest significant time into searching among multiple ads, and many buyers prefer on-line auctions, such as eBay (www.ebay.com). Auctions have their own problems, including significant computational costs and asymmetry between buyers and sellers. Furthermore, most auctions do not allow fast sales; for instance, if a seller posts an item on eBay, she can sell it in three or more days, but not sooner.

An exchange-based market ensures symmetry between buyers and sellers, and supports fast-paced trading. Examples of liquid markets include the traditional stock and commodity exchanges, as well as currency and bond markets. The main limitation of these exchanges is rigid standardization of tradable items. For instance, the New York Stock Exchange allows trading of about 3,100 securities, and a buyer or seller has to indicate a specific item, such as IBM 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. She may also need to specify her preferences; for example, she may indicate that a white car is acceptable but less desirable than a red car. An exchange for nonstandard commodities should satisfy the following requirements:

- Allow complex constraints in specifications of buy and sell orders.

- Support fast-paced trading for markets with millions of orders.

- Include optimization techniques that maximize traders' satisfaction.

A recent project at the University of South Florida has been aimed at developing an electronic exchange for complex goods. Johnson [2001] has defined related trading semantics and developed an exchange system that supports a market with 300,000 orders. Hu [2002] has extended order semantics and developed indexing structures for fast identification of matches between buy and sell orders.

We have continued this work, and developed algorithms for fast identification of most preferable matches, which maximize the satisfaction of market participants. We have tested these algorithms using a suite of artificial markets, as well as a car market and a commercial-paper market. The tests have shown that the new system is as fast as the earlier versions, which used suboptimal-matching algorithms; it scales to markets with 300,000 orders and usually processes 100 to 1,000 new orders per second.

Chapter 2

Previous work

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 been focused on effective auction systems [Bichler, 2000b; Ronen, 2001], optimal matching in various auctions [Ygge and Akkermans, 1997; Monderer and Tennenholtz, 2000], bidding strategies [Tesauro and Das, 2001], and general-purpose systems for auctions and exchanges. It has led to successful Internet auctions, such as eBay (www.ebay.com) and Yahoo Auctions (auctions.yahoo.com). Recently, researchers have developed several *combinatorial auctions,* which allow buying and selling sets of commodities rather than individual items.

Combinatorial auctions. A traditional combinatorial auction allows bidding on a set of fully specified items. For example, Katie may bid on a red Mustang and black Corvette for a total price of $40,000; in this case, she will get both cars together or nothing. An advanced auction may allow disjunctions; for instance, Katie may specify that she wants either a red Mustang and black Corvette or, alternatively, two silver BMWs. On the other hand, standard combinatorial auctions do not allow incompletely specified items, such as a Mustang of any color.

Rothkopf *et al.* [1998] gave a detailed analysis of combinatorial auctions and described semantics of combinatorial bids that allowed fast matching. Nisan discussed alternative semantics for combinatorial bids, formalized the problem of searching for optimal and near-optimal matches, and proposed a linear-programming solution [Nisan, 2000; Lavi and Nisan, 2000]. Zurel and Nisan [2001] developed a system for finding near-optimal matches, based on a combination of approximate linear programming with optimization heuristics. It could quickly clear an auction with 1,000 items and 10,000 bids, and its average approximation error was less than 1%.

Sandholm [1999] developed several efficient algorithms for one-seller combinatorial auctions, and showed that they scaled to a market with about 1,000 bids. Sandholm and his colleagues later improved the original algorithms and implemented a system that processed several thousand bids [Sandholm, 2000a; Sandholm and Suri, 2000; Sandholm *et al.*, 2001a]. They developed a mechanism for determining a trader's preferences and converting them into a compact representation of combinatorial bids [Conen and Sandholm, 2001]. They also described several special cases of bid processing that allowed polynomial solutions, proved the NP-completeness of more general cases, and tested various heuristics for NP-complete cases [Sandholm *et al.*, 2001b].

Sakurai *et al.* [2000] developed an algorithm for finding near-optimal matches in combinatorial auctions based on a synergy of iterative-deepening A* with limited-discrepancy search. It processed auctoins with up to 5,000 bids, and its approximation error was under 5%. Hoos and Boutilier [2000] applied stochastic local search to finding near-optimal matches; their system could clear auctions with 500 items and 10,000 bids. Akcoglu *et al.* [2000] represented a combinatorial auction as a graph; its nodes were bids, and its edges were conflicts between bids. This representation led to the development of a linear-time approximation algorithm for clearing the auction.

Fujishima proposed an approach for enhancing standard auction rules, analyzed trade-offs between optimality and running time, and presented two related algorithms [Fujishima *et al.*, 1999a; Fujishima *et al.*, 1999b]. The first algorithm ensured optimal matching and scaled to about 1,000 bids, whereas the second found near-optimal matches for a market with 10,000 bids.

Leyton-Brown *et al.* [2000] investigated combinatorial auctions that allowed bidders to specify a number of items; for instance, a buyer could bid on ten identical cars. They described a branch-and-bound search algorithm for finding optimal matches, which could quickly process markets with fifteen item types and 2,500 bids.

Lehmann *et al.* [1999] investigated heuristic algorithms for combinatorial auctions and identified cases that allowed *truthful bidding,* which meant that users did not benefit from providing incorrect information about their intended maximal bids. Gonen and Lehmann [2000, 2001] studied branch-and-bound heuristics for processing combinatorial bids and integrated them with linear programming. Mu'alem and Nisan [2002] also investigated truthful-bidding combinatorial auctions, described conditions for ensuring truthful bidding, and proposed approximation algorithms for clearing the auctions that satisfied these conditions.

Yokoo *et al.* [2001a, 2001b] considered a problem of *false-name bids,* that is, manipulation of prices by creating fictitious users and submitting bids without intention to buy; they proposed auction rules that discouraged such bids. Suzuki and Yokoo [2002] studied another security problem in combinatorial auctions; they investigated techniques for clearing an auction without revealing the content of bids to the auctioneer. They described a distributed dynamic-programming algorithm that found matches without revealing the bids to the auction participants or to any central "auctioneer" system; however, its complexity was exponential in the number of items.

Andersson *et al.* [2000] compared the main techniques for combinatorial auctions and proposed an integer-programming representation that allowed richer bid semantics. Wurman *et al.* [2001] analyzed a variety of previously developed auctions and identified the main components of an automated auction, including bid semantics, clearing mechanisms, rules for placing and canceling bids, and policies for hiding information from other users. They proposed a standardized format for describing the components of each specific auction.

Researchers have also investigated the application of auction algorithms to non-financial settings, such as scheduling problems [Wellman *et al.*, 2001], management of resources in wide-area networks [Chen *et al.*, 2001], and co-ordination of services performed by different companies [Preist *et al.*, 2001].

The reader may find a detailed survey of combinatorial auctions in the review article by de Vries and Vohra [2001]. Although the developed systems can efficiently process several thousand bids, their running time is super-linear 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 an 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 [1993] analyzed auctions that allowed negotiating not only the price but also the quality of a commodity. A bid in these auctions was a function that specified a desired trade-off between price and quality. Cripps and Ireland [1994] considered a similar setting and suggested several strategies for bidding on price and quality.

Sandholm and Suri [2001b] described a mechanism for imposing nonprice constraints in combinatorial auctions, such as budget constraints and limit on the number of winners; they showed that these constraints sometimes increased the auction complexity, and sometimes reduced the complexity. They have also studied combinatorial auctions that allowed bulk discounts [Sandholm and Suri, 2001a]; that is, they enabled a bidder to specify a dependency between item price and order size. Lehmann *et al.* [2001] also considered the dependency of price on order size, showed that the corresponding problem of finding best matches was NP-hard, and developed a greedy approximation algorithm.

Bichler discussed a market that would allow negotiations on any attributes of a commodity [Bichler *et al.*, 1999; Bichler, 2000a]; for instance, a car buyer could set a fixed price and negotiate the options and service plan. He analyzed several alternative versions of this model, and concluded that it would greatly increase the economic utility of auctions; however, he pointed out the difficulty of implementing it and did not propose any computational solution.

Jones extended the semantics of combinatorial auctions and allowed buyers to use complex constraints [Jones, 2000; Jones and Koehler, 2000]; for instance, a car buyer could bid on a vehicle that was less than three-years old, or on the fastest available vehicle. They suggested an advanced semantics for these constraints, which allowed compact description of complex bids; however, they did not allow complex constraints in sell orders. They implemented an algorithm that found near-optimal matches, but it scaled only to one thousand bids.

Boutilier and Hoos [2001] developed a general propositional language for specifying bids in combinatorial auctions, which allowed a compact representation of most bids. Conen and Sandholm [2002] described a system that helped the participants of combinatorial auctions to specify their bids; it elicited the preferences of an auction participant and used them to define appropriate bids.

This initial work leaves many open problems, which include the use of complex constraints with general preference functions, symmetric treatment of buy and sell orders, and design of efficient matching algorithms for advanced semantics.

Exchanges. Economists have extensively studied traditional stock exchanges; for example, see the historical review by Bernstein [1993] and the textbook by Hull [1999]. They have focused on exchange dynamics and related mathematics, rather than on efficient algorithms [Cason and Friedman, 1996; Cason and Friedman, 1999; Bapna *et al.*, 2000]. Several computer scientists have also studied trading dynamics and proposed algorithms for finding the market equilibrium [Reiter and Simon, 1992; Cheng and Wellman, 1998; Andersson and Ygge, 1998].

Successful on-line exchanges include electronic communication networks, such as REDI (www.redibook.com) and Island (www.island.com). The directors of large stock and commodity exchanges are also considering electronic means of trading. For example, the Chicago Mercantile Exchange has deployed the Globex system, which supports trading around the clock.

Some auction researchers have investigated the related theoretical issues; they have viewed exchanges as a variety of auction markets, called *continuous double auctions.* In particular, Wurman *et al.* [1998a] proposed a theory of exchange markets and implemented a general-purpose system for auctions and exchanges, which processed traditional fully specified orders. Sandholm and Suri [2000] developed an exchange for combinatorial orders, but it could not support markets with more than 1,000 orders. Blum *et al.* [2002] explored methods for improving liquidity of standardized exchanges. Kalagnanam *et al.* [2000] investigated techniques for placing orders with complex constraints and identifying matches between them. They developed network-flow algorithms for finding optimal matches in simple cases, and showed that more complex cases were NP-complete. The complexity of their algorithms was super-linear in the number of orders, and the resulting system did not scale beyond a few thousand orders.

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

General-purpose systems. Computer scientists have developed several systems for auctions and exchanges, which vary from specialized markets to general-purpose tools for building new markets. The reader may find a survey of most systems in the review articles by Guttman *et al.* [1998a, 1998b] and Maes *et al.* [1999].

Kumar and Feldman [1998] built an Internet-based system that supported several standard auctions, including open-cry auctions, single-round sealed-bid auctions, and multiple-round auctions. Chavez and his colleagues designed an on-line agent-based auction; they built intelligent agents that negotiated on behalf of buyers and sellers [Chavez and Maes, 1996; Chavez *et al.*, 1997]. Vetter and Pitsch [1999] constructed a more flexible agent-based system that supported several types of auctions. Preist [1999a; 1999b] developed a similar distributed system for exchange markets. Bichler designed an electronic brokerage service that helped buyers and sellers to find each other and to negotiate through auction mechanisms [Bichler *et al.*, 1998; Bichler and Segev, 1999].

Benyoucef *et al.* [2001] considered a problem of simultaneous negotiations for interdependent goods in multiple markets, and applied a workflow management system to model the negotiation process. Their system helped a user to purchase a combinatorial package of goods in noncombinatorial markets. Boyan *et al.* [2001] also built a system for simultaneous bidding in multiple auctions; they applied beam search with simple heuristics to the problem of buying complementary goods in different auctions. Babaioff and Nisan [2001] studied the problem of integrating multiple auctions across a supply chain, and proposed a mechanism for sharing information among such auctions.

Wurman and Wellman built a general-purpose system, called the Michigan Internet AuctionBot, that could run a variety of different auctions [Wellman, 1993; Wellman and Wurman, 1998; Wurman *et al.*, 1998b; Wurman and Wellman, 1999]; however, they restricted the users to simple fully specified bids. Their system included scheduler and auctioneer procedures, related databases, and advanced interfaces. Hu *et al.* [1999] created agents for bidding in the Michigan AuctionBot; they used regression and learning techniques to predict the behavior of other bidders. Later, Hu *et al.* [2000] designed three types of agents and showed that their relative performance depended on the strategies of other auction participants. Hu and Wellman [2001] developed an agent that learned the behavior of its competitors and adjusted its strategy accordingly. Wurman [2001] considered a problem of building general-purpose agents that simultaneously bid in multiple auctions.

Parkes built a fast system for combinatorial auctions, but it worked only for markets with up to one hundred users [Parkes, 1999; Parkes and Ungar, 2000]. Sandholm created a more powerful auction server, configurable for a variety of markets, and showed its ability to process several thousand bids [Sandholm, 2000a; Sandholm, 2000b; Sandholm and Suri, 2000].

All these systems have the same limitation as commercial on-line exchanges; they require fully specified bids and do not support the use of constraints.

Chapter 3

Motivating example

We give an example of 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. For instance, a seller may offer a red Mustang, made in 1999, with 35,000 miles.

The exchange allows placing buy and sell orders, analogous to the orders in a stock market. 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 1999, 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 manufacturer may offer a brand-new Mustang of any color for $18,000.

The exchange system searches for matches between buy and sell orders, and generates corresponding *fills,* that is, transactions that satisfy both buyers and sellers. In the previous example, it will determine that a brand-new red Mustang for $18,500 satisfies both the buyer and the seller (Figure 3.1). If the system finds several matches for an order, it chooses the match with the best price. For example, the buy order in Figure 3.2 will trade with the cheaper of the two sell orders.

The system allows a user to trade several identical items by specifying a size for

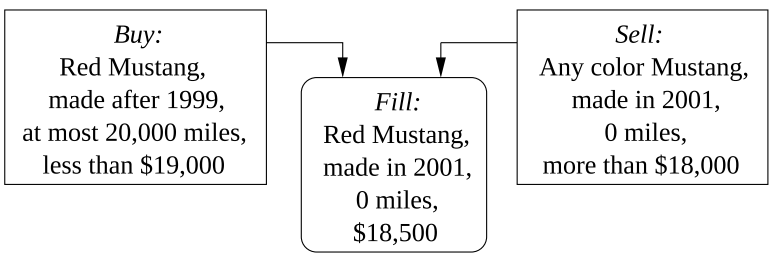


Figure 3.1: Matching orders and the resulting trade. When the system finds a match between two orders, it generates a fill, which is a trade that satisfies both parties.

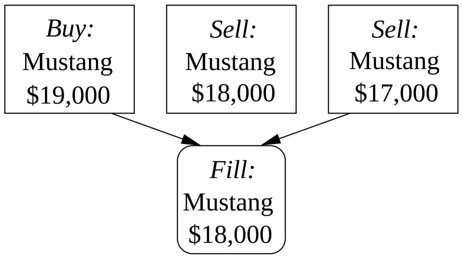


Figure 3.2: Choosing the match with the best price.

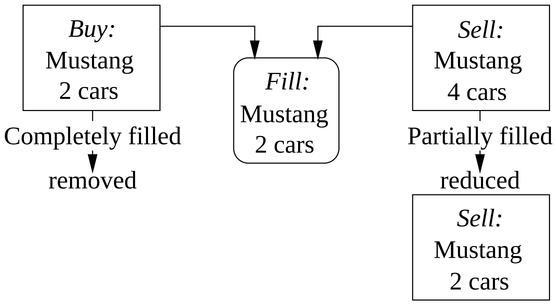


Figure 3.3: Example of order sizes. When the system finds a match, it completely fills the smaller order and reduces the size of the larger order.

an order. For example, a dealer can place an order to sell four Mustangs; then, the system can match it with a smaller buy order (Figure 3.3) and later find a match for the remaining cars. In addition, the user can specify a minimal acceptable size of a transaction. For instance, the dealer may place an order to sell four Mustangs, and indicate that she wants to trade at least two cars.

A user can specify that she is willing to trade any of several items. For example, she can place an order to buy either a Mustang or Camaro. If a user describes a set of items, she can indicate that the price depends on an item. For instance, she may offer $18,500 for a Mustang and $17,500 for a Camaro; furthermore, she may offer an extra $500 if a car is red, and subtract $1 for every ten miles on its odometer. A user can also specify her preferences for choosing among potential trades; for instance, she may indicate that a red Mustang is better than a white Mustang, and that a Mustang for $19,000 is better than a Camaro for $18,000.

Chapter 4

General exchange model

We describe a general model of trading complex commodities using a car market as an example. We formalize the concept of buy and sell orders, and consider a trading environment that allows hard and soft constraints in the order specification.

4.1 Orders

A specific market includes a certain set of items that can potentially be bought and sold; we denote it by M, which stands for *market set.* In the car market, it includes all vehicles that have ever been made, as well as vehicles that can be made in the future.

When a trader makes a purchase or sale, she has to specify a set of acceptable items, denoted I, which stands for *item set;* it must be a subset of M, that is, $I \subseteq M$. In addition, a trader should specify a limit on the acceptable price; for instance, Katie may be willing to pay $19,000 for a red Mustang, but only $18,500 for a black Mustang, and even less for a Camaro. 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 $\mathit{Price}(i)$. For a buyer, $\mathit{Price}(i)$ is the maximal acceptable price; for a seller, it is the minimal acceptable price.

We use the term *buy order* to refer to a buyer's item set and price function; when a buyer announces her desire to trade, we say that she has *placed an order.* Similarly, a *sell order* is a seller's constraints that define the offered merchandise. We say that a buy order *matches* a sell order if the buyer's constraints are consistent with the seller's constraints, thus allowing for a mutually acceptable trade. For instance, if Katie is willing to pay $19,000 for a red Mustang, and Laura offers a red Mustang for $18,000, then their orders match.

4.2 Quality functions

Buyers and sellers may have preferences among acceptable trades, which depend on a specific item i and its price p. For instance, Katie may prefer a red Mustang for \$19,000 to a black Camaro for \$18,000.

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 the user would rather trade i_1 at price p_1 than i_2 at p_2. We assume that negative quality corresponds to unacceptable trades; that is, if $Qual(i, p) < 0$, the user will not trade item i at price p.

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

We do not require a user to specify a quality function for each order; by default, quality is defined through price. To define this default function, we denote the user's price function by *Price*, and the price of an actual purchase or sale of an item i by p. The default function is the difference between the price limit and actual price, divided by the price limit:

- *For buy orders:* $Qual_b(i, p) = \frac{Price(i) - p}{Price(i)}$.
- *For sell orders:* $Qual_s(i, p) = \frac{p - Price(i)}{Price(i)}$.

For instance, if a customer is wiling to pay \$19,000 for a Mustang, and she gets an opportunity to buy it for \$18,500, then the default quality is $\frac{\$19{,}000 - \$18{,}500}{\$19{,}000} = 0.026$.

4.3 Order sizes

If a user wants to trade several identical items, she can include their quantity in the order specification; for example, Katie can place an order to buy two sports cars. We assume that an order size is a natural number; thus, we enforce discretization of continuous commodities, such as orange juice.

The user can specify not only an overall order size but also a minimal acceptable size. For instance, suppose that a Toyota wholesale agent is selling one thousand cars, and she works only with dealerships that are buying at least twenty vehicles. Then, she may specify that the overall size of her order is one thousand, and the minimal size is twenty. In addition, the user can indicate that a transaction size must be divisible by a certain number, called a *size step.* For example, a wholesale agent may specify that she is selling cars in blocks of ten.

To summarize, an order may include six elements:

- Item set, $I \subseteq M$
- 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, $Step$

To define a match between a buy order and sell order, we denote the item set of a buy order by I_b, its price function by $Price_b$, its quality function by $Qual_b$, and its size parameters by Max_b, Min_b, and $Step_b$. Similarly, we denote the parameters of a sell order by I_s, $Price_s$, $Qual_s$, Max_s, Min_s, and $Step_s$. The two orders match if they satisfy the following constraints:

- There is an item $i \in I_b \cap I_s$, such that $Price_s(i) \leq Price_b(i)$.
- There is a price p, such that
  - $Price_s(i) \leq p \leq Price_b(i)$, and

  - $Qual_b(i, p) \geq 0$ and $Qual_s(i, p) \geq 0$.

- There is a mutually acceptable size value, *size*, such that

  - $Min_b \leq size \leq Max_b$,
  - $Min_s \leq size \leq Max_s$, and
  - *size* is divisible by $Step_b$ and $Step_s$.

4.4 Market attributes

The set M of all possible items may be very large, which means that we cannot explicitly represent all items. To avoid this problem, we define this set by a list of attributes and possible values for each attribute. As a simplified example, we describe a car by four attributes: *Model, Color, Year,* and *Mileage.*

Formally, every attribute is a set of values; for instance, the *Model* set may include all car models, *Color* may include standard colors, *Year* may include the integers from 1896 to 2002, and *Mileage* may include the real values from 0 to 500,000. The market set M is a Cartesian product of the attribute sets; in this example, $M = Model \times Color \times Year \times Mileage$. If the market includes n attributes, each item is an n-tuple; in the car example, it is a quadruple that specifies the model, color, year, and mileage. We assume that every attribute is one of the three types:

- Set of explicitly listed values, such as the car model.
- Interval of integers, such as the year.
- Interval of real values, such as the mileage.

The value of a commodity may monotonically depend on some of its attributes; for example, the quality of a car decreases with an increase in mileage. When a market attribute has this property, we say that it is *monotonically decreasing.* To formalize this concept, suppose that a market has n attributes, and we consider the kth attribute. We denote attribute values of a given item by $i_1, \ldots, i_k, \ldots, i_n$, and a transaction price by p. The kth attribute is monotonically decreasing if all price and quality functions satisfy the following constraints:

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

- *Buy monotonicity:* If $Qual_b$ is a quality function for a buy order, and $i_k \leq i'_k$, then, for every two items $(i_1, \ldots, i_{k-1}, i_k, i_{k+1}, \ldots, i_n)$ and $(i_1, \ldots, i_{k-1}, i'_k, i_{k+1}, \ldots, i_n)$, and every price p, we have $Qual_b(i_1, \ldots, i_k, \ldots, i_n, p) \geq Qual_b(i_1, \ldots, i'_k, \ldots, i_n, p)$.

- *Sell monotonicity:* If $Qual_s$ is a quality function for a sell order, and $i_k \leq i'_k$, then, for every two items $(i_1, \ldots, i_{k-1}, i_k, i_{k+1}, \ldots, i_n)$ and $(i_1, \ldots, i_{k-1}, i'_k, i_{k+1}, \ldots, i_n)$, and every price p, we have $Qual_s(i_1, \ldots, i_k, \ldots, i_n, p) \leq Qual_s(i_1, \ldots, i'_k, \ldots, i_n, p)$.

Similarly, if the quality of commodities 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 making.

4.5 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. For example, suppose that Katie has placed an order for two sports cars, and Laura is selling four red Mustangs. If the prices of these orders match, Katie may purchase two red Mustangs from Laura; in this case, we say that two red Mustangs is a fill for her order. Formally, a fill consists of three parts: a specific item i, its price p, and the number of purchased items, denoted *size.*

If (I_b, $Price_b$, $Qual_b$, Max_b, Min_b, $Step_b$) is a buy order, and (I_s, $Price_s$, $Qual_s$, Max_s, Min_s, $Step_s$) is a matching sell order, then a fill $(i, p, size)$ must satisfy the following conditions:

- $i \in I_b \cap I_s$

- $Price_s(i) \leq p \leq Price_b(i)$

- $Qual_b(i, p) \geq 0$ and $Qual_s(i, p) \geq 0$

- $\max(Min_b, Min_s) \leq size \leq \min(Max_b, Max_s)$

- *size* is divisible by $Step_b$ and $Step_s$

If both buyer and seller specify a set of items, the resulting fill can contain any item $i \in I_b \cap I_s$. Similarly, we may have some freedom in selecting the price and size of the fill; the heuristics for making these choices depend on a specific implementation.

- *Item choice:* If $I_b \cap I_s$ includes several items, we may choose an item to maximize either the buyer's quality function or the seller's quality.
- *Price choice:* The default strategy is to split the price difference between the buyer and seller, which means that $p = \frac{Price_b(i)+Price_s(i)}{2}$.
- *Size choice:* We assume that buyers and sellers 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. In Figure 4.1, we give an algorithm that finds the maximal fill size for two matching orders.

After generating a fill, the system reduces the sizes of the respective orders. If the reduced size of an order 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.

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 $Step_b$,
and the size specification of a matching sell order, Max_s, Min_s, and $Step_s$.
The output is the fill size for these orders.

Find the least common multiple of $Step_b$ and $Step_s$:
$step := \frac{Step_b \cdot Step_s}{\text{GCD}(Step_b, Step_s)}$

Find the maximal acceptable size, divisible by $step$:
$size := \lfloor \frac{\min(Max_b, Max_s)}{step} \rfloor \cdot step$

Verify that it is not smaller than the minimal acceptable sizes:
If $size \geq Min_b$ and $size \geq Min_s$, then return $size$
Else, return 0 (no acceptable size)

GCD($Step_b, Step_s$)
The algorithm determines the greatest common divisor of $Step_b$ and $Step_s$.

$small := \min(Step_b,\ Step_s)$
$large := \max(Step_b,\ Step_s)$
Repeat while $small \neq 0$:
$rem := large \bmod small$
$large := small$
$small := rem$
Return $large$

Figure 4.1: Computing the fill size for two matching orders.

Chapter 5

Order representation

We describe the representation of orders in the implemented system. We explain the use of Cartesian products for coding item sets, and then describe the price functions, quality functions, and sizes.

Cartesian products. When a trader places an order, she has to specify a set of acceptable values for each market attribute, which is called an *attribute set.* Thus, if a market includes n attributes, the order description contains n attribute sets. For example, Katie may indicate that she is buying an Echo or Tercel, the acceptable colors are white, silver, and gold, the car should be made after 1999, and it should have at most 30,000 miles.

To give a formal definition, we denote the set of all possible values for the first attribute by M_1, the set of all values for the second attribute by M_2, and so on. The trader has to specify a set $I_1 \subseteq M_1$ of values for the first attribute, a set $I_2 \subseteq M_2$ of values for the second attribute, and so on. The resulting set I of acceptable items is the Cartesian product of the attribute sets; that is, $I = I_1 \times I_2 \times \cdots \times I_n$.

A trader can use specific values or ranges for each attribute; for instance, she can specify a desired year as 2002 or as a range from 1999 to 2002. Note that ranges work only for numeric attributes, such as year and mileage. A trader can also specify a list of several values or ranges. For instance, if Katie is interested in white, silver, and gold cars, she can specify the set of colors as {`white`, `silver`, `gold`}. If she is interested in cars made before 1950 and after 1999, she can specify the year as {[1896..1950], [1999..2002]}.

Unions and filters. A trader can define an item set I as the union of several Cartesian products. For example, if she wants to buy either a used Camry or a new

Echo, she can specify the following set:

$$I = \{\texttt{Camry}\} \times \{\texttt{white}, \texttt{silver}, \texttt{gold}\} \times [1999..2002] \times [0..30{,}000]$$
$$\cup \{\texttt{Echo}\} \times \{\texttt{white}, \texttt{silver}, \texttt{gold}\} \times \{2002\} \times [0..200].$$

Furthermore, the trader can indicate that she wants to avoid certain items, by providing a *filter function* that prunes undesirable items. Formally, it is a Boolean function on the set I that gives FALSE for unwanted items. We implement it by an arbitrary C++ procedure that inputs an item description and returns TRUE or FALSE. To summarize, the representation of an item set consists of two parts:

- A union of Cartesian products,
  $I = I1_1 \times I1_2 \times \ldots \times I1_n \cup \ldots \cup Ik_1 \times Ik_2 \times \ldots \times Ik_n$
- A filter function, *Filter*: $I \to \{\text{TRUE}, \text{FALSE}\}$,
  implemented by a C++ procedure

Price, quality, and size. If a price function is a constant, the trader specifies it by a numeric value, called a *price threshold.* If an item set is the union of several Cartesian products, the trader can specify a separate threshold for each product. For instance, if Katie's item set is the union of used Camries and new Echoes, she can indicate that she is paying \$15,000 for a Camry and \$12,000 for an Echo. If several Cartesian products overlap, and the trader has specified different thresholds for these products, then we use the tightest threshold for their intersection; that is, we use the lowest threshold for buy orders, and the highest threshold for sell orders.

If a price is not constant, the trader specifies it by an arbitrary C++ procedure that inputs an item and outputs the corresponding price limit. If an order includes both a threshold and price function, the system uses the tighter of the two. If the market includes monotonic attributes, the price functions must satisfy the monotonicity condition of Section 4.4.

The representation of a quality function is also an arbitrary C++ procedure; it inputs an item description and price, and outputs a numeric quality value. If a user does not provide any quality function, the system uses the default quality measure defined through the price function (Section 4.2). All quality functions must be monotonic on price (Section 4.2); if some attributes are monotonic, the quality must also satisfy the monotonicity condition of Section 4.4.

Finally, the size specification is the same as in the general model; it includes the overall size, minimal acceptable size, and size step.

Chapter 6

Indexing structure

We describe the data structures for fast identification of matches between buy and sell orders. We first explain the overall architecture and then present the indexing structures. We refer to the orders that are currently in the system as *pending orders*.

6.1 Architecture

The system maintains a description of market attributes and a collection of pending orders (Figure 6.1). It includes a central structure for indexing of pending orders, implemented by two trees (see Section 6.2). This structure allows indexing of orders with fully specified items; for example, it can include an order to sell a red Mustang made in 1999, but it cannot contain an order to buy any red car made after 1999. If we can put an order into the indexing structure, we call it an *index order*. If an order includes a set of items, rather than a fully specified item, the matcher adds it to an unordered list of *nonindex orders*. In Figure 6.2, we give an example of four index orders and four nonindex orders. The indexing structure allows fast retrieval of index orders that match a given order. On the other hand, the system does not identify matches between two nonindex orders.

In Figure 6.3, we show the main cycle of the matcher, which alternates between processing new orders and finding matches for pending nonindex orders. When it

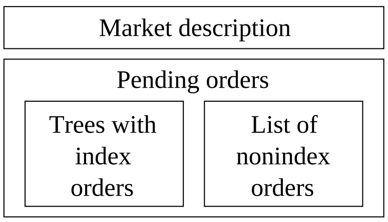


Figure 6.1: Main data structures.

Market description

Attribute 1: Model — Attribute 3: Year

Attribute 2: Color — Attribute 4: Mileage

Pending orders

Trees with index orders	*List of nonindex orders*
sell: red Mustang, made in 1999, 30,000 miles	buy: any red car, made after 2000
sell: white Camry, made in 1998, 42,000 miles	sell: any sports car, made in 2001
sell: red Echo, made in 1995, 65,000 miles	buy: any green Mustang, at most 5,000 miles
buy: green Saturn, made in 1997, 58,000 miles	sell: any Echo, made in 1995

Figure 6.2: Example of index and nonindex orders.

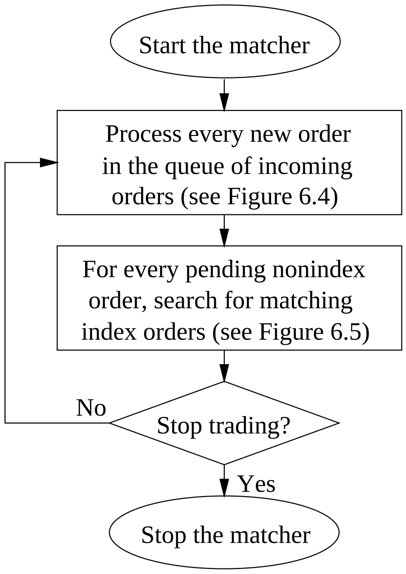


Figure 6.3: Top-level loop of the matcher engine.

receives a new order, it immediately searches for matching index orders (Figure 6.4). If there are no matches, and the new order is an index order, then the system adds it to the indexing structure. Similarly, if the matcher fills only part of a new index order, it stores the remaining part in the indexing structure. If the system gets a nonindex order and does not find a complete fill, it adds the unfilled part to the list of nonindex orders.

For example, suppose that Laura places an order to sell a red Mustang, made in 1999, with 30,000 miles. The system immediately looks for matching index orders; if it does not find a match, it adds the order to the indexing structure. If Katie later places a buy order for a sports car, the system identifies the match with Laura's order.

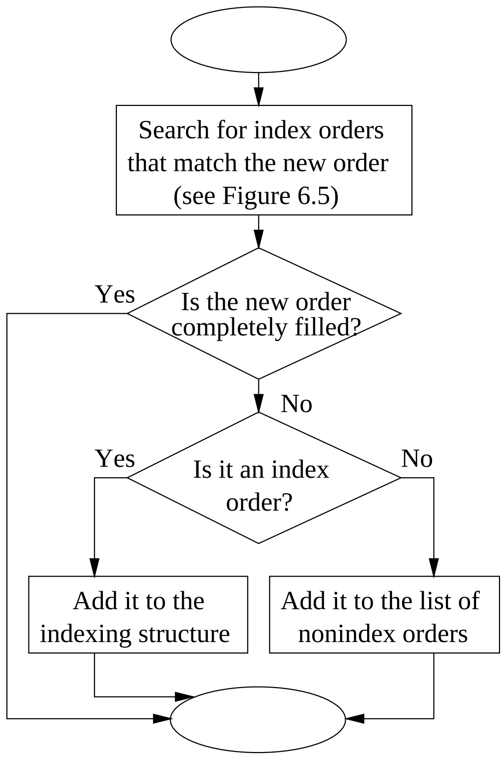


Figure 6.4: Processing of a new order.

After processing all new orders, the system tries to fill pending nonindex orders, which include not only the new arrivals, but also the old unfilled orders. For each nonindex order, it identifies matching index orders (Figure 6.5). For example, consider the market in Figure 6.2, and suppose that Laura places an order to sell a green Mustang, made in 2001, with zero miles. Since the market has no matching index orders, the system adds this new order to the indexing structure. After processing all new orders, it tries to fill the nonindex orders, and determines that Laura's order is a match for the old order to buy any green Mustang.

6.2 Indexing trees

We have implemented an indexing structure for orders with fully specified items, which do not include ranges or lists of values. The structure consists of two identical trees: one is for buy orders, and the other is for sell orders.

In Figure 6.6, we show an indexing tree for sell orders; its height is equal to the number of market attributes, and each level corresponds to one of the attributes. The

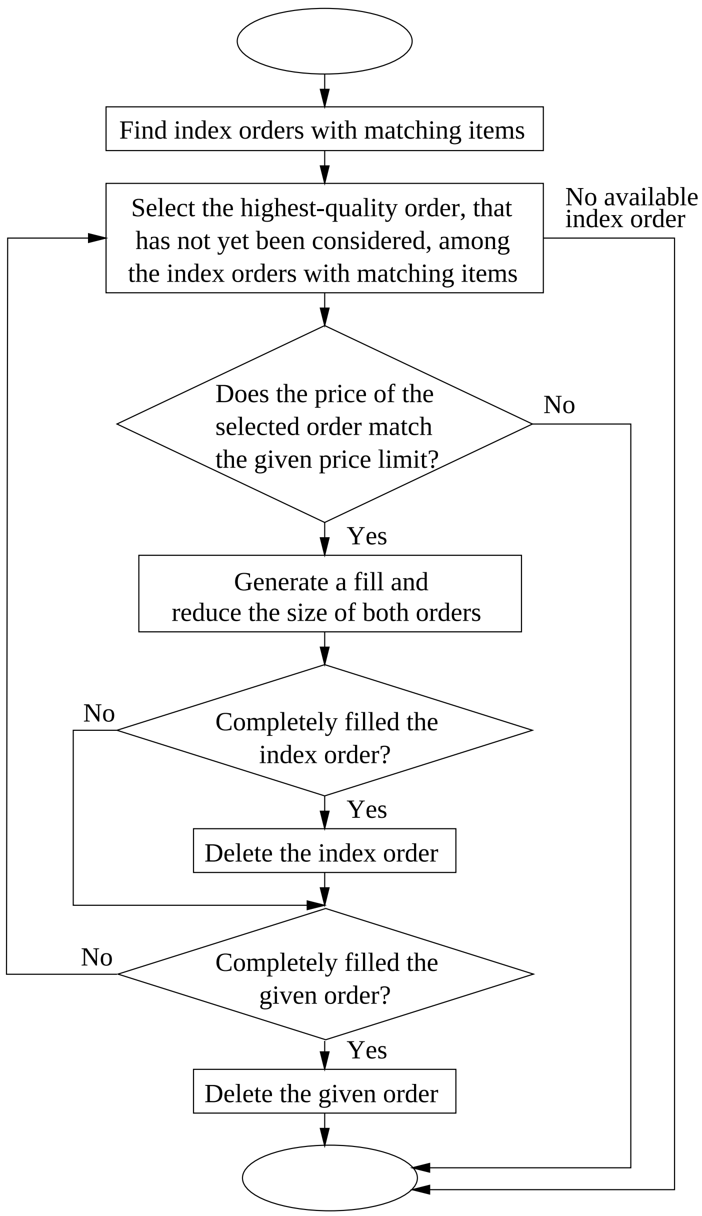


Figure 6.5: Search for index orders that match a given order.

root node encodes the first attribute, and its children represent different values of this attribute; in Figure 6.6, each child of the root corresponds to some car model. 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 the $(i+1)$th level 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; for instance, if nobody is selling an Echo, the root has no child for Echo.

Every nonleaf node includes a red-black tree that allows fast retrieval of its children with specific values. A leaf of the indexing tree includes orders with identical items, which may have different prices and sizes. Each leaf includes a red-black tree that indexes the corresponding orders by price.

The nodes of an indexing tree include summary data that help to find matching orders. Every node contains the following data about the orders in the corresponding subtree:

- The total number of orders and the total of their sizes.
- The minimal and maximal price.
- The minimal and maximal value for each numeric attribute.
- The time of the latest addition of an order.

For example, consider node 2 in Figure 6.6; the subtree rooted in this node includes nine orders. If the newest of these orders was placed at 2pm, the summary data in node 2 are as follows:

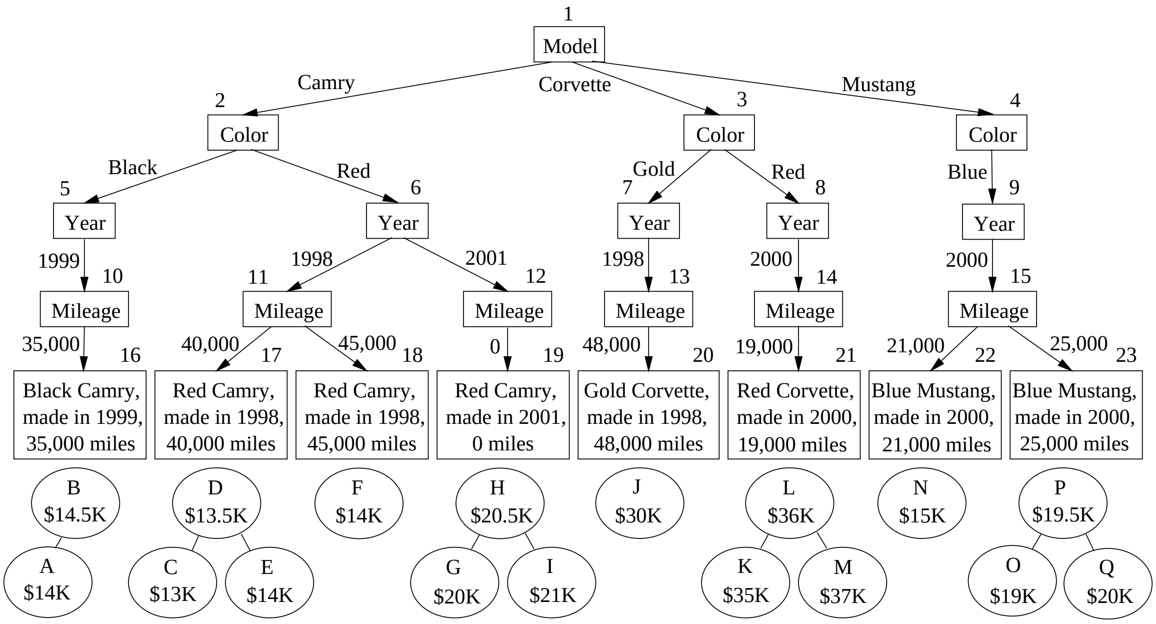
- Number of orders: 9
- Total size: 14
- Prices: $13,000..21,000
- Years: 1998..2001
- Mileages: 0..45,000
- Latest addition: 2pm

6.3 Basic tree operations

When a user places or removes an index order, the system has to update the indexing tree. We describe algorithms for adding a new order to the indexing structure and deleting an old order.

	Model	Color	Year	Mileage	Price	Size
A	Camry	Black	1999	35,000	14,000	2
B	Camry	Black	1999	35,000	14,500	1
C	Camry	Red	1998	40,000	13,000	1
D	Camry	Red	1998	40,000	13,500	2
E	Camry	Red	1998	40,000	14,000	2
F	Camry	Red	1998	45,000	14,000	2
G	Camry	Red	2001	0	20,000	2
H	Camry	Red	2001	0	20,500	1
I	Camry	Red	2001	0	21,000	1
J	Corvette	Gold	1998	48,000	30,000	1
K	Corvette	Red	2000	19,000	35,000	2
L	Corvette	Red	2000	19,000	36,000	1
M	Corvette	Red	2000	19,000	37,000	1
N	Mustang	Blue	2000	21,000	15,000	2
O	Mustang	Blue	2000	25,000	19,000	1
P	Mustang	Blue	2000	25,000	19,500	2
Q	Mustang	Blue	2000	25,000	20,000	5

(a) List of index orders.



(b) Indexing tree.

Figure 6.6: Indexing tree with seventeen orders.

Adding an order. When a user places an index order, the system adds it to the corresponding leaf; for example, if Laura places an order to sell a black Camry, made in 1999, with 35,000 miles, the system adds it to node 16 in Figure 6.7. If the leaf is not in the tree, the matcher adds the appropriate new branch; for example, if Laura offers to sell a white Camry, the matcher adds the dashed branch in Figure 6.7.

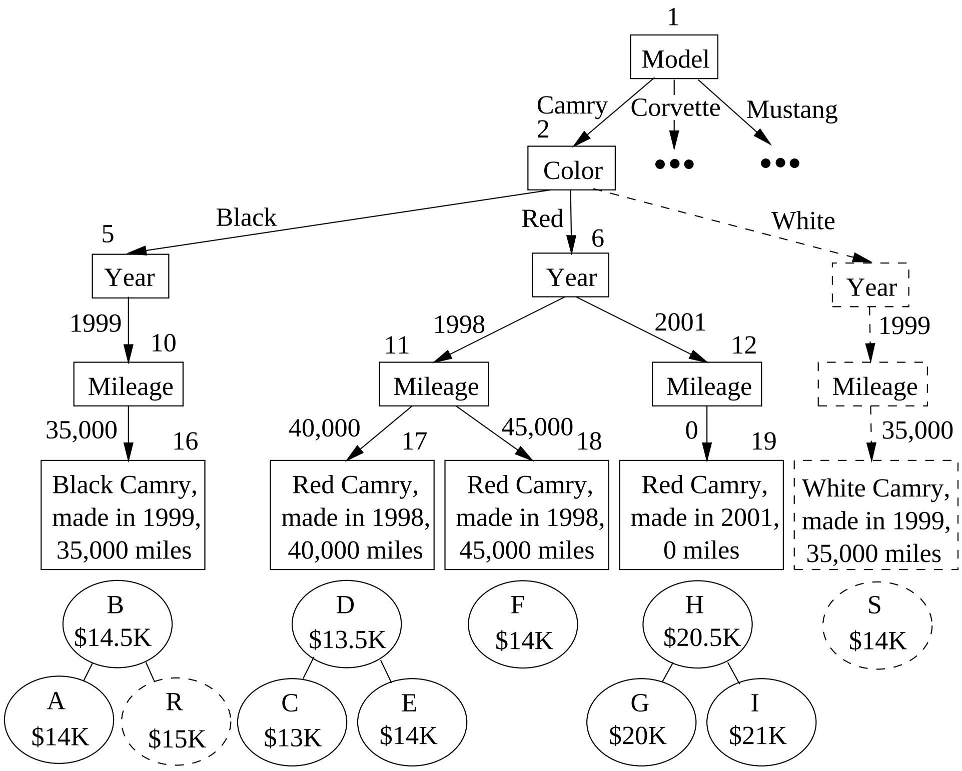
After adding a new order, the system modifies the summary data of the ancestor nodes. Note that every summary value is the minimum, maximum, or sum of the order values. In Figure 6.8, we give the algorithms for updating the total size and minimal price; the update of the other values is similar. These algorithms perform one pass from the leaf to the root, and their running time is proportional to the height of the tree; thus, if the market includes n attributes, the time is $O(n)$.

Deleting an order. When the matcher 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 indexing tree; for example, if the matcher fills order F in Figure 6.6, it removes node 18. If the deleted node is the only leaf in some subtree, the system removes this subtree; for instance, the deletion of order J leads to the removal of nodes 7, 13, and 20.

After deleting an order, the system updates the summary data in the ancestor nodes. In Figure 6.9, we give procedures for updating the total size and minimal price; the modification of the other data is similar. The update time depends on the number n of market attributes, and on the number of children of the ancestor nodes, $c_1, c_2, ..., c_n$. If a summary value is the sum of the order values, the update time is $O(n)$; if it is the minimum or maximum of order values, the time is $O(c_1+c_2+...+c_n)$.

	Model	Color	Year	Mileage	Price	Size
R	Camry	Black	1999	35,000	15,000	2
S	Camry	White	1999	35,000	14,000	1

(a) Two new orders.



(b) Indexing tree with new orders.

Figure 6.7: Adding orders to an indexing tree. We show new orders by dashed ovals. If the tree does not have the leaf for a new order, the system adds the proper branch.

ADD-SIZE(*new-size*, *leaf*)
The algorithm inputs the size of a newly added order
and the corresponding leaf of the indexing tree.

$node := leaf$
Repeat while $node \neq$ NIL:
    $total\text{-}size[node] := total\text{-}size[node] + new\text{-}size$
    $node := parent[node]$

ADD-PRICE(*new-price*, *leaf*)
The algorithm inputs the price of a newly added order
and the corresponding leaf of the indexing tree.

$node := leaf$
Repeat while $node \neq$ NIL and $min\text{-}price[node] > new\text{-}price$:
    $min\text{-}price[node] := new\text{-}price$
    $node := parent[node]$

Figure 6.8: Updating the summary data after addition of an order. We show the update of the total size (ADD-SIZE) and minimal price (ADD-PRICE).

DEL-SIZE(*old-size*, *leaf*)
The algorithm inputs the size of a deleted order,
along with the leaf from which the order is deleted.

node := *leaf*
Repeat while *node* $\neq$ NIL:
  total-size[*node*] := *total-size*[*node*] − *old-size*
  node := *parent*[*node*]

DEL-PRICE(*old-price*, *leaf*)
The algorithm inputs the price of a deleted order,
along with the leaf from which the order is deleted.

If *min-price*[*leaf*] < *old-price*, then terminate
Update the minimal price of the leaf:
  min-price[*leaf*] := $+\infty$
  For every *order* in the leaf:
    If *min-price*[*leaf*] > *price*[*order*],
      then *min-price*[*leaf*] := *price*[*order*]
Update the minimal price of its ancestors:
  node := *leaf*
  Repeat while *min-price*[*node*] > *old-price*
      and *parent*[*node*] $\neq$ NIL and *min-price*[*parent*[*node*]] = *old-price*:
    node := *parent*[*node*]
    min-price[*node*] := $+\infty$
    For every *child* of *node*:
      If *min-price*[*node*] > *min-price*[*child*],
        then *min-price*[*node*] := *min-price*[*child*]

Figure 6.9: Updating the summary data after deletion of an order. We show the update of the total size (DEL-SIZE) and minimal price (DEL-PRICE).

Chapter 7

Search for matches

We describe two techniques for the retrieval of matching orders from an indexing tree. The first algorithm is based on the depth-first search developed by Johnson [2002] during his work on the earlier version of the system. The second technique is a novel retrieval algorithm, based on the best-first search in an indexing tree, which allows fast retrieval of the highest-quality matches.

7.1 Depth-first search

In Figure 7.1, we give an algorithm that retrieves matching leaves of an indexing tree for a given item set. The FIND-LEAVES subroutine finds all matches for a Cartesian product. It identifies all children of the root that match the first element of the Cartesian product, and then recursively processes the corresponding subtrees. For example, suppose that a buyer is looking for a Camry or Mustang made after 1998, with any color and mileage, and the tree of sell orders is as shown in Figure 7.3. The subroutine determines that nodes 2 and 4 match the model, and then recursively 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. If a given order includes a union of several Cartesian products, we call the FIND-LEAVES subroutine for each product. If an order includes a filter function, the subroutine uses it to prune inappropriate leaves.

To avoid search in the branches that have not changed since the previous search, we compare the time stamp of the given order with the latest addition time of each branch. Similarly, we compare the time stamp of the given order with the placement times of index orders in the matching leaves. The time stamp of an order is the time of the previous search for matches. We prune the branches whose latest addition times are smaller than the time stamp of the given order, and skip the index orders

MATCHING-LEAVES(I, *filter, time-stamp, num-leaves, root*)
The algorithm inputs an item description I, represented by a union of Cartesian products, a corresponding filter function and time stamp, a limit on the number of retrieved leaves, and the root of an indexing tree. It returns a set of leaves that match the item description.

leaves := $\emptyset$ (set of matching leaves)
num-left := *num-leaves* (limit on the number of leaves)
For each Cartesian product $I_1 \times I_2 \times \ldots \times I_n$ in the union I:
    Call FIND-LEAVES($I_1 \times I_2 \times \ldots \times I_n$, *filter, time-stamp, leaves, num-left, root*)
    If *num-left* $= 0$, then return *leaves*
Return *leaves*

FIND-LEAVES($I_1 \times I_2 \times \ldots \times I_n$, *filter, time-stamp, leaves, num-left, node*)
The subroutine inputs a Cartesian product $I_1 \times I_2 \times \ldots \times I_n$, a corresponding filter function and time stamp, a set of matching leaves, a limit on the number of retrieved leaves, and a node of the indexing tree. It finds the matching children of the given node, recursively processes the respective subtrees, and identifies matching leaves in these subtrees.

If *latest-addition-time*[*node*] $\leq$ *time-stamp*, then terminate
If *node* is a leaf and *filter*(*node*) = TRUE, then:
    Add *node* to *leaves*
    num-left := *num-left* $- 1$
If *node* is not a leaf, then:
    Identify all children of *node* that match $I_{level[node]}$
    For each matching *child*:
        Call FIND-LEAVES($I_1 \times I_2 \times \ldots \times I_n$, *filter, time-stamp, leaves, num-left, child*)
        If *num-left* $= 0$, then terminate

Figure 7.1: Retrieval of matching leaves.

MATCHING-ORDERS(*order, leaves*)

The algorithm inputs an order and the leaves that match the order's item set.
It identifies matching orders in these leaves and completes the respective trades.

$Quality$:= $Qual[order]$
For each *leaf* in *leaves*:
    $current\text{-}order[leaf]$:= $best\text{-}price\text{-}order[leaf]$
    $quality[leaf]$:= $Quality(current\text{-}order[leaf])$
Build a priority queue for *leaves*, sorted by $quality[leaf]$
Repeat while $Max[order] \geq Min[order]$ and $Quality(current\text{-}order[top\text{-}leaf[queue]]) \geq 0$:
    $leaf := top\text{-}leaf[queue]$
    $best\text{-}order := current\text{-}order[leaf]$
    $current\text{-}order[leaf] := next\text{-}by\text{-}price[current\text{-}order[leaf]]$
    $quality[leaf] := Quality(current\text{-}order[leaf])$
    Update the position of *leaf* in the priority queue
    If $placement\text{-}time[best\text{-}order] > time\text{-}stamp[order]$, then:
        $size$:= FILL-SIZE($Max[order]$, $Min[order]$, $Step[order]$,
                $Max[best\text{-}order]$, $Min[best\text{-}order]$, $Step[best\text{-}order]$)
        If $size > 0$, then:
            Generate a fill for *order* and *best-order*
            $Max[order] := Max[order] - size$
            $Max[best\text{-}order] := Max[best\text{-}order] - size$
            If $Max[best\text{-}order] < Min[best\text{-}order]$, then remove *best-order* from the market
If $Max[order] < Min[order]$, then remove *order* from the market
Else, set $time\text{-}stamp[order]$ to the current time

Figure 7.2: Retrieval of matching orders.

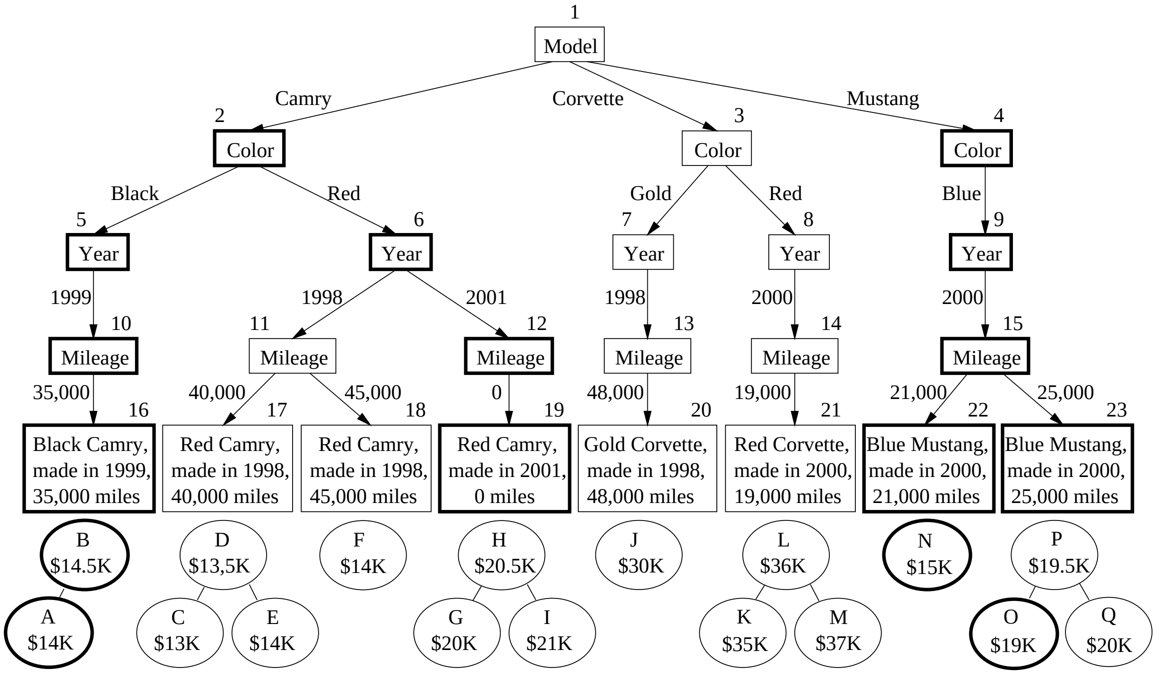


Figure 7.3: Retrieval of matches for an order to buy six Camries or Mustangs made after 1998. We show the matching nodes and retrieved orders by thick lines.

whose placement times are smaller than the time stamp.

If an order matches a large number of leaves, the retrieval may take considerable time. To prevent this problem, we can impose a limit on the number of retrieved leaves. For instance, if we allow at most three matches, and a user places an order to buy any Camry, then the system retrieves the three leftmost leaves in Figure 7.3. We use this limit to control the trade-off between the speed and quality of matches. A small limit ensures the efficiency but reduces the chance of finding the best match.

After the system identifies all matching leaves, it selects the highest-quality orders from these leaves, according to the quality function of the given order. In Figure 7.2, we give an algorithm for identifying best matches, which arranges matching leaves in a priority queue by the quality of the best order in each leaf. At every step, it processes the best match that has not yet been considered. For example, suppose that a buyer places an order for six Camries or Mustangs made after 1998, and her quality measure depends only on price (Figure 7.3). The system first retrieves order A, with price \$14,000 and size 2, then order B with price \$14,500, then order N with price \$15,000 and size 2, and finally order O with price \$19,000.

7.2 Best-first search

If an order matches a large number of leaves, the depth-first search that identifies all matches may take significant time, whereas the search with a limit on the number of matches may not find the best match. We use best-first search to guarantee the optimality without sacrificing the efficiency.

We define the *lowest nonmonotonic level* in the indexing tree as the level that corresponds to the last nonmonotonic attribute. For instance, the first two attributes in the car market are nonmonotonic, whereas the last two are monotonic; thus, level 2 is the lowest nonmonotonic level. A node's *quality estimate* is an estimated quality of the best order in the subtree rooted at this node. To compute this estimate, we use the node's summary data to construct the best possible order we could find in the corresponding subtree, and determine the quality of this order. We can compute it only if the node is below the lowest nonmonotonic level.

In Figures 7.4–7.7, we give the best-first search algorithm for a tree with sell orders; the algorithm for a buy-order tree is similar. To find the best match for a buy order with one Cartesian product, we arrange matching nodes into a priority queue by their quality estimates. At each step, we expand the top node from the queue and add its matching children to the queue; we repeat this process until finding the best match. Note that we can add a node to the queue only if it is below the lowest nonmonotonic level. Therefore, we first find all matching nodes at the lowest nonmonotonic level, and add their matching children to the queue.

If a given buy order includes a union of several Cartesian products, we create a separate priority queue for each product and arrange these queues into an "outer" priority queue, sorted by the quality of their top nodes. At each step, the algorithm expands the top node of the top "inner" queue. If it is a nonleaf node, the algorithm identifies its matching children, calculates their quality estimates, and adds them to the corresponding inner queue. If the node is a leaf, the algorithm uses the best-price order from this leaf to generate a fill. If the given buy order is completely filled, the search terminates. Otherwise, the algorithm takes the next matching order from the leaf and calculate its quality. If this quality is no smaller than the highest quality among other nodes, the algorithm uses the order to generate the next fill. After processing the leaf node, the algorithm sets its quality to the quality of the best unprocessed order, and adds it back to the corresponding inner queue.

BEST-FIRST-SEARCH(*order*, *root*)
The algorithm inputs a given order and the root of the indexing tree.
It finds the highest-quality matches for the given order.

```
Create an empty outer priority queue, outer-queue
For i = 1 to the number of Cartesian products in the union I[order]:
    Create a new empty priority queue, inner-queue, for the ith product
    num[inner-queue] := i
    MONO-NODES(i, order, root, inner-queue)
    Enqueue(outer-queue, inner-queue)
While quality[top-node[top-queue[outer-queue]]] ≥ 0 and Max[order] ≥ Min[order]:
    inner-queue := Dequeue(outer-queue)
    node := Dequeue(inner-queue)
    If node is not a leaf, then:
        i := num[inner-queue]
        k := level[node]
        Identify all node's children that match the kth element of the ith Cartesian product
        For each matching child:
            If latest-addition-time[child] > time-stamp[order], then:
                quality[child] := QUALITY-ESTIMATE(child, Qual[order])
                If quality[child] ≥ 0, then Enqueue(inner-queue, child)
    If node is a leaf and Filter[order](node) = TRUE, then:
        MATCHING-ORDERS(order, node,
            max(quality[top-node[top-queue[outer-queue]]], quality[top-node[inner-queue]]))
        If Max[order] ≥ Min[order] and priority[node] ≥ 0,
            then Enqueue(inner-queue, node)
    Enqueue(outer-queue, inner-queue)
If Max[order] ≥ Min[order], then set time-stamp[order] to the current time
Else, remove order from the market
```

Figure 7.4: Retrieval of the highest-quality matches.

MATCHING-ORDERS(*order*, *leaf*, *min-quality*)
The algorithm inputs a given order, a matching leaf, and a quality value. It finds matching orders in the leaf whose quality values are no smaller than the given quality.

While $Max[order] \geq Min[order]$ and $quality[leaf] \geq min\text{-}quality$:
    $best\text{-}order := current\text{-}order[leaf]$
    $current\text{-}order[leaf] := next\text{-}by\text{-}price[current\text{-}order[leaf]]$
    $quality[leaf] := Qual[order](item[leaf], price[current\text{-}order[leaf]])$
    If $placement\text{-}time[best\text{-}order] > time\text{-}stamp[order]$, then:
        $size :=$ FILL-SIZE($Max[order]$, $Min[order]$, $Step[order]$,
                $Max[best\text{-}order]$, $Min[best\text{-}order]$, $Step[best\text{-}order]$)
        If $size > 0$, then:
            Generate a fill for *order* and *best-order*
            $Max[order] := Max[order] - size$
            $Max[best\text{-}order] := Max[best\text{-}order] - size$
            If $Max[best\text{-}order] < Min[best\text{-}order]$, then remove *best-order* from the market

Figure 7.5: Retrieval of matches from a leaf node.

MONO-NODES(*i, order, node, inner-queue*)
The algorithm inputs the number of a Cartesian product, a given order, a node at or above the lowest nonmonotonic level, and a priority queue. It finds the matching children of the node, recursively processes the respective subtrees, and adds the matching nodes below the lowest nonmonotonic level to the priority queue.

If $latest\text{-}addition\text{-}time[node] < time\text{-}stamp[order]$, then terminate
Identify all children of *node* that match the corresponding element of the ith product
If *node* is above the lowest nonmonotonic level, then:
    For each matching *child*:
        Call MONO-NODES(*i, order, child, inner-queue*)
If *node* is at the lowest nonmonotonic level, then:
    For each matching *child*:
        If $latest\text{-}addition\text{-}time[child] > time\text{-}stamp[order]$, then:
            $quality[child] :=$ QUALITY-ESTIMATE($child$, $Qual[order]$)
            If $quality[child] \geq 0$, then Enqueue(*inner-queue*, *child*)

Figure 7.6: Adding matching nodes to a priority queue.

QUALITY-ESTIMATE(*node*, *Quality*)
The algorithm inputs a node of the indexing tree and a quality function.
It computes the quality estimate for the node.

For $i = 1$ to $level[node] - 1$:
    Set $item[i]$ to the corresponding attribute value
For $i = level[order]$ to n:
    Set $item[i]$ to the corresponding best value in the summary data
Return $Quality(item, \textit{min-price}[node])$

Figure 7.7: Calculating the quality estimate for a node.

Chapter 8

Performance

We compare the efficiency of the best-first search algorithm with the earlier depth-first search versions of the system. We give results for artificial market data, and then show the performance for two real-world markets.

8.1 Artificial markets

We describe the results of controlled experiments, which show the system's efficiency for different search strategies and market sizes.

Control variables. We have implemented an experimental setup that allows control over five parameters: search strategy, number of attributes, number of alternative values for an attribute, number of orders, and average number of matches per order.

Search strategy: We have compared the best-first search with two versions of the depth-first strategy. The first version of the depth-first algorithm searches for all matching orders. In the second version, we limit the number of retrieved matches; specifically, the system finds at most ten matching leaves in the indexing tree and retrieves at most ten orders from each leaf.

Attributes: The number of market attributes determines the complexity of traded items. We have considered markets with one, three, and ten attributes.

Values: We have controlled the number of values per attribute; we have experimented with 2, 16, and 1,024 values.

Orders: We have varied the number of orders from four to 2^{18}, that is, 262,144. Recall that the system's top-level loop involves processing new orders and matching old pending orders (Figure 6.3). We have controlled the total number of orders; the number of new orders in the input queue has been the same as the number of pending orders. For example, when experimenting with a four-order market, we have placed two pending orders and two new orders. We have randomly generated new and pending orders, which include an equal number of buys and sells.

Matching density: We define the *matching density* as the mean percentage of sell orders that match a given buy order; in other words, it is the probability that a randomly selected buy order matches a randomly chosen sell. We have experimented with four density values: 0.001, 0.01, 0.1, and 1.

Measured variables. We have considered three search strategies, three different settings for the number of attributes, three settings for the number of values per attribute, seventeen settings for the number of orders, and four settings for matching density. For each combination of these settings, we have run two independent experiments and measured the time of processing new orders, time of matching old orders, and maximal throughput of the system.

Processing time: We have measured the time of processing new orders, which is the first part of the system's main loop (Figure 6.3). This time is proportional to the number of new orders; it also depends on several other factors, including the number of attributes and pending orders.

Matching time: We have also measured the time of matching old orders, which is the second part of the main loop (Figure 6.3). The total of processing and matching time is the overall length of the main loop, which determines the system's speed.

Maximal throughput: We have determined the maximal acceptable frequency of placing new orders. If the system gets more orders per second, the number of unprocessed orders keeps growing, and the matcher eventually has to reject some of them.

Summary graphs. We show the dependency of the system's performance on each of the control variables in Figures 8.1–8.6. We use two performance measurements: the time of one pass through the system's main loop, and the system's throughput. For each control variable, we consider three settings of the other three variables. For each of the three settings, we give two graphs with the dependency of the performance of the three search strategies on the selected variable; the first graph is in logarithmic scale, and the second is in linear scale. We also give a more detailed summary of these experiments in Appendix A.

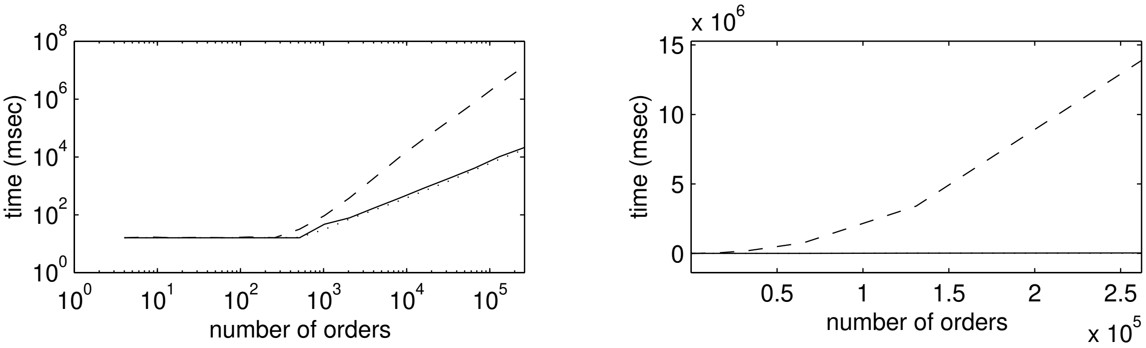
In Figures 8.1 and 8.2, we show how the performance depends on the number of orders. The main-loop time is approximately linear in the number of orders. The throughput in small markets grows with the number of orders; it reaches an upper limit when the market grows to about two hundred orders, and then decreases with further increase in the number of orders.

In Figures 8.3 and 8.4, 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.

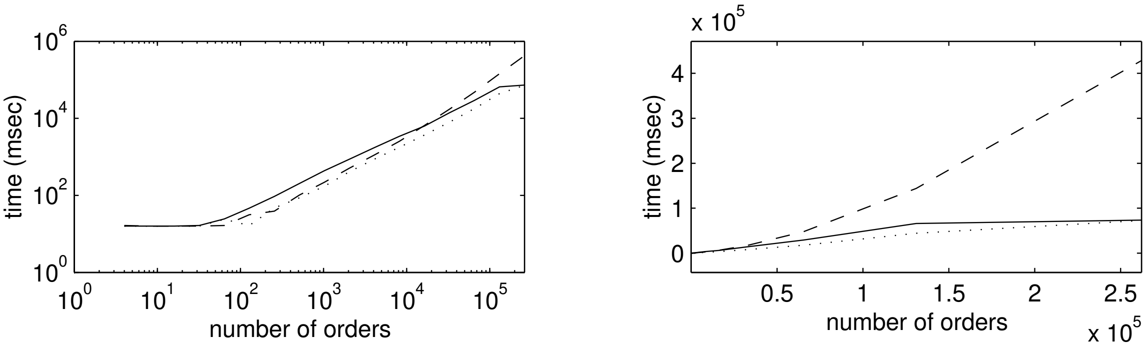
In Figures 8.5 and 8.6, we show how the system's behavior changes with the matching density. We have not found any monotonic dependency between the density and the performance; the increase of the matching density sometimes leads to faster matching and sometimes slows down the system.

The best-first search strategy is much faster than the depth-first search that identifies all matches; the saving factor for large markets is between 1.0 and 750.0, and its mean value is 121.8. Thus, the new system is more effective for finding optimal matches than the old depth-first version.

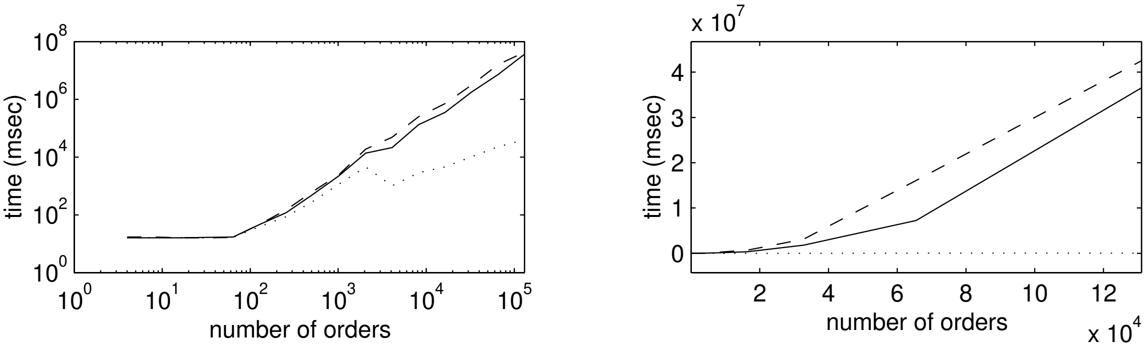
The speed of the best-first search is usually close to that of the depth-first search with a limit on the number of matches; that is, the optimal-matching system is as fast as the old system that could not find optimal matches. A notable exception is the performance in ten-attribute markets with large number of values per attribute. For these markets, the best-first search is slower than the limited depth-first search by a factor of ten to hundred.



(a) Tests with one attribute, two values per attribute, and matching density of 0.001.

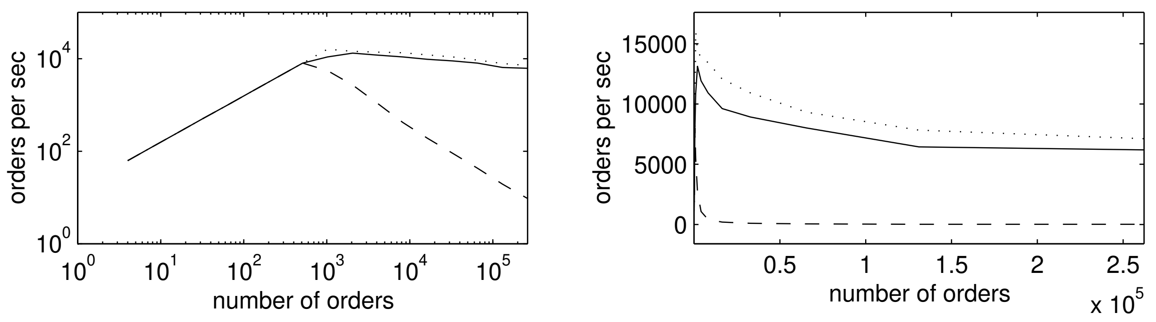


(b) Tests with three attributes, sixteen values per attribute, and matching density of 0.01.

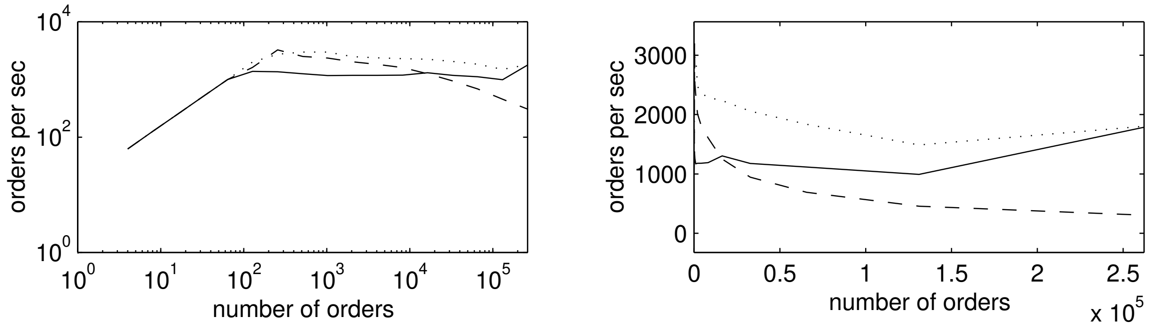


(c) Tests with ten attributes, 1,024 values per attribute, and matching density of 1.

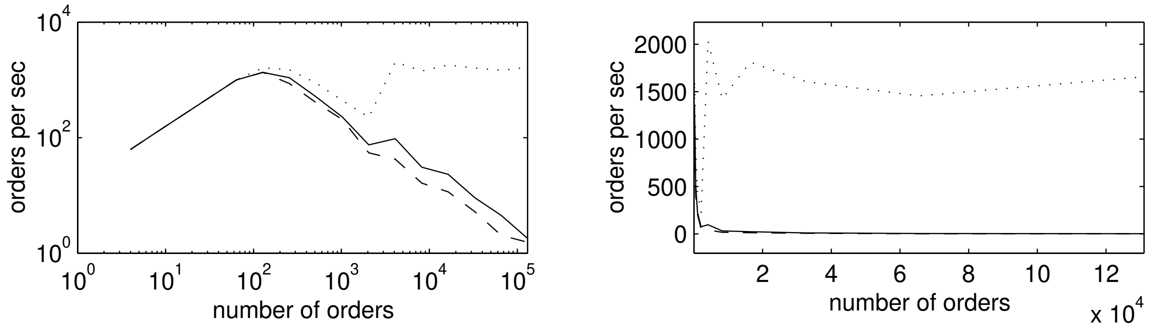
Figure 8.1: Dependency of the total matching time on the number of orders. We show the performance of the best-first search (solid lines), depth-first search that identifies all matches (dashed lines), and depth-first search with a limit on the number of matches (dotted lines). The graphs on the left are in logarithmic scale, whereas the graphs on the right are in linear scale.



(a) Tests with one attribute, two values per attribute, and matching density of 0.001.

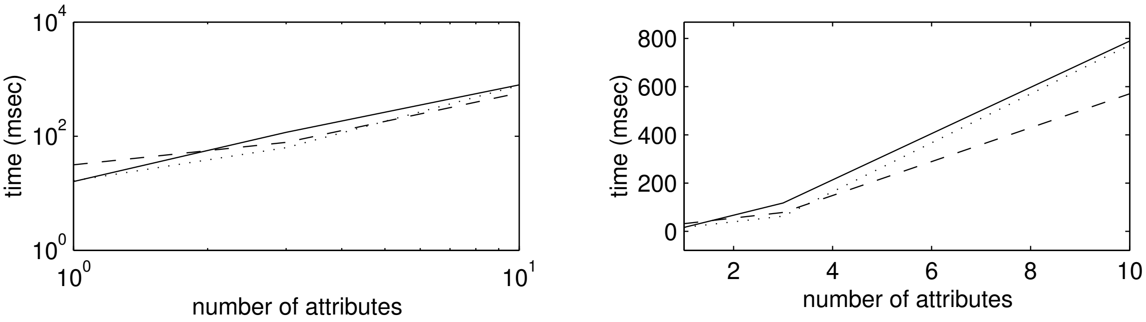


(b) Tests with three attributes, sixteen values per attribute, and matching density of 0.01.

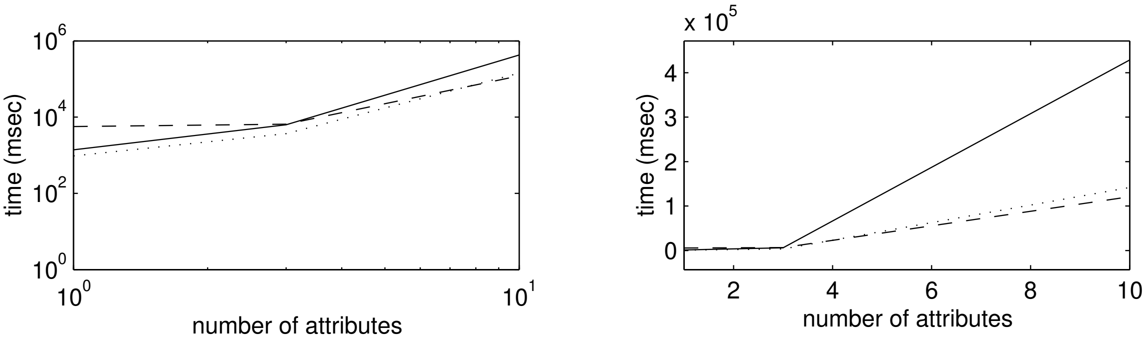


(c) Tests with ten attributes, 1,024 values per attribute, and matching density of 1.

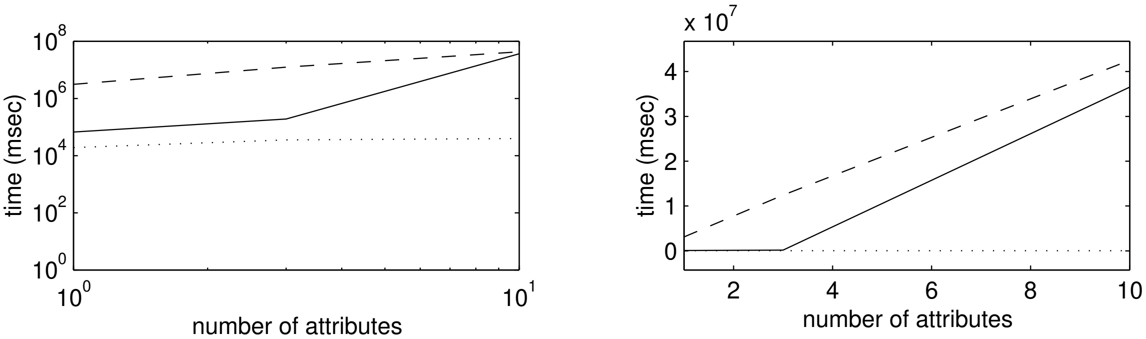
Figure 8.2: Dependency of the throughput on the number of orders. The legend is the same as in Figure 8.1.



(a) Tests with two values per attribute, 512 orders, and matching density of 0.001.

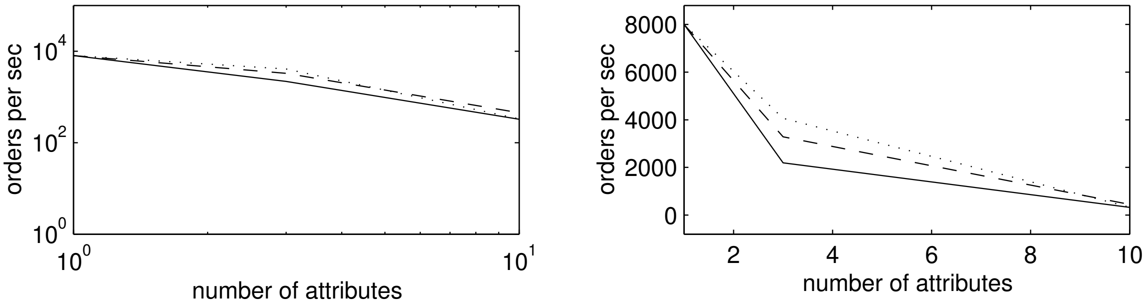


(b) Tests with sixteen values per attribute, 16,384 orders, and matching density of 0.01.

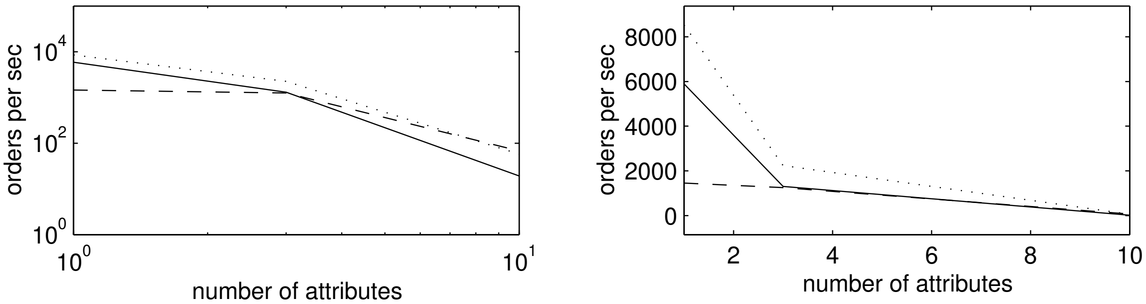


(c) Tests with 1,024 values per attribute, 131,072 orders, and matching density of 1.

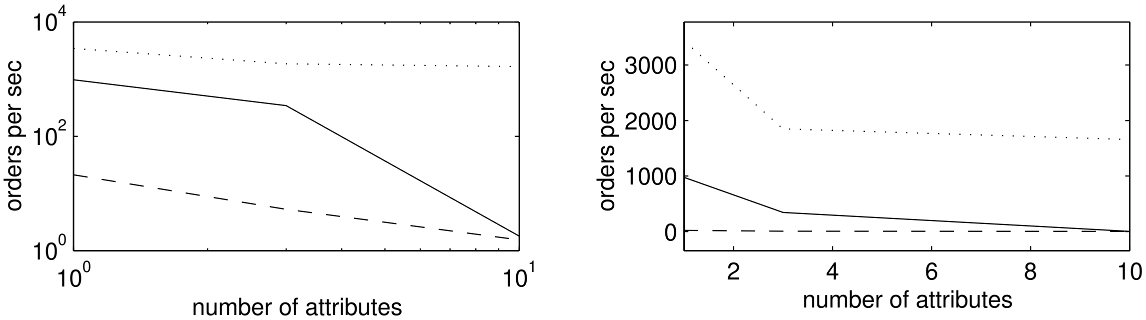
Figure 8.3: Dependency of the total matching time on the number of attributes. The legend is the same as in Figure 8.1.



(a) Tests with two values per attribute, 512 orders, and matching density of 0.001.

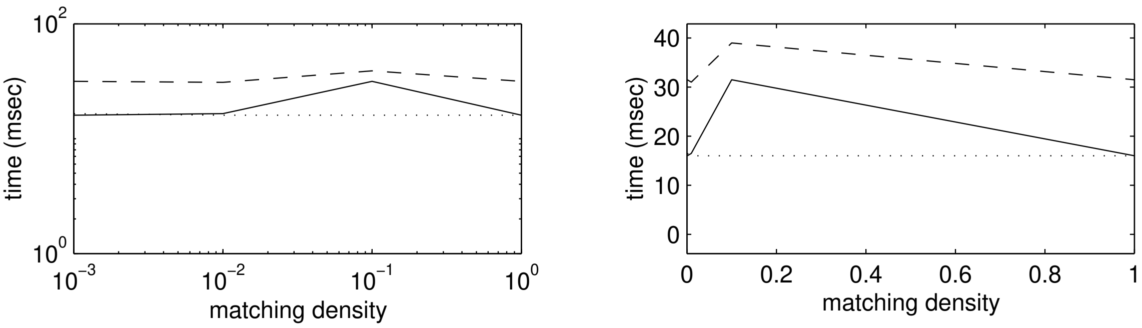


(b) Tests with sixteen values per attribute, 16,384 orders, and matching density of 0.01.

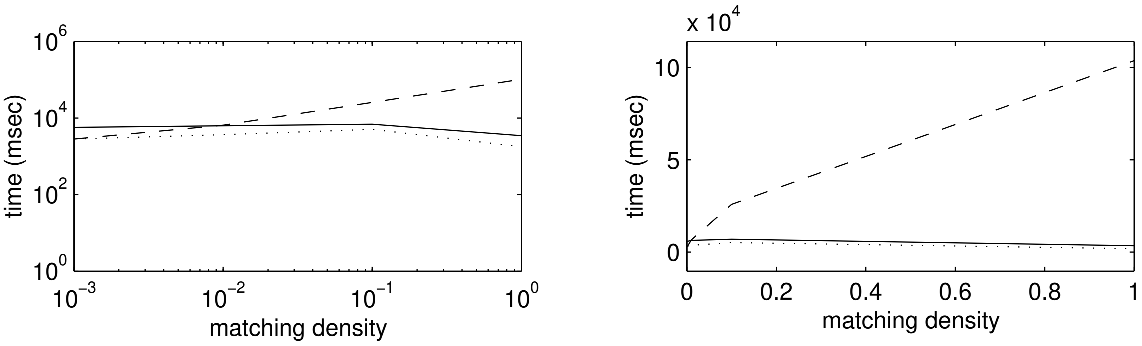


(c) Tests with 1,024 values per attribute, 131,072 orders, and matching density of 1.

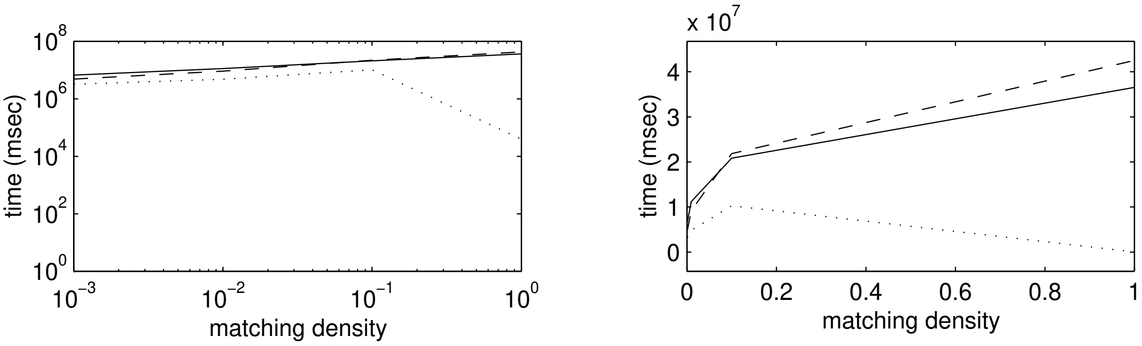
Figure 8.4: Dependency of the throughput on the number of attributes. The legend is the same as in Figure 8.1.



(a) Tests with one attribute, two values per attribute, and 512 orders.

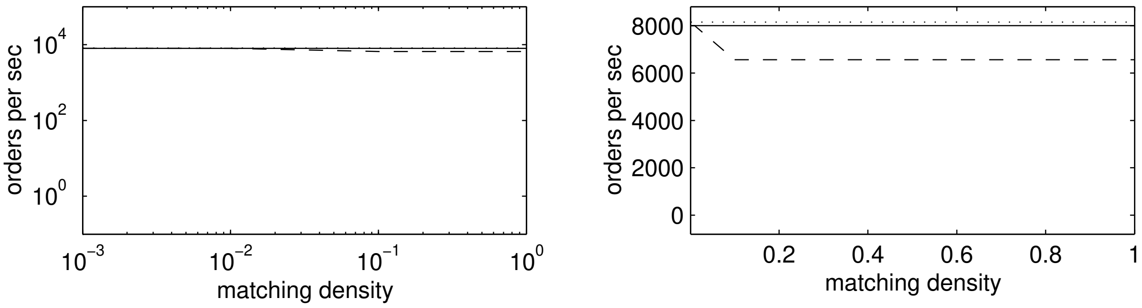


(b) Tests with three attributes, sixteen values per attribute, and 16,384 orders.

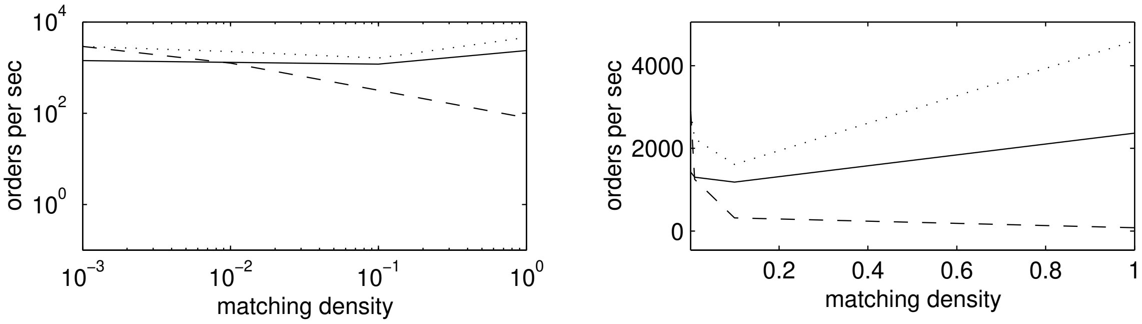


(c) Tests with ten attributes, 1,024 values per attribute, and 131,072 orders.

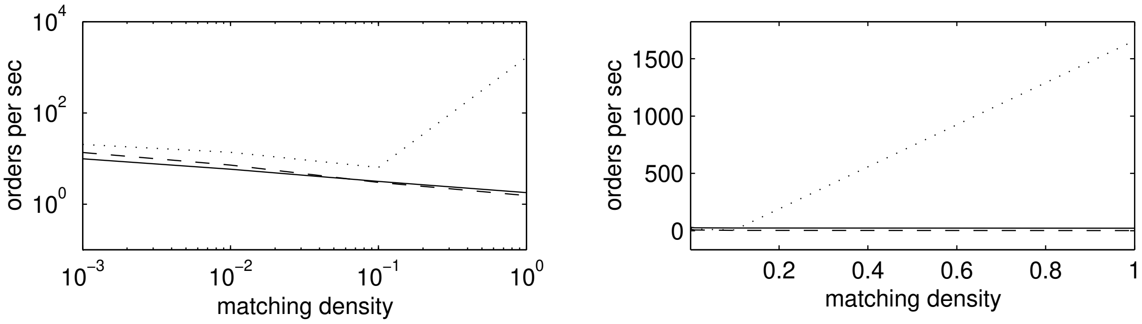
Figure 8.5: Dependency of the total matching time on the matching density. The legend is the same as in Figure 8.1.



(a) Tests with one attribute, two values per attribute, and 512 orders.



(b) Tests with three attributes, sixteen values per attribute, and 16,384 orders.



(c) Tests with ten attributes, 1,024 values per attribute, and 131,072 orders.

Figure 8.6: Dependency of the throughput on the matching density. The legend is the same as in Figure 8.1.

8.2 Real markets

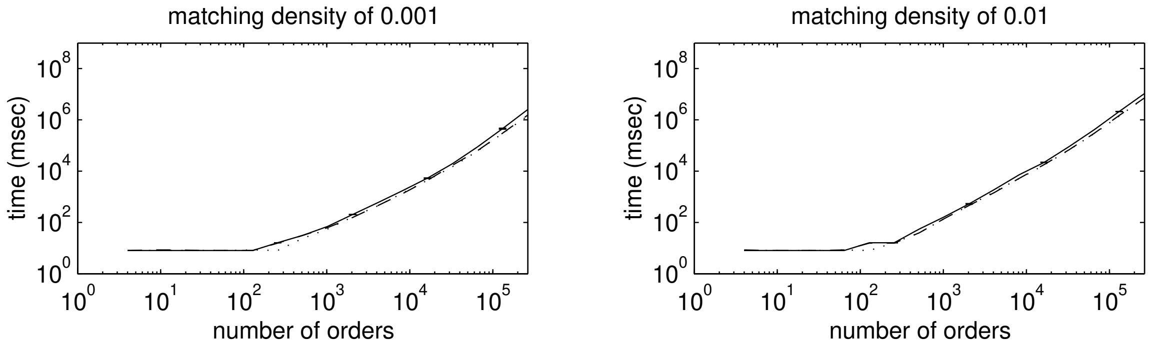
We have applied the system to an extended used-car market and to a commercial-paper market.

Used cars. We have considered a car market that includes all models offered by AutoNation (www.autonation.com), defined by eight attributes:

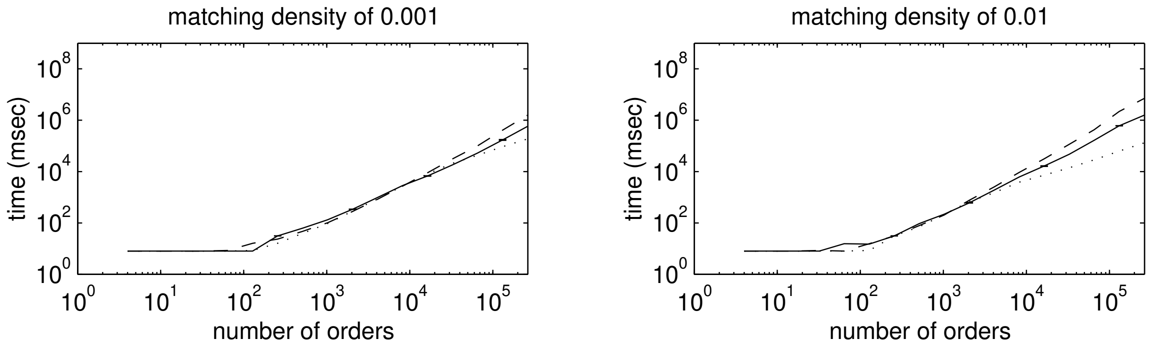
(1) Transmission (2 values): Manual, automatic.
(2) Number of doors (3 values): Two, three, four.
(3) Interior color (7 values): Black, gray, white, tan, brown, blue, red.
(4) Exterior color (52 values): All colors offered by AutoNation.
(5) Model (257 values): All models offered by AutoNation.
(6) Year (106 values): Integers from 1896 to 2001.
(7) Option package (1,024 values): Standard packages offered by AutoNation.
(8) Mileage (500,001 values): Integers from 0 to 500,000.

We have run experiments with up to 262,144 orders; the control variables have included the search strategy, number of orders, and matching density. We have considered the three search strategies, seventeen settings for the number of orders, and four settings for matching density. For each combination of settings, we have run two experiments; we show the results in Figures 8.7 and 8.8, and plot the dependency of the system's performance on the control variables in Figure 8.9–8.12.

The results in the car market are similar to the artificial-test results. The system supports markets with 262,144 orders, and it usually processes 40 to 4,000 new orders per second. The best-first search is more efficient than the depth-first search that identifies all matches; the saving factor in large markets varies from 1.0 to 8.4, with mean at 3.5. For markets with low matching density, the speed of the best-first strategy is close to that of the depth-first search with limited number of matches. On the other hand, for large high-density markets, the best-first search strategy is about hundred times slower than the limited depth-first search.

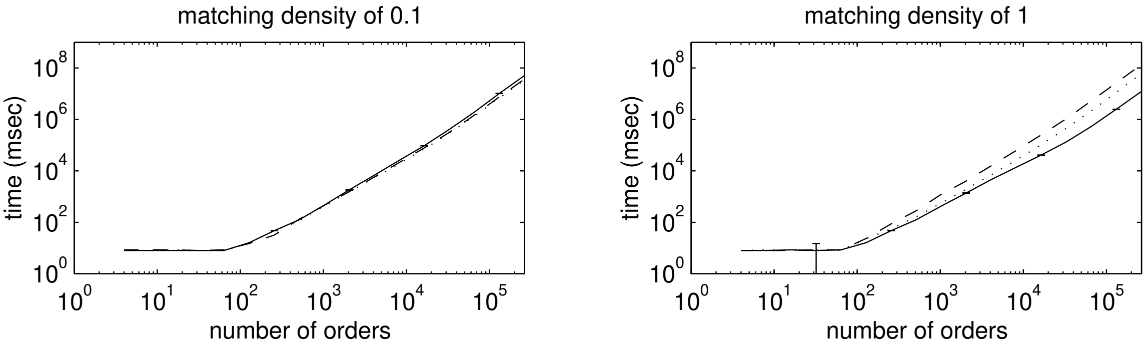


(a) Time of processing the new orders.

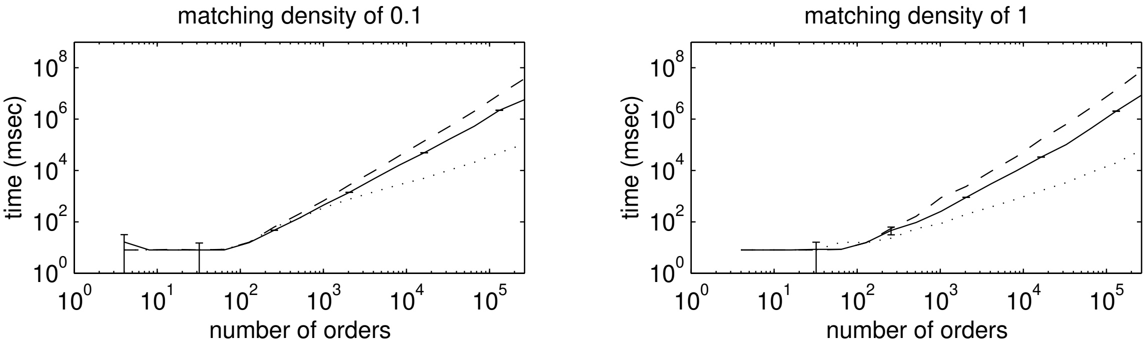


(b) Time of matching the old orders.

Figure 8.7: Performance in the car market for matching density of 0.001 (left) and 0.01 (right). We show the performance of the best-first search (solid lines), depth-first search that identifies all matches (dashed lines), and depth-first search with a limit on the number of matches (dotted lines).

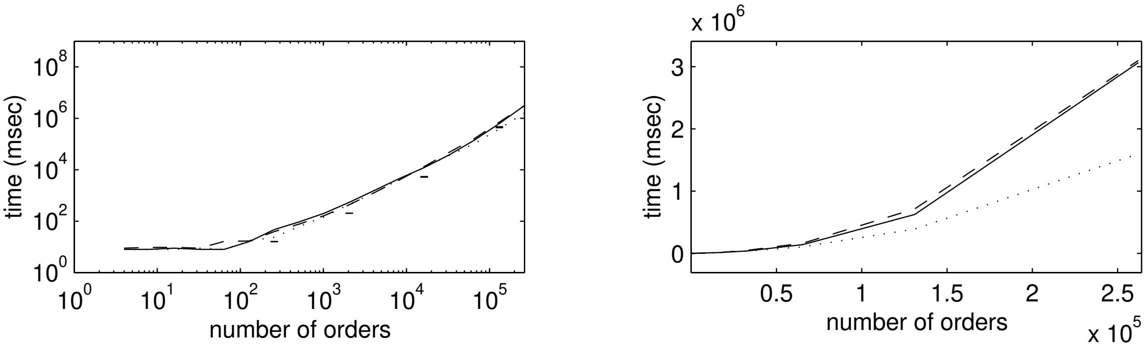


(a) Time of processing the new orders.

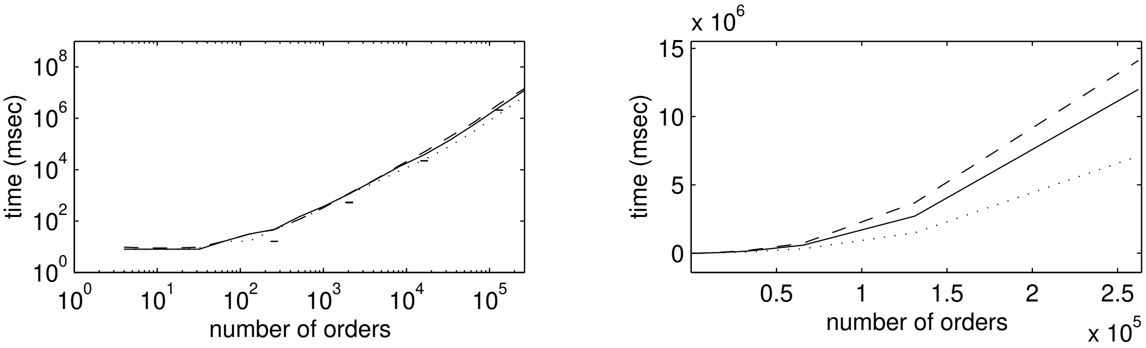


(b) Time of matching the old orders.

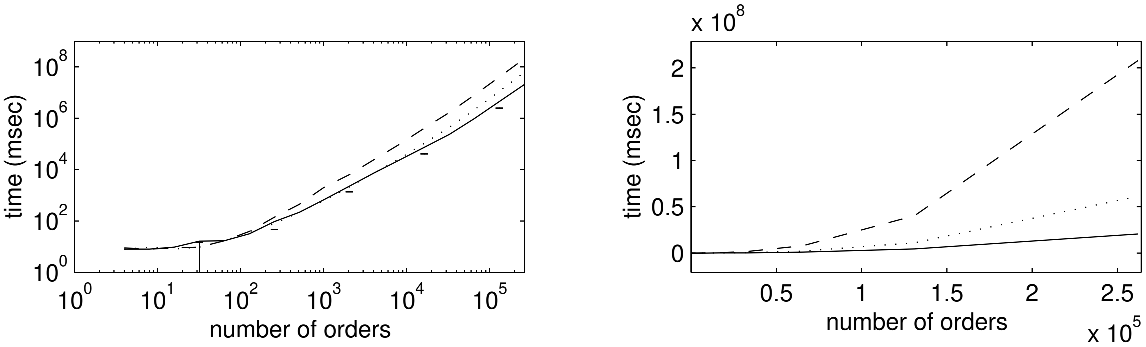
Figure 8.8: Performance in the car market for matching density of 0.1 (left) and 1 (right). The legend is the same as in Figure 8.7.



(a) Tests with matching density of 0.001.

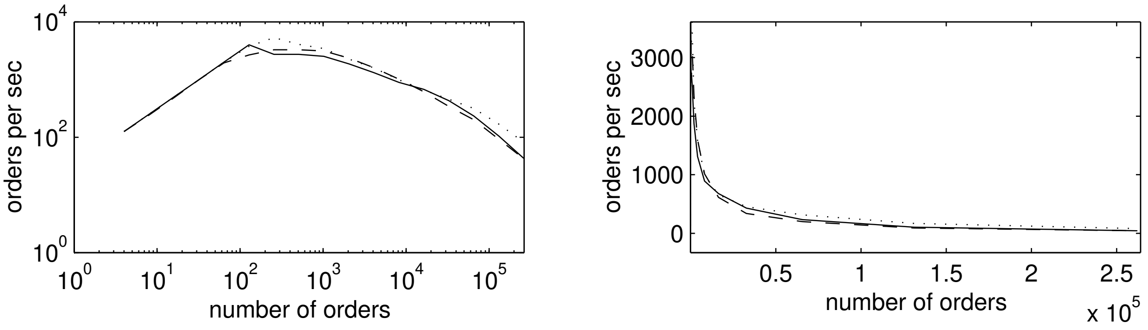


(b) Tests with matching density of 0.01.

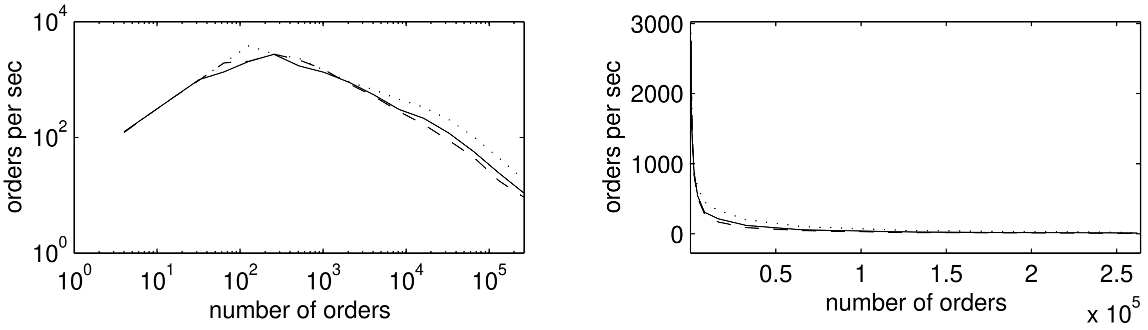


(c) Tests with matching density of 1.

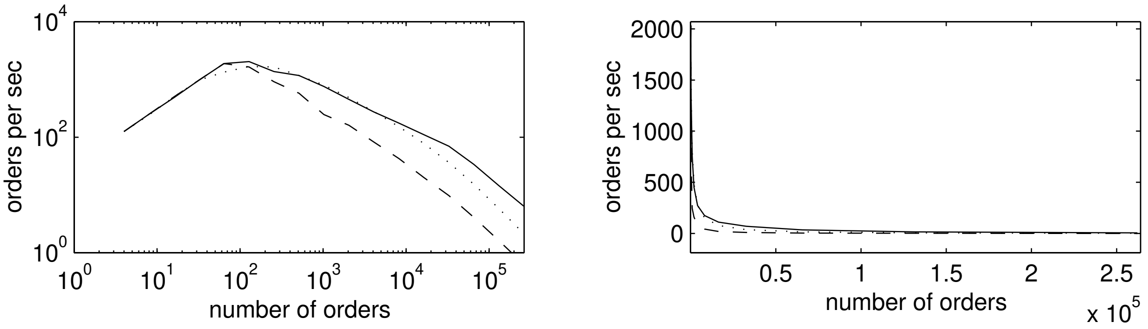
Figure 8.9: Dependency of the total matching time on the number of orders. We give the results for the best-first search (solid lines), depth-first search that identifies all matches (dashed lines), and depth-first search with a limit on the number of matches (dotted lines). The graphs on the left are in logarithmic scale, whereas the graphs on the right are in linear scale.



(a) Tests with matching density of 0.001.

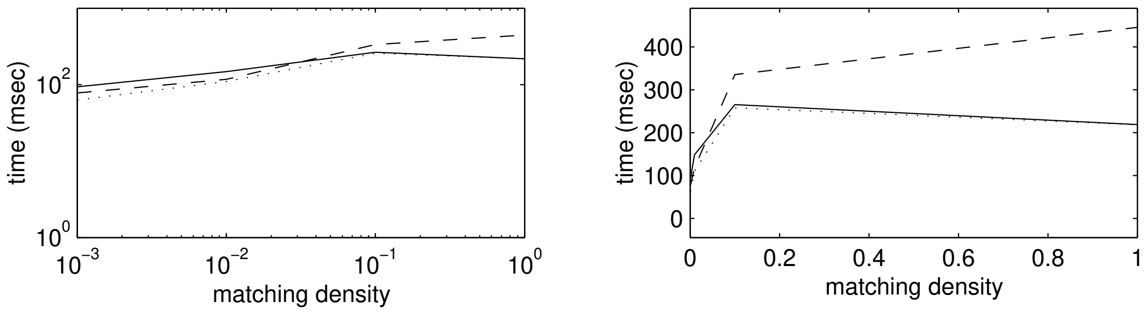


(b) Tests with matching density of 0.01.

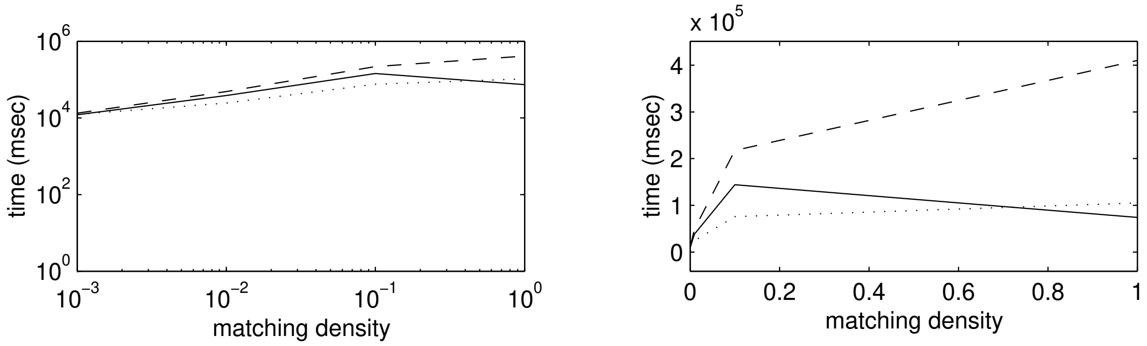


(c) Tests with matching density of 1.

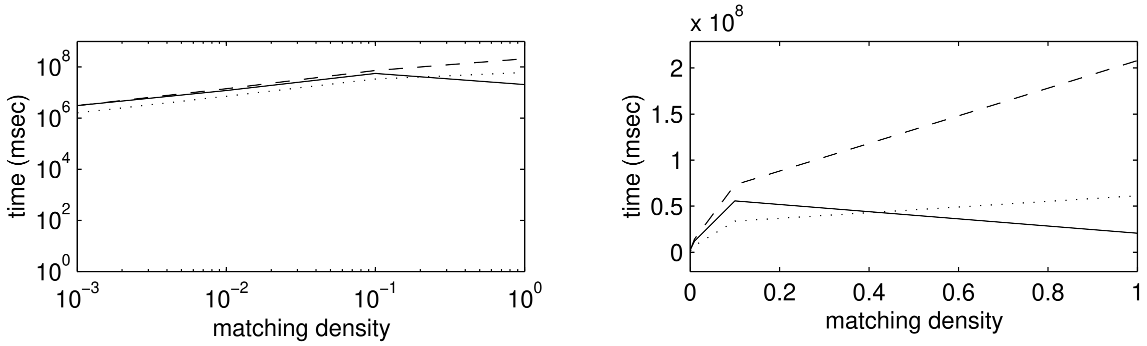
Figure 8.10: Dependency of the throughput on the number of orders. The legend is the same as in Figure 8.9.



(a) Tests with 512 orders.

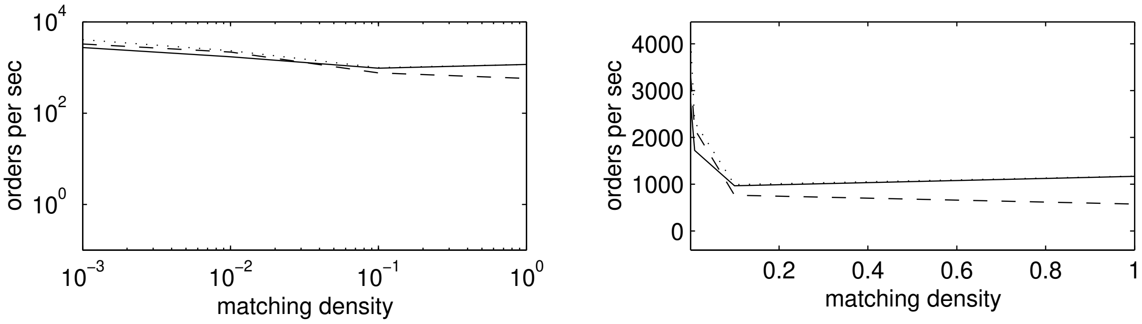


(b) Tests with 16,384 orders.

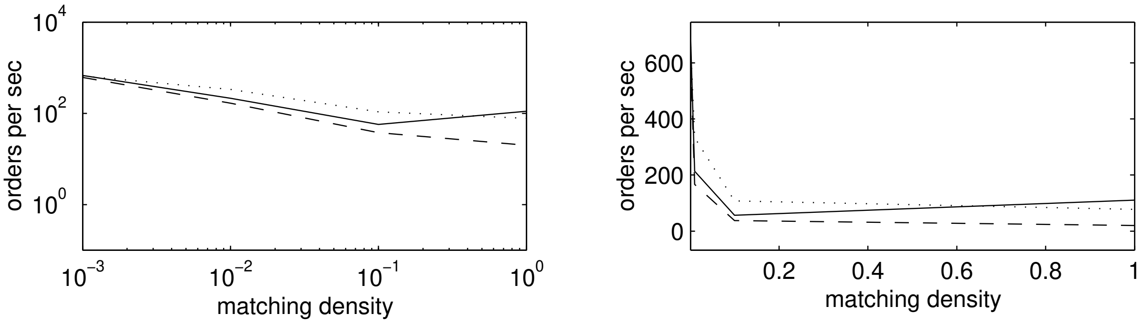


(c) Tests with 262,144 orders.

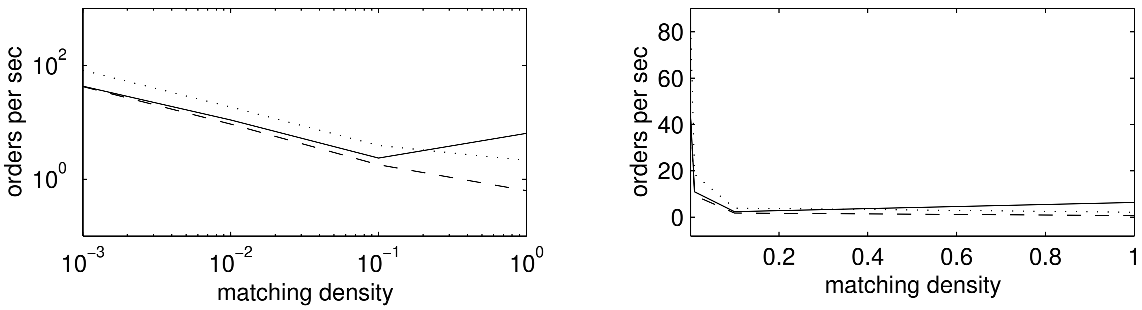
Figure 8.11: Dependency of the total matching time on the matching density. The legend is the same as in Figure 8.9.



(a) Tests with 512 orders.



(b) Tests with 16,384 orders.



(c) Tests with 262,144 orders.

Figure 8.12: Dependency of the throughput on the matching density. The legend is the same as in Figure 8.9.

Commercial paper. When a large company needs a short-term loan, it may issue *commercial paper,* which is a fixed-interest "promissory note" similar to a bond. The company sells commercial paper to investors for a certain period of time, and later returns their money with interest; the payment day is called the *maturity date.* The main difference from bonds is duration of the loan; commercial paper is issued for a short term, from one week to nine months. The appropriate interest depends on the current rate of US Treasury bonds, company's reputation, and paper's time until maturity.

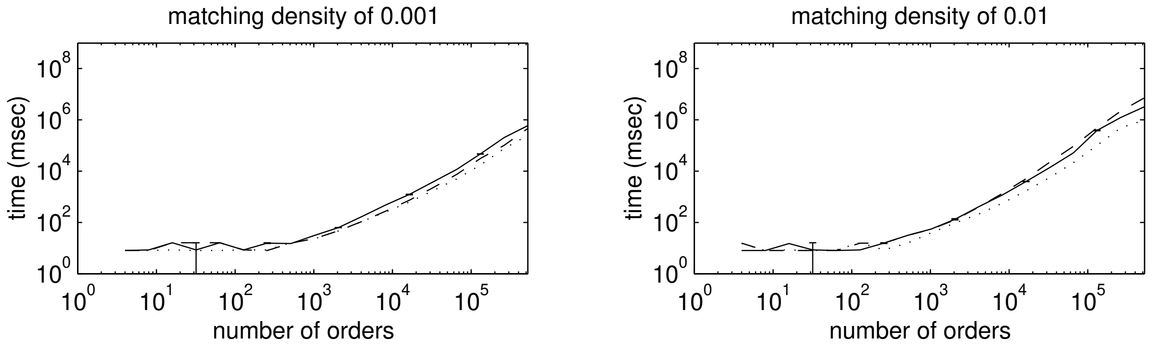
After investors buy a commercial paper, they may resell it on a secondary market before the maturity date. For example, suppose that Katie has bought a three-month paper in May, and then decided that she needs money in June. Then, she may resell the paper and keep part of the interest; if the rate has not changed, she will get one-month interest. On the other hand, if the interest rate of the company's paper has changed, the sale price may be different. If Katie is lucky, she may get more than one-month interest, but in an unfavorable case she may get smaller interest or even lose part of her investment.

We have described commercial paper by two attributes:

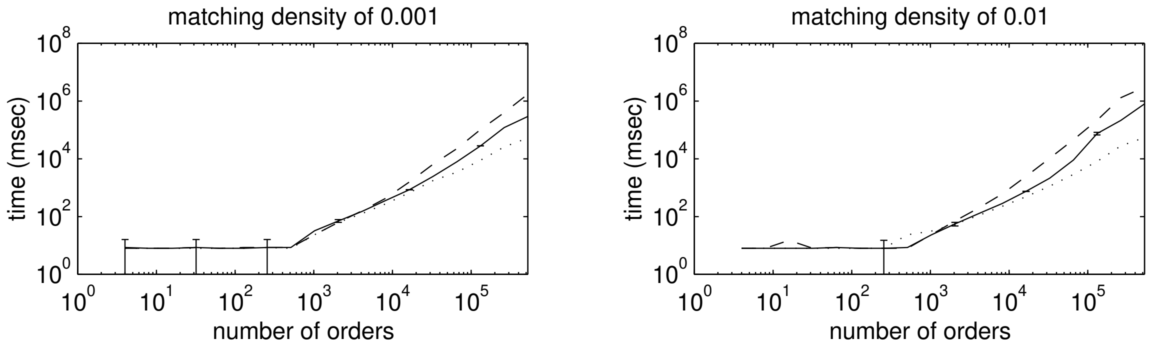
(1) Company (5000 values): 5000 US companies.
(2) Maturity date (2550 values): Business days from year 2001 to 2010.

We have run experiments with up to 524,288 orders; we show the results in Figures 8.13 and 8.14, and plot the dependency of the system's performance on the control variables in Figures 8.15–8.18. The experiments have confirmed that the system scales to large markets, and that its performance in real-life markets is close to the artificial-test results.

The best-first search system usually processes 100 to 10,000 new orders per second; it outperforms the depth-first search that identifies all matches by a factor of 2.3 to 8.8, with mean at 4.5. On the other hand, it is slower than the limited depth-first search; thus, the search for optimal matches takes more time than the suboptimal matching. This speed difference is especially significant in markets with high matching density; in particular, if the density is 1, the best-first search is hundred times slower than the limited depth-first search.

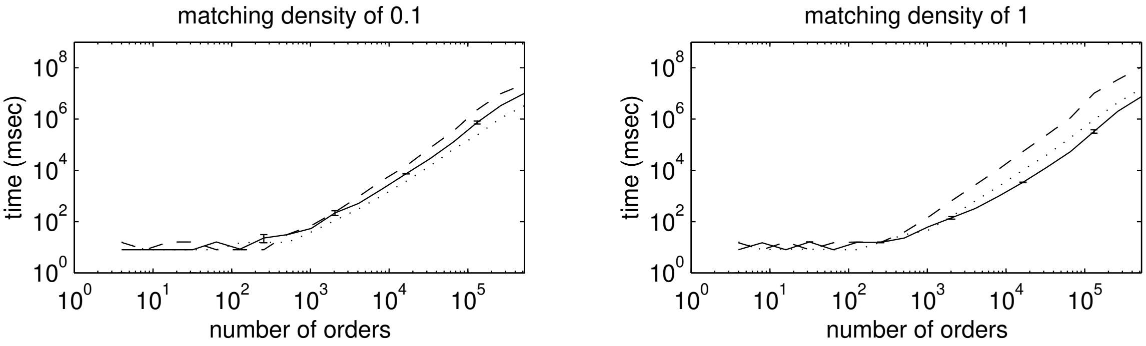


(a) Time of processing the new orders.

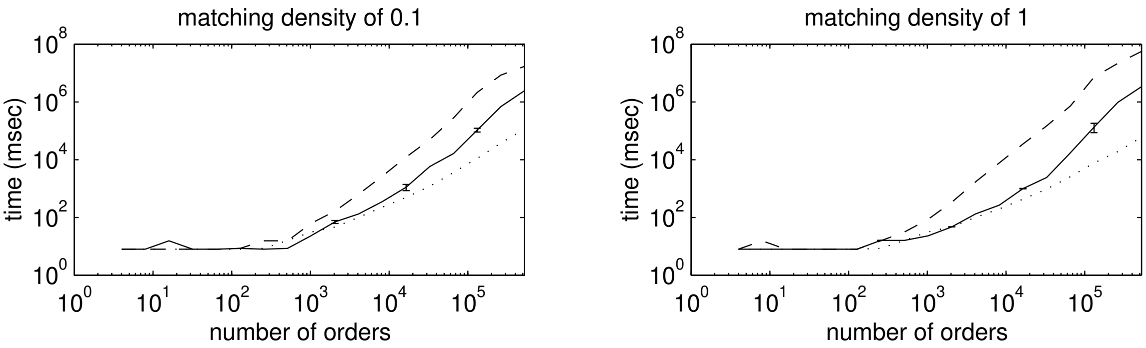


(b) Time of matching the old orders.

Figure 8.13: Performance in the commercial-paper market for matching density of 0.001 (left) and 0.01 (right). We show the performance of the best-first search (solid lines), depth-first search that identifies all matches (dashed lines), and depth-first search with a limit on the number of matches (dotted lines).

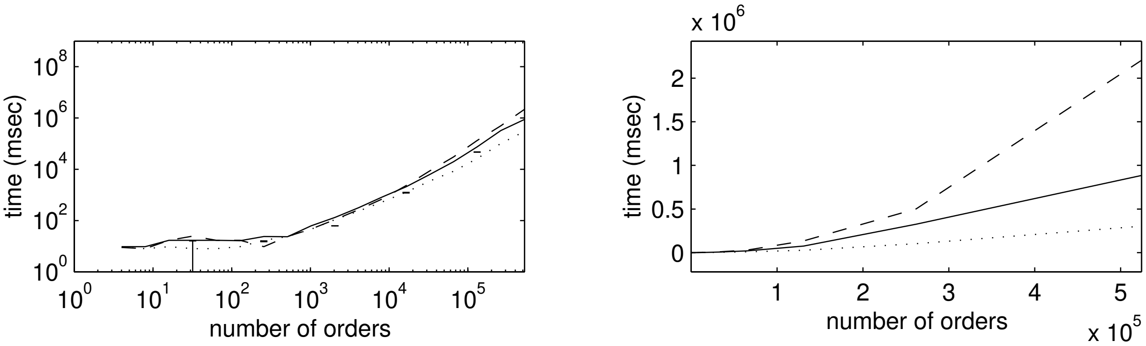


(a) Time of processing the new orders.

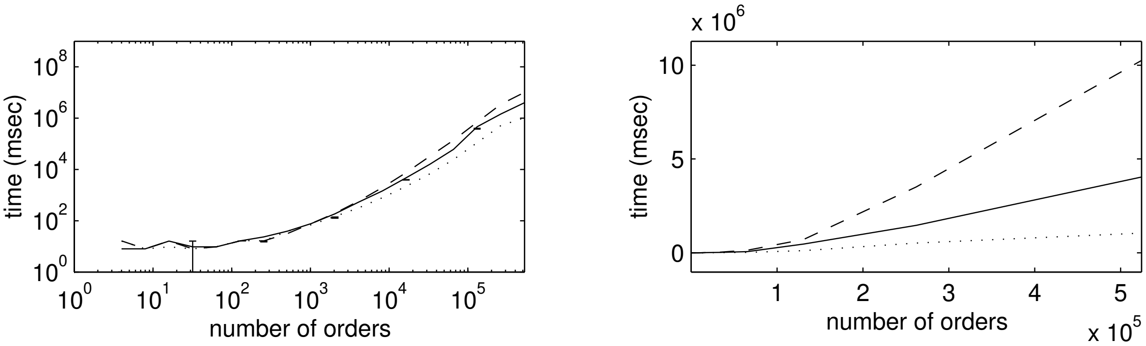


(b) Time of matching the old orders.

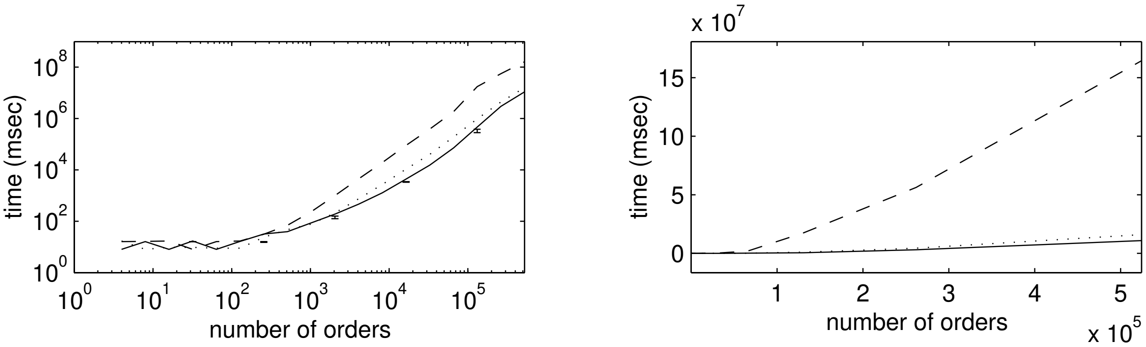
Figure 8.14: Performance in the commercial-paper market for matching density of 0.1 (left) and 1 (right). The legend is the same as in Figure 8.13.



(a) Tests with matching density of 0.001.

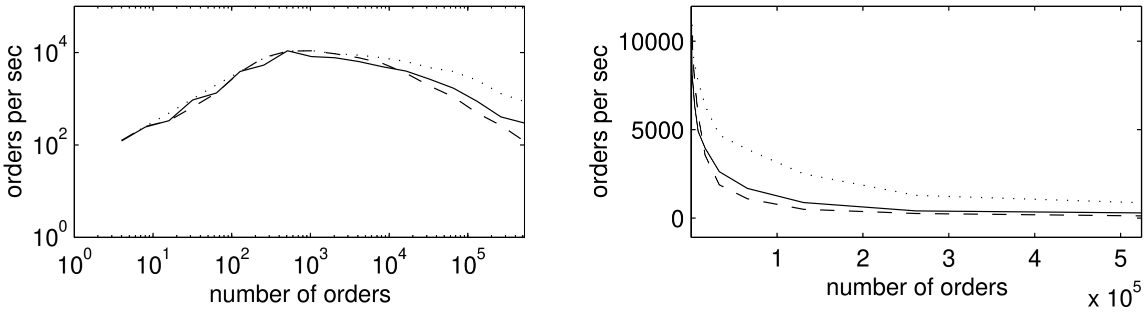


(b) Tests with matching density of 0.01.

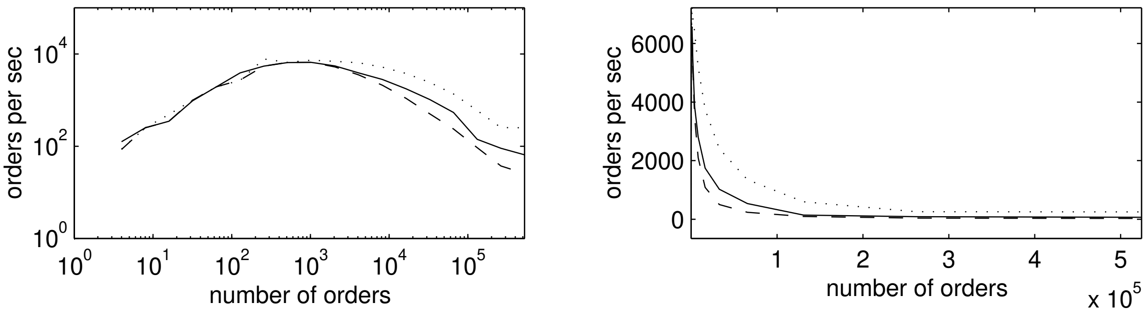


(c) Tests with matching density of 1.

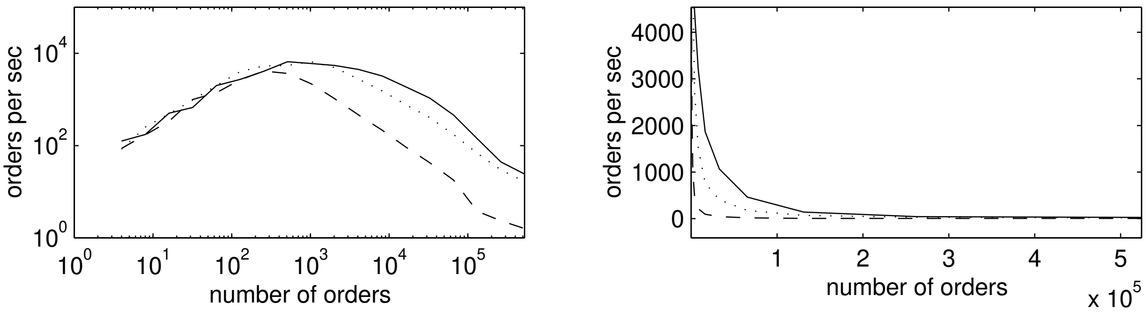
Figure 8.15: Dependency of the total matching time on the number of orders. We give the results for the best-first search (solid lines), depth-first search that identifies all matches (dashed lines), and depth-first search with a limit on the number of matches (dotted lines). The graphs on the left are in logarithmic scale, whereas the graphs on the right are in linear scale.



(a) Tests with matching density of 0.001.

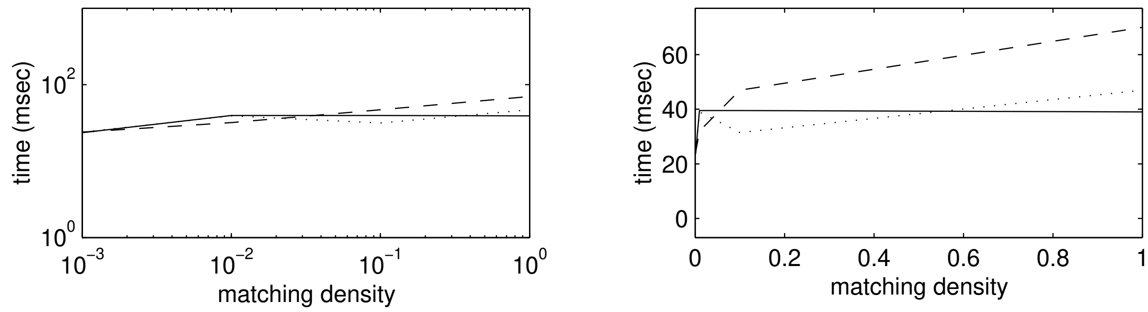


(b) Tests with matching density of 0.01.

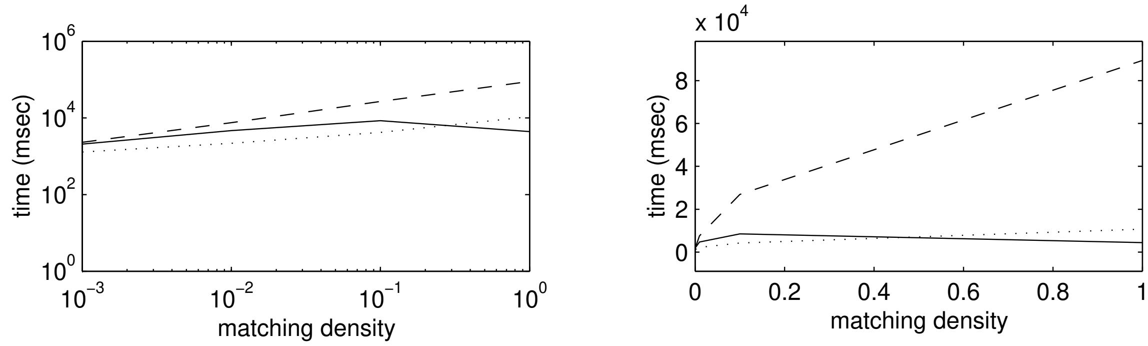


(c) Tests with matching density of 1.

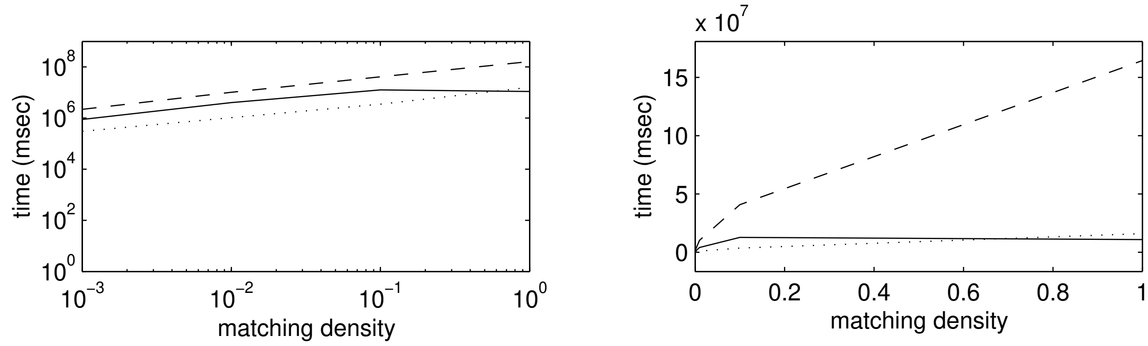
Figure 8.16: Dependency of the throughput on the number of orders. The legend is the same as in Figure 8.15.



(a) Tests with 512 orders.

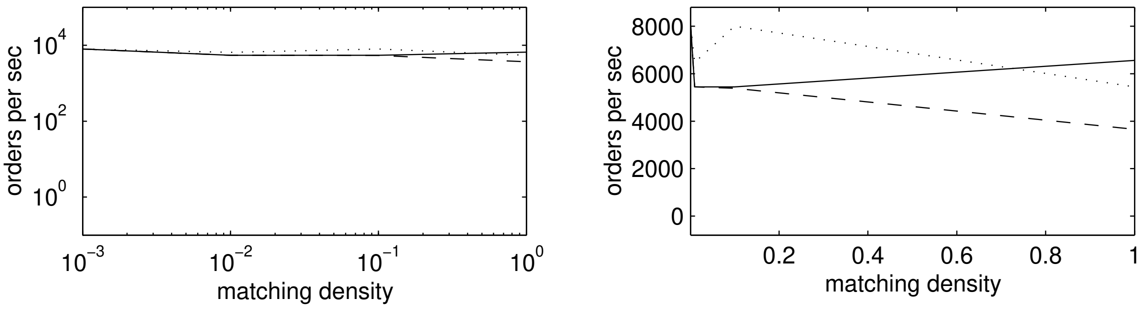


(b) Tests with 16,384 orders.

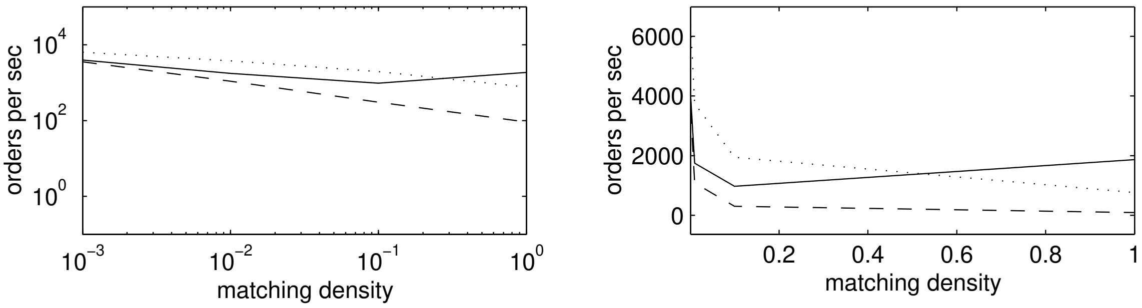


(c) Tests with 524,288 orders.

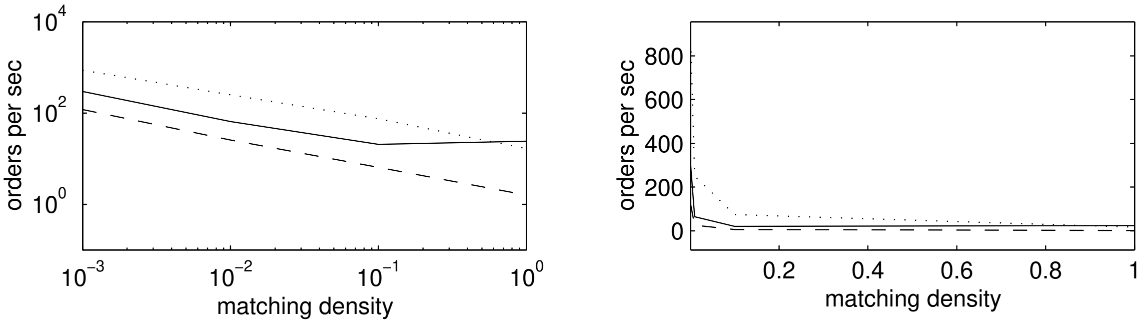
Figure 8.17: Dependency of the total matching time on the matching density. The legend is the same as in Figure 8.15.



(a) Tests with 512 orders.



(b) Tests with 16,384 orders.



(c) Tests with 524,288 orders.

Figure 8.18: Dependency of the throughput on the matching density. The legend is the same as in Figure 8.15.

Chapter 9

Concluding remarks

Although researchers have long realized the importance of exchange markets, they have not applied the exchange model to trading complex commodities. The reported work is a step toward the development of automated complex-commodity exchanges, based on the formal model proposed by Johnson [2001] and Hu [2002].

We have defined price and quality functions, which allow traders to specify price constraints and preference among potential trades, and developed algorithms for fast identification of highest-quality matches between buy and sell orders. These algorithms help to maximize the satisfaction of traders and enforce "fair" choices among available matches, which are consistent with financial-industry rules of fair trading.

The implemented system supports markets with up to 300,000 orders, and it processes hundreds of new orders per second. Its speed is close to the speed of the earlier versions of the system, which did not use price and quality functions and did not guarantee finding best matches.

References

[Akcoglu *et al.*, 2002] Karhan Akcoglu, James Aspnes, Bhaskar DasGupta, and Ming-Yang Kao. Opportunity cost algorithms for combinatorial auctions. In Erricos John Kontoghiorghes, Berç Rustem, and Stavros Siokos, editors, *Applied Optimization: Computational Methods in Decision-Making, Economics and Finance*. Kluwer Academic Publishers, Boston, MA, 2002. To appear.

[Andersson and Ygge, 1998] Arne Andersson and Fredrik Ygge. Managing large scale computational markets. In *Proceedings of the Thirty-First Hawaii International Conference on System Sciences*, volume VII, pages 4–13, 1998.

[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.

[Babaioff and Nisan, 2001] Moshe Babaioff and Noam Nisan. Concurrent auctions across the supply chain. In *Proceedings of the Third* ACM *Conference on Electronic Commerce*, pages 1–10, 2001.

[Bakos, 2001] Yannis Bakos. The emerging landscape for retail e-commerce. *Journal of Ecomonic Perspectives*, 15(1):69–80, 2001.

[Bapna *et al.*, 2000] Ravi Bapna, Paulo Goes, and Alok Gupta. A theoretical and empirical investigation of multi-item on-line auctions. *Information Technology and Management*, 1(1):1–23, 2000.

[Benyoucef *et al.*, 2001] Morad Benyoucef, Sarita Bassil, and Rudolf K. Keller. Workflow modeling of combined negotiations in e-commerce. In *Proceedings of the Fourth International Conference on Electronic Commerce Research*, pages 348–359, 2001.

[Bernstein, 1993] Peter L. Bernstein. *Capital Ideas: The Improbable Origins of Modern Wall Street*. The Free Press, New York, NY, 1993.

[Bichler and Segev, 1999] Martin Bichler and Arie Segev. A brokerage framework for Internet commerce. *Distributed and Parallel Databases*, 7(2):133–148, 1999.

[Bichler *et al.*, 1998] Martin Bichler, Arie Segev, and Carrie Beam. An electronic broker for business-to-business electronic commerce on the Internet. *International Journal of Cooperative Information Systems*, 7(4):315–329, 1998.

[Bichler *et al.*, 1999] Martin Bichler, Marion Kaukal, and Arie Segev. Multi-attribute auctions for electronic procurement. In *Proceedings of the First* IAC *Workshop on Internet Based Negotiation Technologies*, 1999.

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

[Bichler, 2000b] Martin Bichler. A roadmap to auction-based negotiation protocols for electronic commerce. In *Proceedings of the Thirty-Third Hawaii International Conference on System Sciences*, 2000.

[Blum *et al.*, 2002] Avrim Blum, Tuomas W. Sandholm, and Martin Zinkevich. Online algorithms for market clearing. In *Proceedings of the Thirteenth Annual* ACM-SIAM *Symposium on Discrete Algorithms*, 2002.

[Boutilier and Hoos, 2001] Craig Boutilier and Holger H. Hoos. Bidding languages for combinatorial auctions. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 1211–1217, 2001.

[Boyan *et al.*, 2001] Justin Boyan, Amy Greenwald, R. Mike Kirby, and Jon Reiter. Bidding algorithms for simultaneous auctions. In *Proceedings of the Third* ACM *Conference on Electronic Commerce*, pages 115–124, 2001.

[Cason and Friedman, 1996] Timothy N. Cason and Daniel Friedman. Price formation in double auction markets. *Journal of Economic Dynamics and Control*, 20:1307–1337, 1996.

[Cason and Friedman, 1999] Timothy N. Cason and Daniel Friedman. Price formation and exchange in thin markets: A laboratory comparison of institutions. In

Peter Howitt, Elisabetta de Antoni, and Axel Leijonhufvud, editors, *Money, Markets and Method: Essays in Honour of Robert W. Clower*, pages 155–179. Edward Elgar, Cheltenham, United Kingdom, 1999.

[Chavez and Maes, 1996] Anthony Chavez and Pattie Maes. Kasbah: An agent marketplace for buying and selling goods. In *Proceedings of the First International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, pages 75–90, 1996.

[Chavez *et al.*, 1997] Anthony Chavez, Daniel Dreilinger, Robert Guttman, and Pattie Maes. A real-life experiment in creating an agent marketplace. In *Proceedings of the Second International Conference on the Practical Application of Intelligent Agents and Multi-Agent Technology*, pages 159–178, 1997.

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

[Chen *et al.*, 2001] Chunming Chen, Muthucumaru Maheswaran, and Michel Toulouse. On bid selection heuristics for real-time auctioning for wide-area network resource management. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 2001.

[Cheng and Wellman, 1998] John Cheng and Michael Wellman. The WALRAS algorithm: A convergent distributed implementation of general equilibrium outcomes. *Computational Economics*, 12(1):1–24, 1998.

[Conen and Sandholm, 2001] Wolfram Conen and Tuomas W. Sandholm. Minimal preference elicitation in combinatorial auctions. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, Workshop on Economic Agents, Models, and Mechanisms*, pages 71–80, 2001.

[Conen and Sandholm, 2002] Wolfram Conen and Tuomas W. Sandholm. Partial-revelation VCG mechanism for combinatorial auctions. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, 2002.

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

[de Vries and Vohra, 2002] Sven de Vries and Rakesh V. Vohra. Combinatorial auctions: A survey. INFORMS *Journal of Computing*, 2002. To appear.

[Feldman, 2000] Stuart Feldman. Electronic marketplaces. IEEE *Internet Computing*, 4(4):93–95, 2000.

[Fujishima *et al.*, 1999a] Yuzo Fujishima, Kevin Leyton-Brown, and Yoav Shoham. Speeding up ascending-bid auctions. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, pages 554–563, 1999.

[Fujishima *et al.*, 1999b] Yuzo Fujishima, Kevin Leyton-Brown, and Yoav Shoham. Taming the computational complexity of combinatorial auctions: Optimal and approximate approaches. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, volume 1, pages 548–553, 1999.

[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.

[Guttman *et al.*, 1998a] Robert H. Guttman, Alexandros G. Moukas, and Pattie Maes. Agent-mediated electronic commerce: A survey. *Knowledge Engineering Review*, 13(2):147–159, 1998.

[Guttman *et al.*, 1998b] Robert H. Guttman, Alexandros G. Moukas, and Pattie Maes. Agents as mediators in electronic commerce. *International Journal of Electronic Markets*, 8(1):22–27, 1998.

[Hoos and Boutilier, 2000] Holger H. Hoos and Craig Boutilier. Solving combinatorial auctions using stochastic local search. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 22–29, 2000.

[Hu and Wellman, 2001] Junling Hu and Michael P. Wellman. Learning about other agents in a dynamic multiagent system. *Journal of Cognitive Systems Research*, 1:67–79, 2001.

[Hu *et al.*, 1999] Junling Hu, Daniel Reeves, and Hock-Shan Wong. Agents participating in Internet auctions. In *Proceeings of the* AAAI *Workshop on Artificial Intelligence for Electronic Commerce*, 1999.

[Hu *et al.*, 2000] Junling Hu, Daniel Reeves, and Hock-Shan Wong. Personalized bidding agents for online auctions. In *Proceedings of the Fifth International Conference on the Practical Application of Intelligent Agents and Multi-Agents*, pages 167–184, 2000.

[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.

[Hull, 1999] John C. Hull. *Options, Futures, and Other Derivatives*. Prentice Hall, Upper Saddle River, NJ, fourth edition, 1999.

[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.

[Jones and Koehler, 2000] Joni L. Jones and Gary J. Koehler. Multiple criteria combinatorial auction: A B2B allocation mechanism for substitute goods. In *Proceedings of the Americas Conference on Information Systems*, 2000.

[Jones, 2000] Joni L. Jones. *Incompletely Specified Combinatorial Auction: An Alternative Allocation Mechanism for Business-to-Business Negotiations*. PhD thesis, Warrington College of Business, University of Florida, 2000.

[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.

[Klein, 1997] Stefan Klein. Introduction to electronic auctions. *International Journal of Electronic Markets*, 7(4):3–6, 1997.

[Kumar and Feldman, 1998] Manoj Kumar and Stuart I. Feldman. Internet auctions. In *Proceedings of the Third* USENIX *Workshop or Electronic Commerce*, pages 49–60, 1998.

[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.

[Lehmann *et al.*, 1999] Daniel Lehmann, Liadan Ita O'Callaghan, and Yoav Shoham. Truth revelation in rapid, approximately efficient combinatorial auctions. In *Proceedings of the First* ACM *Conference on Electronic Commerce*, pages 96–102, 1999.

[Lehmann *et al.*, 2001] 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.

[Leyton-Brown *et al.*, 2000] Kevin Leyton-Brown, Yoav Shoham, and Moshe Tennenholtz. An algorithm for multi-unit combinatorial auctions. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 56–61, 2000.

[Maes *et al.*, 1999] Pattie Maes, Robert H. Guttman, and Alexandros G. Moukas. Agents that buy and sell: Transforming commerce as we know it. *Communications of the* ACM, 42(3):81–91, 1999.

[Monderer and Tennenholtz, 2000] Dov Monderer and Moshe Tennenholtz. Optimal auctions revisited. *Artificial Intelligence*, 120:29–42, 2000.

[Mu'alem and Nisan, 2002] Ahuva Mu'alem and Noam Nisan. Truthful approximation mechanisms for restricted combinatorial auctions. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence*, 2002.

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

[Parkes and Ungar, 2000] David C. Parkes and Lyle H. Ungar. Iterative combinatorial auctions: Theory and practice. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence*, pages 74–81, 2000.

[Parkes, 1999] David C. Parkes. *i*Bundle: An efficient ascending price bundle auction. In *Proceedings of the First* ACM *Conference on Electronic Commerce*, pages 148–157, 1999.

[Preist *et al.*, 2001] Chris Preist, Andrew Byde, Claudio Bartolini, and Giacomo Piccinelli. Towards agent-based service composition through negotiation in multiple auctions. Technical Report HPL-2001-71, Hewlett Packard, 2001.

[Preist, 1999a] Chris Preist. Commodity trading using an agent-based iterated double auction. In *Proceedings of the Third Annual Conference on Autonomous Agents*, pages 131–138, 1999.

[Preist, 1999b] Chris Preist. Economic agents for automated trading. In Alex L. G. Hayzelden and John Bigham, editors, *Software Agents for Future Communication Systems*, pages 207–220. Springer-Verlag, Berlin, Germany, 1999.

[Reiter and Simon, 1992] Stanley Reiter and Carl Simon. Decentralized dynamic processes for finding equilibrium. *Journal of Economic Theory*, 56(2):400–425, 1992.

[Ronen, 2001] Amir Ronen. On approximating optimal auctions. In *Proceedings of the Third* ACM *Conference on Electronic Commerce*, pages 11–17, 2001.

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

[Sakurai *et al.*, 2000] Yuko Sakurai, Makoto Yokoo, and Koji Kamei. An efficient approximate algorithm for winner determination in combinatorial auctions. In *Proceedings of the Second* ACM *Conference on Electronic Commerce*, pages 30–37, 2000.

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

[Sandholm and Suri, 2001b] Tuomas W. Sandholm and Subhash Suri. Side constraints and non-price attributes in markets. In *Proceedings of the International Joint Conference on Artificial Intelligece, Workshop on Distributed Constraint Reasoning*, 2001.

[Sandholm *et al.*, 2001a] 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.

[Sandholm *et al.*, 2001b] Tuomas W. Sandholm, Subhash Suri, Andrew Gilpin, and David Levine. Winner determination in combinatorial auction generalizations. In *Proceedings of the International Conference on Autonomous Agents, Workshop on Agent-Based Approaches to* B2B, pages 35–41, 2001.

[Sandholm, 1999] Tuomas W. Sandholm. An algorithm for optimal winner determination in combinatorial auctions. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence*, volume 1, pages 542–547, 1999.

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

[Sandholm, 2000b] Tuomas W. Sandholm. eMediator: A next generation electronic commerce server. In *Proceedings of the Fourth International Conference on Autonomous Agents*, pages 341–348, 2000.

[Suzuki and Yokoo, 2002] Koutarou Suzuki and Makoto Yokoo. Secure combinatorial auctions by dynamic programming with polynomial secret sharing. In *Proceedings of the Sixth International Financial Cryptography Conference*, 2002.

[Tesauro and Das, 2001] Gerald Tesauro and Rajarshi Das. High-performance bidding agents for the continuous double auction. In *Proceedings of the Third* ACM *Conference on Electronic Commerce*, pages 206–209, 2001.

[Turban, 1997] Efraim Turban. Auctions and bidding on the Internet: An assessment. *International Journal of Electronic Markets*, 7(4):7–11, 1997.

[Vetter and Pitsch, 1999] Michael Vetter and Stefan Pitsch. An agent-based market supporting multiple auction protocols. In *Proceedings of the Workshop on Agents for Electronic Commerce and Managing the Internet-Enabled Supply Chain*, 1999.

[Wellman and Wurman, 1998] Michael P. Wellman and Peter R. Wurman. Real time issues for Internet auctions. In *Proceedings of the First* IEEE *Workshop on Dependable and Real-Time E-Commerce Systems*, 1998.

[Wellman *et al.*, 2001] Michael P. Wellman, William E. Walsh, Peter R. Wurman, and Jeffrey K. MacKie-Mason. Auction protocols for decentralized scheduling. *Games and Economic Behavior*, 35:271–303, 2001.

[Wellman, 1993] Michael P. Wellman. A market-oriented programming environment and its application to distributed multicommodity flow problems. *Journal of Artificial Intelligence Research*, 1:1–23, 1993.

[Wrigley, 1997] Clive D. Wrigley. Design criteria for electronic market servers. *International Journal of Electronic Markets*, 7(4):12–16, 1997.

[Wurman and Wellman, 1999] Peter R. Wurman and Michael P. Wellman. Control architecture for a flexible Internet auction server. In *Proceedings of the First* IAC *Workshop on Internet Based Negotiation Technologies*, 1999.

[Wurman *et al.*, 1998a] 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.

[Wurman *et al.*, 1998b] 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.

[Wurman *et al.*, 2001] Peter R. Wurman, Michael P. Wellman, and William E. Walsh. A parametrization of the auction design space. *Games and Economic Behavior*, 35(1–2):304–338, 2001.

[Wurman, 2001] Peter R. Wurman. Toward flexible trading agents. In *Proceedings of the* AAAI *Spring Symposium on Game Theoretic and Decision Theoretic Agents*, pages 134–140, 2001.

[Ygge and Akkermans, 1997] Fredrik Ygge and Hans Akkermans. Duality in multi-commodity market computations. In *Proceedings of the Third Australian Workshop on Distributed Artificial Intelligence*, pages 65–78, 1997.

[Yokoo *et al.*, 2001a] Makoto Yokoo, Yuko Sakurai, and Shigeo Matsubara. Bundle design in robust combinatorial auction protocol against false-name bids. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 1095–1101, 2001.

[Yokoo *et al.*, 2001b] Makoto Yokoo, Yuko Sakurai, and Shigeo Matsubara. Robust combinatorial auction protocol against false-name bids. *Artificial Intelligence*, 130(2):167–181, 2001.

[Zurel and Nisan, 2001] Edo Zurel and Noam Nisan. An efficient approximate allocation algorithm for combinatorial auctions. In *Proceedings of the Third* ACM *Conference on Electronic Commerce*, pages 125–136, 2001.

Appendices

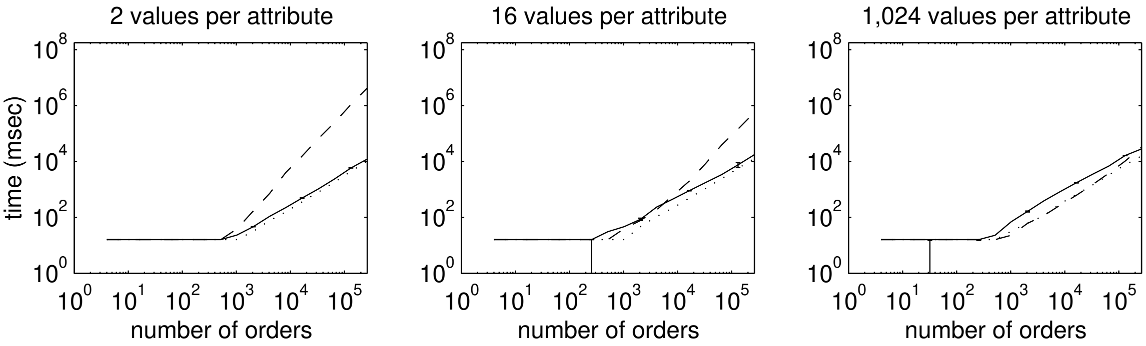
Appendix A

Artificial markets

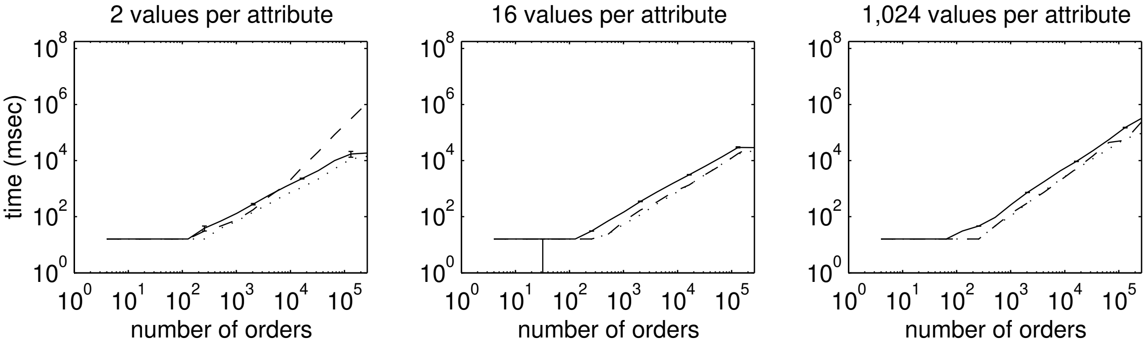
We give detailed results of artificial-market experiments described in Section 8.1.

A.1 Processing time

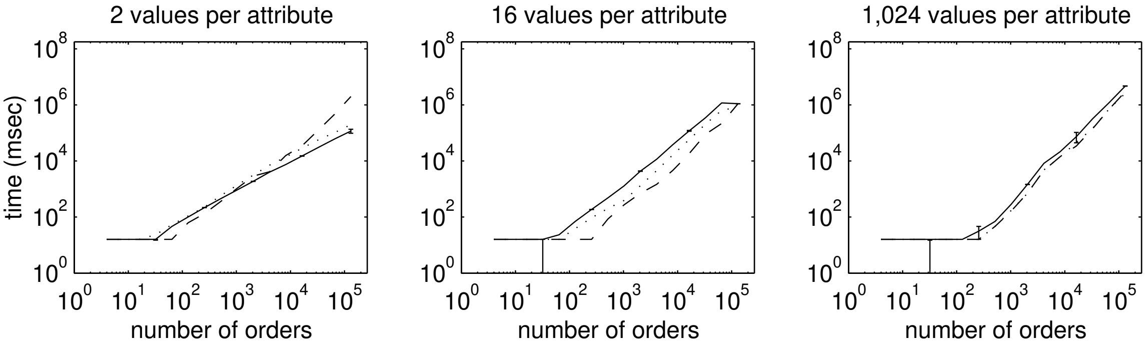
We show the mean time of processing new orders in the first half of the system's top-level loop (Figure 6.3) for various settings of the control variables. We also mark the minimal and maximal values of the time measurements.



(a) Market with one attribute.

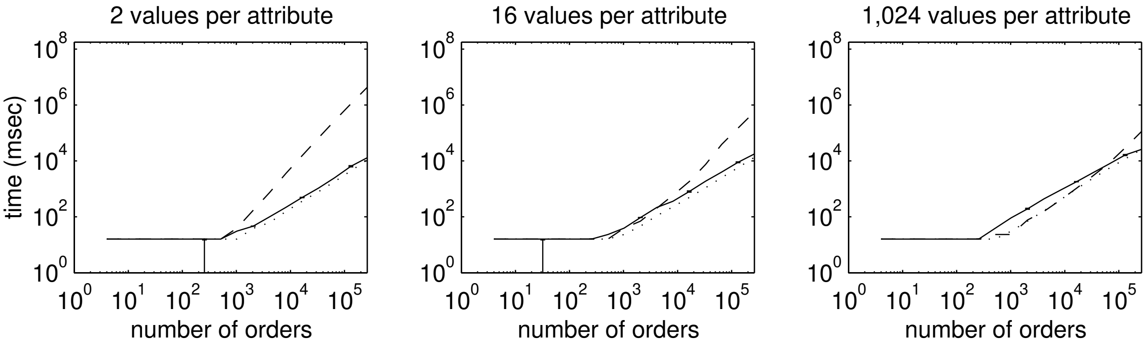


(b) Market with three attributes.

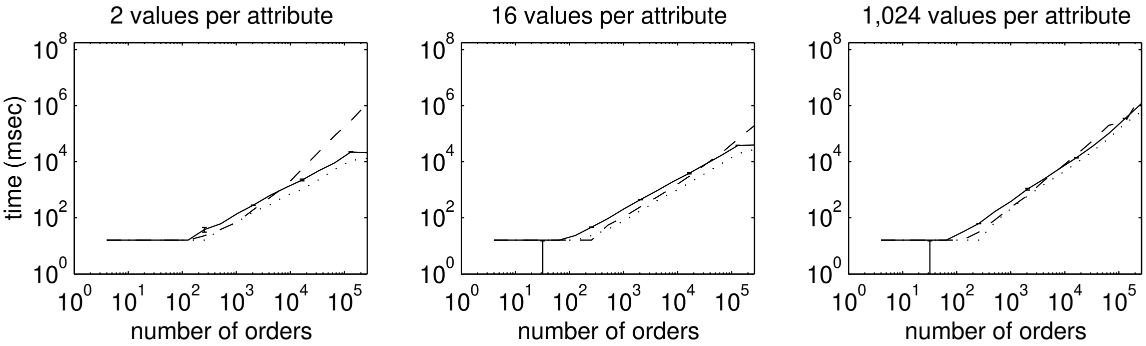


(c) Market with ten attributes.

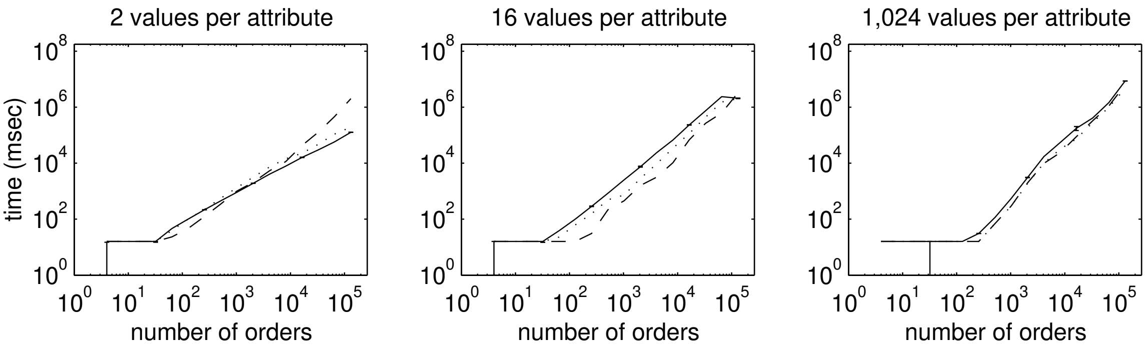
Figure A.1: Time of processing the new orders for matching density of 0.001. We consider markets with 2 values per attribute (left), 16 values per attribute (middle), and 1,024 values per attribute (right). We show the dependency of the processing time on the number of orders for the best-first search (solid lines), depth-first search that identifies all matches (dashed lines), and depth-first search with a limit on the number of matches (dotted lines). Both horizontal and vertical scales are logarithmic, and the vertical bars mark the minimal and maximal values of the time measurements.



(a) Market with one attribute.

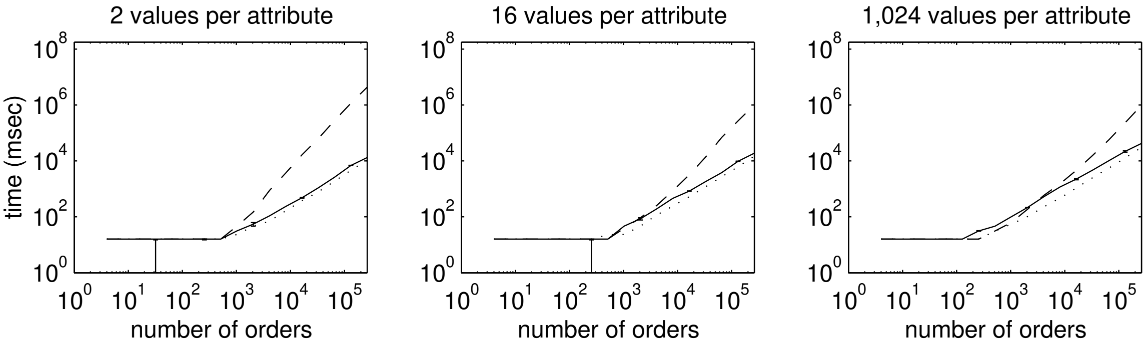


(b) Market with three attributes.

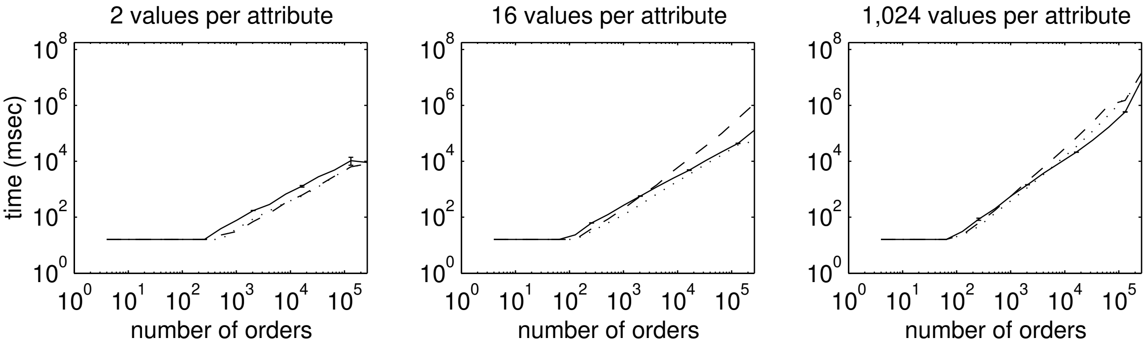


(c) Market with ten attributes.

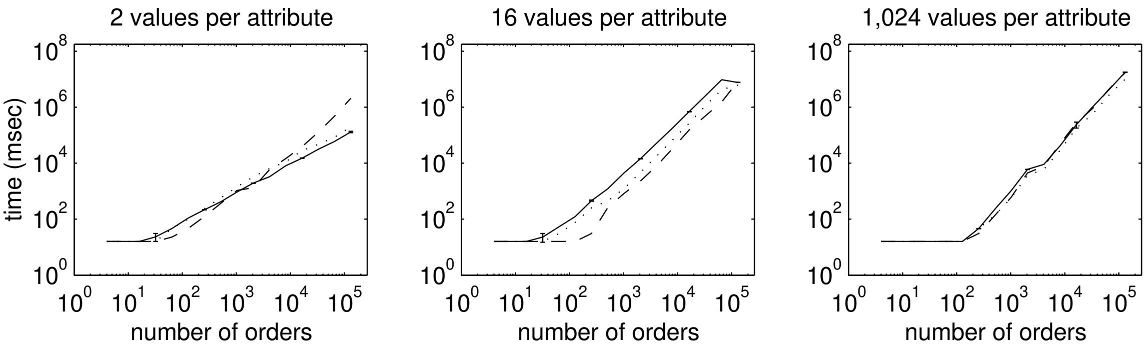
Figure A.2: Time of processing the new orders for matching density of 0.01. The legend is the same as in Figure A.1.



(a) Market with one attribute.

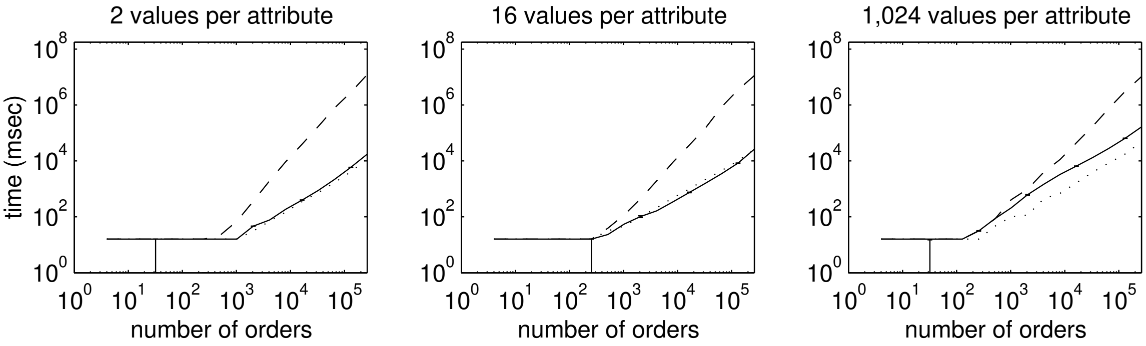


(b) Market with three attributes.

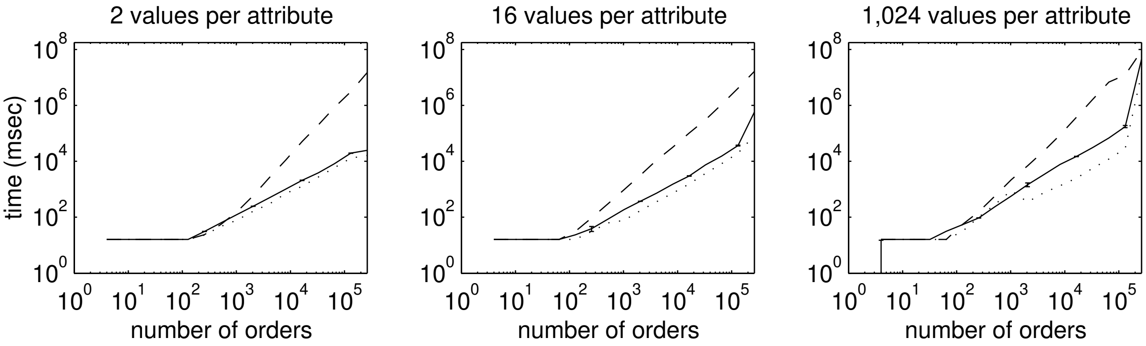


(c) Market with ten attributes.

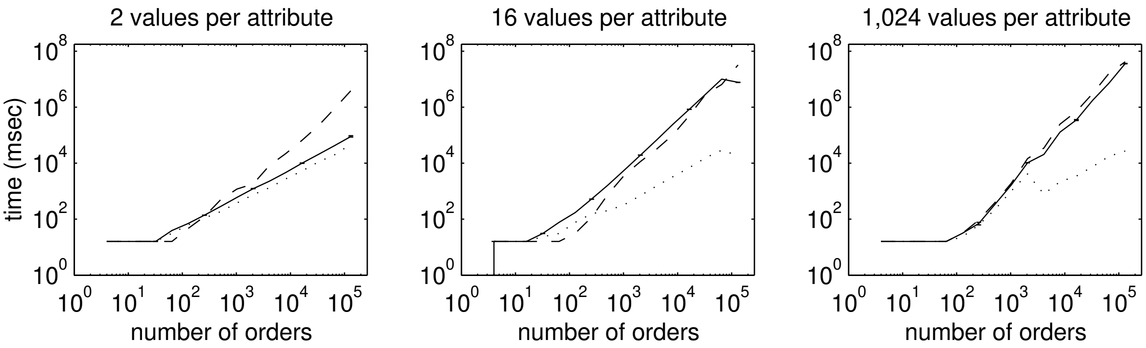
Figure A.3: Time of processing the new orders for matching density of 0.1. The legend is the same as in Figure A.1.



(a) Market with one attribute.



(b) Market with three attributes.

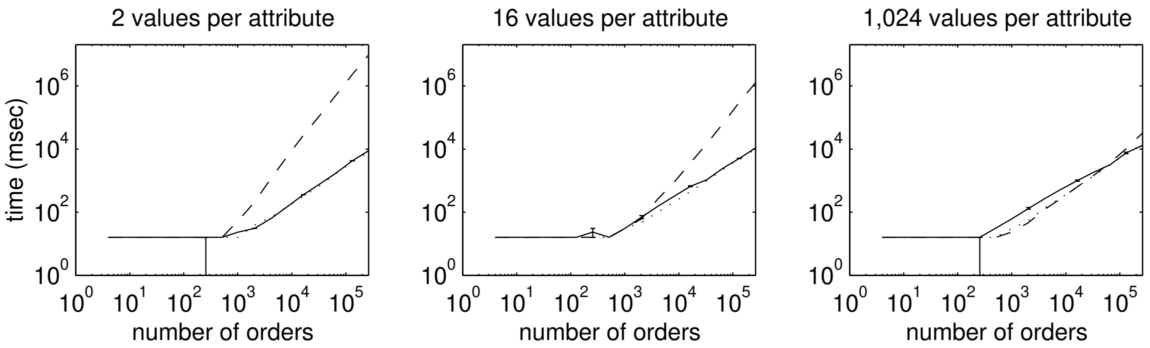


(c) Market with ten attributes.

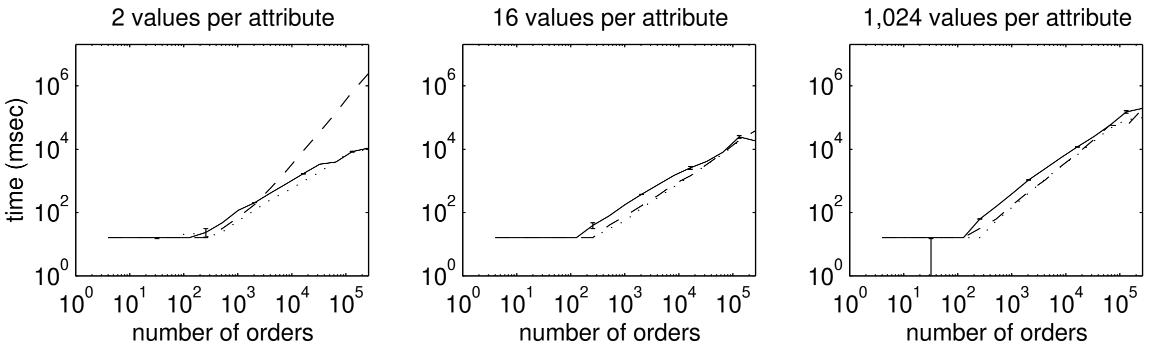
Figure A.4: Time of processing the new orders for matching density of 1. The legend is the same as in Figure A.1.

A.2 Matching time

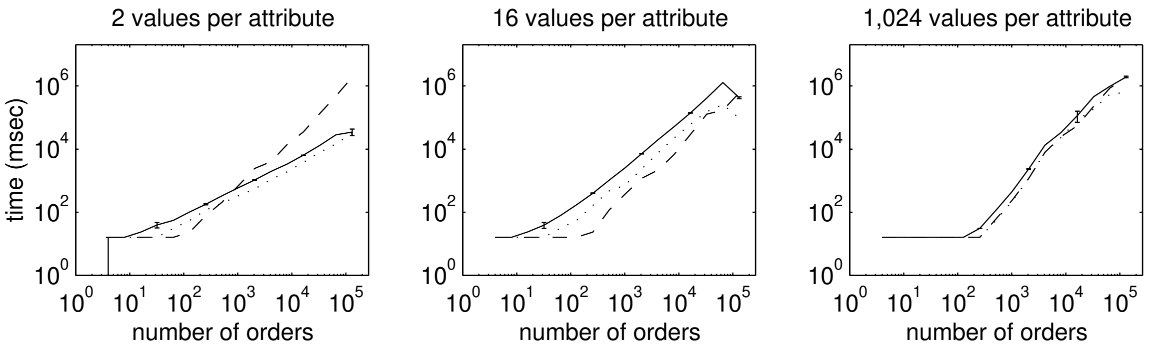
We give the mean time of matching all pending buy orders in the second half of the system's top-level loop (Figure 6.3), along with the minimal and maximal values of the time measurements.



(a) Market with one attribute.

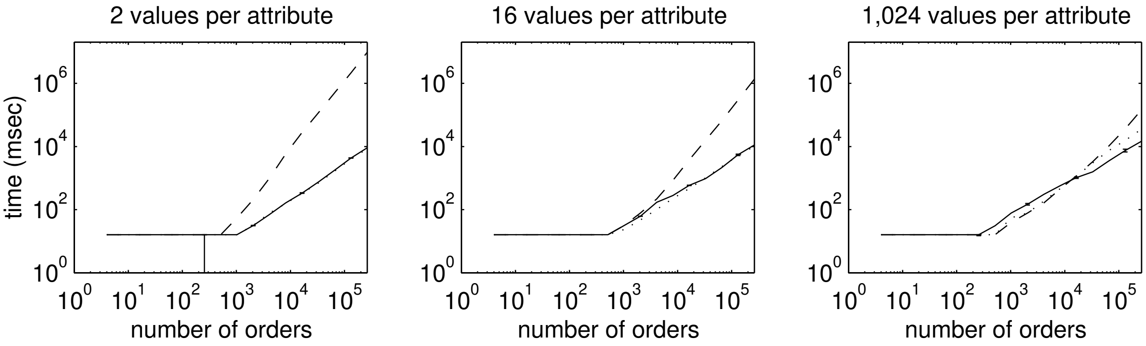


(b) Market with three attributes.

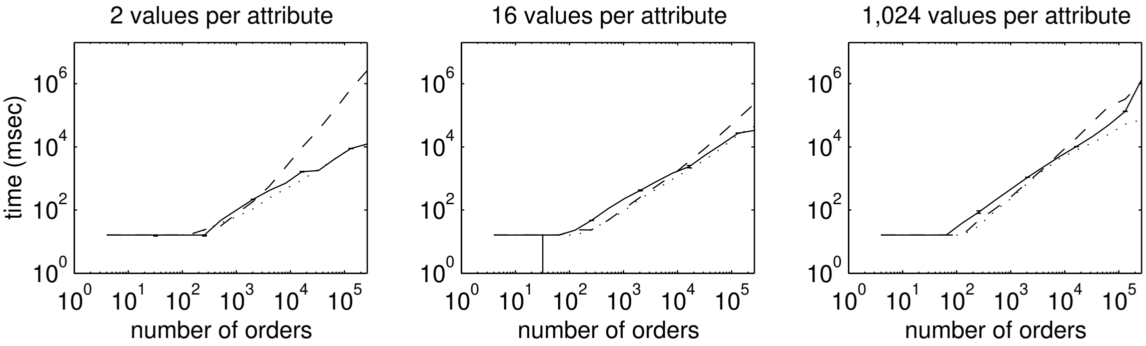


(c) Market with ten attributes.

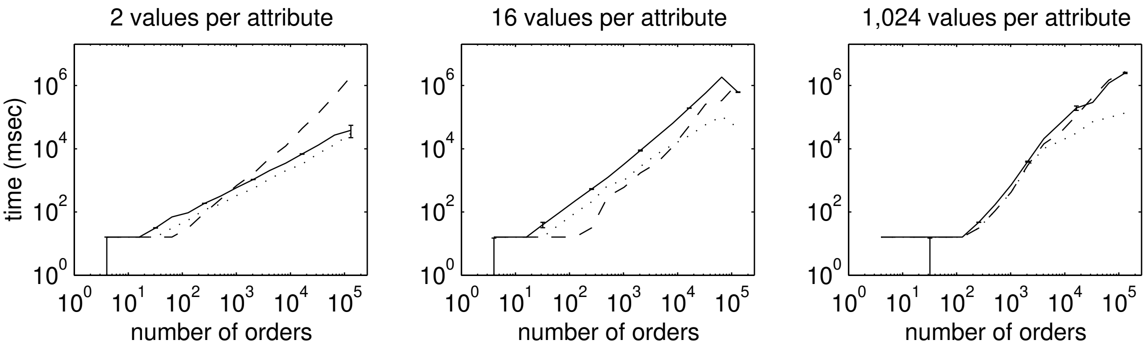
Figure A.5: Time of matching the pending orders for matching density of 0.001. We consider markets with 2 values per attribute (left), 16 values per attribute (middle), and 1024 values per attribute (right). We show the dependency of the matching time on the number of orders for the best-first search (solid lines), depth-first search that identifies all matches (dashed lines), and depth-first search with a limit on the number of matches (dotted lines). Both horizontal and vertical scales are logarithmic, and the vertical bars mark the minimal and maximal values of the time measurements.



(a) Market with one attribute.

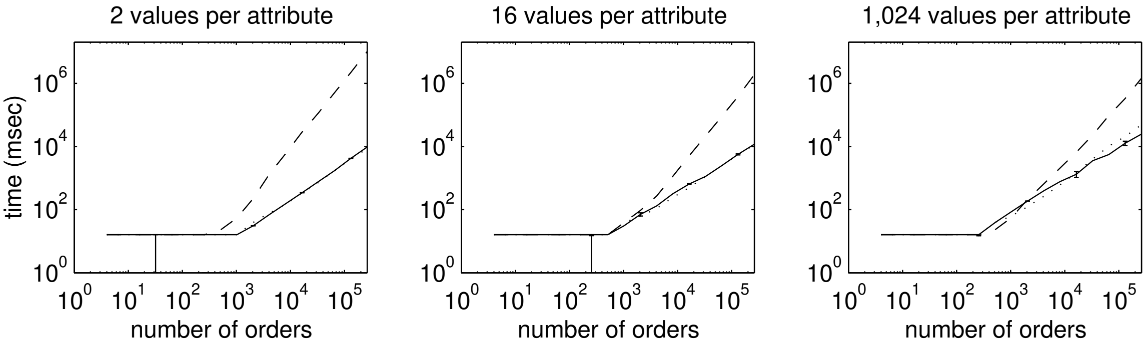


(b) Market with three attributes.

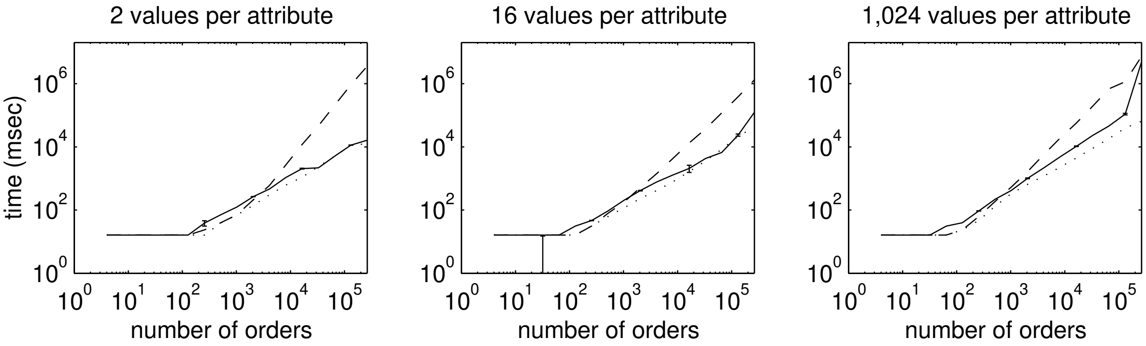


(c) Market with ten attributes.

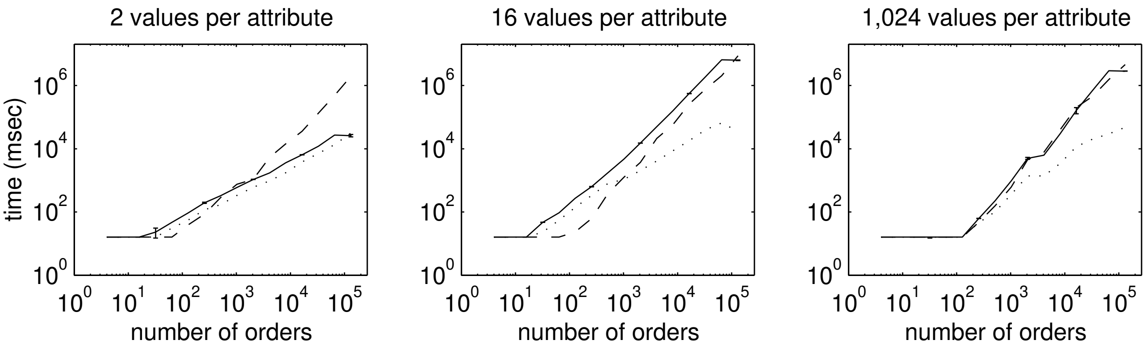
Figure A.6: Time of matching the pending orders for matching density of 0.01. The legend is the same as in Figure A.5.



(a) Market with one attribute.

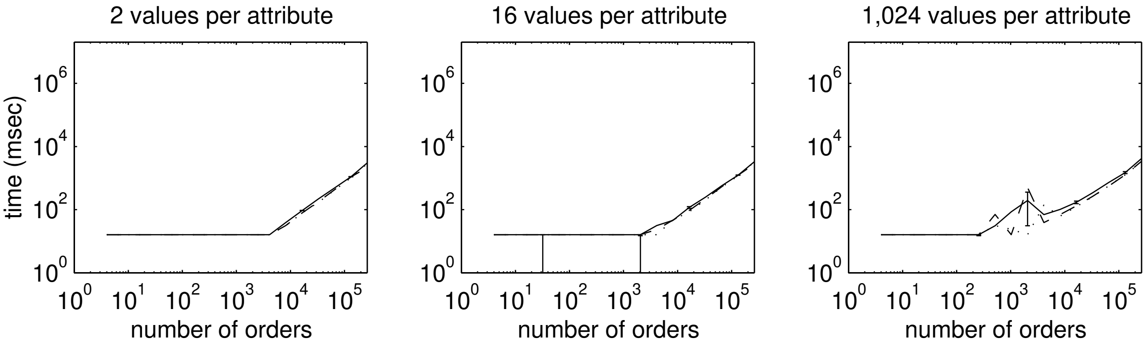


(b) Market with three attributes.

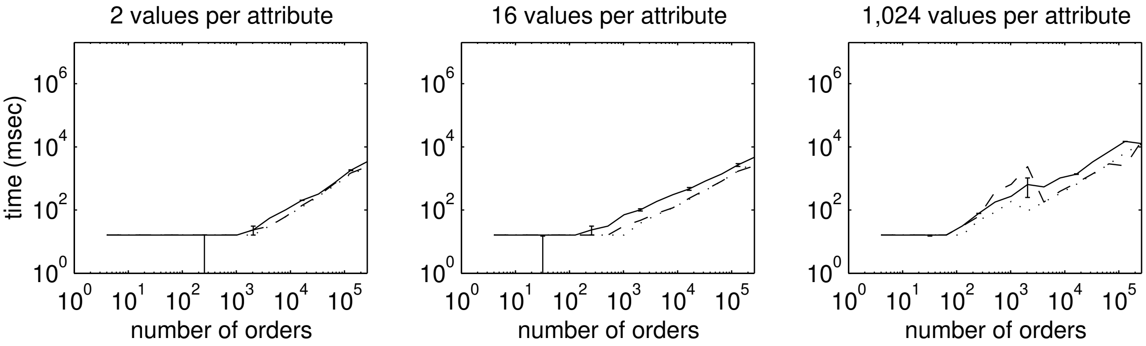


(c) Market with ten attributes.

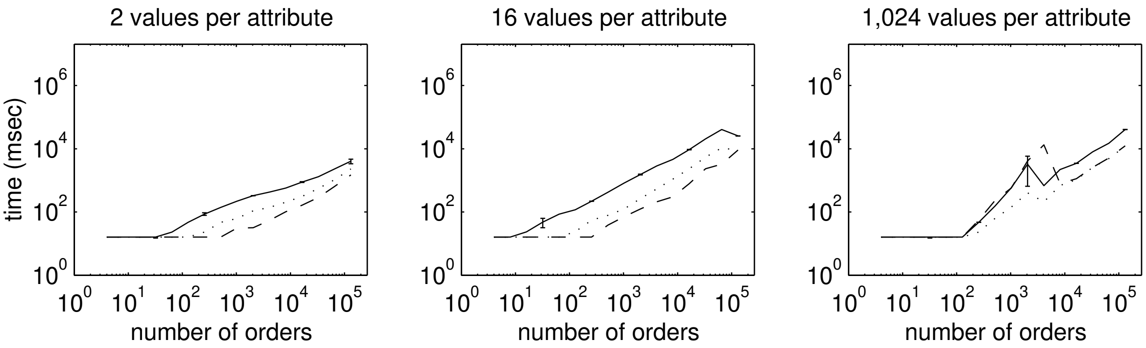
Figure A.7: Time of matching the pending orders for matching density of 0.1. The legend is the same as in Figure A.5.



(a) Market with one attribute.



(b) Market with three attributes.

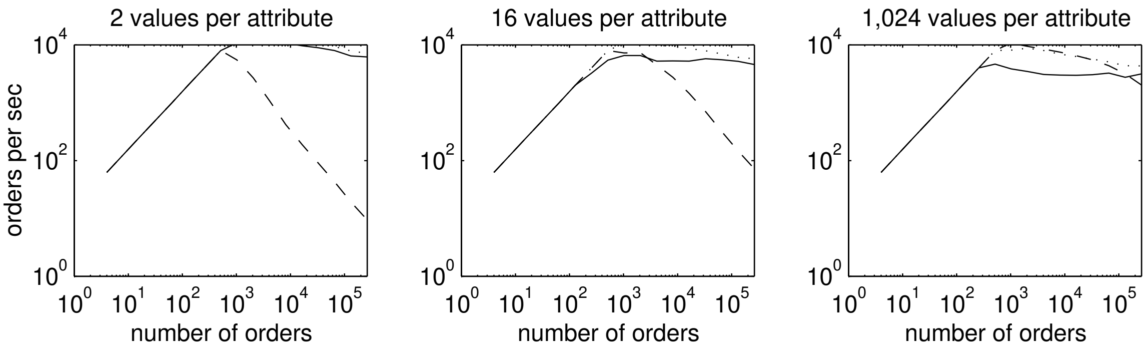


(c) Market with ten attributes.

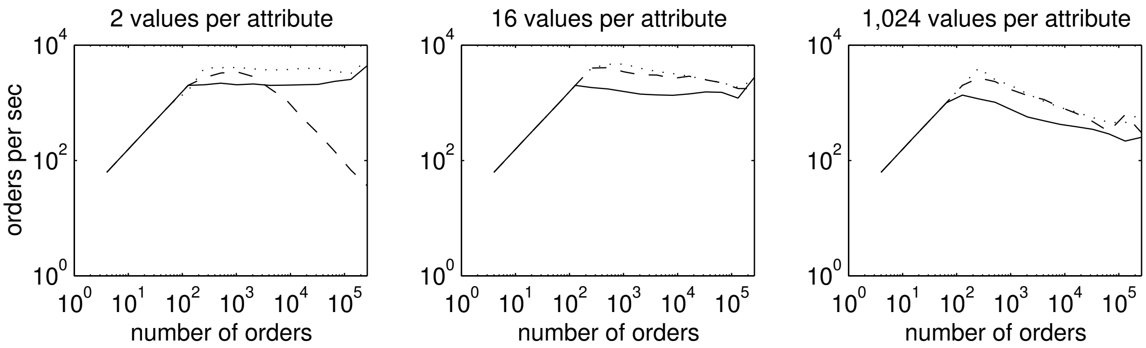
Figure A.8: Time of matching the pending orders for matching density of 1. The legend is the same as in Figure A.5.

A.3 Maximal throughput

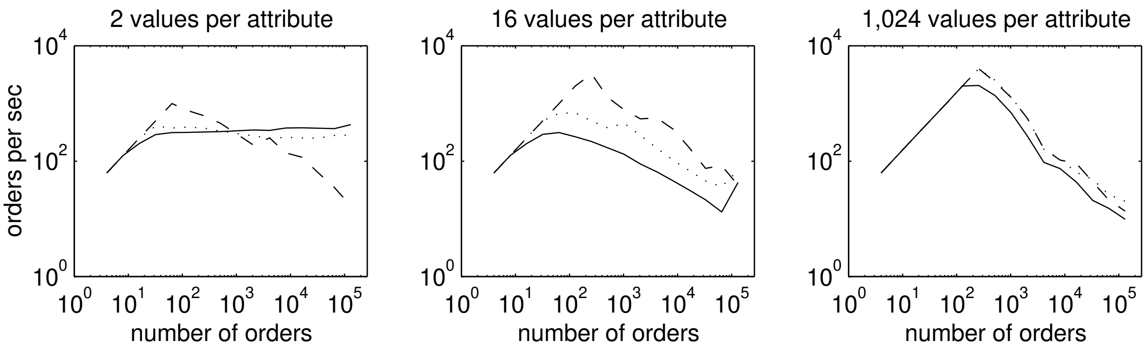
We show the limit on the number of new orders per second. If the matcher gets more orders, it rejects some of them.



(a) Market with one attribute.

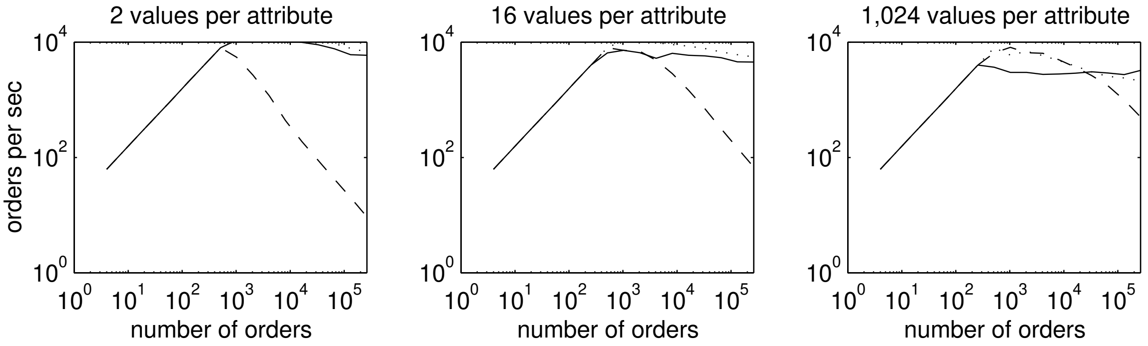


(b) Market with three attributes.

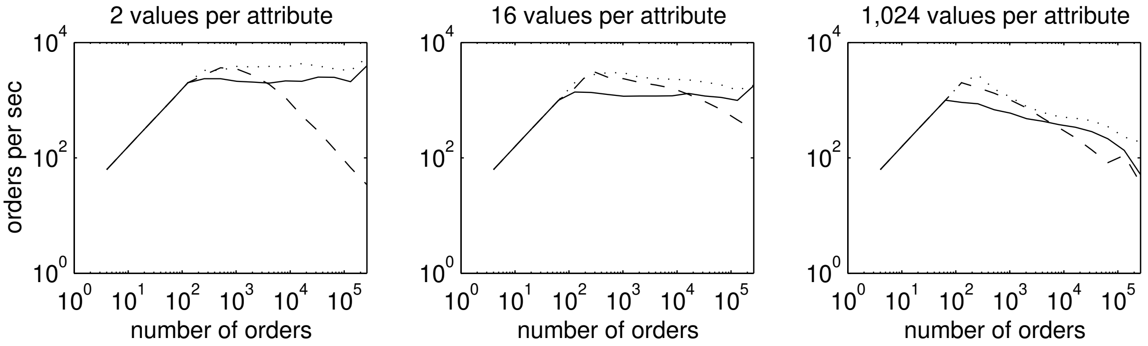


(c) Market with ten attributes.

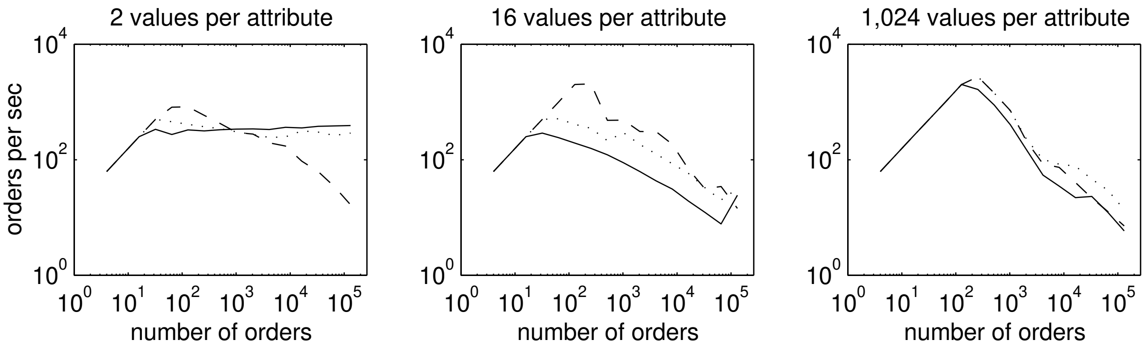
Figure A.9: Maximal number of orders per second for matching density of 0.001. We consider markets with 2 values per attribute (left), 16 values per attribute (middle), and 1,024 values per attribute (right). We show the dependency of the system's throughput on the number of orders for the best-first search (solid lines), depth-first search that identifies all matches (dashed lines), and depth-first search with a limit on the number of matches (dotted lines). Both horizontal and vertical scales are logarithmic.



(a) Market with one attribute.

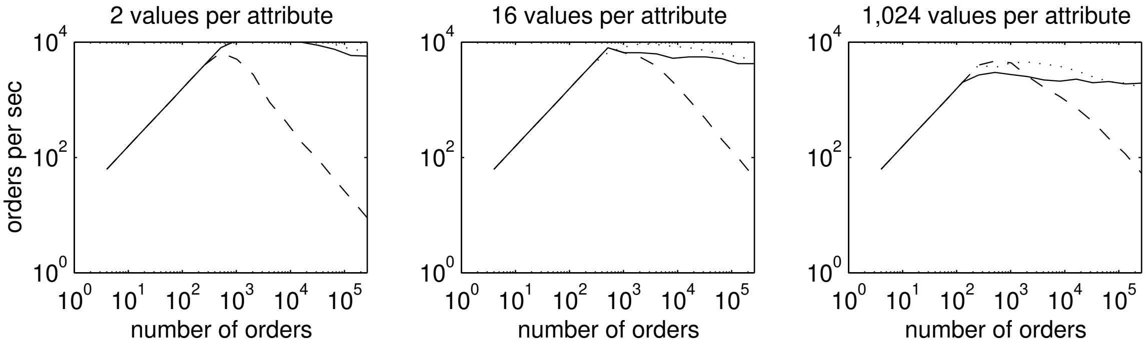


(b) Market with three attributes.

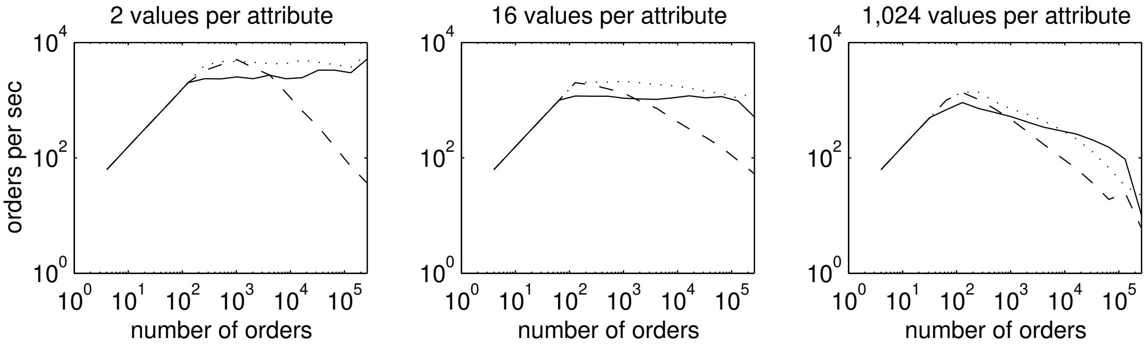


(c) Market with ten attributes.

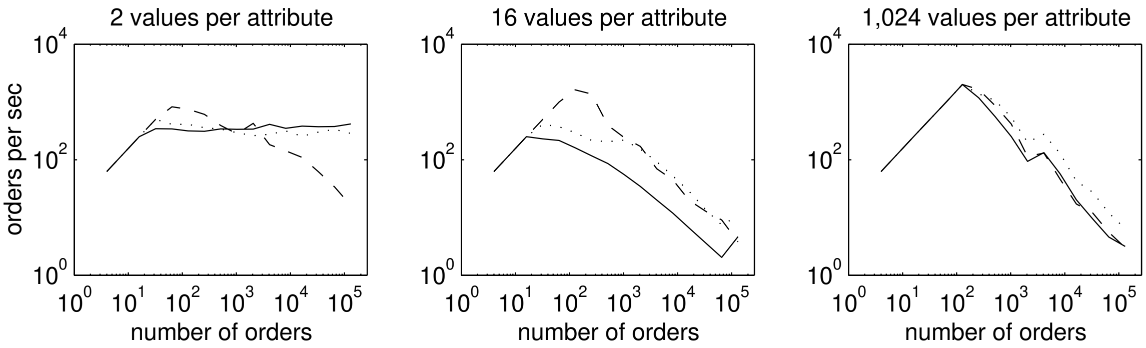
Figure A.10: Maximal number of orders per second for matching density of 0.01. The legend is the same as in Figure A.9.



(a) Market with one attribute.

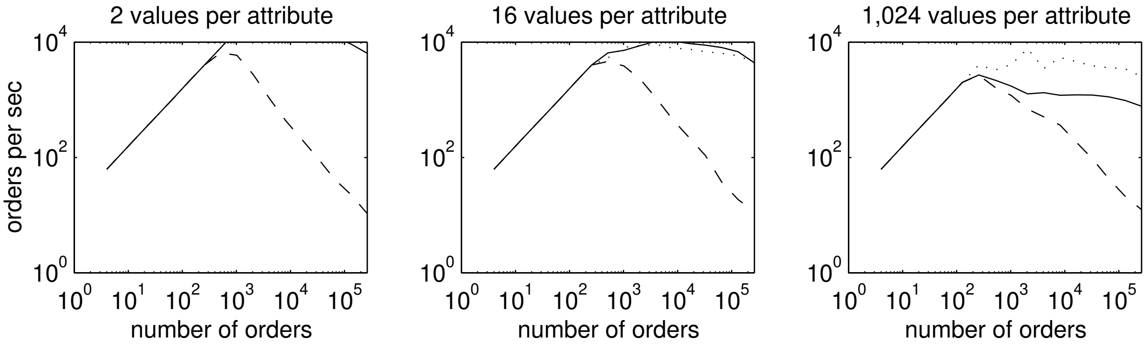


(b) Market with three attributes.

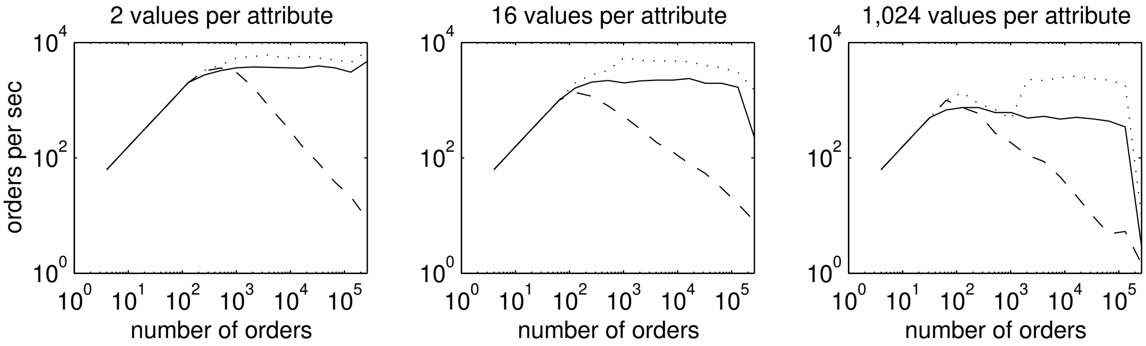


(c) Market with ten attributes.

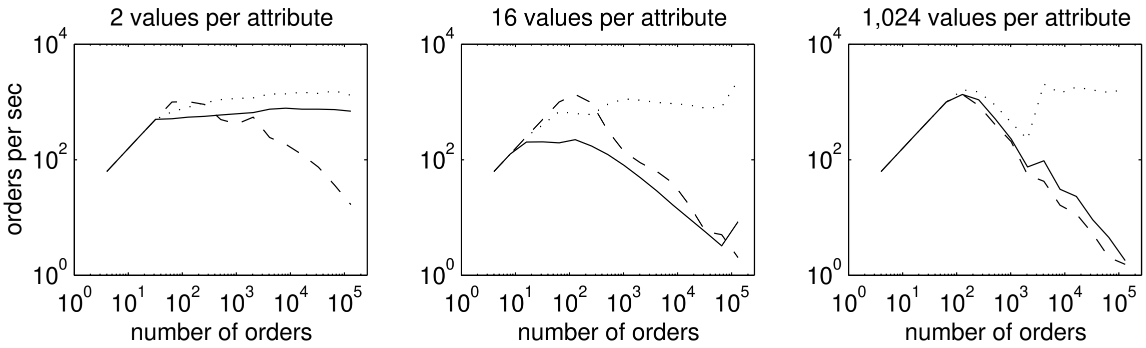
Figure A.11: Maximal number of orders per second for matching density of 0.1. The legend is the same as in Figure A.5.



(a) Market with one attribute.



(b) Market with three attributes.



(c) Market with ten attributes.

Figure A.12: Maximal number of orders per second for matching density of 1. The legend is the same as in Figure A.5.