Exchanges for Complex Commodities:

Toward a General-Purpose System for On-Line Trading

by

John Hershberger

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

Major Professor: Eugene Fink, Ph.D
Member: Rafael Perez, Ph.D
Member: Dmitry B. Goldgof, Ph.D

Date of Approval:
August 20, 2003

Keywords: E-commerce, electronic trading, exchange markets,
multi-attribute commodities.

Acknowledgements

I gratefully acknowledge the help of Eugene Fink, who has supervised my thesis work and guided me through all steps of research and writing. I am also grateful to Rafael Perez for his continuous support of my studies, and Dmitry Goldgof for his valuable comments and suggestions.

I am forever grateful to my girlfriend, Michelle Elliott, for the inexhaustible comfort and reassurance she has given me over all the years of my undergraduate and graduate studies. I also wish to thank my parents, Richard Hershberger and Diane Lynch, for the constant support and freedom they have given me.

Table of Contents

List of Figures

Exchanges for Complex Commodities:

Toward a General-Purpose System for On-Line Trading

John Hershberger

ABSTRACT

The modern economy includes a variety of markets, and the Internet has opened opportunities for efficient on-line trading. Researchers have developed algorithms for various auctions, which have become a popular means for on-line sales. They have also designed algorithms for exchange-based markets, similar to the traditional stock exchange, which support fast-paced trading of rigidly standardized securities. In contrast, there has been little work on exchanges for complex nonstandard commodities, such as used cars or collectible stamps.

We propose a formal model for trading of complex goods, and present an automated exchange for a limited version of this model. The exchange allows the traders to describe commodities by multiple attributes; for example, a car buyer may specify a model, options, color, and other desirable properties. Furthermore, a trader may enter constraints on the acceptable items rather than a specific item; for example, a buyer may look for any car that satisfies certain constraints, rather than for one particular vehicle.

We present an extensive empirical evaluation of the implemented exchange, using artificial data, and then give results for two real-world markets, used cars and commercial paper. The experiments show that the system supports markets with up to 260,000 orders, and generates one hundred to one thousand trades per second.

Chapter 1

Introduction

1.1 Motivation

Economists define *market* as "an arrangement which permits numerous buyers and sellers of related commodities to carry on extensive business transactions on a regular, organized basis" [Trenton, 1964]. The modern economy includes a wide variety of markets, from cars to software to office space.

The supply chain between a manufacturer and customer may include several middlemen. For instance, customers usually buy cars through dealerships, which in turn acquire cars from manufacturers; the sale of used cars may also involve dealers, who serve as middlemen in the secondary market. This problem exists not only in broker-to-consumer transactions, but also in broker-to-broker markets, where many transactions are slow and require either salespeople on both sides or intermediary brokers. The most certain sign of inefficiency in the modern economy is the army of salespeople and brokers, who make their living by acting as middlemen.

In addition, most items are bought and sold not only from the manufacturer, but also on a secondary market, that is, they may be bought and re-sold multiple times. For example, phone companies often re-sell unused phone minutes, and airlines re-sell unused seats. Liquidity is essential, since the commodity may be lost if it is not sold by some predetermined time. For example, unsold seats in an airplane become useless once the airplane is off the ground. These resales increase the cost of goods, since they include commissions for the middlemen. To improve the efficiency of the economy, a more efficient secondary market is necessary. This market would not require highly paid brokers and middlemen, a significant investment of time and effort, or a considerable risk of trading at a sub-optimal price. The recent growth of the Internet has opened opportunities for reducing the number of middlemen [Klein, 1997; Turban, 1997; Wrigley, 1997], and many companies have experimented with direct sales over the web. Middlemen are also using the Internet to increase the volume of their sales and reduce expenses. Furthermore, many companies specialize in the development of electronic marketplaces, which include bulletin boards, auctions, and exchanges.

Electronic bulletin boards are similar to traditional newspaper classifieds. These boards vary from newsgroup postings to on-line sale catalogs, and they help buyers and sellers find each other; however, they often require a user to invest significant effort into searching among multiple ads. For this reason, many buyers prefer on-line auctions, such as eBay (www.ebay.com).

Auctions have their own problems, which include significant computational costs, transaction delays, and asymmetry between buyers and sellers. A traditional auction requires a buyer to bid on a specific item. It helps sellers to obtain the highest price, but limits buyers' flexibility. A *reverse auction* requires a seller to bid on a customer's order; thus, it benefits buyers, and restricts the sellers' flexibility. Furthermore, auctions limit the liquidity, that is, they may cause significant transaction delays. For example, if a seller posts an item on eBay, she can sell it in three or more days, depending on the selected duration of the auction, but not sooner. Thus, auctions are not appropriate for fast sales, which are essential in many markets.

An exchange-based market does not have these problems: it ensures symmetry between buyers and sellers, and supports fast-paced trading. Examples of liquid markets include the traditional stock and commodity exchanges, such as the New York Stock Exchange and Chicago Mercantile Exchange, as well as currency and bond exchanges. For instance, a trader can buy or sell any public stock in seconds, at the best available price. Although stocks have long served as an example of an efficient market, trading in other industries has not reached this efficiency.

The main limitation of traditional exchanges is rigid standardization of tradable items. For instance, the New York Stock Exchange allows trading of about 3,100 securities, and the buyer or seller has to indicate a specific item, such as IBM stock. For most goods and services, however, the description is much more complex. For instance, a car buyer may need to specify a make, model, options, color, and other desirable features. Furthermore, she usually has a certain flexibility and may accept any car that satisfies her constraints, rather than looking for one specific vehicle. For example, she may be willing to get any red Mustang with air conditioning.

Building an exchange for such complex commodities is a major open problem. An effective trading system should satisfy the following requirements:

- Allow complex constraints in specifications of buy and sell orders
- Support fast-paced trading for large markets, with millions of orders
- Include optimization techniques that maximize the traders' satisfaction
- Ensure the "fairness" of the market, according to financial industry standards
- Allow a user to select preferred trades among matches for her order

1.2 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.

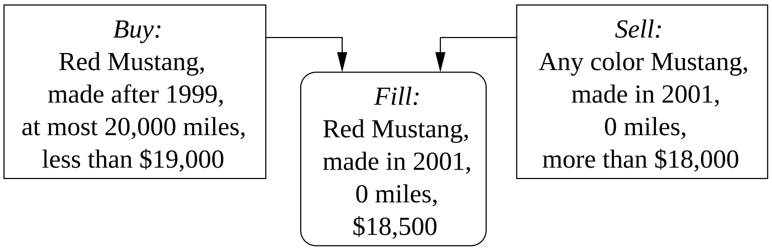


Figure 1.1: Matching orders and the resulting trade. When the system finds a match between two orders, it generates a fill, which is a trade that satisfies both parties.

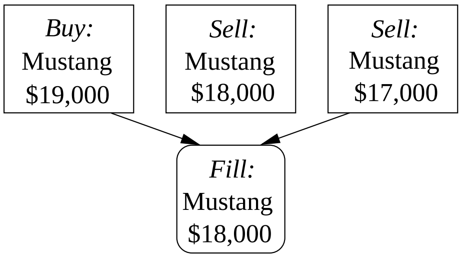


Figure 1.2: Choosing the match with the best price.

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 1.1).

If the system finds several matches for an order, it chooses the match with the best price. For example, the buy order in Figure 1.2 will trade with the cheaper of the two sell orders.

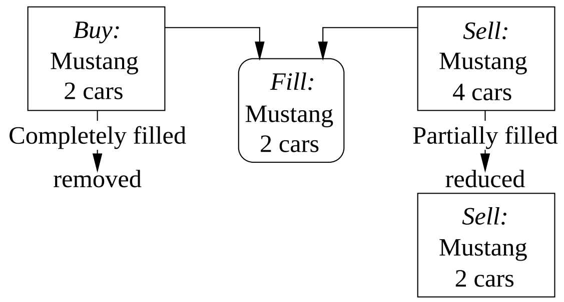


Figure 1.3: Example of order sizes. When the system finds a match, it completely fills the smaller order and reduces the size of the larger order.

The system allows a user to trade several identical items by specifying a size for 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 1.3) 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 example, 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.

1.3 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; Bichler and Werthner, 2000; Ronen, 2001; Ronen and Saberi, 2002], optimal matching in various auctions [Ygge and Akkermans, 1997; Monderer and Tennenholtz, 2000; Kastner *et al.*, 2002], bidding strategies [Tesauro and Das, 2001; He and Leung, 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 efficient systems for *combinatorial auctions,* which allow buying and selling sets of commodities rather than individual items. They have considered not only auctions with completely specified commodities, but also markets that allow the user to negotiate desirable features of merchandise.

1.3.1 Combinatorial auctions

A traditional combinatorial auction allows bidding on a set of fully specified items. For example, a buyer 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, a buyer 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%. Hu and Shi [2002] later refined Rothkopf's analysis, helping to improve optimal matching in combinatorial auctions.

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; Sandholm, 2002; Sandholm and Suri, 2003]. 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 auctions 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. They later analyzed the empirical hardness of optimizing combinatorial auctions, which can lead to more consistent combinatorial auction benchmarks, and improved combinatorial auction algorithms [Leyton-Brown *et al.*, 2002].

Tennenholtz [2002] developed polynomial-time algorithms for finding optimal matches in several types of restricted combinatorial auctions.

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.

Xia *et al.* [2003] investigated multiple techniques of pricing bundles of goods and market clearing, while ensuring the truthfulness of all bidders. They also studied various methods of determining individual item prices, which is useful for evaluating combinatorial auctions. They concluded that popular pricing mechanisms are too restrictive in combinatorial auctions, and they fall short in guaranteeing truthful bids, which lowers the optimality of the auction.

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 nonfinancial 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.

1.3.2 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. Moore *et al.* [1990] gave a decision-theoretic approach to information retrieval, in which the best available record in a file is retrieved based on user prefereces. The primary drawback of their method is that some user preferences are not representable according to their restrictions. Cagno *et al.* [2001] studied the probability of winning in competitive bidding with multi-attribute decision making techniques. Uncertainty stemmed from the evaluation criteria of the seller and the bidding profiles of the competing bidders. They described a technique that accounts for this uncertainty and attempts to maximize the probability of a winning bid, based on current knowledge of the seller and competitors.

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 and Kaukal, 1999; 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; Jones and Koehler, 2002]; 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.

Tewari *et al.* [2003] devised a location-based brokering scheme allowing for geographically nearby bidders to gain preference over those bidding from afar. Their system was useful for markets in which fast delivery is essential, such as restaurants and "impulse shopping" from consumers.

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. Burmeister *et al.* [2002] designed a "package-oriented" approach to bidding in multi-attribute auctions. A package consisted of a combination of multiple attributes, which the buyer assigns a value to as a whole, rather than to each attribute separately. However, in this approach the individual attribute evaluations become transparent to the seller.

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.

1.3.3 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). In addition to the securities exchanges, exchanges for other goods and services have been developed. For instance, GreenOnline (www.greenonline.com) allows traders to exchange goods and services within a variety of environmental markets. These environmental exchanges are specialized auctions in which public, private, and non-profit buyers and sellers trade environmental pollution credits, assets associated with regulatory offsets, and other goods and services. Keever and Alcorn [2000] gives a brief overview of the evolution of GreenOnline and environmental exchanges.

Contreras *et al.* [2001] built a system to simulate the power exchange market of a deregulated electric energy industry. Bidding proposals and interaction between supplier and consumer was simulated. The system could be used to evaluate how the selection of winning bids depends on the auction model used in the exchange. Richter and Sheble [1998] developed a system composed of companies buying and selling power in a regional exchange. Genetic algorithms encoded the bidding strategies used by traders, and the bidding strategies adapted as the traders' behavior changed. The system was useful for evaluating whether or not a bidding strategy will be successful in a practical electricity exchange; however, the system's use in other markets has not been determined.

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. Weinhardt and Gomber [1999] developed a prototype multi-agent system for automating single auctions within an off-exchange bond market. The agents widened the search space of potential traders, accelerated the trading process, and helped reduce the need for intermediary brokers.

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.

1.3.4 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], Maes *et al.* [1999], and Huhns and Vidal [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 Kaukal, 1999; Bichler and Segev, 1999]. Bichler later developed an advanced decision analysis engine capable of incorporating many attributes into bid selection [Bichler *et al.*, 2001]. The system was able to make proper decisions with hundreds of bids, and only required the user to initially rank a subset of the bids on desirability.

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. Piccinelli *et al.* [2001] designed a distributed negotiation system allowing service providers to subcontract aspects of its service to other providers, and determine the price to pay each. Service roles were negotiated within a reverse-combinatorial-auction, in which potential subcontractors bid on combinations of service roles of the original service provider.

Dumas *et al.* [2002] presented a formal model of negotiating agent behavior, consisting of numerous modules and a knowledge base. The model helped in the development of agents able to carry out simultaneous negotiations; however, communication between agents was limited to simple messages, and it did not allow transmitting trading rules.

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.

Rahwan *et al.* [2002] created trading agents that would each individually negotiate with other traders, and a coordinating agent to direct any further action. In this system, agents could be introduced or removed dynamically; however, the trading agents had no knowledge of each other's progress. 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.

Cliff [1998] designed agents that utilized genetic algorithms in order to optimize their bidding strategies. The genetic algorithms automated the setting of a learning rate, which was defined as the rate at which the system adjusts its output toward some target output. He later showed that a genetic algorithm could also optimize the particular market mechanism under which the trader agents operate [Cliff, 2002]. Sample market mechanisms include the continuous double auction, Dutch auction, and English auction.

The genetic algorithm was able to evolve traditional market mechanisms into new hybrid mechanisms, which had no real-world representation; thus, they were only suited for artificial traders.

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].

Maamar [2002] investigated the abilities required for a software agent to perform adequately in an auction setting. He found that agent-based e-commerce systems required human interaction at multiple stages of the trading process, concluding that further research is needed before fully automated e-commerce support becomes realistic.

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.

1.3.5 Industrial systems

Several companies have released software products useful in the e-commerce industry. These systems can typically be used to provide e-Sourcing solutions for businesses around the world. *e-Sourcing* refers to the process by which a company determines the optimal distributor(s) from which to buy its supply of a needed commodity, with optimality based on buyer preferences. The e-Sourcing process usually involves order planning, RFQ generation, RFQ evaluation, negotiation, settlement, and order execution. With the creation and evaluation of an RFQ (Request-for-Quote) comes the need for a decision support and auction system to determine which bid the buyer should purchase from. An optimal sourcing mechanism will result in lower total acquisition costs for the buyer.

IBM Research has developed the Multidimensional Analysis Platform, or MAP, to provide decision support for e-Sourcing. MAP consists of tools to elicit buyer preferences for multi-attribute bid evaluation using decision analysis techniques, a bid evaluation engine that determines the optimal set of bids a buyer should accept, and a visualization tool to compare multiple bids across multiple attributes. In addition, the system may be linked to an existing auction platform to carry out complex auctions for practical business trading.

The Emptoris Sourcing Platform from Emptoris and the Profit Optimization Suite from Rapt are e-Sourcing systems that provide commercial bid analysis via integer programming and constraint programming. These products analyze bids from multiple suppliers, and then select an optimal set of bids, based on user preferences. These products are well-suited for simple objectives, such as minimizing the total cost for the buyer. However, these systems do not handle the optimization of multiple attributes well, which is vital in the trading of complex commodities. Moreover, these systems also do not give users a thorough explanation of results, and they do not allow users to readily look through the available bids themselves.

Another approach to bid-analysis may be found in Frictionless Commerce's Enterprise Software and Perfect Commerce's Perfect Application Suite. These systems carry out complex decision making by decomposing complex decisions into smaller, simpler pieces that can later be recombined into the larger aggregate decision. These systems rely on the user to assign relative weights to different attribute values found in a potential bid. These weights may then be added to yield a numeric overall value for a bid. The system then sorts the bids based on the their overall value and the user selects the winning bids. The major drawback with this approach is that it relies heavily on the appropriate setting of weights by the user, which is often hard to guarantee. The user normally has little help in assigning relative weights, and will encounter difficulty when the number of attributes grows into the dozens.

California Software Company, Ltd. released the eBiz Market Server software, which includes engines for both auctions and exchanges. The eBiz Auction Engine supports several popular auction types, such as Dutch, English, and Sealed Bid. The software can be used to provide an auction platform for business trading. The eBiz Exchange Engine provides an exchange architecture that allows for periodic clearing and continuous clearing exchanges. The exchange can return to a trader the best matching orders in the system, and traders can then commence negotiation and order execution. The eBiz Market Server software also offers e-Sourcing solutions via its RFQ Engine. Businesses may investigate potential suppliers based on specified criteria, and the system can determine the optimal set of suppliers for the buyer.

TripleHop Technologies developed the ShopMatcher software, which uses artificial intelligence methods to learn the buying behavior of a potential customer. ShopMatcher learns a customer's shopping patterns with repeated transactions with the consumer. It then adapts itself to meet the demands of the consumer by making product recommendations that are most apt to lead to a transaction. This enhances the trading experience for both the buyer and seller, allowing the consumer to quickly find what she is looking for, while increasing the seller's chances of completing a transaction. For companies with a large number of online customers, this technology could lead to increased revenue.

1.3.6 Contributions

The review of previous work has shown that techniques for trading complex commodities are still limited. Researchers have investigated several auction models, as well as exchanges for standardized securities, but they have not applied the exchange model to complex goods. The main open problems are (1) design of an automated exchange for complex securities, (2) analysis of related trading rules, and (3) development of a rigorous theory of complex exchanges. The work reported here is a step toward addressing these problems.

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. Gong [2002] has developed algorithms for fast identification of most preferable matches, which maximize the satisfaction of market participants.

1.3.7 Chapter summaries

Chapter 1: *Introduction.*

We give the primary motivation of this work and the current limitations found in modern trading systems, such as electronic bulletin boards, auctions, and exchanges. We present an example exchange that allows transactions involving multi-attribute goods between a buyer and seller. We then review previous research in combinatorial auctions, semantics, exchanges, general-purpose systems, and industry, while specifying the related open problems.

Chapter 2: *General exchange model.*

We introduce a general model for trading complex goods, and formally define market concepts including orders, buyers and sellers, market attributes, order execution, and market fairness. We also describe additional components within the exchange system, such as order flexibility, quality heuristics, and price averaging.

Chapter 3: *Order representation.*

We specify an instance of the general exchange model in a used-car market, and thoroughly describe order semantics including item sets, price, quality, and size. We also delineate user interaction and order manipulation, such as cancellation, modification, and confirmation.

Chapter 4: *Indexing structure.*

We explain the high-level architecture of the exchange system, along with the data structures and algorithms used to maintain the matcher engine. We discuss and illustrate the steps involved in matching a buy order to a sell order, and present typical order scenarios that the exchange system may encounter.

Chapter 5: *Search for matches.*

We give algorithms for two search strategies of order matching, describe the constituent data necessary to carry out the searches, and examine possible variations of these strategies. In addition, we discuss the advantages and drawbacks of each search method, and the inherent trade-offs between them.

Chapter 6: *Performance.*

We describe experiments with both artificial and real market data, and enumerate the control variables and measured variables. We summarize the performance of the exchange system in each data set, while comparing the performance of the different search methods with one another.

Chapter 7: *Concluding remarks.*

We reexamine the motivation of this work and its contribution to pertinent research, and discuss future challenges.

Chapter 2

General Exchange Model

We describe a general model of trading complex commodities, using an example car market. We formalize the concept of buy and sell orders, consider a trading environment that allows hard and soft constraints in the order specification, and discuss methods for representing combinations of purchases and sales. In Chapters 3 and 4, we present an automated exchange supporting a limited version of this general model.

2.1 Orders

We begin by defining buy and sell orders, which include descriptions of commodities, price and size specifications, and traders preferences among acceptable transactions. We then state conditions of a match between a buy order and sell order.

2.1.1 Buyers and sellers

When a buyer looks for a certain item, she usually has some flexibility; that is, she is willing to buy any of several acceptable items. For example, suppose that a buyer is looking for a sports car; then, she may be willing to buy one of several models, such as a Corvette, Camaro, or Viper. For each of these models, the buyer has to determine the maximal acceptable price. In addition, a buyer usually has preferences among acceptable items; for instance, the buyer may prefer Corvettes to other models, and she may prefer black cars to red ones. The preferences may depend on the price, features, or other factors, such as service quality or delivery date.

Similarly, when a dealer sells a vehicle she has to decide on a minimal acceptable price. For instance, a seller may be selling a Camaro for no less than \$15,000 and a Corvette for no less than \$20,000. If the seller offers multiple items, she may prefer some sales to others. For example, a seller may prefer to sell the Corvette for \$20,000, rather than the Camaro for \$15,000. If the buyer came to the seller to purchase a sports car, then the seller would try to sell the Corvette before offering the Camaro.

If a buyer's constraints match a seller's constraints, then they may *trade;* that is, the buyer may purchase an item from the seller. If a buyer finds several acceptable items, possibly provided by different sellers, she will buy the best available item, where the notion of "best" depends on her subjective preferences. Likewise, a seller may be able to choose the most attractive deal among several offers.

We use the term *buy order* to refer to a buyer's set of constraints, particularly requirements and preferences. For example, a buyer's wish to purchase a sports car can be expressed as an *order* for a sports car, and her price limits and preferences will be a part of this order. When a buyer announces her desire to trade, we say that she has *placed an order.*

Similarly, a *sell order* is a seller's set of constraints, defining the offered merchandise. For example, a seller may place an order to sell a Camaro or Corvette, and her order may also include price limits and preferences. If a buy and sell order match, they may result in a trade between the corresponding parties.

2.1.2 Definition of an order

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.* This set may be very large or even infinite; in the car market, it includes all vehicles that have ever been made, as well as the cars that can be made in the future. The choice of a market set limits the objects that may be traded, but it does not guarantee that all of the objects will be trade. For instance, if we restrict M to cars, the market will not allow trading of bicycles or golf carts. On the other hand, if the market includes an item such as a Star Wars pod racer, it may never be traded.

When a customer makes a purchase or sale, she needs to specify a set of acceptable items, denoted I, which stands for *item set;* it must be a subset of M, that is, $I \subseteq M$. For example, if a buyer shops for a brand-new sports car, then her set I includes all new sports vehicles.

In addition, a customer should specify a limit on the acceptable price, which may depend on specific items in I. For instance, a buyer may be willing to pay \$17,000 for a red Mustang, but only \$16,000 for a black Mustang, and even less for a Camaro. Formally, a price limit is a real-valued function defined on the set I, whose values are nonnegative; for each item $i \in I$, it defines a certain limit, $Price(i)$. If a customer is buying an item, then $Price(i)$ is the maximal acceptable price. For a seller, on the other hand, it is the minimal acceptable price. To summarize, a buy or sell order must include two elements (see Figure 2.1a):

- a set of items, $I \subseteq M$, and
- a price function, $Price\colon I \to \mathrm{R}$,
  where R is the set of nonnegative real-valued prices.

The prices in consumer markets are usually in dollars or other currencies; however, traders in some specialized markets may use different price measures. For example, mortgage brokers often view the interest rate as the price of a mortgage. The properties of such price measures may differ from those of dollar prices. In particular, the price may not be additive; for instance, if a customer takes a 8% loan and a 6% loan, the overall interest is not 14%.

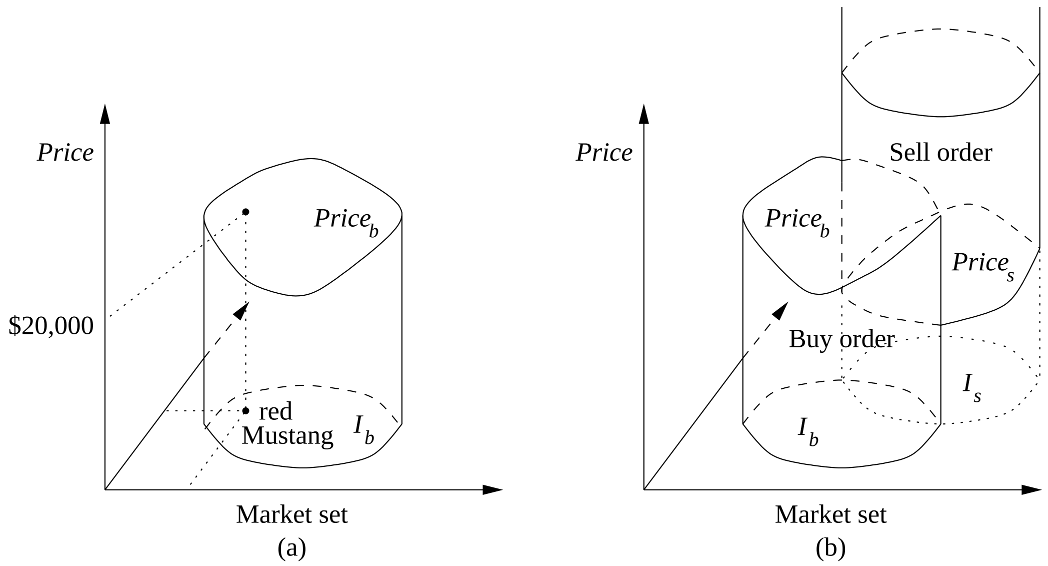


Figure 2.1: An example of a buy order (a) and a match (b). The horizontal plane represents the market set M, and the vertical axis is price R. The buyer is interested in a certain set I_b of cars, with different price limits; in particular, she would buy a red Mustang for $20,000. Her order matches the sell order shown on the right.

We allow such price measures and do not require the use of dollar prices. The only requirement is that a price increase always benefits a seller, and a price decrease always benefits a buyer. In other words, the buyer is interested in finding the lowest available price for a given item, whereas the seller tries to get the maximum possible price. For instance, bank customers look for low-interest loans, whereas bankers try to get high interests.

We say that a buy order *matches* a sell order if the buyer's constraints are consistent with the seller's constraints, thus allowing a mutually acceptable trade (Figure 2.1b). For example, if a buyer is willing to pay $20,000 for a red Corvette, and a seller is ready to sell a red Corvette for $19,000, then their orders match.

Formally, let $(I_b, Price_b)$ be a buy order and $(I_s, Price_s)$ be a sell order. These orders match if some item i satisfies both buyer and seller, at a mutually acceptable price:

$$\text{there exists } i \in I_b \cap I_s \text{ such that } Price_b(i) \geq Price_s(i).$$

2.1.3 Quality function

Buyers and sellers may have preferences among acceptable trades, which depend on a specific item i and its price p. For instance, a buyer may prefer a red Mustang for \$20,000 to a black Corvette for \$22,000.

We define these preferences as a real-valued function $Qual(i, p)$, which assigns a numeric quality to each pair. The larger values correspond to "better" items; that is, if $Qual(i_1, p_1) > Qual(i_2, p_2)$, then a customer would rather pay p_1 for i_1 than p_2 for i_2. For example, a buyer's quality function would satisfy the following inequality:

$$Qual(\texttt{red-Mustang}, \$25{,}000) > Qual(\texttt{black-Chevrolet}, \$20{,}000).$$

Each customer may use her own quality function; furthermore, she may specify different functions for different orders. Note that we define quality as a *totally ordered* function, which is a simplification. In real life, customers sometimes reason in terms of partially ordered functions. For instance, a buyer may believe that a \$25,000 Mustang is better than a \$20,000 Corvette, but she may be undecided between a \$25,000 Mustang and an \$18,000 Corvette.

Also note that buyers prefer lower prices, whereas sellers try to get as much money as possible, which means that all 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, $Qual_b(i, p_1) \geq Qual_b(i, p_2)$.

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

We do not require a user to specify a quality function for each order; by default, quality is defined through price. This default quality is a function of a transaction price and its difference from the users price limit. For example, buying a Toyota Echo for \$11,000 is better than buying it for \$12,000; as another example, if a user has specified a \$12,000 price limit for an Echo and a \$19,000 limit for a Mustang, then buying a Mustang for \$11,000 is better than buying an Echo for \$11,000.

To formalize this rule, we denote the users price function by *Price*, and the price of an actual purchase or sale of an item i by p. The default quality function must satisfy the following conditions for every item i and price p:

- For buy orders: If $Price_1(i) \leq Price_2(i)$, then $Qual_1(i, p) \leq Qual_2(i, p)$.

- For sell orders: If $Price_1(i) \leq Price_2(i)$, then $Qual_1(i, p) \geq Qual_2(i, p)$.

Naturally, the larger the gap between the price limit and actual price, the better the deal; that is, the more the user saves, the more she likes the transaction. Note that if the price limit is a constant then the quality is simply based on prices, that is, a cheaper match is better for a buyer, and a more expensive match is preferable for a seller.

We have considered two default functions, and a user can choose either of them. The first function is the difference between the price limit and actual price:

- For buy orders: $Qual_b(i, p) = Price(i) - p$.

- For sell orders: $Qual_s(i, p) = p - Price(i)$.

This default is typical for financial and wholesale markets; intuitively, the quality of a transaction depends on a users savings. For example, suppose that a car dealer wants to purchase either ten Mustangs for \$19,000 each or ten Echoes for \$12,000 each. Suppose further that she finds Mustangs for \$17,500 and Echoes for \$11,000. If she buys Mustangs, she saves $(\$19,000 - \$17,500) \cdot 10 = \$15,000$. On the other hand, if she acquires Echoes, her savings are only $(\$12,000 - \$11,000) \cdot 10 = \$10,000$. Thus, the first transaction is more attractive.

The other default function is the ratio of the price difference to the price limit:

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

This default is traditional for consumer markets; it shows a users percentage savings. For instance, if a customer is willing to pay \$19,000 for a Mustang, and she gets an opportunity to buy it for \$17,500, then the transaction quality is $\frac{\$19{,}000 - \$17{,}500}{\$19{,}000} = 0.08$. If she is also willing to pay \$12,000 for an Echo and finds that it is available for \$11,000, the quality of buying it is $\frac{\$12{,}000 - \$11{,}000}{\$12{,}000} = 0.09$, which is preferable to the Mustang.

2.1.4 Order sizes

If a user wants to buy or sell several identical items, she may include their number in the order specification; for example, a buyer can place an order to buy two sports cars, and a seller can announce a sale of one thousand Corvettes. We assume that the order size is a natural number, that is, the market participants buy and sell whole items. This assumption is somewhat restrictive, since it enforces discretization of continuous commodities, such as copper or orange juice.

The user may specify not only the overall order size, but also the minimal acceptable size. For instance, suppose that a wholesale agent for Chevrolet needs to sell one thousand cars. Furthermore, she has no time for individual sales, and works with dealerships that are buying at least ten cars at once. She may then specify that the overall size of her sell order is one thousand, and the minimal acceptable size is ten. If the minimal size equals the overall size, we say that the order is *all-or-none*. For example, the agent may offer ten cars and specify that her minimal size is also ten; then, she will sell either nothing or ten cars at once.

In addition, the user can indicate that a transaction size must be divisible by a certain number, called a *size step*. For example, stock traders often buy and sell stocks in blocks of hundred. As another example, a wholesale agent may specify that she is selling cars in blocks of twenty; in this case, she would be willing to sell twenty or forty cars, but not thirty.

To summarize, an order may include six elements:

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

The item set, price limit, and size specification are hard constraints that determine whether a buy order matches a sell order, whereas the quality function serves as both hard and soft constraints. Rejection of a negative quality is a hard constraint, whereas choice of large values among positive-quality transactions is a soft constraint.

To define the matching conditions, 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.

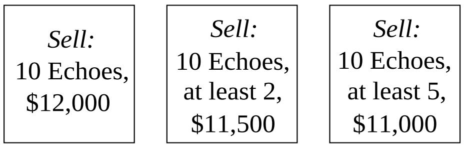


Figure 2.2: Example of a bulk discount. If a dealer is offering a lower price for bulk purchases, she has to place several orders with different prices and minimal sizes.

- 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 $Step_s$

The price and quality functions in this model do not depend on a transaction size, which is a simplification, because sellers sometimes offer discounts for bulk orders. For example, a car dealer may give a discount to a customer who purchases two cars at once, and an even larger discount to a buyer of five cars. In such cases, a seller can place several orders with different price limits and minimal sizes, as illustrated in Figure 2.2. If a seller wants to complete only one of these orders, she can use the disjunctive-order mechanism described in Section 2.3.1.

2.1.5 Market attributes

The set M of all possible items may be very large, which means that we cannot explicitly represent all items. For example, we probably cannot make a catalog of all feasible cars, since it would include a separate entry for each possible combination of models, colors, features, and other attributes that describe a specific vehicle. To avoid this problem, we define a set M by a list of attributes and possible values of each attribute. As a simplified example, we may define a used car by four attributes: *Model, Color, Year,* and *Mileage.* Then, a user describes a specific car by substituting values for these attributes; for example, a seller may offer a red Mustang, made in 1998, with 30,000 miles.

Formally, every attribute is a set of values; for instance, the *Model* set may include all car models, *Color* may include all visible wavelengths, *Year* may include the integer values from 1896 to 2001, and *Mileage* may include real values from 0 to 500,000. The market set M is a *Cartesian product* of these attribute sets; in this example, $M = \mathit{Model} \times \mathit{Color} \times \mathit{Year} \times \mathit{Mileage}$. If the market includes n attributes, then each item is an n-tuple; in the car example, it is a quadruple that specifies the model, color, year, and mileage.

The Cartesian-product representation is a simplification, based on the assumption that all items in the market have the same attributes. Some markets do not satisfy this assumption; for example, if we trade chariots and Star Wars pod-racers on the same market, we may need two different sets of attributes. We further limit the model by assuming that every attribute set has one of three types:

- A set of explicitly listed values, such as car models
- An interval of integer numbers, such as year
- An interval of real values, such as 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. If a customer is willing to buy a certain car with 20,000 miles, she will agree to accept an identical vehicle with 10,000 miles for the same price. That is, a buyer will always accept smaller mileage if it does not affect other aspects of the transaction.

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.

Note that monotonic attributes are numeric, and we cannot apply this notion to an unordered set of values, such as car models. Also note that we do not consider partially ordered attribute sets, which is a simplification, because some attributes may be "partially monotonic." For example, Camry LX (a deluxe model) is definitely better than Camry CE (a basic model), whereas the choice between Camry CE and Sienna CE depends on a specific customer. In addition, observe that a monotonic attribute is a generalization of price, that is, it may be subject for negotiation, with the other attributes fixed. For example, the shipping service may negotiate a delivery date.

Theoretically, we can view the price as one of the monotonic attributes; however, its use in the implemented system is different from the other attributes.

2.2 Order execution

We introduce the notion of a *fill*, which is a specific transaction between buyers and sellers. We first consider a trade between one buyer and one seller, and then define fills for transactions that involve multiple buyers and sellers. We use this notion to define conditions of an acceptable multi-order transaction.

2.2.1 Fills

When a buy order matches a sell order, the corresponding parties can complete a trade, which involves the delivery of appropriate items to the buyer for an appropriate price. We use the term $fill$ to refer to the traded items and their price. For example, suppose that a buyer has placed an order for two sports cars, and a seller is selling three red Mustangs. If the prices of these orders match, the buyer may purchase two red Mustangs from the seller; 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 $Step_s$

Note that a fill consists of a specific item, price, and size; unlike an order, it cannot include a set of items or a range of sizes. Furthermore, all items in a fill have the same price; for instance, a fill (`red-Mustang`, \$18,000, 2) means that a buyer has purchased two red Mustangs for \$18,000 each. If she had bought these cars for different prices, we would represent them as two different fills for the same order.

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 buyers quality or the sellers quality. A more complex heuristic may search for an item that maximizes the overall satisfaction of the buyer and seller.

- *Price choice*: The default strategy is to split the price difference between a buyer and seller, which means that $p = \frac{Price_b(i) + Price_s(i)}{2}$. Another standard option is to favor either the buyer or the seller; that is, we may always use $p = Price_b(i)$ or, alternatively, we may always use $p = Price_s(i)$.

- *Size choice*: We assume that buyers and sellers are interested in trading at the maximal size, or as close to the maximum as possible; thus, the fill has the largest possible size. This default is the same as in financial markets.

In Figure 2.3, we give an algorithm that finds the maximal fill size for two matching orders. The GCD function determines the greatest common divisor of $Step_b$ and $Step_s$ using Euclid's algorithm. The main procedure finds the least common multiple of $Step_b$ and $Step_s$, denoted *step*, which equals $\frac{Step_b \cdot Step_s}{\text{GCD}(Step_b, Step_s)}$. Then, it computes the greatest size, divisible by *step*, that is between $\max(Min_b, Min_s)$ and $\min(Max_b, Max_s)$. If no fill size satisfies these constraints, the algorithm returns zero, which means that the size specification of the buy order does not match that of the sell order.

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

If $Price_b(i) \geq Price_s(i)$, then return $\frac{Price_b(i)+Price_s(i)}{2}$;
else, return NONE (no acceptable price)

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

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 $step$:
$size = \lfloor \frac{min(Max_b, Max_s)}{step} \rfloor$
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($Step_b, Step_s$)
$small = \min(Step_b, Step_s)$
$large = \max(Step_b, Step_s)$
Repeat while $small \neq 0$:
$rem = large \bmod small$
$large = small$
$small = rem$
Return $large$

Figure 2.3: Computing the price (FILL-PRICE) and size (FILL-SIZE) of a fill for two matching orders.

After getting a fill, the trader may keep the initial order, reduce its size, or remove the order; the default option is the size reduction. For example, if a seller has ordered a sale of three cars and gotten a two-car fill, the size of her order becomes one. If the reduced size is zero, we remove the order from the market. If the size remains positive but drops below the minimal acceptable size Min, the order is also removed. The process of generating a fill and then reducing the buy and sell order is called the *execution* of these orders. In Figure 2.4, we illustrate four different scenarios of order execution.

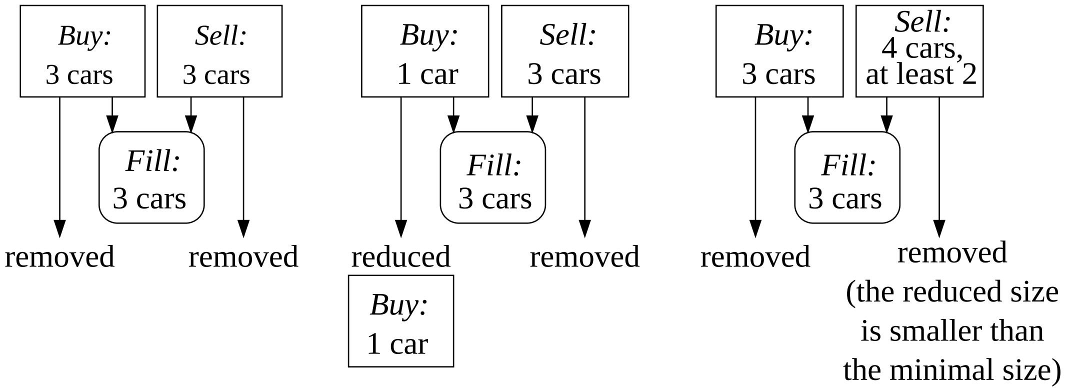


Figure 2.4: Examples of order execution.

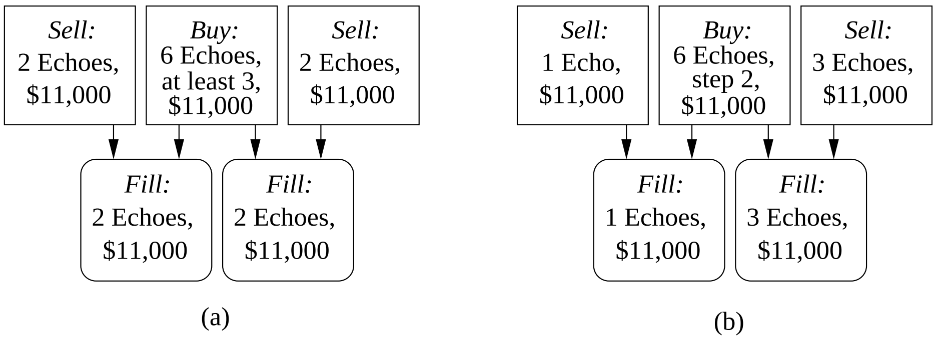


Figure 2.5: Examples of multi-order transactions.

2.2.2 Multi-fills

If a user specifies a minimal order size, she may indicate that she will accept a trade with multiple matching orders if their *total* size is no smaller than her minimal size. For example, the buy order in Figure 2.5(a) does not match either of the sell orders; however, if the user allows trades with multiple matching orders, we can generate the transaction shown in Figure 2.5(a). If the user specifies the size step, then the total size of a multi-order transaction must be divisible by this step (Figure 2.5b).

To formalize this concept, suppose that a buyer has placed an order $(I_b, Price_b, Qual_b, Max_b, Min_b, Step_b)$, and she is willing to trade with multiple sell orders. Suppose further that sellers have placed k orders, denoted as follows:

$(I_1, Price_1, Qual_1, Max_1, Min_1, Step_1)$

$(I_2, Price_2, Qual_2, Max_2, Min_2, Step_2)$

$\ldots$

$(I_k, Price_k, Qual_k, Max_k, Min_k, Step_k)$

Then, the buy order matches these k sell orders if they satisfy the following conditions.

- For every $j \in [1 \ldots k]$, there is an item $i_j \in I_b \cap I_j$, such that $Price_j(\ i_j) \leq Price_b(\ i_j)$
- For every $j \in [1 \ldots k]$, there is a price p_j, such that
    - $Price_j(\ i_j) \leq p_j \leq Price_b(\ i_j)$
    - $Qual_b(\ i_j,\ p_j) \geq 0$ and $Qual_j(\ i_j,\ p_j) \geq 0$
- There are acceptable sizes, $size_1, size_2, \ldots, size_k$, such that
    - For every $j \in [1 \ldots k]$, $Min_j \leq size_j \leq Max_j$
    - For every $j \in [1 \ldots k]$, $size_j$ is divisible by $Step_j$
    - $Min_b \leq size_1 + size_2 + \ldots + size_k \leq Max_b$
    - $size_1 + size_2 + \ldots + size_k$ is divisible by $Step_b$

Similarly, we can define a match between a sell order and multiple buy orders. In addition, we can allow transactions that involve multiple buy orders and multiple sell orders, as shown in Figure 2.6. We will define the conditions for such transactions in Section 2.2.3.

We refer to the result of a multi-order transaction as a *multi-fill*, which is a set of several fills for a given order. Since a multi-fill can include both purchases and sales, we denote the purchase sizes by positive integers, and the sale sizes by negative integers. For instance, the orders in Figure 2.6 get the following multi-fills:

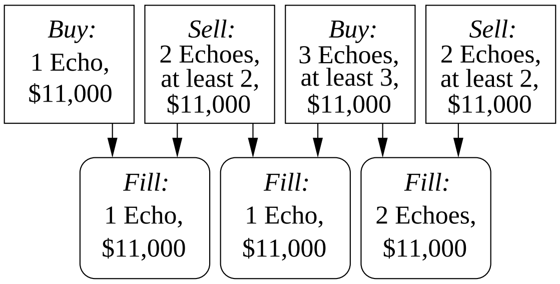


Figure 2.6: Example of a transaction that involves multiple buy and sell orders.

- Buy_1: (`Echo`, \$11,000, 1)
- $Sell_2$: (`Echo`, \$11,000, -1), (`Echo`, \$11,000, -1)
- Buy_3: (`Echo`, \$11,000, 1), (`Echo`, \$11,000, 2)
- $Sell_4$: (`Echo`, \$11,000, -2)

As another example, the multi-fill (`Camry`, \$20,000, 1), (`Echo`, \$11,000, -2) means that a trader has bought a Camry for \$20,000 and sold two Echoes for \$11,000 each.

We say that two multi-fills have the same item set if they include the same commodities, not necessarily at the same price. For example, the multi-fill (`Echo`, \$11,000, 1), (`Echo`, \$12,000, 1) has the same item set as (`Echo`, \$11,000, 2); in this example, both multi-fills represent the purchase of two Echoes. As another example, (`Camry`, \$20,000, 2), (`Echo`, \$11,000, 1), (`Echo`, \$12,000, -2) includes the same item set as (`Camry`, \$20,000, 3), (`Camry`, \$21,000, -1), (`Echo`, \$11,000, -1); both multi-fills represent a purchase of two Camries and sale of an Echo. Finally, we define the *empty multi-fill*, denoted $\emptyset$, as the empty set of fills.

2.2.3 Equivalence of multi-fills

We next observe that different multi-fills may be equivalent from a traders point of view. For instance, buying two Echoes for \$10,000 each and immediately selling one of them for the same price is equivalent to buying one car; that is, the multi-fill (`Echo`, \$10,000, 2), (`Echo`, \$10,000, -1) is equivalent to (`Echo`, \$10,000, 1). As another example, if two fills include the same set of items and the same total price, most traders would consider them identical; thus, (`Echo`, \$10,000, 1), (`Tercel`, \$12,000, 1) is equivalent to (`Echo`, \$11,000, 1), (`Tercel`, \$11,000, 1). If a multi-fill $M\text{-}Fill_1$ is equivalent to $M\text{-}Fill_2$, we write "$M\text{-}Fill_1 \equiv M\text{-}Fill_2$."

An exact definition of equivalence may vary across markets. For example, if the price is in dollars, buying two identical items for prices p_1 and p_2 is equivalent to buying each item for $\frac{p_1 + p_2}{2}$. On the other hand, if we consider the sale of mortgages and view the interest rate as a price, this averaging rule may not work because of nonlinear growth of compound interests.

To formalize the concept of equivalence, we first define the *union of multi-fills*, which is the set of all transactions contained in these multi-fills. For example, the union of (`Echo`, \$10,000, 1) and (`Echo`, \$10,000, 1), (`Tercel`, \$12,000, 1) is a three-element multi-fill (`Echo`, \$10,000, 1), (`Echo`, \$10,000, 1), (`Tercel`, \$12,000, 1). This definition is different from the standard union of sets since a multi-fill may include multiple identical elements. We denote the multi-fill union by "+" to distinguish it from the set union:

$$(i1_1, p1_1, size1_1), \ldots, (i1_m, p1_m, size1_m) + (i2_1, p2_1, size2_1), \ldots, (i2_k, p2_k, size2_k)$$
$$= (i1_1, p1_1, size1_1), \ldots, (i1_m, p1_m, size1_m), (i2_1, p2_1, size2_1), \ldots, (i2_k, p2_k, size2_k).$$

A multi-fill equivalence is defined for a specific market, and it may be different for different markets. Formally, it is a relation between multi-fills that satisfies the following properties:

- Standard properties of equivalence:
  - $M\text{-}Fill \equiv M\text{-}Fill$ (reflexivity)
  - If $M\text{-}Fill_1 \equiv M\text{-}Fill_2$, then $M\text{-}Fill_2 \equiv M\text{-}Fill_1$ (symmetry).
  - If $M\text{-}Fill_1 \equiv M\text{-}Fill_2$ and $M\text{-}Fill_2 \equiv M\text{-}Fill_3$, then $M\text{-}Fill_1 \equiv M\text{-}Fill_3$ (transitivity).

- A transaction that involves zero items is equivalent to the empty multi-fill:
  $(i, p, 0) \equiv \emptyset$

- Buying or selling identical items separately, at the same price, is equivalent to buying or selling them together:
  $(i, p, size_1), (i, p, size_2) \equiv (i, p, size_1 + size_2)$.

- The union operation preserves the equivalence:
  If $\textit{M-Fill}_1 \equiv \textit{M-Fill}_2$, then $\textit{M-Fill}_1 + \textit{M-Fill}_3 \equiv \textit{M-Fill}_2 + \textit{M-Fill}_3$.

These conditions are the required properties of the multi-fill equivalence in all markets; in a specific market, the equivalence may have additional properties. For example, $(i, p, 1), (i, p, 1)$ is always equivalent to $(i, p, 2)$. On the other hand, $(i, p_1, 1), (i, p_2, 1)$ may be equivalent to $(i, \frac{p_1 + p_2}{2}, 2)$ in some markets, such as car trading, but not in other markets, such as mortgage sales.

We use the concept of equivalence to define conditions for a multi-order transaction, such as the trade in Figure 2.6. Specifically, we can execute a transaction that involves k orders, denoted $Order_1, Order_2, \ldots, Order_k$, if there exist multi-fills $\textit{M-Fill}_1, \textit{M-Fill}_2, \ldots, \textit{M-Fill}_k$, such that

- For every $j \in [1 \ldots k]$, $\textit{M-Fill}_j$ matches $Order_j$

- $\textit{M-Fill}_1 + \textit{M-Fill}_2 + \ldots + \textit{M-Fill}_k \equiv \emptyset$

For example, we can select the following multi-fills for orders in Figure 2.6:

- (`Echo`, \$11,000, 1) matches Buy_1

- (`Echo`, \$11,000, -1), (`Echo`, \$11,000, -1) matches $Sell_2$

- (`Echo`, \$11,000, 1), (`Echo`, \$11,000, 2) matches Buy_3 item (`Echo`, \$11,000, -2) matches $Sell_4$

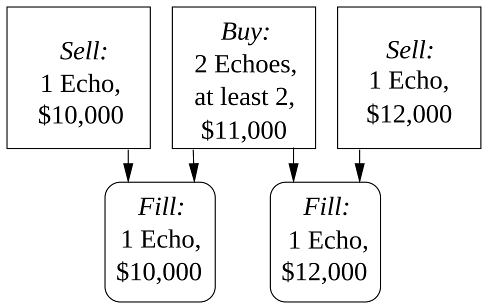


Figure 2.7: Example of price averaging.

The union of these multi-fills is equivalent to the empty multi-fill:

$$(\texttt{Echo},\ \$11{,}000,\ 1) + (\texttt{Echo},\ \$11{,}000,\ -1),\ (\texttt{Echo},\ \$11{,}000,\ -1)$$
$$+\ (\texttt{Echo},\ \$11{,}000,\ 1),\ (\texttt{Echo},\ \$11{,}000,\ 2) + (\texttt{Echo},\ \$11{,}000,\ -2) \equiv \emptyset.$$

2.2.4 Price averaging

A trader may sometimes accept a multi-fill even if it does not satisfy the conditions of a multi-fill. For example, consider the transaction in Figure 2.7. The price of the second fill does not match the buy order, but the overall price of the two fills is acceptable. The buyer pays $22,000 for two cars; thus, their average price matches the buyers price limit. When placing an order, the trader has to specify whether she will accept such price averaging.

Since the price may not be in dollars, we cannot directly compute the total price of a multi-fill. For example, if the price of a mortgage is the interest rate, the overall interest of a multi-fill is not the sum of its elements rates. To allow price averaging, we define a *payment* for a multi-fill. Intuitively, it represents a dollar amount delivered by a buyer or received by a seller, and the units of payment may differ from price units. For example, when a homebuyer negotiates a mortgage, she may use interest as a price measure; after receiving the mortgage, she will repay it in dollars. Formally, a payment is a real-valued function *Pay* on multi-fills that has the following properties:

- If a trader does not buy or sell any items, the payment is zero:

  $Pay(\emptyset) = 0$.

- The payment is proportional to the number of items:

  $Pay((i, p, size)) = size \cdot Pay((i, p, 1))$.

- The payment for multiple fills equals the sum of respective payments:

  $Pay((i_1, p_1, size_1), \ldots, (i_k, p_k, size_k))$

  $= Pay((i_1, p_1, size_1)) + \ldots + Pay((i_k, p_k, size_k))$.

- Equivalent multi-fills incur the same payment:

  If $M\text{-}Fill_1 \equiv M\text{-}Fill_2$, then $Pay(M\text{-}Fill_1) = Pay(M\text{-}Fill_2)$.

- A buyers payment is monotonically increasing on price:

  If $p_1 \leq p_2$, then $Pay((i, p_1, 1)) \ \leq \ Pay((i, p_2, 1))$.

Since a payment is monotonic on price, both buyers and sellers want to reduce their payments. For buyers, this reduction means paying less money; for sellers, it means getting more money, which is represented by a smaller negative value. For example, a car seller would rather get the $-\$12{,}000$ payment than the $-\$11{,}000$ payment, which means that she prefers selling her vehicle for \$12,000 rather than for \$11,000. A buyer's payment may be negative, which means that a seller pays the buyer for accepting an undesirable item. For example, if the seller wants to dispose of a broken car, she may pay \$100 for pulling it away; in this case, the buyers payment is $-\$100$.

Note that a payment depends not only on price but also on specific items; that is, $Pay((i_1, p, 1))$ may be different from $Pay((i_2, p, 1))$. For example, the payment for a 6% fifteen-year mortgage is different from the payment for a 6% thirty-year mortgage. Also note that the total payment of all transaction participants is zero. For example, consider the trade in Figure 2.7. The buyer's payment is \$22,000, the first selle'rs payment is

$-\$10{,}000$, and the second seller's payment is $-\$12{,}000$; thus, the overall payment is \$22,000 - \$10,000 - \$12,000 $= 0$.

We can decompose the payment for a multi-fill into the payments for its elements: $Pay((i_1, p_1, size_1), \ \ldots \ , (i_k, p_k, size_k)) = size_1 \cdot Pay((i_1, p_1, 1)) + \ \ldots \ + size_k \cdot Pay((i_k, p_k, 1))$. To simplify this notation, we will usually write $Pay(i, p)$ instead of $Pay((i, p, 1))$.

A user can also define a quality function for multi-fills. Formally, it is a real-valued function $Qual_m$ on multi-fills that satisfies the following constraints:

- If the user does not trade any items, the quality is zero:
  $Qual_m(\emptyset) = 0$.
- The quality function is consistent with the quality of simple fills:
  - If $size > 0$, then $Qual_m((i, p, size)) = Qual_b(i, p)$.
  - If $size < 0$, then $Qual_m((i, p, size)) = Qual_s(i, p)$.
- Equivalent fills have the same quality:
  If $M\text{-}Fill_1 \equiv M\text{-}Fill_2$, then $Qual_m(M\text{-}Fill_1) = Qual_m(M\text{-}Fill_2)$.
- The multi-fill union preserves relative quality of multi-fills:
  If $Qual_m(M\text{-}Fill_1) \leq Qual_m(M\text{-}Fill_2)$, then
  $Qual_m(M\text{-}Fill_1 + M\text{-}Fill_3) \leq Qual_m(M\text{-}Fill_2 + M\text{-}Fill_3)$.

Recall that the quality of simple fills is monotonic on price (Section 2.1.3), which implies that the multi-fill quality is also monotonic on price:

- If $size > 0$ and $p_1 \leq p_2$, then $Qual_m((i, p_1, size)) \geq Qual_m((i, p_2, size))$.
- If $size < 0$ and $p_1 \leq p_2$, then $Qual_m((i, p_1, size)) \geq Qual_m((i, p_2, size))$.

If the user does not provide a multi-fill quality function, we define it as the weighted mean quality of a multi-fills elements. If the multi-fill includes purchases $(i_1, p_1, size_1), \ldots, (i_j, p_j, size_j)$ and sales $(i_{j+1}, p_{j+1}, -size_{j+1}), \ldots, (i_k, p_k, -size_k)$, the default quality is:

$$Qual_m((i_1, p_1, size_1), \ldots, (i_j, p_j, size_j), (i_{j+1}, p_{j+1}, -size_{j+1}), \ldots, (i_k, p_k, -size_k))$$
$$= \frac{size_1 \cdot Qual_b(i_1,p_1) + \ldots + size_j \cdot Qual_b(i_j,p_j) + size_{j+1} + Qual_s(i_{j+1},p_{j+1}) + \ldots + size_k \cdot Qual_s(i_k,p_k)}{size_1 + \ldots + size_j + size_{j+1} + \ldots + size_k}.$$

Now suppose that a trader has placed an order $(I, Price, Qual_m, Max, Min, Step)$, and that she accepts price averaging. Then, a multi-fill $(i_1, p_1, size_1), \ldots, (i_k, p_k, size_k)$ is acceptable if it satisfies the following conditions.

- $i_1, \ldots, i_k \in I$
- $size_1 \cdot Pay(i_1, p_1) + \ldots + size_k \cdot Pay(i_k, p_k)$
  $\leq size_1 \cdot Pay(i_1, Price(i_1)) + \ldots + size_k \cdot Pay(i_k, Price(i_k))$
- $Qual_m((i_1, p_1, size_1), \ldots, (i_k, p_k, size_k)) \geq 0$
- $Min \leq size_1 + \ldots + size_k \leq Max$
- $size_1 + \ldots + size_k$ is divisible by $Step$

2.2.5 Fair trading

Fairness rules are based on the standards of the financial industry: the users must get the best available price for a given quality, the best price must be selected among competing orders, and the system must prefer earlier orders when the price is equal.

Formally, fair trading must satisfy the following constraints:

- When an order gets a fill, there is no better fill on the market, according to the order's preference function.
- When an order gets a fill, there is no equally good fill on the market with an older order.

Note that these conditions do not guarantee optimization; for example, the maximum *surplus*, which is defined as the difference between the respective buyer and seller price limits, is not guaranteed.

A more general concept of fairness is based on the "boys-and-girls" condition. Specifically, each order "wants" other orders, and the orders are prioritized. In this condition, fair trading must ensure that when A matches B and C matches D, it is not the case that (A wants D more than B) and (D wants A more than C).

2.3 Combinatorial orders

A combinatorial order is a collection of several orders with constraints on their execution. A simple example is a *spread*, often used in futures trading, which consists of a buy order and sell order that must be executed at the same time; for instance, a trader may place an order to buy gold futures and simultaneously sell silver futures.

Combinatorial auctions allow larger combinations of bids; for example, a trader can order a simultaneous purchase of a sport utility vehicle, trailer, boat, and two bicycles. Some auctions also support mutually exclusive bids; for instance, a user can indicate that she needs either a boat or two bicycles.

We describe combinatorial orders in the proposed exchange model, which include mutually exclusive orders, simultaneous transactions, and chains of consecutive trades.

2.3.1 Disjunctive orders

A *disjunctive-order* mechanism is for traders who want to execute one of several alternative transactions. For example, if the buyer wants to sell one of her three cars, she can place the order in Figure 2.8(a). As another example, if a buyer has a trailer, she can either buy an old sport utility vehicle (SUV) for pulling it or sell the trailer (Figure 2.8b). We have to guarantee that a trader does not get fills for two different elements of a dis-

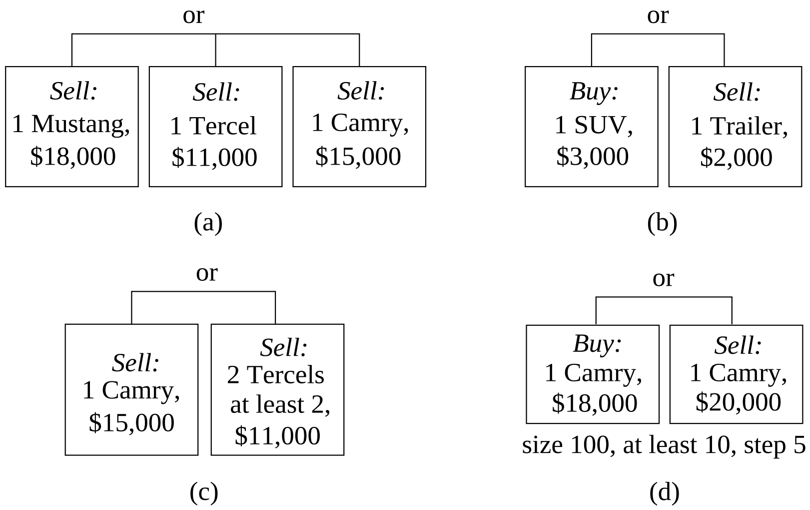


Figure 2.8: Examples of disjunctive orders.

junctive order. For example, if a buyer places the order in Figure 2.8(a), she will sell at most one of her cars.

If a trader specifies a size for some elements of a disjunctive order, these elements must be all-or-none orders; that is, their minimal sizes must be the same as the overall sizes. For example, a buyer may place an order to sell a Camry or two Tercels, as shown in Figure 2.8(c).

A disjunctive order as a whole can also have a size, which is equivalent to placing several identical orders. For example, suppose that a buyer has specified size five for the order in Figure 2.8(a). Then, she will sell five cars, and each car will be a Mustang, Tercel, or Camry. As another example, if she specifies size five for the order in Figure 2.8(b), she will complete five transactions, and each transaction will be either a purchase of a sport utility vehicle or a sale of a trailer; for instance, she may end up buying two sport utility vehicles and selling three trailers.

In addition, a disjunctive order can have a minimal size and size step. For example, suppose that a dealer is buying Camries for $18,000 and reselling them for $20,000, and she is interested in bulk transactions that involve at least ten cars. She may place the order in Figure 2.8(d); its minimal size is ten, and its step is five. If the minimal size of a disjunctive order is the same as the maximal size, it is an all-or-none order. In this case, it may be an element of another disjunctive order; it may also be an element of a conjunctive order, described in Section 2.3.2.

If a trader uses quality functions in a disjunctive order, she must specify a function for every element of a disjunction. If the trader does not specify quality functions, we use the same default as for simple orders. We utilize quality functions not only for selecting the best fill for each element of a disjunction, but also for selecting among fills for different elements. For example, suppose that a trader has placed the disjunction in Figure 2.8(b), and that she has specified a quality function $Qual_b$ for the buy element and $Qual_s$ for the sell element. Suppose further that she has found an old Explorer for \$2,500, and that she can sell the trailer for \$2,200. If $Qual_b$(`Explorer`, \$2,500) > $Qual_s$(`Trailer`, \$2,200), the trader prefers the purchase of the Explorer to the sale of the trailer.

To summarize, a disjunctive order consists of five parts:

- Set of all-or-none orders, $Order_1, Order_2, \ldots, Order_k$
- Optional permission for price averaging
- Overall order size, *Max*
- Minimal acceptable size, *Min*
- Size step, *Step*

A multi-fill *M-Fill* matches a disjunctive order if we can decompose it into m multi-fills, denoted $\textit{Sub-Fill}_1$, $\textit{Sub-Fill}_2, \ldots, \textit{Sub-Fill}_m$, that match elements of the disjunction and satisfy the following constraints.

- $Min \leq m \leq Max$, and m is divisible by *Step*
- Every multi-fill $\textit{Sub-Fill}_j$ matches some element of the disjunction; that is, for every $j \in [1..m]$, there is $l \in [1..k]$ such that $\textit{Sub-Fill}_j$ matches $Order_l$.

- If the order does not allow price averaging, then

  $\textit{M-Fill} \equiv \textit{Sub-Fill}_1 + \textit{Sub-Fill}_2 + \ldots + \textit{Sub-Fill}_m$.

  If the order allows price averaging, then

  M-Fill includes the same items as $\textit{Sub-Fill}_1 + \textit{Sub-Fill}_2 + \ldots + \textit{Sub-Fill}_m$,

  and $\textit{Pay}(\textit{M-Fill}) = \textit{Pay}(\textit{Sub-Fill}_1 + \textit{Sub-Fill}_2 + \ldots + \textit{Sub-Fill}_m)$.

For example, suppose that a trader has placed the disjunctive order in Figure 2.8(c), and specified that its overall size is six and its minimal acceptable size is three. Then, the multi-fill (`Camry`, \$16,000, −2), (`Tercel`, \$11,500, −2) matches the order since we can decompose this multi-fill into three parts:

(`Camry`, \$16,000, −1) + (`Camry`, \$16,000, −1) + (`Tercel`, \$11,500, −2).

The first and second parts match the left element of the disjunction, and the third part matches the right element. After completing this transaction, we reduce the size of the disjunctive order, as shown in Figure 2.9.

2.3.2 Conjunctive orders

A trader places a *conjunctive order* if she needs to complete several transactions together. For example, if a customer wants to sell her old Tercel and buy a new Echo, she may place the order in Figure 2.10(a). As another example, if a trader plans to buy a sport utility vehicle, trailer, and boat, she may place the order in Figure 2.10(b).

We have to guarantee that the trader gets a fill for all elements of a conjunction at the same time. For example, the conjunction in Figure 2.10(a) can simultaneously trade with two simple orders (Figure 2.11a). As a more complex example, it can be a part of a transaction that involves several conjunctive orders (see Figures 2.11b and 2.11c).

A disjunctive order may be an element of a conjunction. For example, if a customer wants to buy a trailer, boat, and one of several alternative vehicles, she can place the order

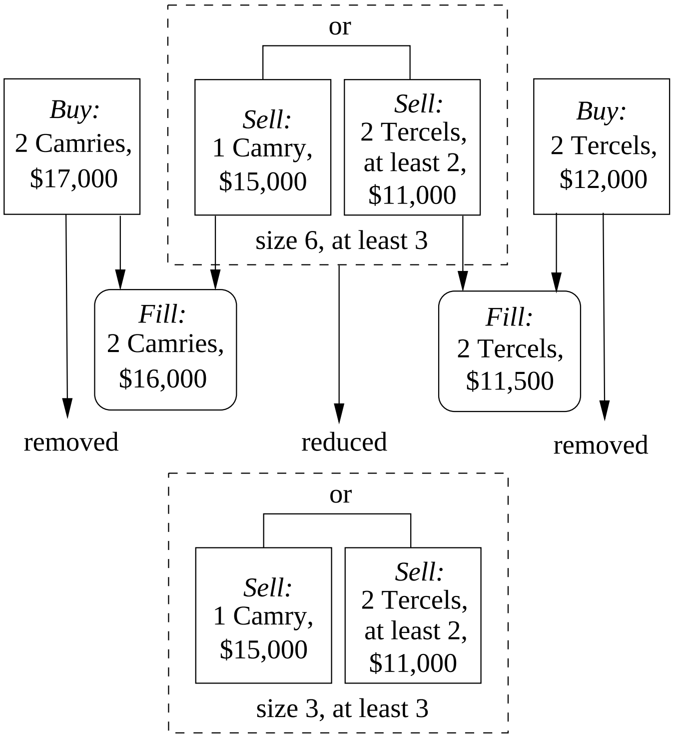


Figure 2.9: Example of a transaction that involves a disjunctive order.

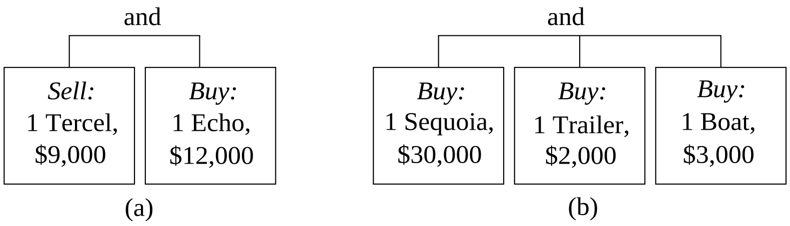


Figure 2.10: Examples of conjunctive orders.

in Figure 2.12(a). Furthermore, a conjunctive order may be an element of a disjunction, and a trader may nest several conjunctions and disjunctions (Figure 2.12b).

If a trader specifies sizes for some elements of a conjunctive order, these elements must be all-or-none. For example, a customer may place an order to sell an old Tercel and buy two new Echoes (Figure 2.13a). As another example, she may sell a Tercel and buy two new cars, where each new car is either an Echo or a Civic (Figure 2.13b).

A conjunctive order as a whole may have a size, which is equivalent to placing several identical orders; in addition, it may have a minimal size and size step. For instance, if a

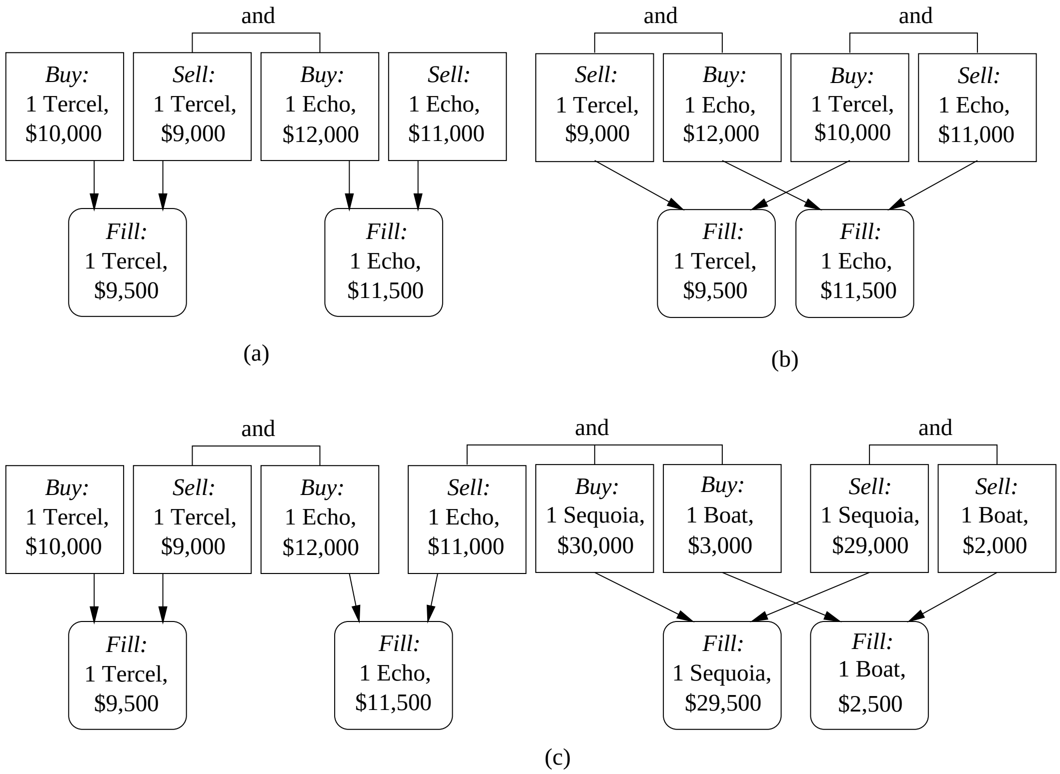


Figure 2.11: Example transactions that involve conjunctive orders.

trader places the order in Figure 2.14(a), she may complete two, four, or six conjunctive transactions; each transaction will involve selling a Tercel and buying an Echo. If the minimal size of a conjunctive order is the same as the overall size, then it is an all-or-none order, and it can be an element of a disjunction or another conjunction.

When a trader places a conjunctive order, she is usually interested in the price of the overall transaction rather than the prices of its elements. For example, suppose that a customer is selling her old Tercel and buying an Echo, and she is willing to spend $3,000 for this transaction. She may sell the Tercel for $9,000 and buy an Echo for $12,000; alternatively, she may sell her old car for $8,000 and buy a new one for $11,000.

We allow two mechanisms for specifying a price limit for the overall transaction. First, a trader can set a payment limit for a conjunctive order, along with price limits for its elements. For instance, she may place the order shown in Figure 2.14(b); in this case, she wants to get at least $5,000 for her old Tercel and pay at most $15,000 for a new

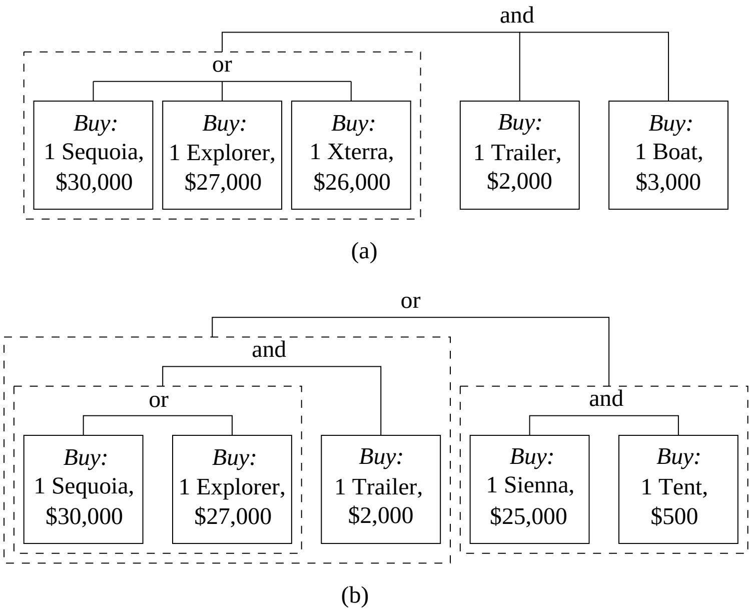


Figure 2.12: Examples of nested disjunctions and conjunctions.

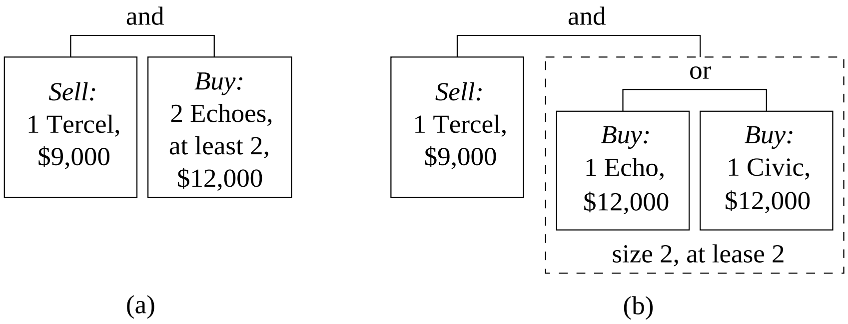


Figure 2.13: Examples of size specifications in conjunctive orders.

Echo, and her total cash spending must be at most \$3,000. Thus, she is willing to sell her Tercel for \$5,000 and buy an Echo for \$8,000, and she is also willing to sell her car for \$12,000 and buy a new one for \$15,000. If the overall conjunctive order has a size, then the price limit is for size 1. For example, if a buyer specifies a size of ten for a conjunctive order and trades all ten, then the overall payment will by \$80,000, not \$8,000. Recall that the units of payment may differ from price (Section 2.2.3); for example, mortgage brokers may express the price as an interest rate, and the overall payment for a conjunctive order as a dollar amount.

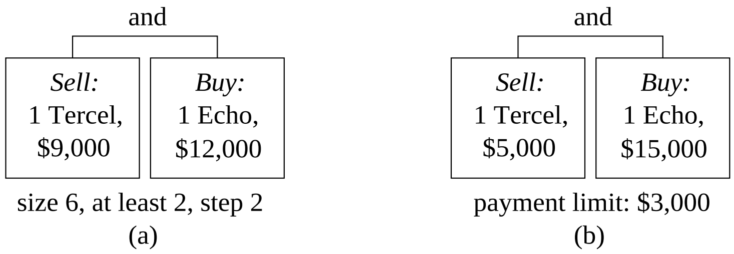


Figure 2.14: Conjunctive orders with a size specification (a) and payment limit (b).

Second, a trader can specify a price limit for each element of a conjunction, and indicate that she will accept any multi-fill that leads to the same total payment, even if the prices of individual elements do not satisfy the price limits. This option is similar to price averaging for simple orders, described in Section 2.2.4. For example, suppose that a trader uses this option for the order in Figure 2.12(a). If she gets a Sequoia with a trailer and boat, the total payment must be at most $\$30,000 + \$2,000 + \$3,000 = \$35,000$. If she gets an Explorer instead of a Sequoia, the total payment must be at most $\$27,000 + \$2,000 + \$3,000 = \$32,000$.

In addition, a trader can specify a multi-fill quality function for a conjunctive order. For instance, suppose that a trader has placed the order in Figure 2.13(a), and she prefers Sequoia to Explorer. Then, her quality function must satisfy the following constraint:

$$\begin{aligned}&Qual_m((\texttt{Explorer}, \$27{,}000, 1), (\texttt{Trailer}, \$2{,}000, 1), (\texttt{Boat}, \$3{,}000, 1))\\ &< Qual_m((\texttt{Sequoia}, \$30{,}000, 1), (\texttt{Trailer}, \$2{,}000, 1), (\texttt{Boat}, \$3{,}000, 1)).\end{aligned}$$

A trader can also specify quality functions for elements of a conjunction, but we do not use them for selecting the best fill; their only use is to reject matches with negative quality.

To summarize, a conjunctive order consists of seven parts:

- Set of all-or-none orders, $Order_1, Order_2, \ldots, Order_k$
- Overall payment limit, *Pay-Max*
- Multi-fill quality function, $Qual_m$
- Optional permission for price averaging

- Overall order size, *Max*
- Minimal acceptable size, *Min*
- Size step, *Step*

We next define a multi-fill that matches a conjunctive order. We first consider a conjunction of size one and then generalize the definition to larger sizes. A multi-fill *M-Fill* matches a conjunctive order of size one if it can be decomposed into multi-fills for the elements of the conjunction, denoted $\mathit{Sub\text{-}Fill}_1, \mathit{Sub\text{-}Fill}_2, \ldots, \mathit{Sub\text{-}Fill}_k$, that satisfy the following conditions.

- For every $j \in [1..k]$, $\mathit{Sub\text{-}Fill}_j$ matches Order_j
- If the order does not allow price averaging, then
  $\mathit{M\text{-}Fill} \equiv \mathit{Sub\text{-}Fill}_1 + \mathit{Sub\text{-}Fill}_2 + \ldots + \mathit{Sub\text{-}Fill}_k$.
  If the order allows price averaging, then
  M-Fill includes the same items as $\mathit{Sub\text{-}Fill}_1 + \mathit{Sub\text{-}Fill}_2 + \ldots + \mathit{Sub\text{-}Fill}_k$,
  and $\mathit{Pay}(\mathit{M\text{-}Fill}) = \mathit{Pay}(\mathit{Sub\text{-}Fill}_1 + \mathit{Sub\text{-}Fill}_2 + \ldots + \mathit{Sub\text{-}Fill}_k)$.
- $\mathit{Pay}(\mathit{M\text{-}Fill}) \leq \mathit{Pay\text{-}Max}$
- $\mathit{Qual}_m(\mathit{M\text{-}Fill}) \geq 0$

For example, the multi-fill (`Tercel`, \$9,000, −1), (`Echo`, \$12,000, 1), (`Civic`, \$12,000, 1) matches the conjunctive order in Figure 2.13(b). To show the match, we decompose it into two parts:

$(\texttt{Tercel}, \$9,000, -1) + (\texttt{Echo}, \$12,000, 1), (\texttt{Civic}, \$12,000, 1)$.

The first part matches the sell element of the conjunction, and the second part matches the disjunctive buy.

If a conjunctive order includes a size specification, then a multi-fill $\mathit{M\text{-}Fill}_s$ matches the order if it can be decomposed into multi-fills $\mathit{M\text{-}Fill}_1, \mathit{M\text{-}Fill}_2, \ldots, \mathit{M\text{-}Fill}_{size}$ that satisfy the following conditions.

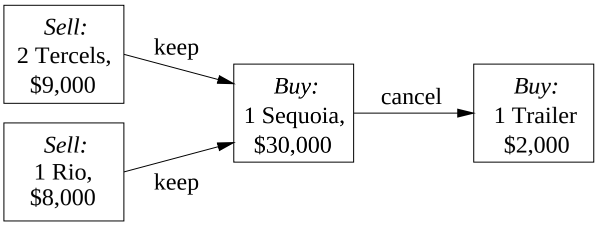


Figure 2.15: Example of a chain order. The trader first sells two Tercels and a Rio, then purchases a Sequoia, and finally acquires a trailer.

- $\mathit{Min} \leq \mathit{size} \leq \mathit{Max}$
- size is divisible by Step
- For every $l \in [1..\mathit{size}]$, $\mathit{M\text{-}Fill}_l$ matches the conjunctive order
- If the order does not allow price averaging, then

  $\mathit{M\text{-}Fill}_s \equiv \mathit{M\text{-}Fill}_1 + \mathit{M\text{-}Fill}_2 + \ldots + \mathit{M\text{-}Fill}_{\mathit{size}}$.

  If the order allows price averaging, then

  $\mathit{M\text{-}Fill}_s$ includes the same items as $\mathit{M\text{-}Fill}_1 + \mathit{M\text{-}Fill}_2 + \ldots + \mathit{M\text{-}Fill}_{\mathit{size}}$,

  and $\mathit{Pay}(\mathit{M\text{-}Fill}_s) = \mathit{Pay}(\mathit{M\text{-}Fill}_1 + \mathit{M\text{-}Fill}_2 + \ldots + \mathit{M\text{-}Fill}_{\mathit{size}})$.

For instance, the conjunctive order in Figure 2.14(a) matches the multi-fill (`Tercel`, \$9,000, −2), (`Echo`, \$12,000, 2), which can be decomposed into two parts:

(`Tercel`, \$9,000, −1), (`Echo`, \$12,000, 1) + (`Tercel`, \$9,000, −1), (`Echo`, \$12,000, 1).

2.3.3 Chain orders

The *chain-order* mechanism allows execution of several transactions in a sequence. To illustrate it, suppose that a buyer plans to sell two Tercels and a Rio, and to purchase a Sequoia. Because of budgetary constraints, she wants to sell all three cars before buying a new one. Suppose further that a buyer wishes to acquire a trailer after buying a Sequoia. In Figure 2.15, we show the sequence of a buyers transactions, which form a chain order.

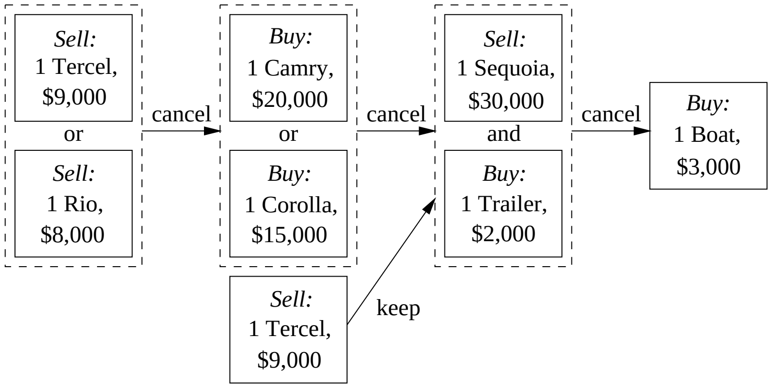


Figure 2.16: Chain order with two simple orders, two disjunctions, and a conjunction.

Formally, a chain order is a directed acyclic graph; its nodes are orders, and edges are temporal constraints. If the graph includes an edge from $order_1$ to $order_2$, we can execute $order_2$ only after we have completely filled $order_1$. For instance, we cannot execute a buyer's buy orders before she sells her Rio and both Tercels.

The elements of a chain order may be combinatorial orders; that is, the chain may include disjunctive orders, conjunctive orders, and even other chains. We do not impose any restrictions on the elements of a chain; in particular, they may not be all-or-none. In Figure 2.16, we show a chain that includes two simple orders, two disjunctions, and a conjunction.

If a trader cancels an element of a chain without getting a fill, she may want to execute the following orders; alternatively, she may want to cancel them. For each edge in the chain, the trader has to specify whether the cancellation of the earlier order causes the cancellation of the later one; in Figure 2.15, we show such specifications. In this example, if a buyer cancels either sale, she is still interested in buying a Sequoia. On the other hand, if she cancels the purchase of a Sequoia, she will not buy a trailer. As another example, the removal of the leftmost disjunction in Figure 2.16 will cause the cancellation of all buy orders.

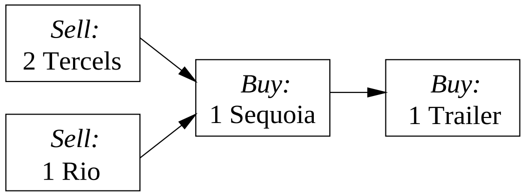


Figure 2.17: Active and inactive elements of a chain order. Thick boxes mark active orders, which can lead to immediate trades.

When placing a chain order, a trader may specify its size, which is equivalent to placing several identical orders. For example, if a buyer specifies that the size of her order in Figure 2.15 is two, she may end up selling four Tercels and two Rios, and buying two Sequoias and two trailers. On the other hand, a chain cannot have a minimal size or size step.

To summarize, a chain order consists of the following parts:

- Set of orders
- Temporal constraints that form a directed acyclic graph
- "Keep" or "cancel" specification for each constraint
- Overall order size

Since the execution of a chain includes several steps, it cannot be an all-or-none order; hence, it cannot be an element of a disjunctive or conjunctive order.

Intuitively, some elements of a chain are *inactive*; that is, they cannot lead to a trade. An element becomes *active* after the execution of all preceding elements. We illustrate this concept in Figure 2.17, where thick boxes mark active orders. The use of chain orders is a special case of activating an order upon certain conditions. We have considered three types of activation conditions in the implemented system: completion of the preceding orders in a chain, reaching a pre-set time, and a request from the user. A related problem is to develop a more general activation mechanism, which we plan to address in future research.

Chapter 3

Order Representation

We have built an exchange system for a special case of the automated trading problem. We describe the semantics of orders in the implemented exchange, and then explain its functionality and overall architecture.

3.1 Item sets

We first describe the representation of item sets and prices in the implemented system, and discuss the related limitations. The representation is less general than the formal model in Chapter 2. In particular, it limits possible item sets and does not allow the use of price and quality functions.

3.1.1 Buy item sets

A buyer may specify a set I of multiple items, but possible sets are limited by the representation. A buyer has to give a set of acceptable values for each attribute, which is called an *attribute set*. Thus, if the market includes n attributes, the buy-order description contains n attribute sets, and the set I is a Cartesian product of these attribute sets. For example, a buyer may indicate that she wants a Mustang or Corvette, the acceptable colors are red, silver, and black, the car should be made in 1998 or later, and it should have no more than 30,000 miles.

To give a formal definition, suppose that the set of all possible values for the first attribute is M_1, the set of all values for the second attribute is M_2, and so on, which means that the market set of all possible items is $M = M_1 \times M_2 \times \ldots \times M_n$. The buyer has to specify a set I_1 of values for the first attribute, where $I_1 \subseteq M_1$, a set of values for the second attribute, $I_2 \subseteq M_2$, and so on. The resulting item set I is the Cartesian product of the specified sets:

$$I = I_1 \times I_2 \times \ldots \times I_n.$$

For instance, the buyer has specified the following item set in the automobile example:

$$I = \{\texttt{Mustang}, \texttt{Corvette}\} \times \{\texttt{red}, \texttt{silver}, \texttt{black}\} \times \{1998, 1999, \ldots\} \times [0 \ldots 30{,}000].$$

Note that an item set in the implemented matcher *must* be a Cartesian product of attribute sets. For example, a buyer cannot describe an item set that includes red Mustangs and black Corvettes, but no black Mustangs.

3.1.2 Sell item sets

A sell order has to include a specific item, rather than a set of acceptable items. For example, a seller can order the sale of a red Mustang made in 1998, which has 10,000 miles; however, she cannot offer a set of various Mustangs made between 1990 and 2000. If she is selling multiple different cars, she needs to place multiple orders.

This limitation is based on the assumption that sellers usually offer specific items; however, some real-world markets do not satisfy this assumption. In particular, it creates problems for trading of services, such as package delivery or carpet cleaning. For instance, a maid service may offer to clean any carpets, rather than a specific carpet in a specific building.

To describe an item, the seller has to provide a value for each attribute; for example, the seller may define the model as `Mustang`, the color as `red`, and so on. If the market includes n attributes, then the definition of a sell item is a sequence of n values, $(i_1, i_2, \ldots, i_n)$, where i_1 is the value of the first attribute, i_2 is for the second attribute, and so on. For example, the seller would define her car as (`Mustang`, `red`, 1998, 10,000).

3.1.3 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, a buyer may indicate that she is purchasing an Echo or Tercel, the acceptable colors are white, silver, and gold, the car should be made after 1998, and it should have at most 30,000 miles.

To give a formal definition, suppose that the set of all possible values for the first attribute is M_1, the set of all values for the second attribute is M_2, and so on, which means that the market set is $M = M_1 \times M_2 \times \ldots \times M_n$. The trader has to specify a set $I_1 \subseteq M_1$ of values for the first attribute, a set $I_2 \subseteq M2$ of values for the second attribute, and so on. The resulting set I of acceptable items is the Cartesian product of the attribute sets:

$$I = I_1 \times I_2 \times \ldots \times I_n.$$

For instance, a buyer may specify the following item set:

$$I = \{\texttt{Echo}, \texttt{Tercel}\} \times \{\texttt{white}, \texttt{silver}, \texttt{gold}\} \times [1999 \ldots 2001] \times [0 \ldots 30,000].$$

3.1.4 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:

$$\begin{aligned} I = {} & \{\texttt{Camry}\} \times \{\texttt{white}, \texttt{silver}, \texttt{gold}\} \times [1995 \ldots 2001] \times [0 \ldots 30,000] \\ & \cup \{\texttt{Echo}\} \times \{\texttt{white}, \texttt{silver}, \texttt{gold}\} \times \{2001\} \times [0 \ldots 200]. \end{aligned}$$

Furthermore, the trader can indicate that she wants to avoid certain items; for instance, a superstitious user may want to avoid black cars with 13 miles on the odometer. In this case, the user has to provide 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,
  $I = I1_1 \times I1_2 \times \ldots \times I1_n \cup I2_1 \times I2_2 \times \ldots \times I2_n \cup \ldots \cup Ik_1 \times Ik_2 \times \ldots \times Ik_n$
- A filter function, $\mathit{Filter} : I \to \{\text{TRUE}, \text{FALSE}\}$,
  implemented by a C++ procedure.

We do not impose restrictions on filter functions; however, if a filter prunes too many items, the system may miss some matches. To avoid this problem, a user should choose Cartesian products that tightly bound the set of acceptable items.

3.1.5 Attribute sets

A buyer may use specific values or ranges; for example, she may specify a desired year as 2001 or as a range from 1998 to 2001. Note that ranges work only for numeric attributes, such as year and mileage.

The specification of a market may include certain *standard sets* of values, such as "all sports cars" or "all American cars", and the buyer may use them in her orders. For example, she may place an order for any American car.

Moreover, the buyer may use unions and intersections in her specification of attribute sets. For instance, suppose that a buyer is interested in Mustangs, Corvettes, and European sports cars; suppose further that we have defined a standard set that includes all European cars, and another standard set that comprises all sports cars. Then, the buyer can represent the desired set of models as follows:

$$\{\texttt{Mustang}, \texttt{Corvette}\} \cup (\texttt{European-cars} \cap \texttt{Sports-cars}).$$

REDUCE-SET(*attribute values* $a_1, a_2, \ldots, a_n$)
The algorithm inputs a disjunctive attribute set of n values.

Return $a_1 \cup a_2 \cup \ldots \cup a_n$

Figure 3.1: Simplifying a disjunctive attribute set. The algorithm returns the union of the attribute values.

A simplification mechanism within the system reduces the complexity of disjunctive attribute sets, thus improving efficiency. For example, the attribute set "1–5 or 4–8" simplifies to "1–8." This mechanism is located outside the matcher module, in the user interface, we give the algorithm in Figure 3.1.

Formally, an attribute set may be:

- A specific value, such as `Mustang` or 2001
- A range of values, such as 1998–2001
- A standard set of values, such as all European cars
- An intersection of several attribute sets
- A union of several sets

3.2 Price, quality, and size

We now explain the representation of price functions, quality functions, and order sizes.

3.2.1 Price

If a price function is a constant, a 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 a buyers 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 defined 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.

We specify a price function by an arbitrary C++ procedure that inputs an item and outputs the corresponding price limit. Note that a trader can specify different functions for different orders. If an order includes both a threshold and price function, the system uses the tighter of the two. For example, if a buyer's threshold for buying an Echo is \$12,000, and her price function returns \$12,500 for a specific vehicle, then the resulting price limit is \$12,000. Price thresholds help to prune unacceptable items, whereas C++ price functions allow more accurate evaluation of the remaining items. If the market includes monotonic attributes, the price functions must satisfy the monotonicity condition in Section 2.1.5. Note that if the price depends on "additional data," then different orders with identical item sets may have unequal price limits, requiring alternative price functions and impacting the efficiency.

3.2.2 Quality

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. The system includes several standard functions, and a trader can select among them and adjust the corresponding parameters. The system allows specifying different quality functions for different orders. If a user does not provide any quality function, the system uses a default quality measure defined through the price function (Section 2.1.3). All quality functions must be monotonic on price (Section 2.1.3); furthermore, if some attributes are monotonic, the quality functions must also satisfy the monotonicity condition of Section 2.1.5.

3.2.3 Size

The implemented size specification is the same as in the general model; it includes the overall size, minimal acceptable size, and size step. A trader can specify whether the system should preserve the minimal size in the case of a partial fill; if not, the system removes the minimal size after a partial fill (see Figure 3.2). The trader can also indicate whether she accepts multi-fills and allows price averaging.

3.3 Cancellations and inactive orders

We describe three mechanisms for removing an order from the market: immediate cancellation, expiration time, and temporary inactivation.

3.3.1 Cancellation

If a trader places an order and does not get a fill, she can later cancel it. If a trader has placed a combinatorial order, she can cancel the entire order or some of its elements. If she deletes an element of a disjunctive or conjunctive order, the other elements remain on market. On the other hand, a cancellation of a chain-order element may cause the deletion of other elements if the chain includes "cancel" constraints. For instance, the removal of the middle element in Figure 2.15 causes the cancellation of the rightmost element.

3.3.2 Expiration time

When placing an order, a trader can specify its *expiration time* with one-second precision. If the system does not find a match by the specified time, it cancels the order. A trader can also place an "immediate-or-cancel" order, which is removed if there is no immediate match. When placing a combinatorial order, a trader can specify an expiration time for

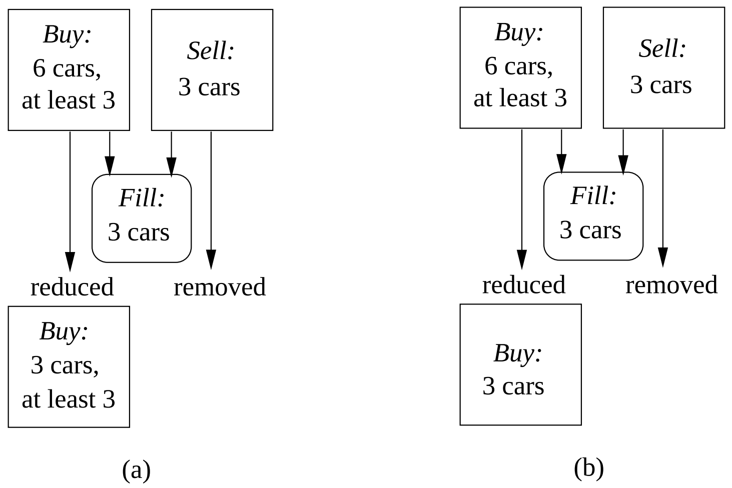
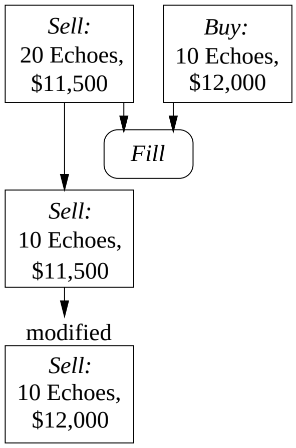


Figure 3.2: Examples of partial fills. If a trader specifies a minimal size, she indicates whether the system should preserve it after a partial fill (a), or remove it after the first fill (b).

the whole order, as well as different times for its elements. The expiration of the whole order leads to the removal of all its elements, whereas the expiration of individual elements does not affect the other elements.

3.3.3 Inactive order

We can mark some orders as *inactive*, which means that they cannot lead to a trade. We have introduced this mechanism for efficiency; it allows temporary removal of an order from the trading process without deleting it from the indexing structure. In particular, we use it to delay trading with inactive elements of a chain. We also enable users to inactivate their orders by hand, and to specify inactivation and reactivation times. The system allows inactivation of combinatorial orders, but it does not support selective inactivation of their elements.



(a)

```
CHANGE-ORDER(old-order )
If size[old-order ] > 10,
   then return "no change"
new-order := old-order
price[new-order ] := $12,000
Return new-order
```

(b)

Figure 3.3: Example of an order modification. If the size of the order drops to ten, the system should increase its price (a). The modification request includes a procedure that checks the size and adjusts the price (b).

3.4 Modifications

A trader can modify her order without removing it from the market. For example, if a buyer has placed an order to buy a gold Toyota Echo for $11,500 and has not gotten a fill, she can increase the price to $12,000. As another example, she can change the item description from "gold Echo" to "any Echo or Tercel." A trader can also define conditions that trigger a modification. For instance, suppose that a seller is selling twenty Echoes for $11,500 each. She can indicate that, if she gets a partial fill of at least ten cars, then her price increases to $12,000 (Figure 3.3a).

We specify a modification request by a C++ procedure that inputs an order and returns its modified version; if an order requires no modification, the procedure returns the "no change" signal. For example, if a seller wants to increase her order price after its size drops to ten, she can use the procedure in Figure 3.3(b). The system includes standard modification functions, and a user can select among them and adjust the appropriate parameters.

A user can specify an activation and expiration time for a modification request. When it becomes active, the system invokes the corresponding procedure. If it returns "no change," the system re-invokes it after any change to the order, which may be caused by a partial fill or by another modification. If a trader changes her mind, she can manually remove an old request. The system cancels a request in the following cases:

- The processing of the request has resulted in a modification
- The request has expired
- The corresponding order has been removed from the market
- The user has manually removed the request

A trader can also place an "immediate-or-cancel" request; if it does not result in an immediate modification, the system does not re-invoke it later.

3.5 Fairness heuristics

If the system identifies multiple matches between buy and sell orders, it may need to choose among them before generating fills. For example, if a buy order matches two different sell orders, the system has to select between the two, and the users usually expect a "fair"choice. We have used help from Michael Foster, a professional trader working for PowerLoom Corporation, to identify standard fairness expectations.

First, if the system has found several matches for the same trade, it should prefer the best-price match. For instance, if a buyer is looking for a sports car and the matcher has found two different orders to sell sports cars, then it has to match the buyer's buy order with the cheaper sell.

Second, if several users compete for the same trade, the system should give priority to the user who offers a better price. For instance, suppose that $Seller_1$ and $Seller_2$ are both selling a Corvette, and $Seller_1$'s price is better. Then, the system should fill $Seller_1$'s order before $Seller_2$'s order. Although traders often view this requirement as different from the first one, both impose the same constraints on the matching process.

Third, if several traders offer the same price, the system should execute their orders on a first-come first-serve basis. Thus, if $Seller_1$ and $Seller_2$ offer Mustangs for the same price, and $Seller_1$ has made her offer before $Seller_2$, then $Seller_1$'s sell order should get priority. Professional traders consider this "chronological" fairness almost as important as price fairness. When a seller makes a low-price offer in a volatile market, she assumes a risk and expects to be rewarded with priority over other sellers who follow her lead.

3.6 Confirmations

When placing an order, a trader can provide not only a description used in automated matching, but also additional information for human traders; for instance, a car dealer can post a picture of a vehicle. The system enables traders to browse through potential matches and choose the most desirable trade.

When a user places an order, she can indicate the need for *confirmation*. In this case, when the system finds matches, it displays their descriptions; if the user confirms some of the matches, the system executes the corresponding trades. For example, a buyer can place an order to buy a silver Corvette and require confirmation; then, she can browse through matching Corvettes and handpick the best match.

When the system finds a match between a buy and sell order, it checks the need for confirmation. If neither order requires confirmation, it immediately executes the trade. If one of the orders needs confirmation, the system notifies the corresponding user and executes the trade upon getting her approval (Figure 3.4a). If both orders require confirmation, it notifies both sides and completes the trade only after getting both approvals (Figure 3.4b). For example, if a buyer requests confirmation for her

Corvette order, and a seller sells a Corvette that also needs confirmation, then the system will complete the transaction only after getting approvals from both the buyer and seller. If the system finds a multi-order trade, it may need more than two confirmations.

A trader can confirm several different matches for her order, which allows the system to execute any of them. For instance, a buyer may confirm several Corvettes, and then she will get one of them.

When the system asks users for confirmation, it does *not* remove the matching orders from the trading process, and it can find other matches for them. If a user delays her confirmation, she may miss a trade, and then the system notifies her that the trade is no longer available (Figure 3.4c). For instance, when the buyer confirms the purchase of a specific Corvette, she may find out that someone else has bought it before her. The system tries to fill orders without confirmation before sending requests for confirmation. This strategy improves the speed of the trading process and reduces the number of "late" confirmations.

3.7 User actions

To summarize, a trader can perform six main operations: place an order, cancel an old order, activate or inactivate an order, place a modification request, cancel an old request, and confirm a trade. The implemented system does not support changes to modification requests; if a user needs to change her old request, she should cancel it and place a new one. We list the main elements of a simple order in Figure 3.5, the elements of a combinatorial order in Figure 3.6, and the elements of a modification request in Figure 3.7. If a user does not specify some of the elements, the system uses the corresponding default values.

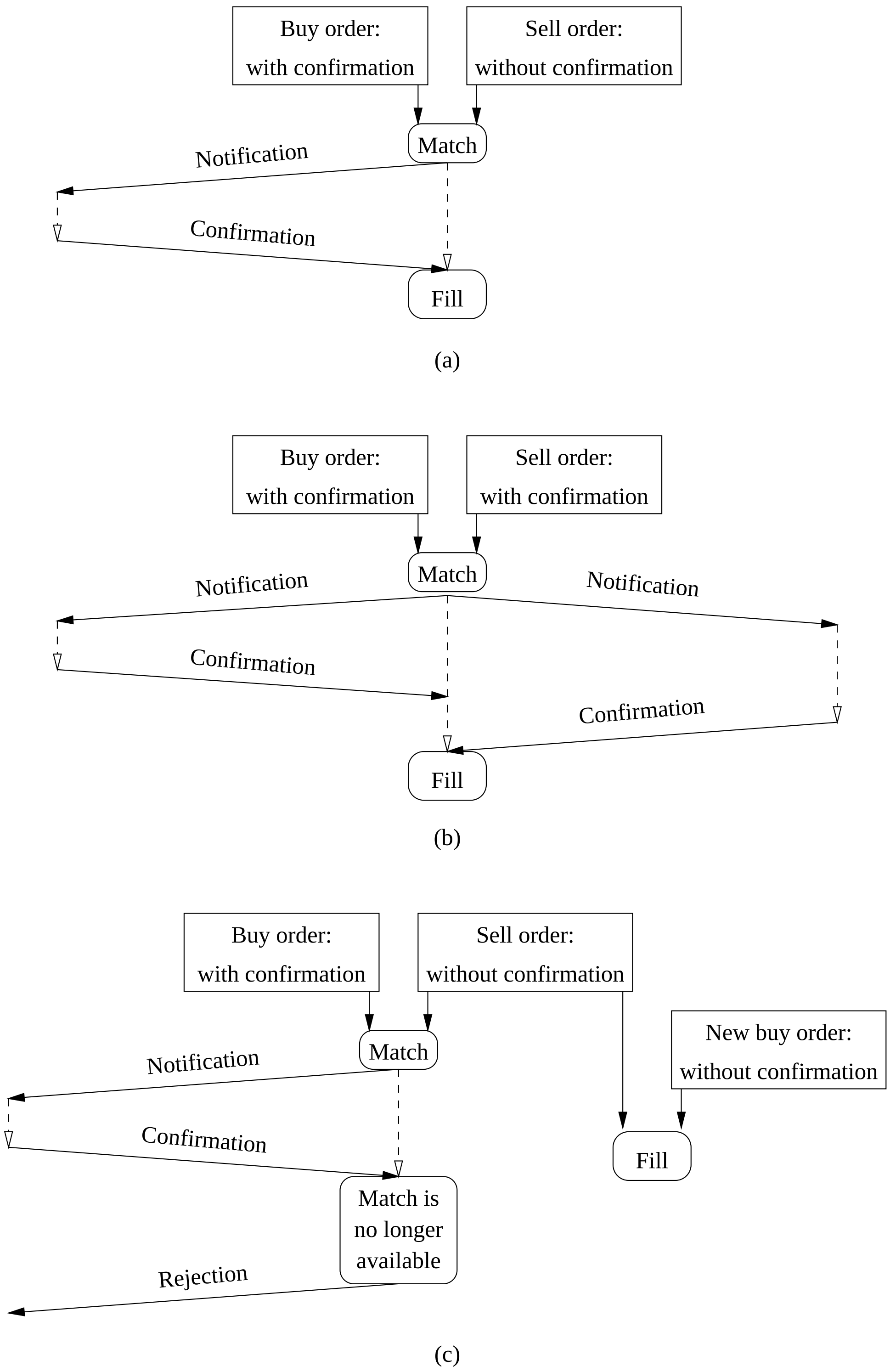


Figure 3.4: Trading with confirmations. If one of the matching orders needs confirmation, the system notifies the corresponding trader and waits for her approval (a). If both orders need confirmation, the system waits for approval from both traders (b). If it finds an alternative match before getting an approval, it executes the corresponding trade and later rejects the confirmation (c).

- *Item Set*
  Union of Cartesian products (no default)
  Filter function (by default, no filter)

- *Price*
  Price threshold for each Cartesian product
  (by default, $-\infty$ for sell orders and $+\infty$ for buy orders)
  Price function (by default, equal to the threshold)
  Quality function (by default, based on the price function)

- *Additional data*
  Data for price, quality, and filter functions (by default, no data)
  Information for human traders (by default, no information)

- *Size*
  Overall order size (by default, one)
  Minimal acceptable size (by default, one)
  Size step (by default, one)

- *Activation and expiration*
  Active or inactive status (by default, active)
  Inactivation time (by default, never)
  Reactivation time (by default, never)
  Expiration time (by default, never)

- *Options*
  Acceptance of multi-fills (by default, accept)
  Acceptance of price averaging (by default, accept)
  Confirmation request (by default, no confirmation)

Figure 3.5: Elements of a simple order and their default values. When a trader places an order, she has to specify an item set, and she may optionally specify the other elements.

- *Disjunctive order:*
  Set of all-or-none orders (no default)
  Size (the same as in a simple order)
  Activation and expiration (the same as in a simple order)
  Acceptance of price averaging (by default, accept)

- *Conjunctive order:*
  Set of all-or-none orders (no default)
  Overall payment limit (by default, $+\infty$)
  Multi-fill quality function (by default, no function)
  Size (the same as in a simple order)
  Activation and expiration (the same as in a simple order)
  Acceptance of price averaging (by default, accept)

- *Chain order:*
  Set of orders (no default)
  Ordering constraints (no default)
  "Keep" or "cancel" specification for each constraint (by default, "keep")
  Overall order size (by default, one)

Figure 3.6: Elements of combinatorial orders.

- Reference to a specific order (no default)
- Modification function that inputs an order and returns either a modified order or "no change" signal (no default)
- Activation time (by default, immediate)
- Expiration time (by default, never)

Figure 3.7: Elements of a modification request.

Chapter 4

Indexing Structure

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

4.1 Architecture

The system consists of a central matcher and multiple user interfaces, which run on separate machines and communicate over the network using an asynchronous messaging protocol. We outline the distributed architecture and explain the main functions of the matcher.

4.1.1 Top-level control

We describe a high-level logic that controls an indexing tree. We discuss different strategies for a matching round, and continuous versus periodic clearing. Both the total throughput of the system, and its response time, depend on the matching strategy chosen. We consider several trade-offs between maximizing the throughput and minimizing the response time. We also consider the fairness of each strategy.

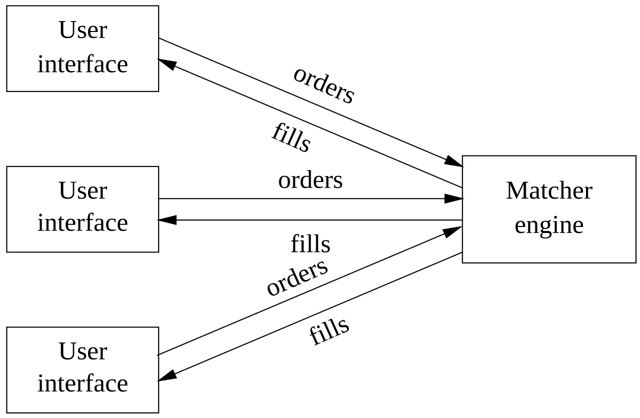


Figure 4.1: The architecture of the trading system.

- Placing an order
- Canceling an order
- Activating or inactivating an order
- Placing a modification request
- Canceling a modification request
- Confirming a trade

Figure 4.2: Main types of messages from a user interface.

4.1.2 User interfaces

The traders enter their orders through interface machines, which send the orders to the matcher engine (Figure 4.1). The central engine serves as a trading pit; it finds matches among orders, generates fills, and sends them to the corresponding interfaces. In Figure 4.2, we list the main types of messages from interfaces, which correspond to the user actions supported by the system (Sections 3.3–3.7).

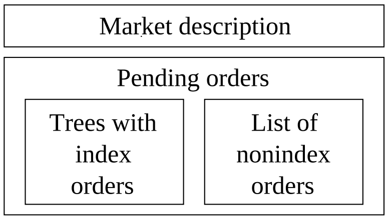


Figure 4.3: Main data structures in the matcher engine.

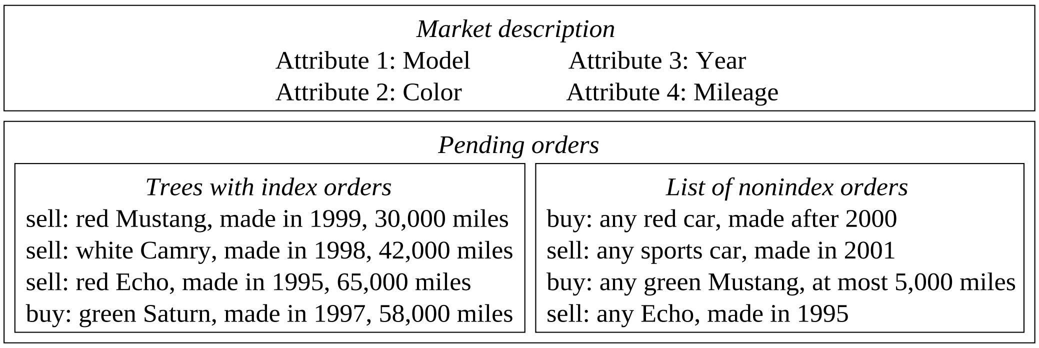


Figure 4.4: Example of index and nonindex orders.

4.1.3 Matcher engine

The matcher maintains a description of market attributes, a collection of pending orders, and a queue of scheduled future events (Figure 4.3). It includes a central structure for indexing of pending orders, implemented by two trees (Section 4.2). 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 4.4, 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. For example, if the orders are as shown in Figure 4.4, and a trader places an order to buy

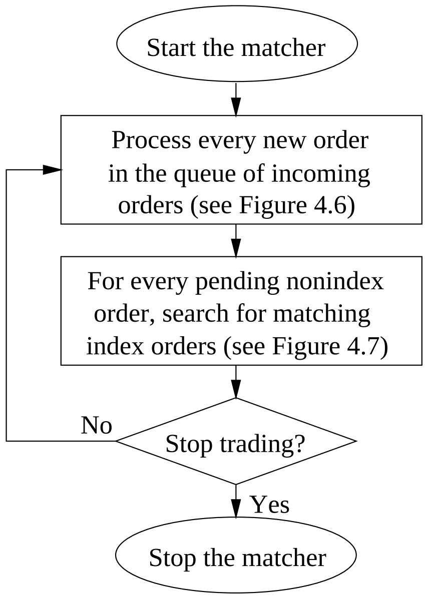


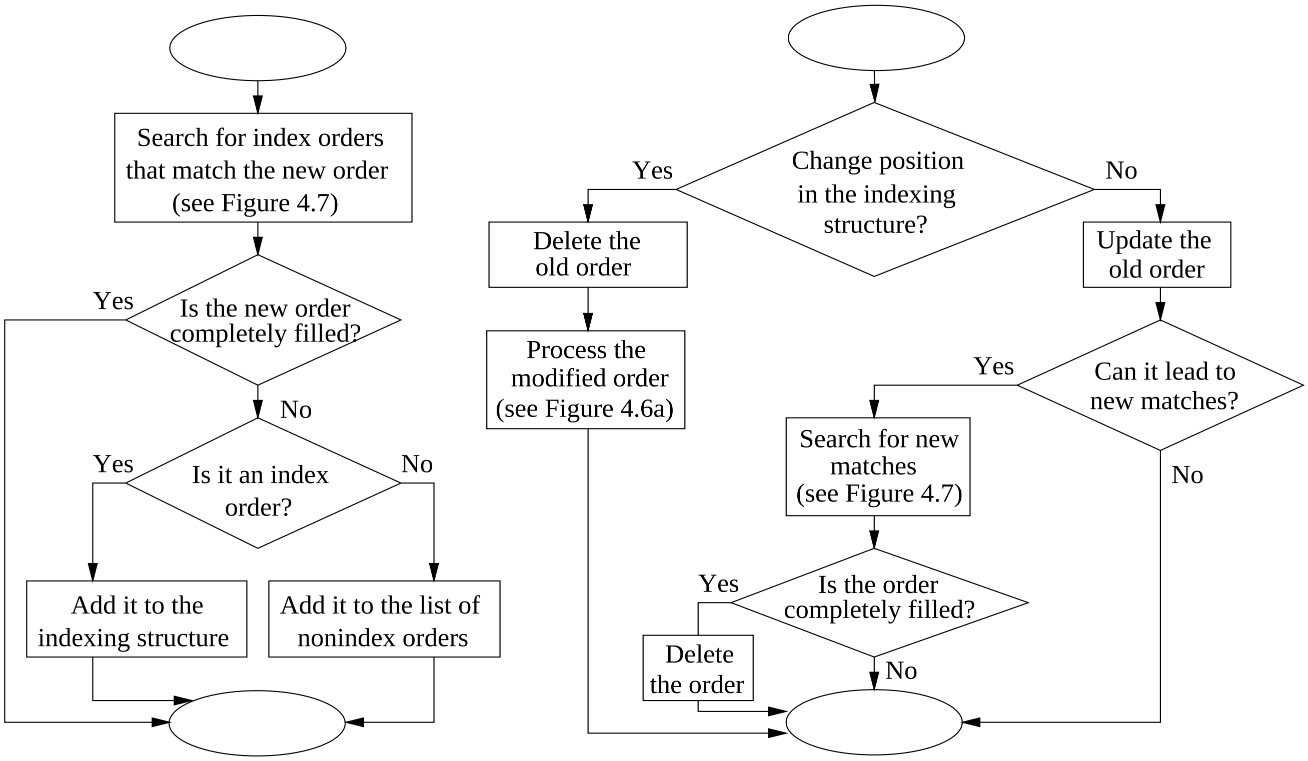
Figure 4.5: Top-level loop of the matcher engine.

a car made after 1997, then the system will find two matches: "sell red Mustang made in 1999" and "sell white Camry made in 1998."

4.1.4 Matching cycle

In Figure 4.5, we show the main cycle of the matcher, which alternates between parsing new messages and searching for matches. When it receives a message with a new order, it immediately searches for matching index orders (Figure 4.6a). 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 a seller 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 a buyer later places



(a) Processing a new order

(b) Modifying an order

Figure 4.6: Addition and modification of an order.

a buy order for a sports car, the system identifies the match with a seller's order, and informs a buyer and seller that they have exchanged a Mustang.

When the system gets a cancellation message, it removes the specified order from the market. When it receives a modification message, it makes the corresponding changes to the specified order (Figure 4.6b). If the changes can potentially lead to new matches, the system immediately searches for index orders that match the modified order; in Figure 4.8, we list all modifications that can result in new matches. For example, if a seller has placed an order to sell a Mustang for \$18,000, and she later reduces its price to \$17,500, then the system immediately looks for new matches. On the other hand, if she increases the price to \$18,500, the system does not search for matches.

After processing all messages, 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, as shown in Figure 4.7. For example, consider the market in Figure 4.4, and suppose that a seller 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 messages, it tries to fill the nonindex orders, and determines that a sellers order is a match for the old order to buy any green Mustang.

In addition to the matching cycle described above, the system may periodically search for new matches for an unmatched order. For a small-scale market, we can use the matching condition directly: for each new order, search all old orders for a match, and select the best match. The complexity of processing an order is linear in the number of pending orders, which is too slow for large markets.

4.1.5 Matching frequency

The matcher keeps track of the "age" of each order, and uses it to avoid repetitive search for matches among the same index orders. If it has already tried to find matches for some order, the matching process will involve search only among new index orders.

If a nonindex order has been on market for a long time, the system matches it less frequently than recent orders. We have implemented a mechanism that determines the intervals between searches for matching index orders; by default, the system increases the length of an interval between consecutive searches in proportion to an order age. If it does not find a match for a new nonindex order, it repeats the search on the next matching cycle, then after two cycles, then after four cycles, and so on; that is, the intervals between searches increase as the powers of two.

4.2 Indexing trees

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

Conceptually, these tree are tries, in which the description of specific orders serves as strings of equal length. These tries differ from traditional tries in that different attributes have different sets of values. In Figure 4.9, 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 root node encodes the first attribute, and its children represent different values of this attribute; in Figure 4.9, 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 $(i+1)$st level corresponds to all orders with a specific value 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. For example, the root in Figure 4.9 includes a red-black tree that indexes its children by model values, as shown in Figure 4.10. 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.

4.2.1 Multiple sell trees

The system uses multiple sell trees, where the recent orders are in the first tree, less recent orders are in the second tree, and so on. This strategy is faster than the use of times by a constant factor, since we do not need to look up old nodes and orders. We plan on addressing variations of this strategy, as well as the potential problems these changes could have on market fairness, as part of future research.

4.2.2 Standard sets

If a market includes standard sets of values, such as "all sports cars" and "all American cars," traders can use them in specifying their orders (Section 3.1). We define standard sets separately for each attribute; for instance, the set of American cars belongs to the "model" attribute. Note that a standard set may be a union of ranges, rather than an explicit list of values. For example, the set of collectable cars could include all cars made before the year 1976.

For every attribute, the system maintains a central table of standard sets, which consists of two parts (Figure 4.11a). The first part includes a sorted list of values for every standard set; it allows determining whether a given value belongs to a specific set, by the binary search in the corresponding list. The second part includes all values that belong to at least one set; for each value, we store a sorted list of sets that include it.

Every node of an indexing tree also includes a table of standard sets; for example, the root node in Figure 4.9 includes a table of sets for the first attribute (Figure 4.11b), and every "color" node includes a separate table of the second-attribute sets. Every set in the table includes a list of pointers to its elements in the red-black tree; for instance, the "American-cars" set points to the "Corvette" and "Mustang" nodes. If the current tree does not contain elements of some sets, we do not add these sets to the table; for example, if the market does not include any orders to sell European cars, then the "European-cars" set is not in the table.

We have implemented the table of sets by a red-black tree, which allows fast addition and deletion of sets, as well as fast retrieval of all values in a given set. For instance, if a buyer looks for American cars, the system retrieves the appropriate children of the "model" node by finding the "American-cars" set and following its pointers.

4.2.3 Summary data

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 or modification of an order

For example, consider node 2 in Figure 4.9; the subtree rooted in this node includes nine orders. If the newest of these orders was placed at 2 pm, the summary data in node 2 is as follows:

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

We also store the time that the oldest order was added to the subtree; this is done for the fairness heuristic, to ensure that earlier orders get preference over newer orders.

4.3 Basic tree operations

When a user places, removes, or modifies an index order, the system has to update the indexing tree. We first describe addition and deletion algorithms, and then explain the modification procedure.

4.3.1 Adding a new order

When a user places an index order, the system adds it to the corresponding leaf; for example, if a seller places an order to sell a black Camry, made in 1999, with 35,000 miles, the system adds it to node 16 in Figure 4.12. If the leaf is not in the tree, the matcher adds the appropriate new branch; for example, if a seller offers to sell a white Mustang, it adds the dashed branch in Figure 4.12.

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 4.13, we give the algorithms for updating the number of orders, 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 $O(n)$.

4.3.2 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 4.9, 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. We show a procedure for removing an order and the corresponding subtree in Figure 4.14.

After deleting an order, the system updates the summary data in the ancestor nodes. In Figure 4.15, we give procedures for updating the number of orders, 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,

$c_1, c_2, \ldots, c_n$. 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 $O(c_1 + c_2 + \ldots + c_n)$.

4.3.3 Modifying an order

If a trader changes the order size, expiration time, or additional data, the change does not affect the structure of the indexing tree; however, the system needs to update the summary data of the ancestor nodes. If a trader modifies the price of an order, the system changes the position of the order in the red-black tree of the leaf, and propagates the price change to the summary data.

Finally, if a trader changes the item specification, the system treats it as the deletion of an old order and addition of a new one. For example, suppose that a seller has placed an order to sell a black Camry, and the indexing tree is as shown in Figure 4.12. If a seller has entered a wrong color, and she later changes it to white, then the system removes the order from the leftmost leaf in Figure 4.12 and adds it to the rightmost leaf.

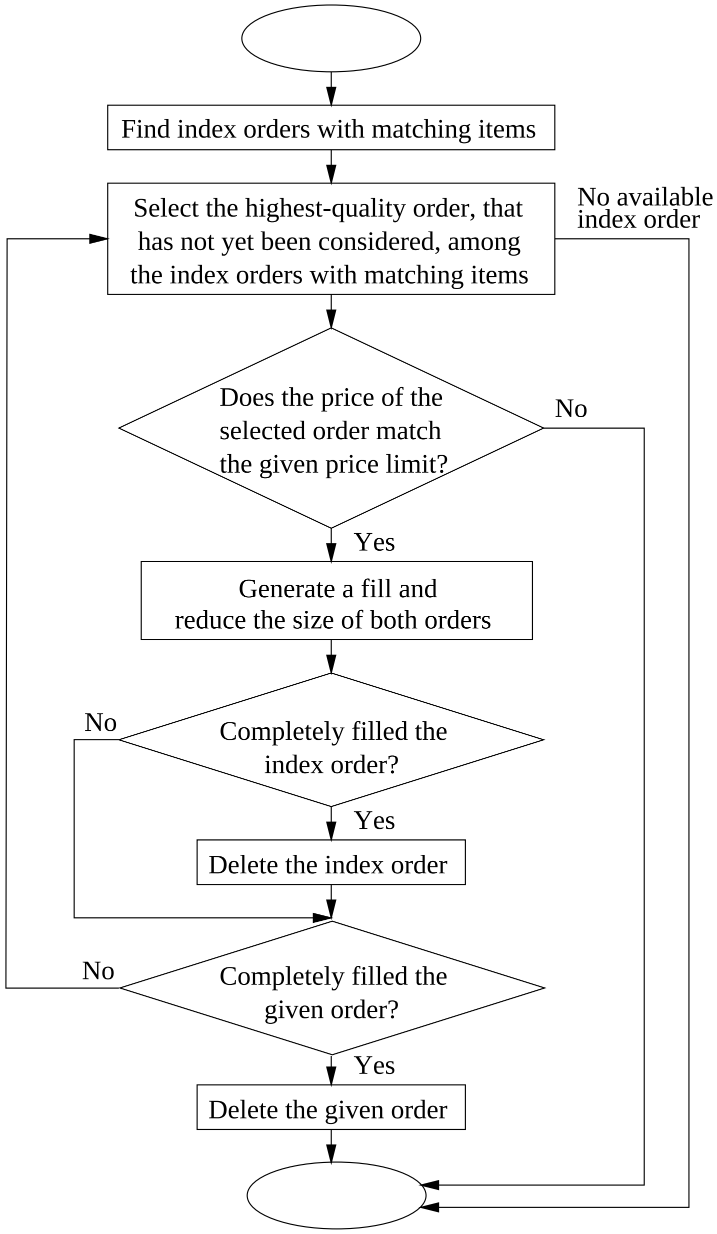


Figure 4.7: Search for index orders that match a given order.

- Changing the item set or additional data
- Increasing the price threshold of a buy order, or decreasing the threshold of a sell order
- Changing the price function or quality function
- Increasing the overall order size or reducing the minimal acceptable size
- Changing the size step in such a way that the new step is not a multiple of the old step
- Activating an inactive order
- Allowing multi-fills or price averaging
- Adding new elements to a disjunctive order, or deleting some elements from a conjunctive order

Figure 4.8: Order modifications that lead to new matches.

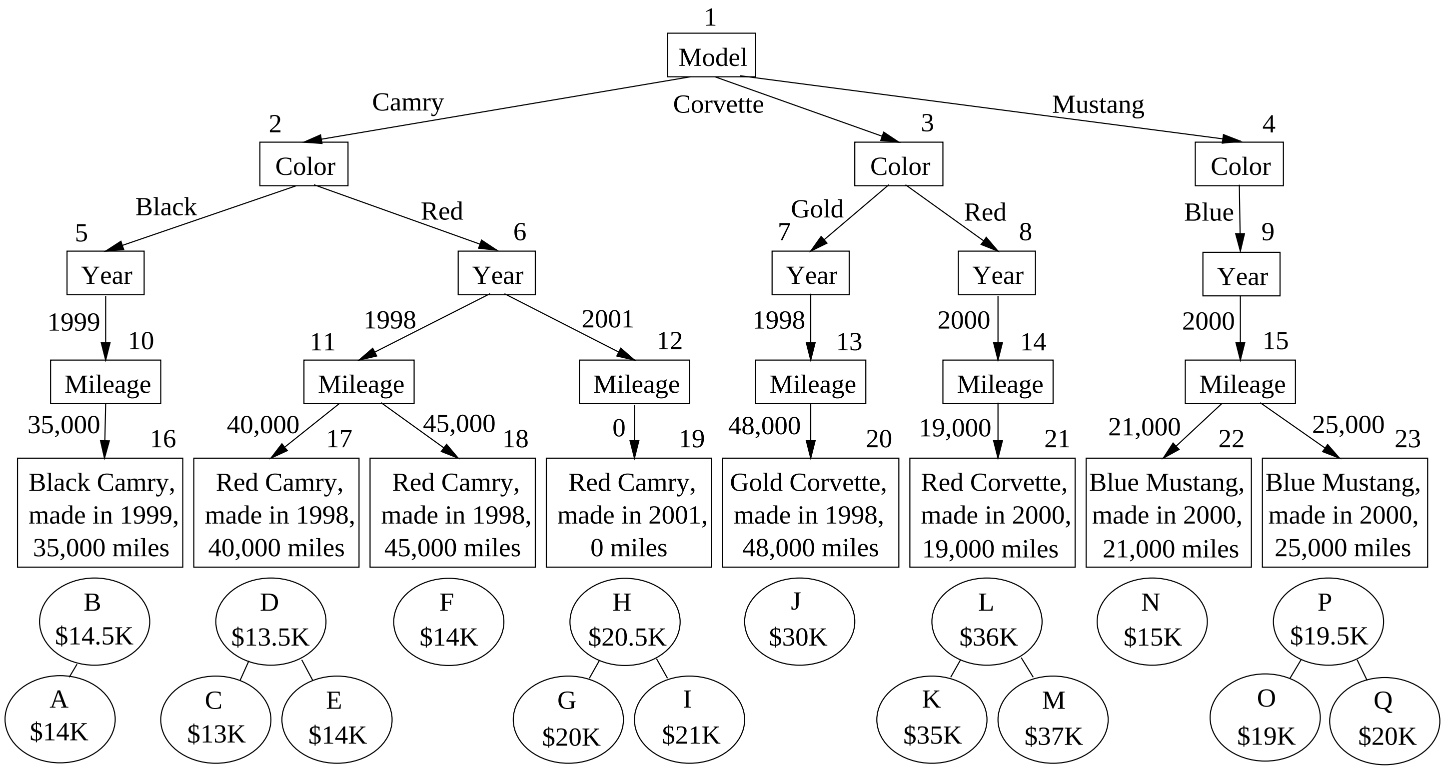


Figure 4.9: Indexing tree with seventeen orders.

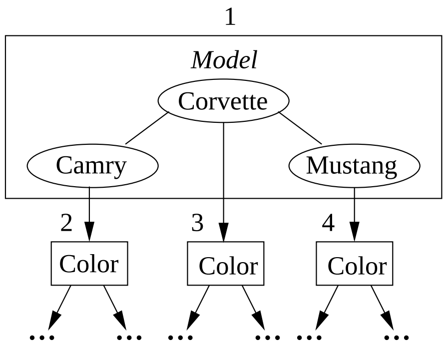
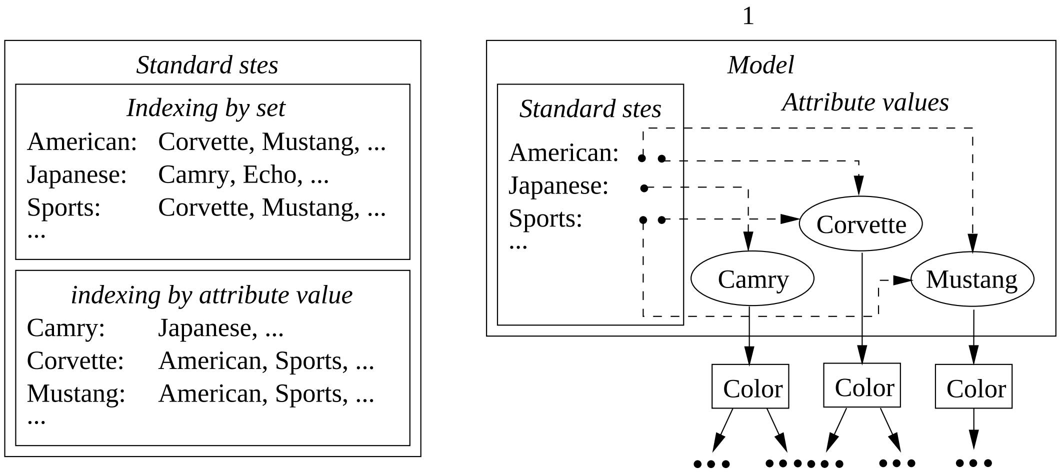


Figure 4.10: Node of an indexing tree. We arrange the attribute values in a red-black tree, and each value points to the corresponding child in the indexing tree.



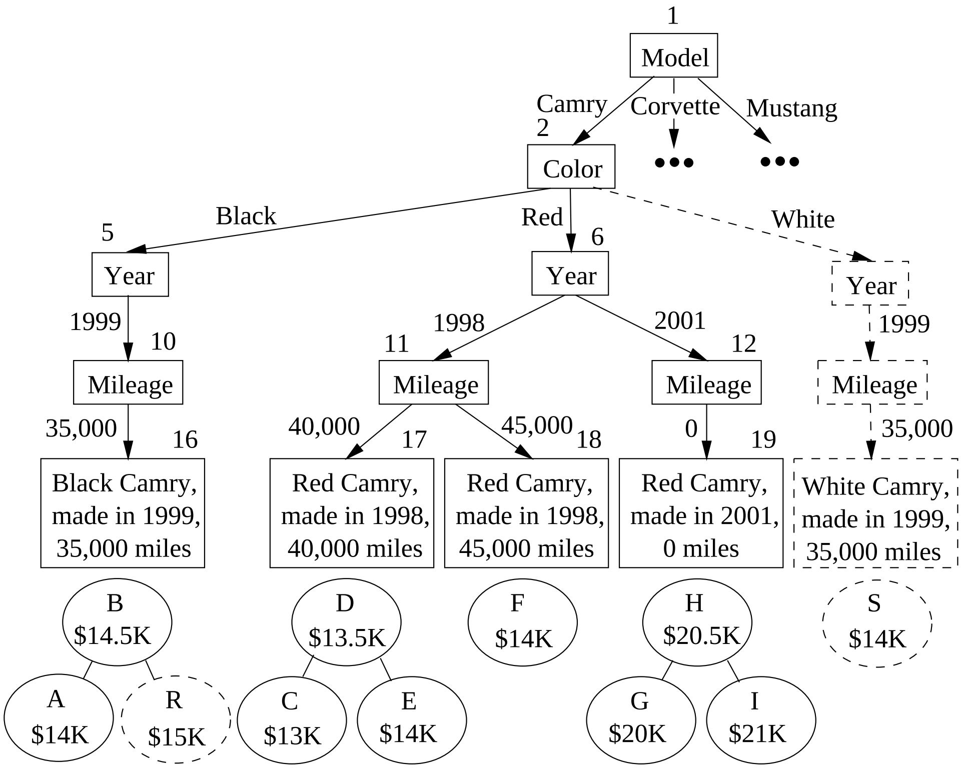
(a) Central table of standard stes.

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

Figure 4.11: Standard sets of values. The market includes a central table of sets (a), and every node in the indexing tree includes a table of sets for the respective attribute (b).

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

ADD-COUNT(*new-size*, *leaf*)
The algorithm inputs the leaf with the newly added order.

node := *leaf*
Repeat while *node* $\neq$ NIL:
    num-orders[*node*] := *num-orders*[*node*] + 1
    node := *parent*[*node*]

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-size[*node*] := *total-size*[*node*] + *new-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-price*[*node*] > *new-price*:
    min-price[*node*] := *new-price*
    node := *parent*[*node*]

Figure 4.13: Updating the summary data after addition of an order. We show the update of the order number (ADD-COUNT), total size (ADD-SIZE) and minimal price (ADD-PRICE).

DEL-ORDER(*order*, *leaf*)
The algorithm inputs an old order and the corresponding leaf.

Remove *order* from *leaf*
If *leaf* includes other orders, then terminate
node := *leaf*
Repeat while *parent*[*node*] $\neq$ NIL and *node* has no children:
    ancestor := *parent*[*node*]
    delete *node*
    node := *ancestor*

Figure 4.14: Deletion of an order. If it has been the only order in some subtree of the indexing tree, the system removes this subtree.

DEL-COUNT(*leaf*)
The algorithm inputs the leaf with a deleted order.

node := *leaf*
Repeat while *node* $\neq$ NIL:
  num-orders[*node*] := *num-orders*[*node*] $-$ 1
  node := *parent*[*node*]

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

Chapter 5

Search for Matches

We describe additional data used by the system in the search for matches, and describe two different search strategies. We then discuss the related fairness heuristics, and give the trade-offs involved between minimizing processing time, while ensuring fairness.

5.1 Additional search information

Each order maintains additional data used for simplifying the matching process. This information includes:

- *Time of placement*: the time that the order was initially added to the tree
- *Time of last modification*: the time that the order was last modified
- *Time of last search*: the time that the order was last searched for a match

We describe two algorithms that identify matches for a given order; the first algorithm is based on depth-first search in an indexing tree, and the second is best-first search. In Figure 5.1, we present the notation for the order and node structures used by the algorithms. We give the depth-first algorithm in Figures 5.2 and 5.3, and the best-first algorithm in Figures 5.4–5.6.

5.2 Depth-first search

The depth-first algorithm consists of two steps; it first finds the leaves of an indexing tree that match a given order (Figure 5.2), and then selects the best matching orders in these leaves (Figure 5.3).

5.2.1 Depth-first search strategies

The depth-first search used to retrieve matching orders may be implemented using alternative schemes listed below, each with its own inherent advantages and drawbacks.

- *Exhaustive search*: We may use DFS to find all leaves that match a given order, and then select matches among them.

- *Node limit*: We impose a limit on the number of explored leaves; this strategy does not improve the speed of finding a match, but it limits the time of an unsuccessful search. Moreover, it limits the time of a successful search that would return too many nodes.

- *Cluster search*: We may begin by exploring the subtrees that include more orders, since they are likely to have more matches. A variation of this strategy is to search for a large total volume of matches within a space.

- *Monotonicity*: We first explore the branches that are likely to have best matches. This technique works if the user does not specify price and quality functions that account for monotonicity. For example, if looking for an inexpensive car made after 1970, we begin with cars made in 1970. Note that this strategy is opposite to fairness, since it looks for the worst model that would satisfy the user's conditions.

5.2.2 Matching leaves

The algorithm in Figure 5.2 retrieves the matching leaves for a given item set, represented by a union of Cartesian products and a filter function.

The PRODUCT-LEAVES subroutine finds the matching leaves for one Cartesian product using depth-first search in the indexing tree. It identifies all children of the root that match the first element of the Cartesian product, and then recursively processes the respective subtrees. For example, suppose that a buyer is looking for a Camry or Mustang made after 2000, with any color and mileage, and the tree of sell orders is as shown in Figure 4.9. The subroutine determines that nodes 2 and 4 match the model, and then processes the two respective subtrees. It identifies three matching nodes for the second attribute, three nodes for the third attribute, and finally four matching leaves; we show these nodes by thick boxes.

If the system already tried to find matches for a given order during the previous execution of the main loop, it skips the subtrees that have not been modified since the previous search. If the order includes a union of several Cartesian products, the system calls the PRODUCT-LEAVES subroutine for each product. If the order includes a filter function, the system uses it to prune inappropriate leaves.

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 leaves, and a buyer places an order for any Camry, then the system retrieves the three leftmost leaves in Figure 4.9. We use this limit to control the trade-off between speed and quality of matches; a small limit ensures the efficiency but reduces the chances of finding the best match.

5.2.3 Best matches

After the system identifies matching leaves, it selects the best matching orders in these leaves, according to the quality function of the given order. In Figure 5.3, we give an algorithm that identifies the highest-quality matches and completes the respective trades. It arranges the leaves in a priority queue by the quality of the best unprocessed match in a leaf. At each step, the algorithm processes the best available match; it terminates after it fills the given order or runs out of matches.

For example, consider the tree in Figure 4.9, and suppose that a buyer places an order for four Camries or Mustangs made after 2000. We suppose further that she uses the default quality measure, which depends only on price. The system first retrieves order A with price $16,000 and size 2, then order B with price $16,500, and finally order O with price $19,000; we show these orders by thick circles.

If the price and quality functions do not use any extra data, then the sorted order on price corresponds to the sorted order on quality, as long as quality is monotonic on price; upon reaching the price and quality limits, we have exhuasted all matches. On the other hand, if they do use additional information, then the ordering may differ. In this case, the default strategy may result in selecting suboptimal matches, or missing matches. An alternative method is to consider all orders in the selected cells, and explicitly sort them on quality, but it is slower. We plan on developing this algorithm in future research.

We apply the matching test each time a sell order is processed. Note that an order may not match because of a size requirement or filter function. If the filter does not use extra any data, we can apply it once to a leaf. If it uses extra data, however, we must apply it to every order in the leaf. If an order matches, we apply the algorithm given in Figure 2.3 to determine the matching size of the fill.

5.3 Best-first search

If some attributes are monotonic, we can use best-first search to find optimal matches, which is usually faster than depth-first search. The best-first algorithm uses a node's summary data to estimate the quality of matches in the node's subtree; at each step, it processes the node with the highest quality estimate.

5.3.1 Quality estimates

We can compute a quality estimate for a node only if all branching in the node's subtree is on monotonic attributes; a node with this property is called *monotonic*. For example, node 6 in Figure 4.9 is monotonic; the branching in its subtree is on year and mileage, which are monotonic attributes. On the other hand, node 2 is not monotonic because its subtree includes branching on color.

In Figure 5.4, we give a procedure that inputs a monotonic node and constructs the best possible item that may be present in the node's subtree, based on the summary data. To estimate the node's quality, the system computes the quality of this item traded at the best possible price from the summary data. For example, consider node 6 in Figure 4.9; all orders in its subtree include red Camries, and the summary data show that the best year is 2003, the best mileage is 5,000, and the best price is \$13,000. Thus, the system computes the quality estimate as $Qual(\texttt{Camry}, \texttt{red}, 2003, 5{,}000, \$13{,}000)$.

5.3.2 Search steps

The best-first algorithm consists of two steps, similar to the steps of the depth-first algorithm. First, it finds all smallest-depth monotonic nodes that match a given order (Figure 5.5); for example, if a buyer is looking for a Camry or Mustang made after 2000, and the tree of sell orders is as shown in Figure 4.9, then the algorithm retrieves nodes 5, 6, and 9. Second, it finds the best matching orders in the subtrees of the selected nodes (Figure 5.6). It arranges the nodes into a priority queue by their quality estimates; at each step, it processes the highest-quality node. If this node is a leaf, the algorithm identifies the best-price matching order in the leaf and completes the respective trade. If the node is not a leaf, the algorithm identifies its children that match the given order, and adds them to the priority queue. The algorithm terminates when it fills the given order or runs out of matches.

5.4 Fairness heuristics

We consider multiple fairness methods, with the difference search techniques using different heuristics as follows:

- *Depth-first search*: We can use the depth first search with heuristics that guide toward a better match. The heuristics are the same as in the best-first search.
- *Beam search*: Beam search is a variety of the best-first search, where we limit the number of the best candidates. We control the trade-off between efficiency and fairness by controlling the breadth limit.
- *Best-price search*: If the user specifies quality but no partial quality function, we may use best-price search as a heuristic. That is, we can find the best-price search and hope that it maximizes the quality function.

5.5 Trade-offs

When implementing the search, we need to consider trade-offs between processing speed and near-fairness guarantees. The fairness definition is related to the quality function. The system must always satisfy hard constraints, defined by the item description, filters, price functions, and quality. On the other hand, the quality ordering is not a hard constraint, and we may not guarantee finding the best match or ensuring a perfect fairness. We consider three approaches.

- Disregard quality preferences and time priorities, and guarantee only hard constraints. If we use this strategy, we optimize for the speed of market clearing. The market is not "completely unfair" since we select matches at random and do not favor specific users.

- Guarantee fairness, that is, ensure that the "boys-and-girls" condition is never violated (subsection 2.2.5); this approach involves two separate problems. First, when searching for a match for a buy order, we need to find the best match, according to the quality function. Second, when a buy order finds a sell order, we need to ensure that there are no better buys that could match with the same sell.

- We consider heuristics that do not guarantee fairness but lead to near-fair choice, and at the same time take less time that the full guarantee. We consider several different heuristics, which lead to different trade-offs. We first consider the approach with no fairness, then complete fairness, and then discuss several trade-offs. The no-fairness approach is based on the depth-first search, the complete fairness approach is based on the best-first search, and near-fairness is depth-first search with fairness heuristics.

Elements of the order structure:

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

Elements of the indexing-tree node structure:

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

Additional elements of the leaf-node structure:

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

Figure 5.1: Notation for the orders and nodes of an indexing tree. Note that the leaf-node structure includes the five elements of the node structure and two additional elements. We use this notation in the pseudocode in Figures 5.2–5.6.

MATCHING-LEAVES($order, root$)

The algorithm inputs an order and the root of an indexing tree.

We denote the order's item set by $I1_1 \times ... \times I1_n \cup ... \cup Ik_1 \times ... \times Ik_n$.

Initialize an empty set of matching leaves, denoted *leaves*

For l from 1 to k,

call PRODUCT-LEAVES($Il_1 \times ... \times Il_n, Filter[order], Search\text{-}Time[order], root, leaves$)

Return *leaves*

PRODUCT-LEAVES($Il_1 \times ... \times Il_n, Filter, Search\text{-}Time, node, leaves$)

The subroutine inputs a Cartesian product $Il_1 \times ... \times Il_n$, a filter function, the previous-search time, a node of the indexing tree, and a set of leaves. It finds the matching leaves in the node's subtree, and adds them to the set of leaves.

If *Search-Time* is larger than *node*'s time of the last order addition, then terminate

If *node* is a leaf and $Filter(Item[node]) =$ TRUE, then add *node* to *leaves*

If *node* is not a leaf:

  Identify all children of *node* that match $I_{Depth[node]+1}$

  For each matching *child*,

call PRODUCT-LEAVES($Il_1 \times ... \times Il_n, Filter, Search\text{-}Time, child, leaves$)

Figure 5.2: Retrieval of matching leaves. The algorithm identifies the leaves of an indexing tree that match the item set of a given order. The PRODUCT-LEAVES subroutine uses depth-first search to retrieve the matching leaves for one Cartesian product.

LEAF-MATCHES(*order*, *leaves*)
The algorithm inputs an order and matching leaves of an indexing tree.

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

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

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

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

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

---

Figure 5.3: Retrieval of matching orders. The algorithm finds the highest-quality matches for a given order and completes the corresponding trades. The LEAF-PRIORITY subroutine adds a given leaf to the priority queue, arranged by the quality of a leaf's best-price unprocessed match. The TRADE subroutine completes the trade between the given order and the best available match.

BEST-ITEM($node$)

The algorithm inputs a monotonic node of an indexing tree.

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

Figure 5.4: Construction of the best possible item. The algorithm inputs a monotonic node and generates the best item that may be present in the subtree rooted at the node.

MATCHING-NODES($order$, $root$)

The algorithm inputs an order and the root of an indexing tree.
We denote the order's item set by $I1_1 \times ... \times I1_n \cup ... \cup Ik_1 \times ... \times Ik_n$.

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

PRODUCT-NODES($Il_1 \times ... \times Il_n$, $Search\text{-}Time$, $node$, $nodes$)

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

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

Figure 5.5: Retrieval of matching monotonic nodes. The algorithm identifies the smallest-depth monotonic nodes that match the item set of a given order. The PRODUCT-NODES subroutine uses depth-first search to retrieve the matching monotonic nodes for one Cartesian product.

NODE-MATCHES(*order*, *nodes*)
The algorithm inputs an order and matching monotonic nodes of an indexing tree.

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

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

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

Figure 5.6: Retrieval of matching orders. The algorithm finds the best matches for a given order and completes the corresponding trades. The NODE-PRIORITY subroutine adds a nonleaf node to the priority queue, arranged by quality estimates. The algorithm also uses four other subroutines: FILL-PRICE (Figure 2.3), LEAF-PRIORITY (Figure 5.3), TRADE (Figure 5.3), and BEST-ITEM (Figure 5.4).

Chapter 6

Performance

We describe experiments with artificial market data, and then show the performance for two real-world markets. We compare the efficiency of the best-first search algorithm with that of depth-first search.

6.1 Artificial markets

We give results of artificial tests with five control variables. We measure the matcher's efficiency for different numbers of market attributes and values per attribute, as well as the dependency of the performance on the number of orders.

6.1.1 Control variables

We have implemented an experimental setup that allows control over five features of artificial markets: search strategy, number of attributes in an item description, 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 attributes determines the complexity of traded items. For example, the description of a public stock includes only one attribute, the stock symbol, whereas the example car market has four attributes: model, color, year, and mileage. The description of a used vehicle in a real market includes more attributes, such as transmission type, options package, and number of previous owners. We have considered artificial markets with one, three, and ten attributes.

- *Values:* The implemented system supports integer and real attributes, as well as explicitly listed values (Section 2.1.5). We have considered only integer attributes in the artificial markets, and we have controlled the number of values per attribute. We have varied this number from two to 1,024; that is, the smallest market has two values for each attribute, and the largest market includes 1,024 values per attribute.

- *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 4.5). 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. For example, if each buy order matches 1% of sell orders, then the matching density is 0.01. We have experimented with four density values: 0.001, 0.01, 0.1, and 1.

6.1.2 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 the matching density. For each combination of settings, we have run two independent experiments; for each experiment, we have measured the time of processing new orders and matching old orders, average time between placing an order and getting a response, and maximal throughput of the system.

- *Processing time:* The first measurement is the time of processing new orders, which is the first part of the system's main loop (Figure 4.6a). 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:* The second measurement is the time of matching old orders, which is the second part of the main loop (Figure 4.6a). The total of processing and matching time is the overall length of the main loop, which determines the system's speed.

- *Response time:* We have recorded the average time between placing an order and getting the system's response. If the system immediately finds a match for the new order, then its response is the corresponding fill; else, it responds with a confirmation message. The response time has varied from a few seconds to more than twenty minutes. This delayed response is often unacceptable in financial markets, but it would not cause problems in consumer markets, such as used cars.

- *Maximal throughput:* Finally, we have determined the maximal acceptable frequency of placing new orders. If the system gets more orders per second, then the number of unprocessed orders in the input queue keeps growing and the matcher eventually has to reject some of them.

6.1.3 Summary graphs

In Figures 6.1–6.12, we show the dependency of the system's performance on each of the control variables. We use three performance measurements: (1) the time of one pass through the system's main loop, which includes processing of new orders and matching of old orders; (2) the mean time between placing a new order and getting a response; and (3) the system's throughput.

For each control variable, we consider three settings of the other four 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.

In Figures 6.1–6.3, we show how the performance depends on the number of orders. The main-loop and response times are linear in the number of orders. The throughput in small markets grows with the number of orders, reaching an upper limit when the market grows to about two hundred orders, and slightly decreasing with further increase in the number of orders.

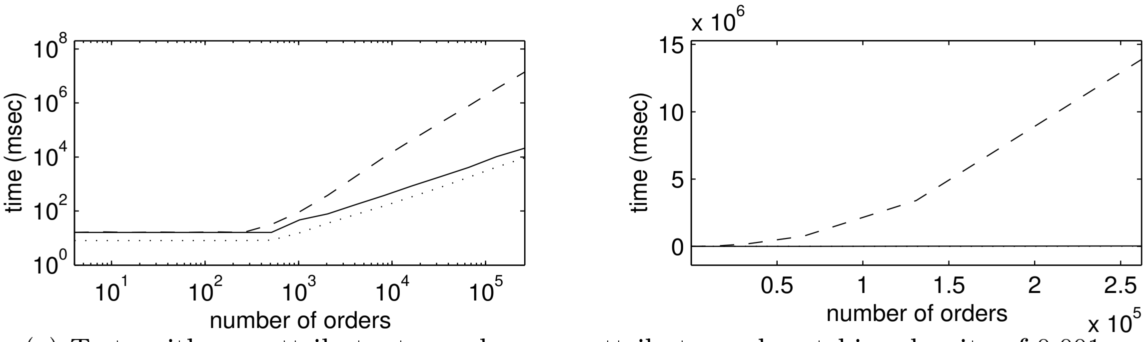
In Figures 6.4–6.6, we give the dependency of the performance on the number of attributes. The main-loop and response times are super-linear in the number of attributes, whereas throughput is in inverse proportion to the same super-linear function.

In Figures 6.7–6.9, we show that the main-loop and response times grow sub-linearly with the number of values per attribute, and throughput slightly decreases with an increase in the number of values.

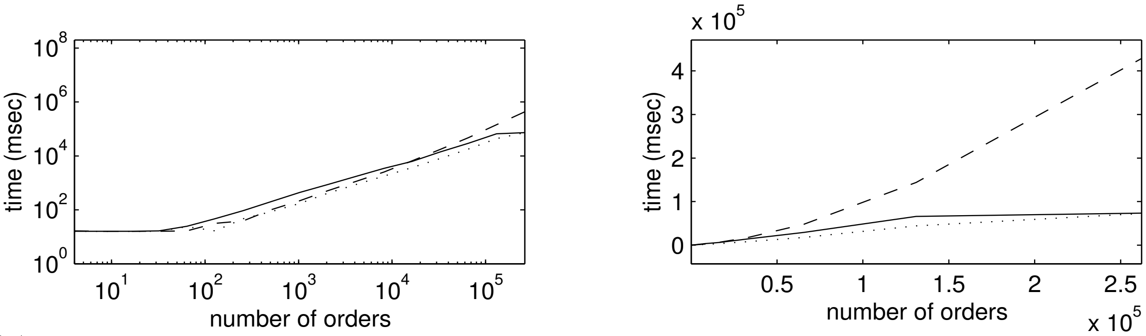
Finally, in Figures 6.10–6.12, we show how the system's behavior changes with the matching density. We have not found any monotonic dependency between the density and the performance; the increase of the matching density sometimes leads to faster matching and sometimes slows down the system.

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.

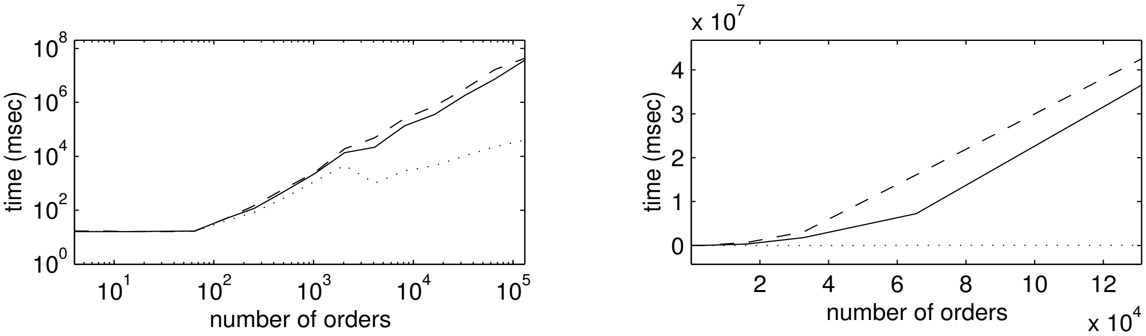
The speed of the best-first search is usually close to that of the depth-first search with a limit on the number of matches; that is, the optimal-matching system is as fast as the old system that could not find optimal matches. A notable exception is the performance in ten-attribute markets with large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

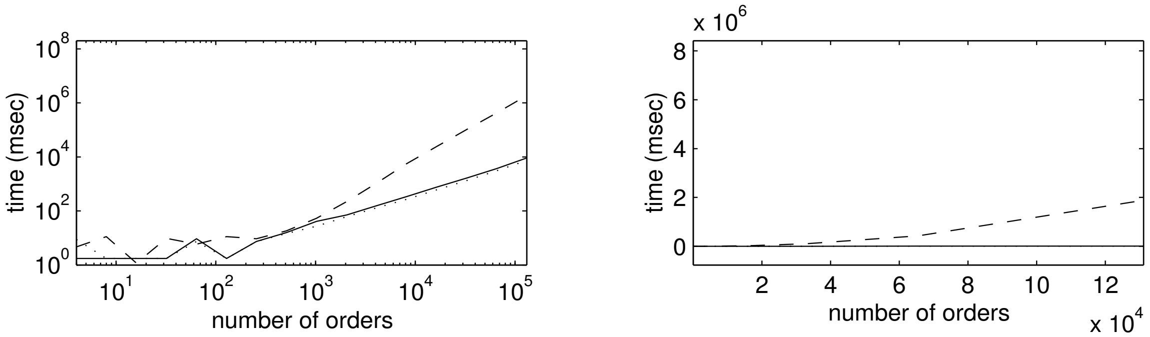


(b) Tests with three attributes, sixteen values per attribute, and matching density of 0.01.

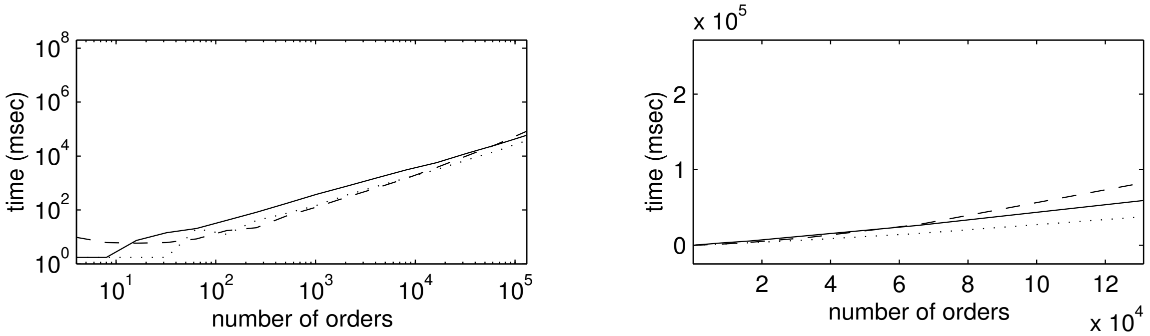


(c) Tests with ten attributes, 1,024 values per attribute, and matching density of 1.

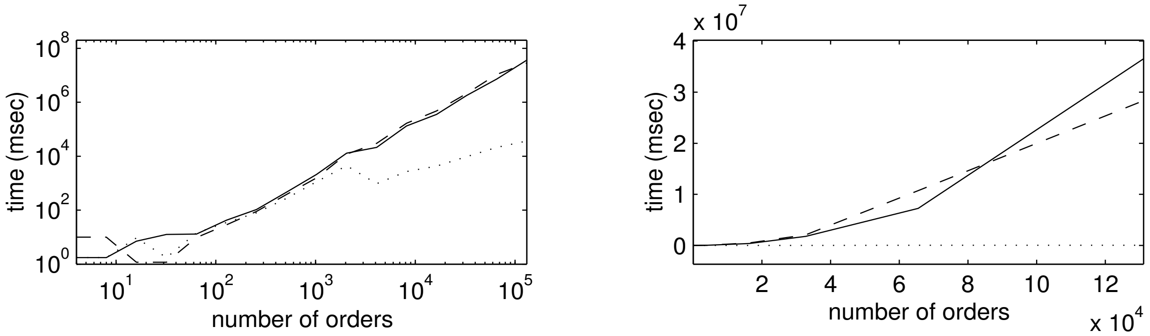
Figure 6.1: Dependency of the total matching time on the number of orders. We show the performance of the best-first search (solid lines), depth-first search that identifies all matches (dashed lines), and depth-first search with a limit on the number of matches (dotted lines). The graphs on the left 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.

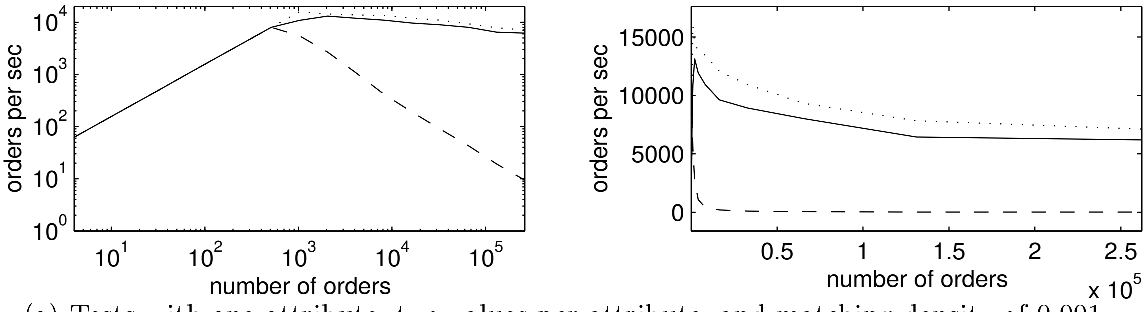


(b) Tests with three attributes, sixteen values per attribute, and matching density of 0.01.

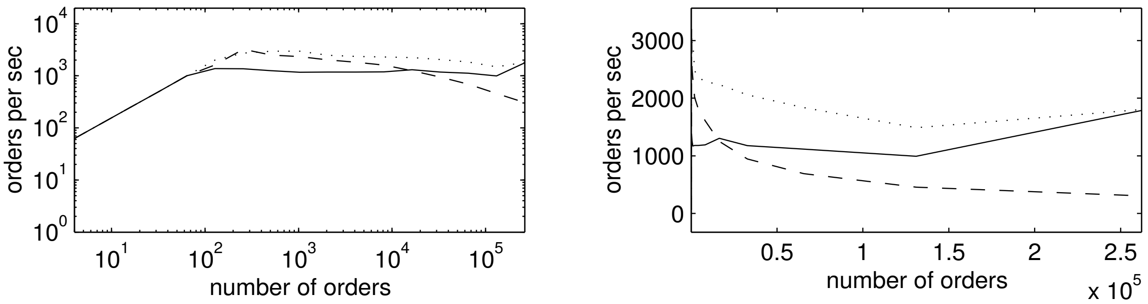


(c) Tests with ten attributes, 1,024 values per attribute, and matching density of 1.

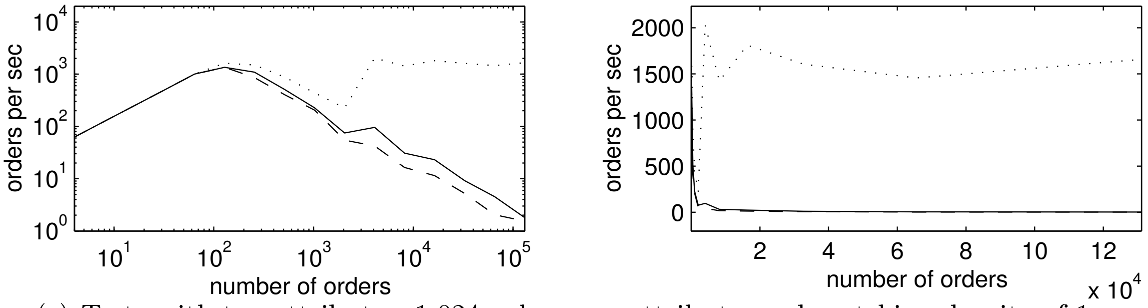
Figure 6.2: Dependency of the response time on the number of orders. The legend is the same as in Figure 6.1.



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

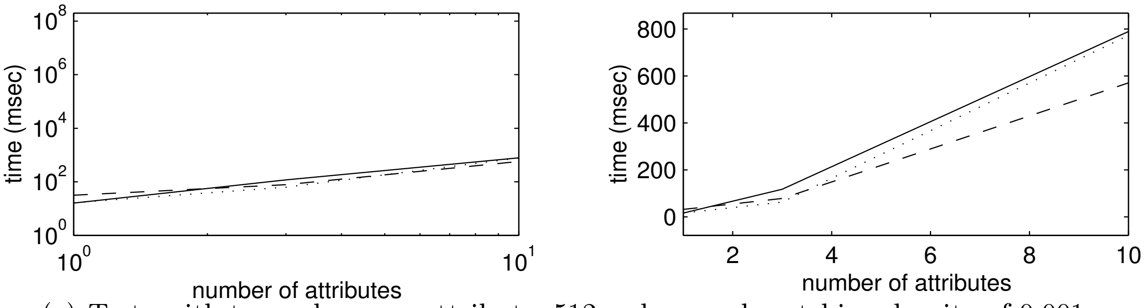


(b) Tests with three attributes, sixteen values per attribute, and matching density of 0.01.

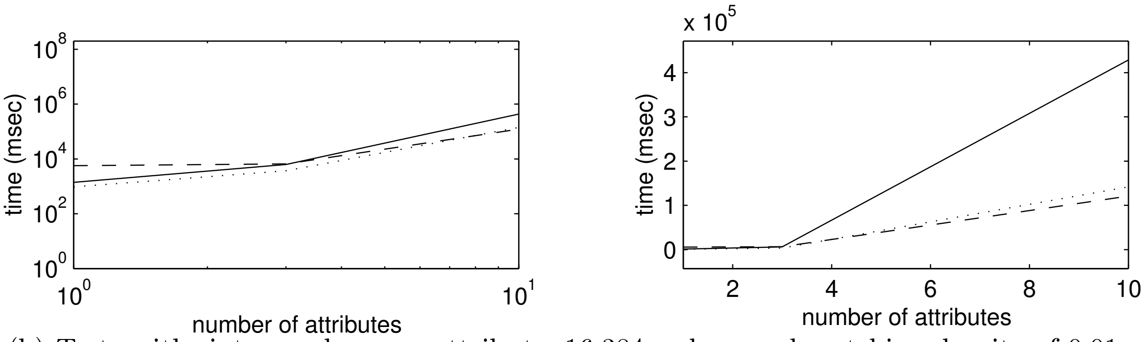


(c) Tests with ten attributes, 1,024 values per attribute, and matching density of 1.

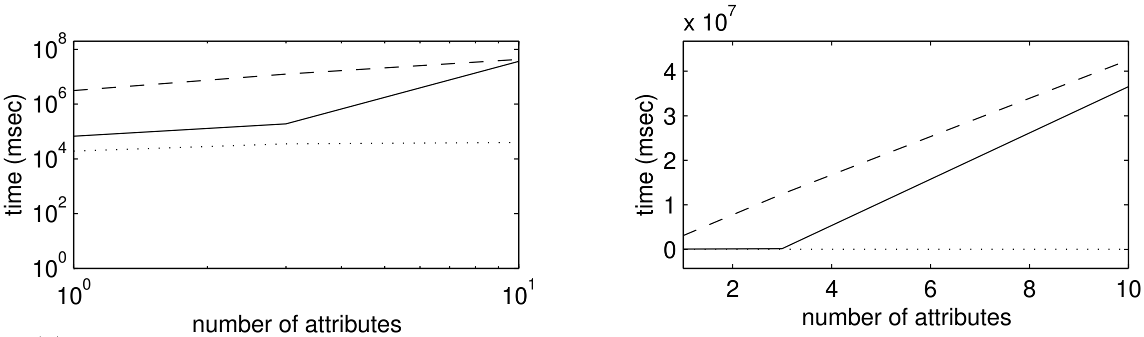
Figure 6.3: Dependency of the throughput on the number of orders. The legend is the same as in Figure 6.1.



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

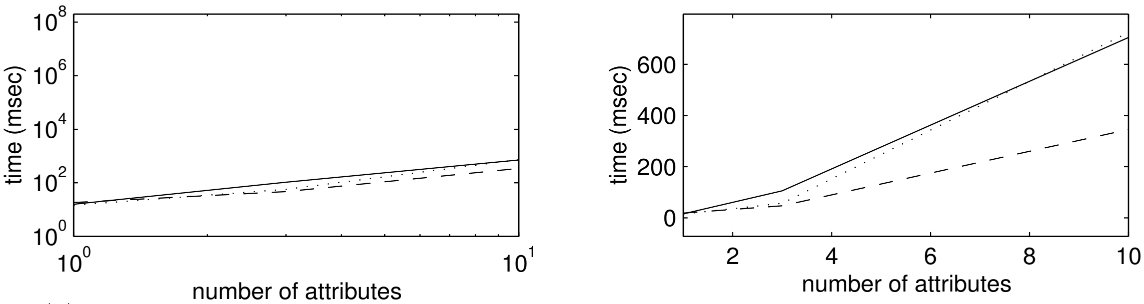


(b) Tests with sixteen values per attribute, 16,384 orders, and matching density of 0.01.

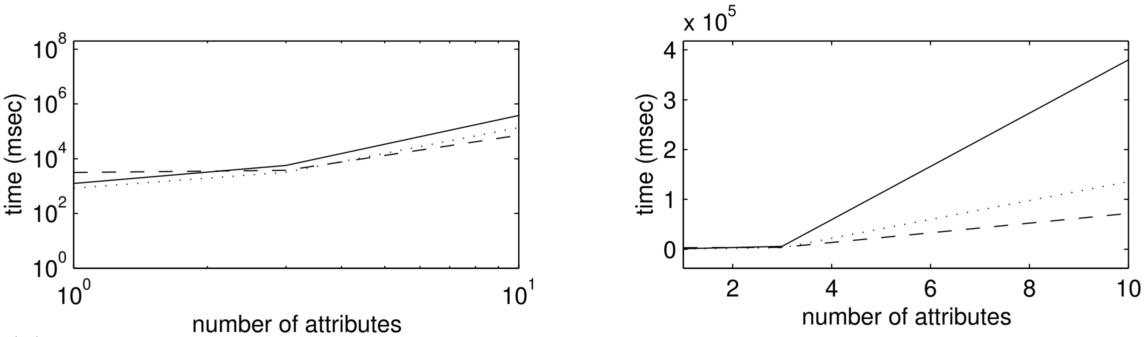


(c) Tests with 1,024 values per attribute, 131,072 orders, and matching density of 1.

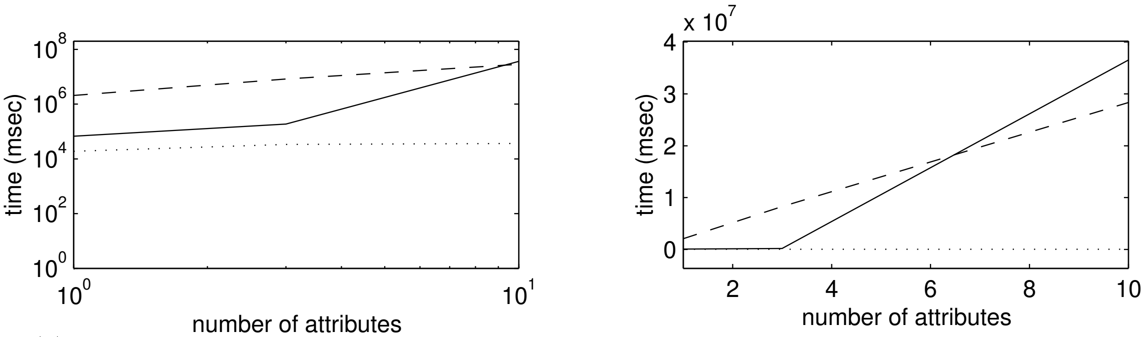
Figure 6.4: Dependency of the total matching time on the number of attributes. The legend is the same as in Figure 6.1.



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

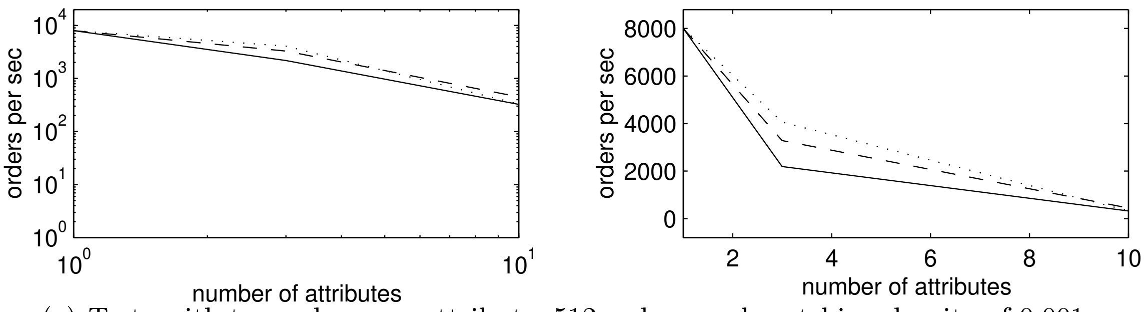


(b) Tests with sixteen values per attribute, 16,384 orders, and matching density of 0.01.

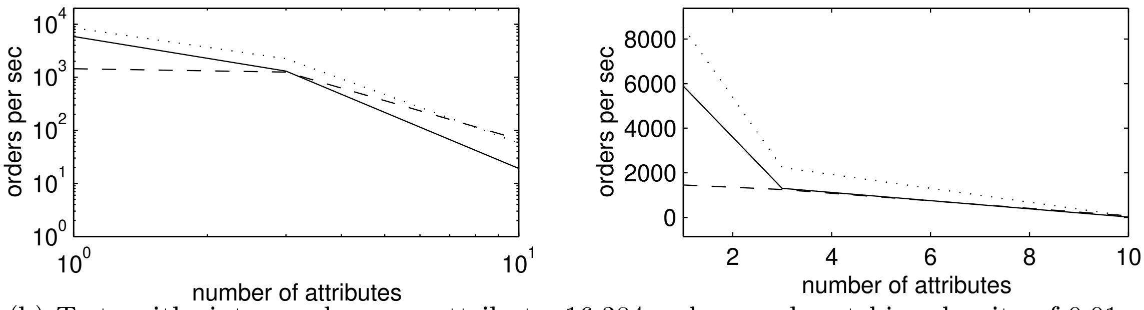


(c) Tests with 1,024 values per attribute, 131,072 orders, and matching density of 1.

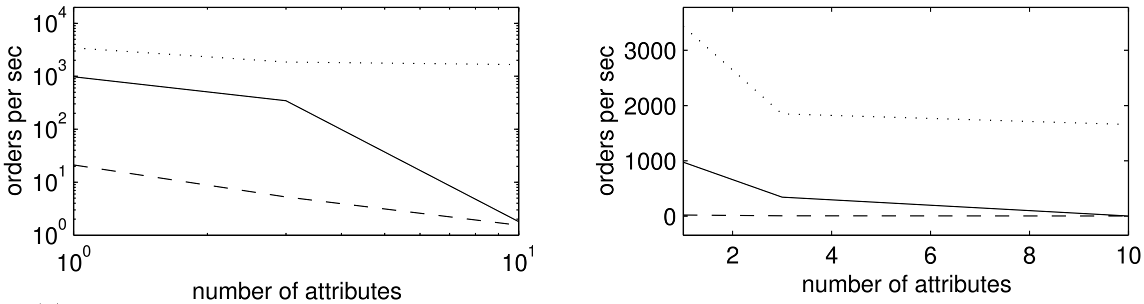
Figure 6.5: Dependency of the response time on the number of attributes. The legend is the same as in Figure 6.1.



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

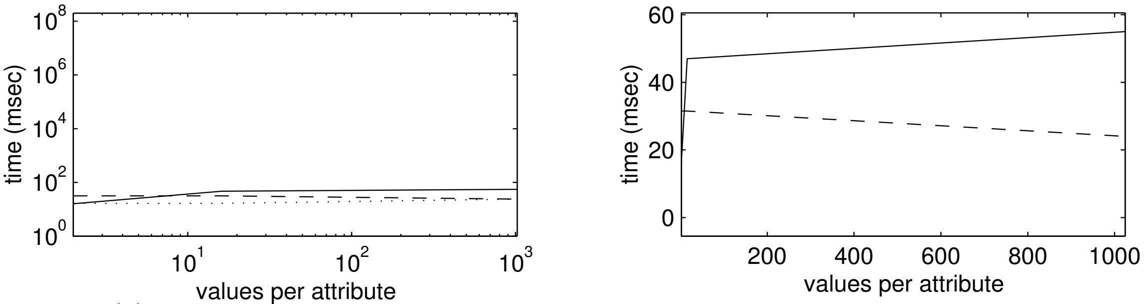


(b) Tests with sixteen values per attribute, 16,384 orders, and matching density of 0.01.

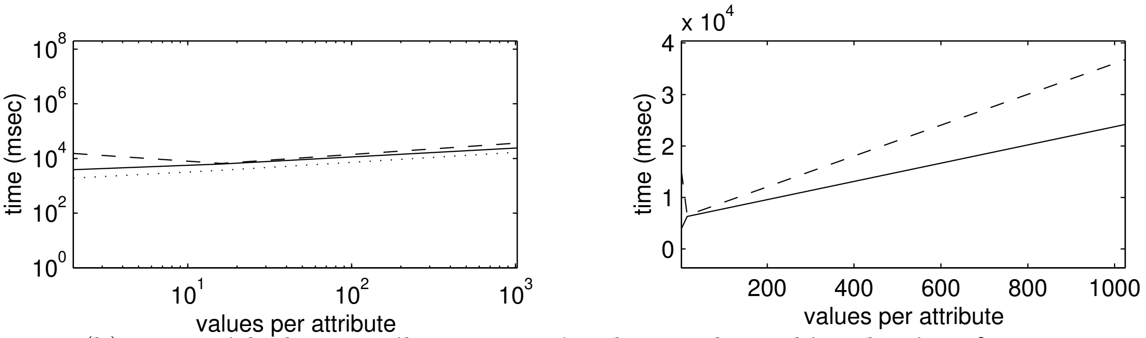


(c) Tests with 1,024 values per attribute, 131,072 orders, and matching density of 1.

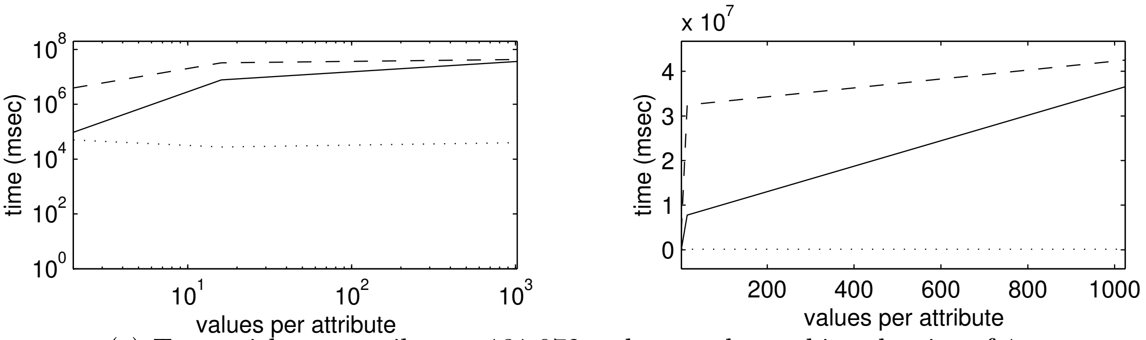
Figure 6.6: Dependency of the throughput on the number of attributes. The legend is the same as in Figure 6.1.



(a) Tests with one attribute, 512 orders, and matching density of 0.001.

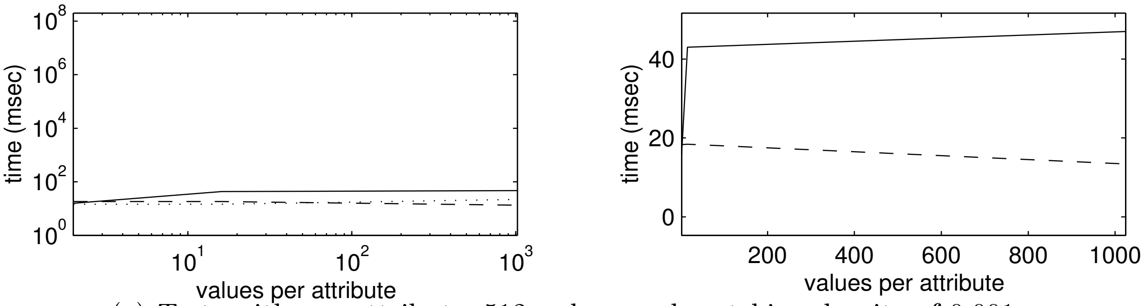


(b) Tests with three attributes, 16,384 orders, and matching density of 0.01.

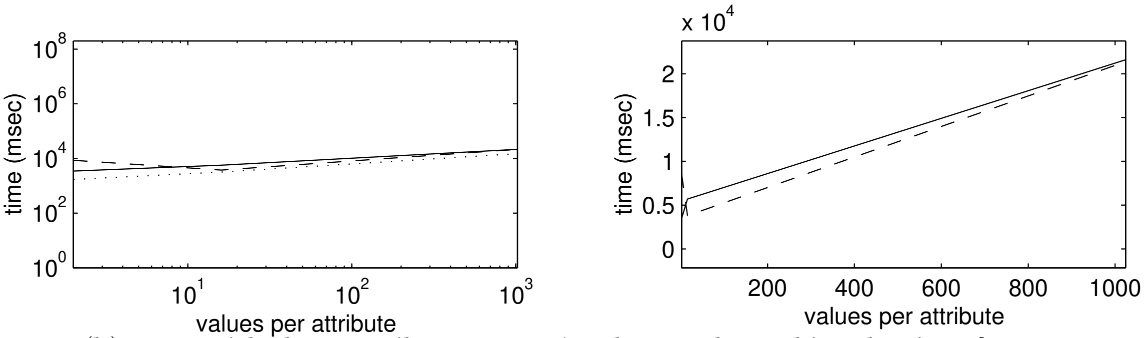


(c) Tests with ten attributes, 131,072 orders, and matching density of 1.

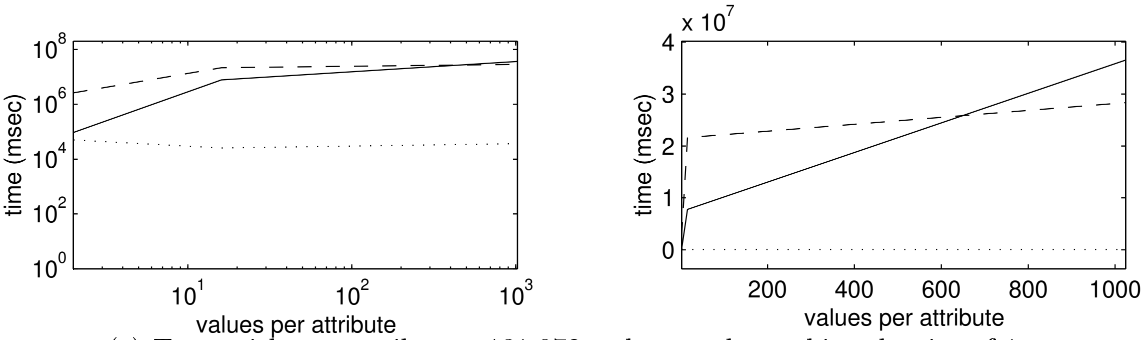
Figure 6.7: Dependency of the total matching time on the number of values per attribute. The legend is the same as in Figure 6.1.



(a) Tests with one attribute, 512 orders, and matching density of 0.001.

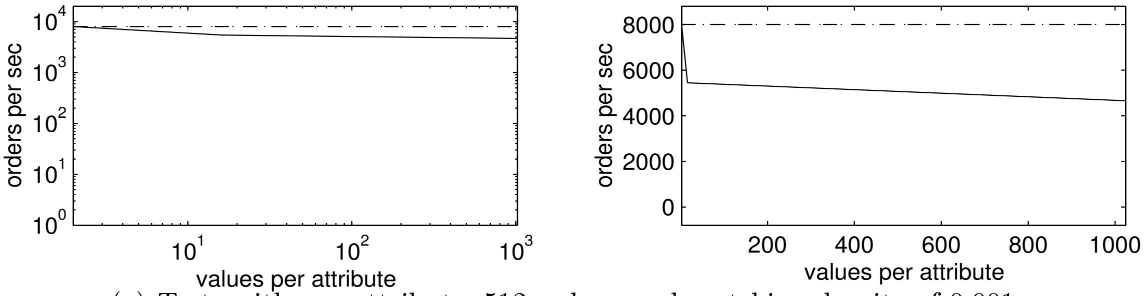


(b) Tests with three attributes, 16,384 orders, and matching density of 0.01.

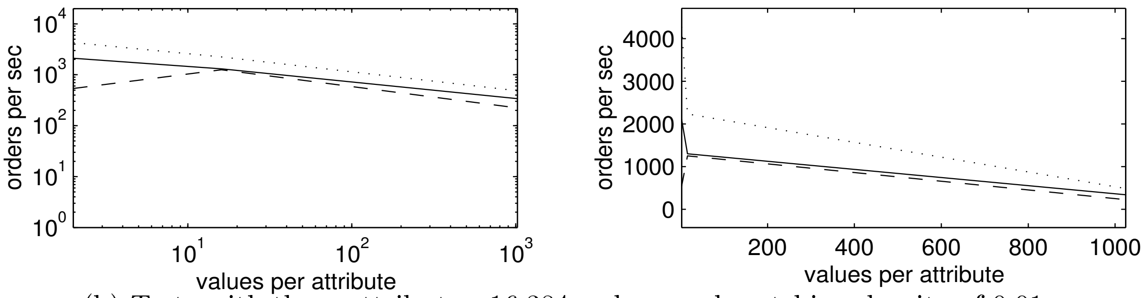


(c) Tests with ten attributes, 131,072 orders, and matching density of 1.

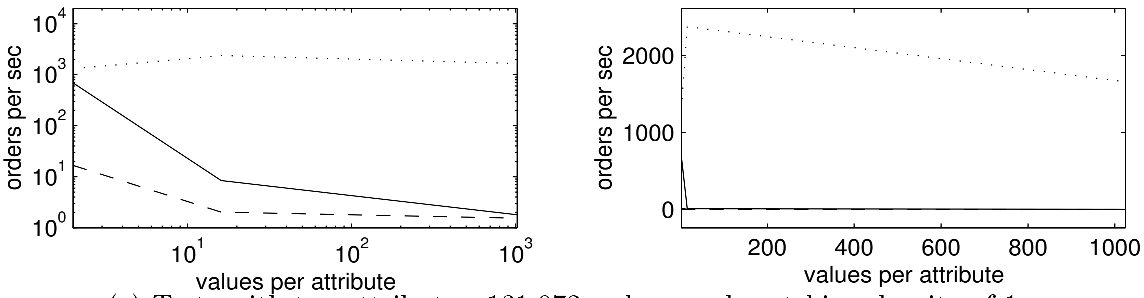
Figure 6.8: Dependency of the response time on the number of values per attribute. The legend is the same as in Figure 6.1.



(a) Tests with one attribute, 512 orders, and matching density of 0.001.

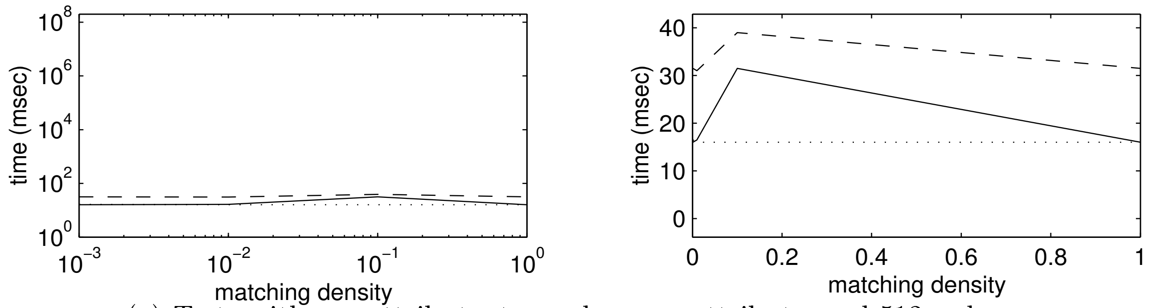


(b) Tests with three attributes, 16,384 orders, and matching density of 0.01.

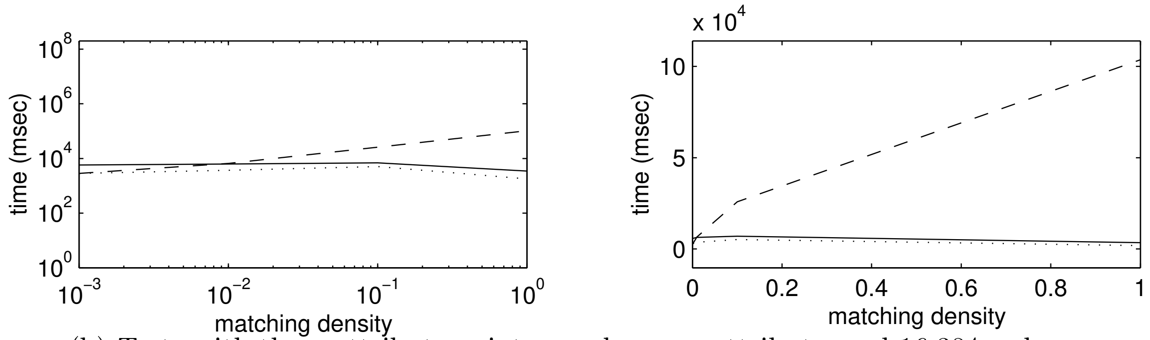


(c) Tests with ten attributes, 131,072 orders, and matching density of 1.

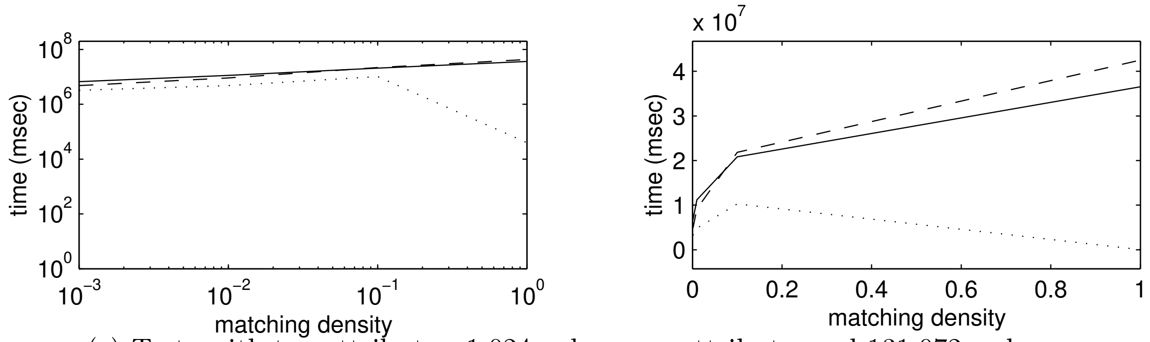
Figure 6.9: Dependency of the throughput on the number of values per attribute. The legend is the same as in Figure 6.1.



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

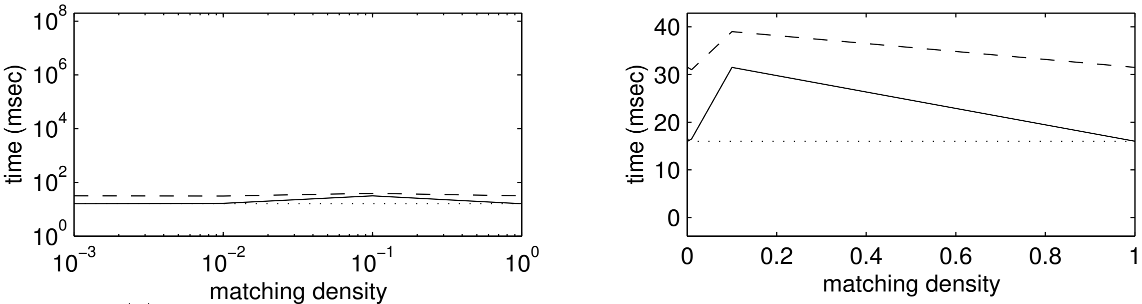


(b) Tests with three attributes, sixteen values per attribute, and 16,384 orders.

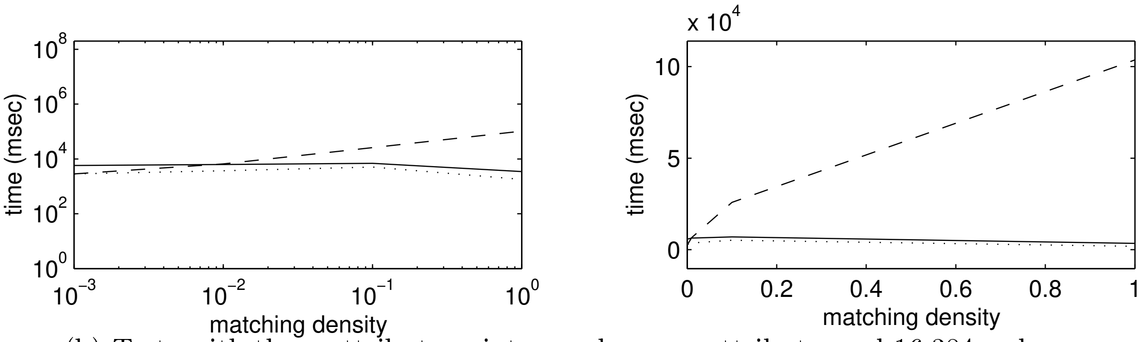


(c) Tests with ten attributes, 1,024 values per attribute, and 131,072 orders.

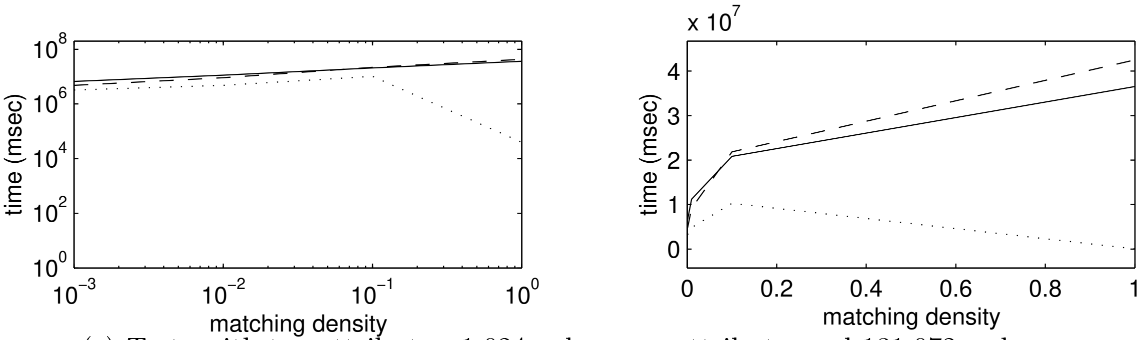
Figure 6.10: Dependency of the total matching time on the matching density. The legend is the same as in Figure 6.1.



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

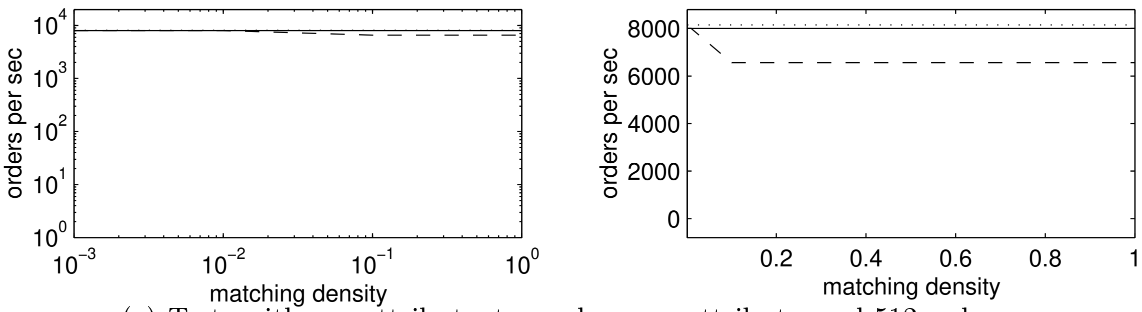


(b) Tests with three attributes, sixteen values per attribute, and 16,384 orders.

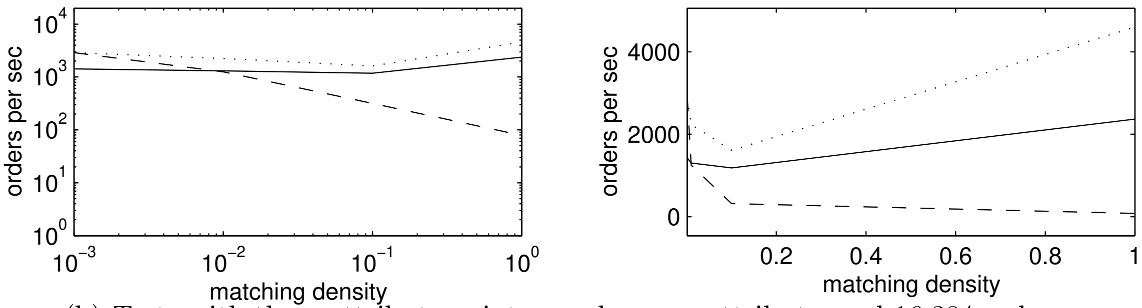


(c) Tests with ten attributes, 1,024 values per attribute, and 131,072 orders.

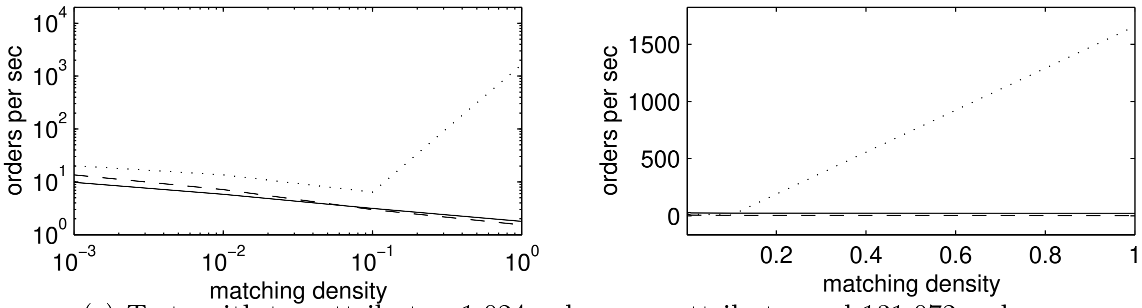
Figure 6.11: Dependency of the response time on the matching density. The legend is the same as in Figure 6.1.



(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 6.12: Dependency of the throughput on the matching density. The legend is the same as in Figure 6.1.

6.2 Real markets

The results of experiments with real-world markets have been similar to those of artificial tests. First, we have applied the system to a used-car market with eight attributes, which is an extended version of the example market. Second, we have experimented with a financial market, which involves trading of commercial paper.

6.2.1 Used cars

We have tested the system on a car market with eight attributes: transmission, number of doors, interior color, exterior color, year, model, options, and mileage (Figures 6.13 and 6.14). The market includes all models offered by AutoNation (www.autonation.com); it comprises seven interior colors, fifty-two exterior colors, 257 models, and 1,024 option packages. We have also defined three standard sets of interior colors, three sets of exterior colors, and forty-three sets of models.

We have run experiments with up to 262,144 orders; the control variables have included the search strategy, number of orders, and the matching density. We have considered the three search strategies, seventeen different settings for the number of orders, and four settings for matching density. For each combination of settings, we have run three experiments, and we show the results in Figures 6.15–6.22.

Observe that the system readily scales to 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 10.1, with mean at 2.5. 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 one hundred times slower than the limited depth-first search.

Attribute 1: Transmission (2 values)
Manual, automatic.

Attribute 2: Number of doors (3 values)
Two, three, four.

Attribute 3: Interior color (7 values)
Black, gray, white, tan, brown, blue, red.

Standard sets
Dark (2 values)
Medium (3 values)
Light (2 values)

Attribute 4: Exterior color (52 values)

Amazon Green	Dark Green Satin	Light Gray
Arizona Beige	Dark Toreador Red	Light Parchment Gold
Atlantic Blue	Deep Emerald Green	Light Sapphire Blue
Autumn Orange	Deep Jewel Green	Malibu Blue
Autumn Red	Deep Wedgewood Blue	Mandarin Gold
Black	Ebony	Medium Brown
Bright Amber	Electric Green	Medium Charcoal Blue
Bright Atlantic Blue	Estate Green	Medium Charcoal Green
Bright Red	Fort Knox Gold	Medium Gray
Bright Silver	Graphite Blue	Medium Royal Blue
Cabernet Red	Harvest Gold	Medium Steel Blue
Charcoal Green	Infra Red	Medium Titanium
Chesapeake Blue	Island Blue	Medium Wedgewood Blue
Chestnut	Ivory Parchment	Midnight Gray
Chrome Yellow	Jewel Green	Performance Red
Cloud White	Laser Red Tinted	Silver
Crystal White	Light Blue	
Dark Blue	Light Brass	

Standard sets
Dark (18 values)
Medium (20 values)
Light (14 values)

Figure 6.13: Attributes 1–4 of the used-car market (also see Figure 6.14).

Attribute 5: Year (106 values)
Integer values from 1896 to 2001.

Attribute 6: Model (257 values)
All models offered by AutoNation (www.autonation.com).

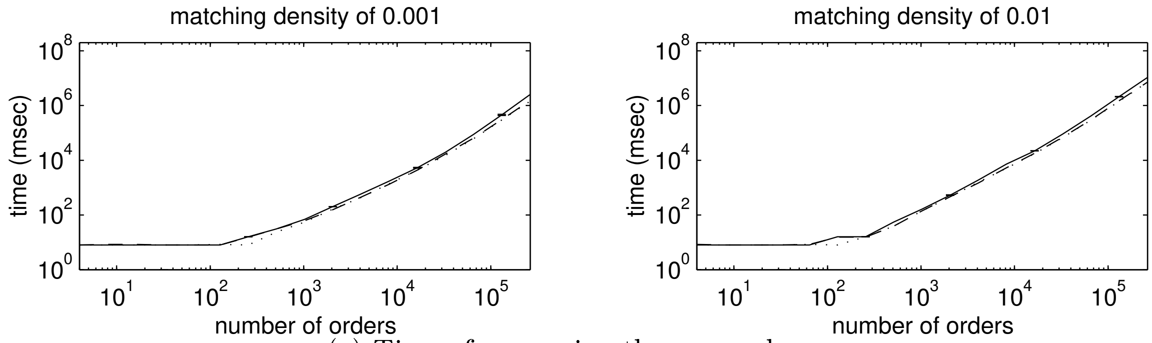
Standard sets

American (138 values)	Chevrolet (26 values)	Mazda (11 values)
European (48 values)	Chrysler (9 values)	Mercedes Benz (9 values)
Japanese (71 values)	Dodge (12 values)	Mercury (5 values)
	Ford (19 values)	Mitsubishi (6 values)
Sedans (139 values)	GMC (10 values)	Nissan (8 values)
Sports Cars (28 values)	Honda (8 values)	Oldsmobile (5 values)
SUVs (57 values)	Hyundai (6 values)	Plymouth (5 values)
Trucks (23 values)	Infiniti (4 values)	Pontiac (8 values)
Vans (10 values)	Isuzu (4 values)	Porsche (2 values)
	Jaguar (4 values)	Saturn (6 values)
Acura (6 values)	Jeep (4 values)	Subaru (3 values)
Audi (6 values)	Kia (5 values)	Suzuki (4 values)
BMW (7 values)	Land Rover (2 values)	Toyota (16 values)
Buick (5 values)	Lexus (8 values)	Volkswagen (7 values)
Cadillac (5 values)	Lincoln (5 values)	Volvo (7 values)

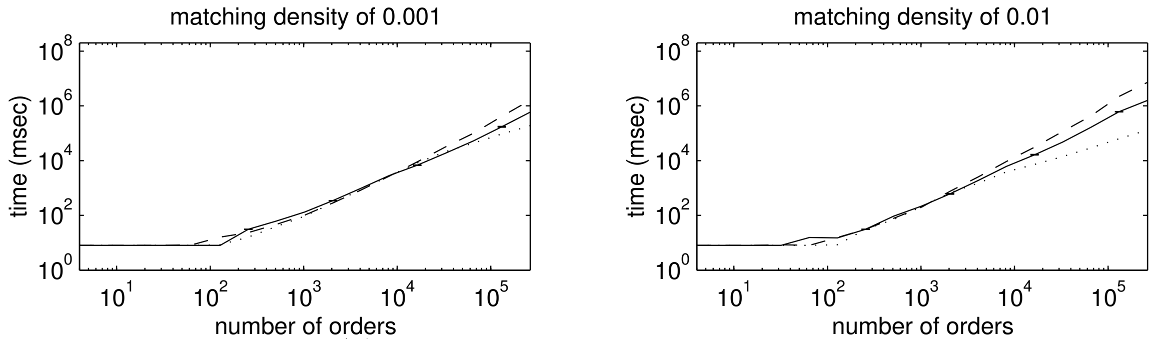
Attribute 7: Option package (1,024 values)
Standard combinations of options offered by AutoNation.

Attribute 8: Mileage (500,001 values)
Integer values from 0 to 500,000.

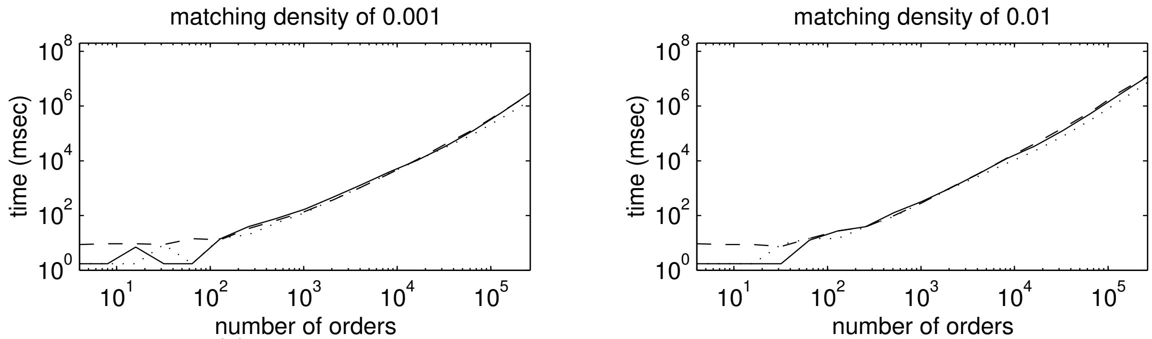
Figure 6.14: Attributes 5–8 of the used-car market (also see Figure 6.13).



(a) Time of processing the new orders.

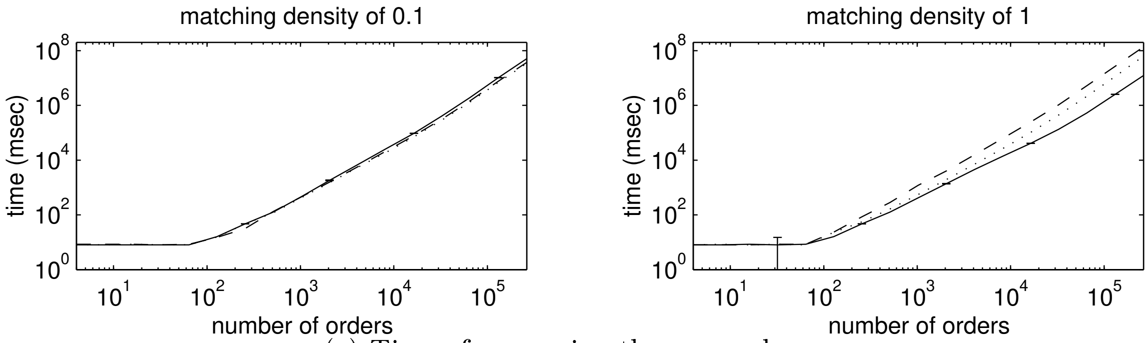


(b) Time of matching the old orders.

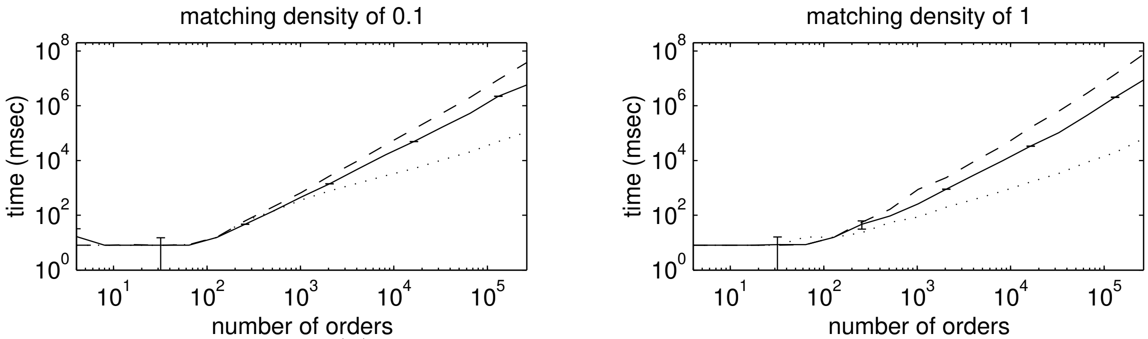


(c) Time between order placement and system response.

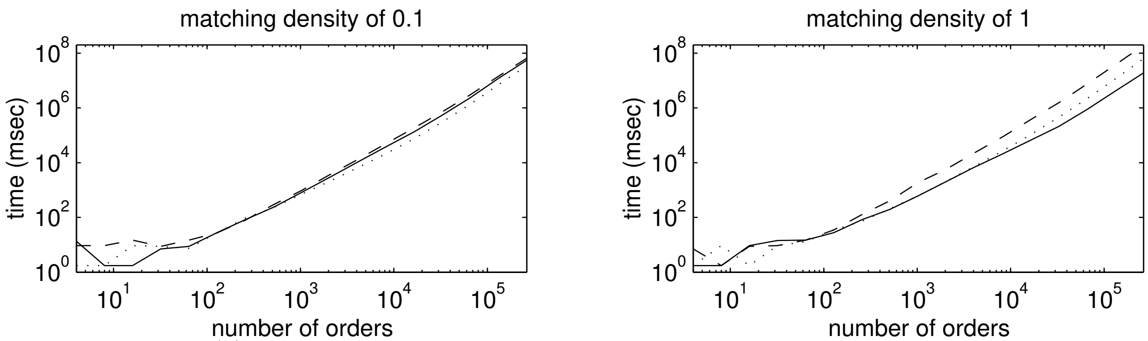
Figure 6.15: Performance in the car market for matching density of 0.001 (left) and 0.01 (right). We show the performance of the best-first search (solid lines), depth-first search that identifies all matches (dashed lines), and depth-first search with a limit on the number of matches (dotted lines).



(a) Time of processing the new orders.

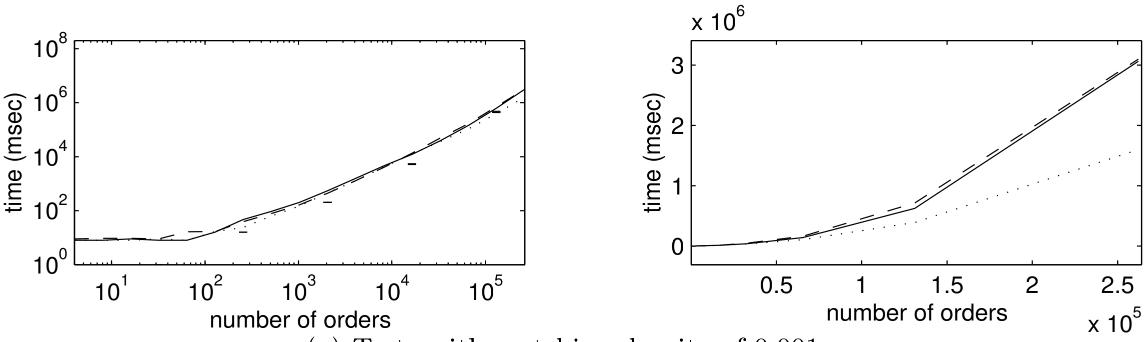


(b) Time of matching the old orders.

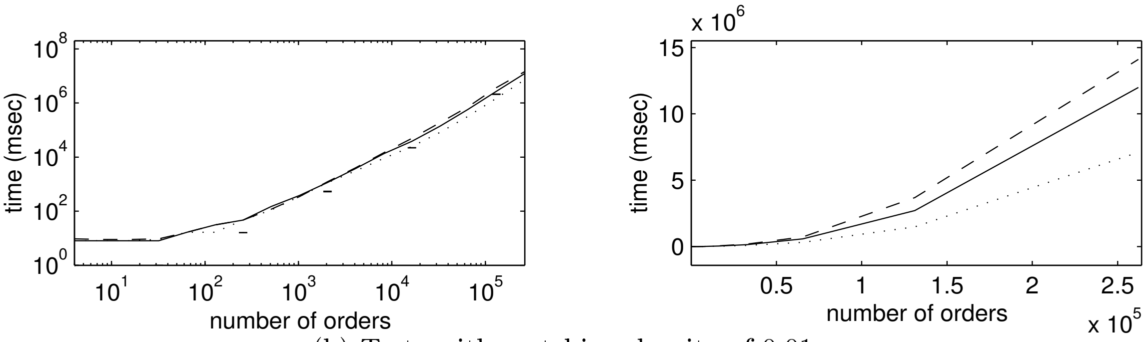


(c) Time between order placement and system response.

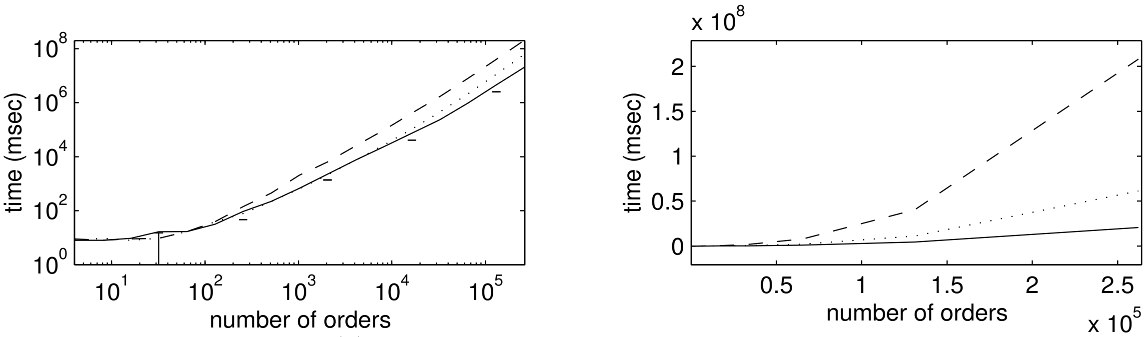
Figure 6.16: Performance in the car market for matching density of 0.1 (left) and 1 (right). The legend is the same as in Figure 6.15.



(a) Tests with matching density of 0.001.

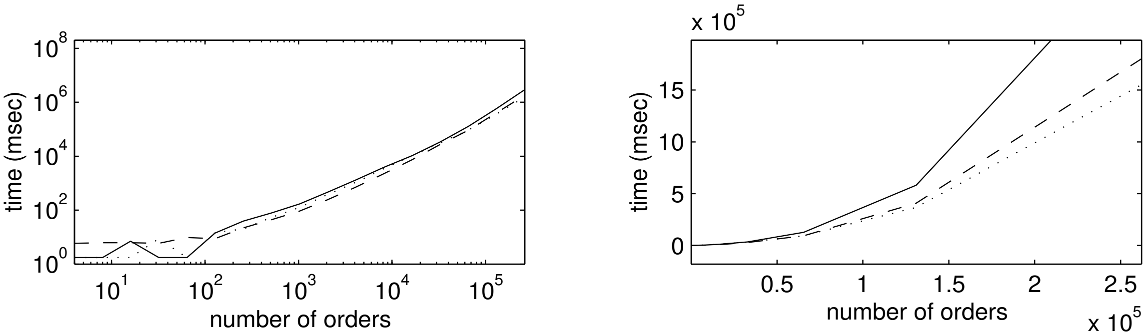


(b) Tests with matching density of 0.01.

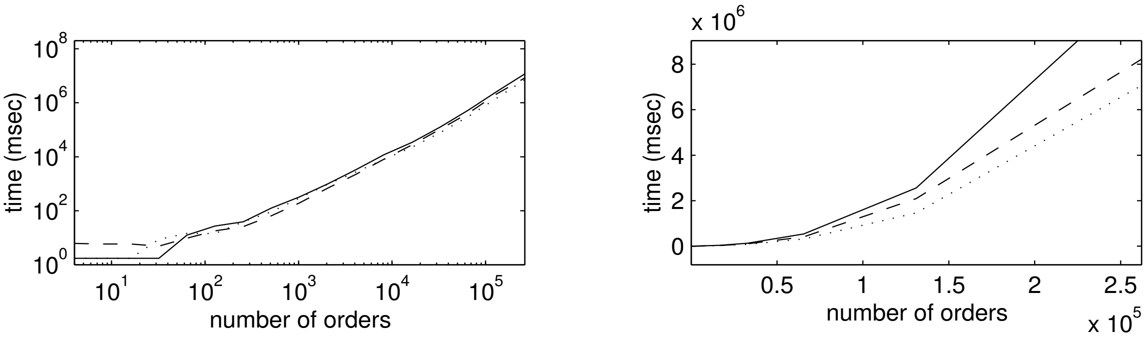


(c) Tests with matching density of 1.

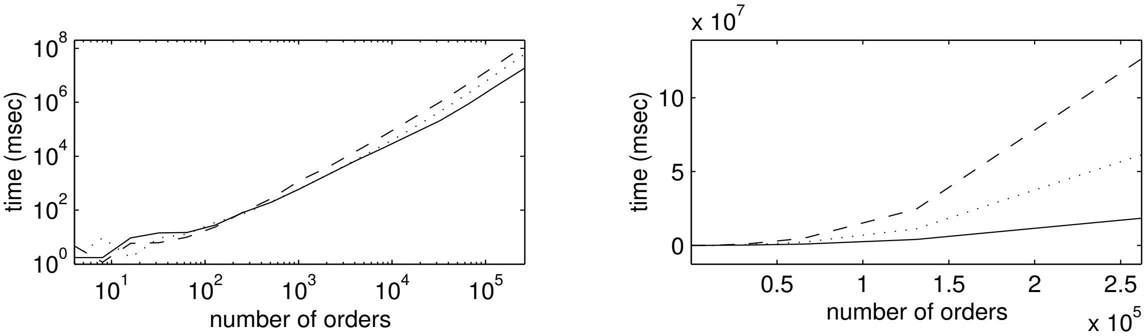
Figure 6.17: Dependency of the total matching time on the number of orders. We give the results for the best-first search (solid lines), depth-first search that identifies all matches (dashed lines), and depth-first search with a limit on the number of matches (dotted lines). The graphs on the left are in logarithmic scale, whereas the graphs on the right are in linear scale.



(a) Tests with matching density of 0.001.

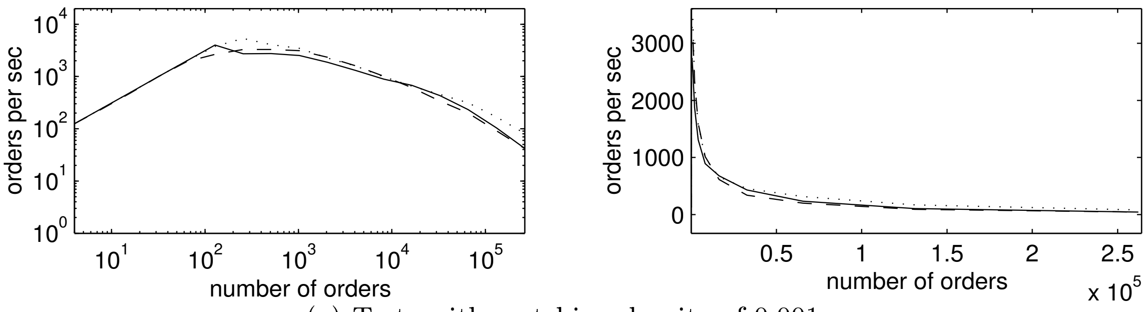


(b) Tests with matching density of 0.01.

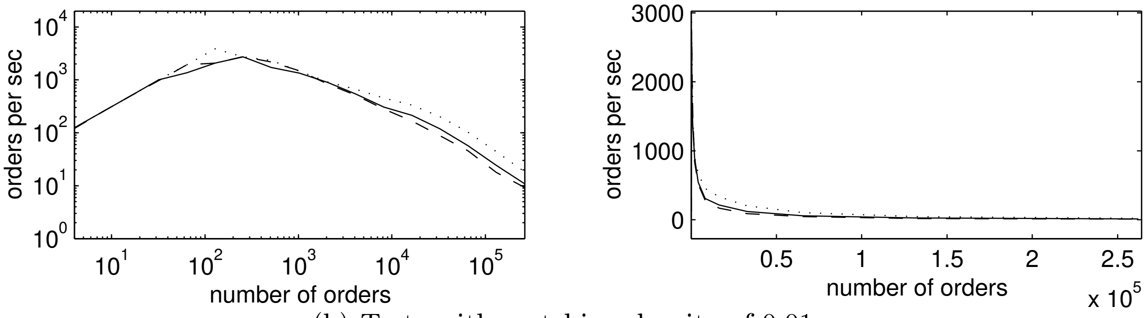


(c) Tests with matching density of 1.

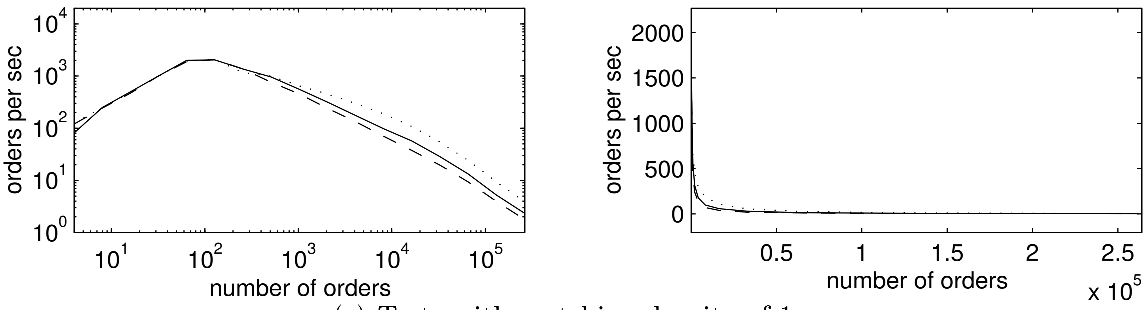
Figure 6.18: Dependency of the response time on the number of orders. The legend is the same as in Figure 6.17.



(a) Tests with matching density of 0.001.

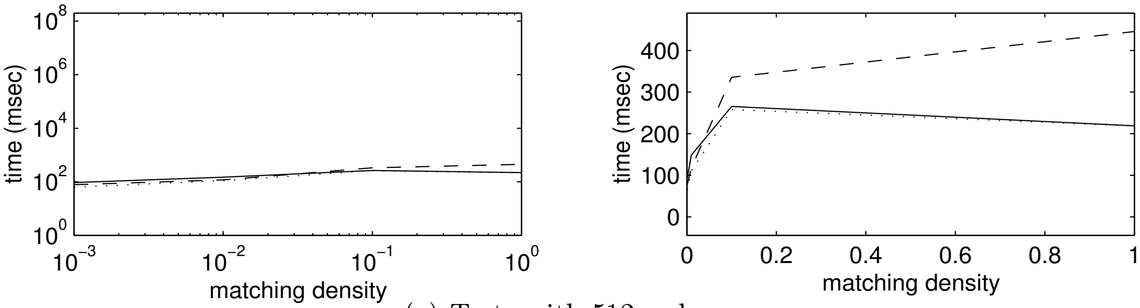


(b) Tests with matching density of 0.01.

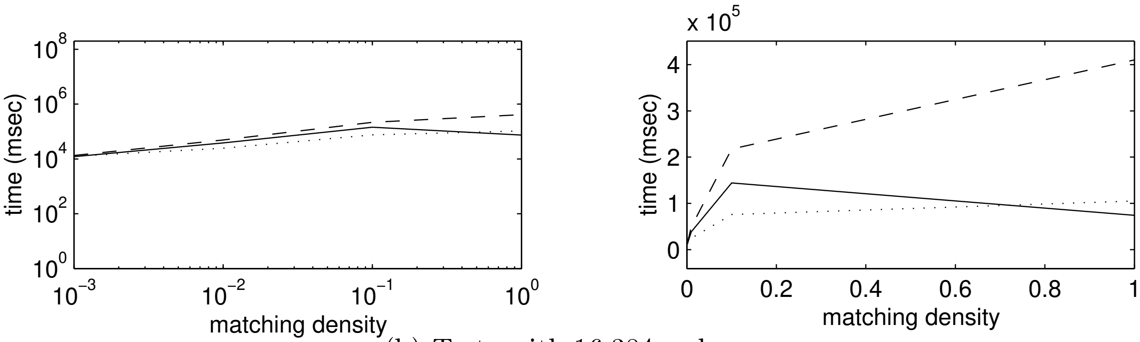


(c) Tests with matching density of 1.

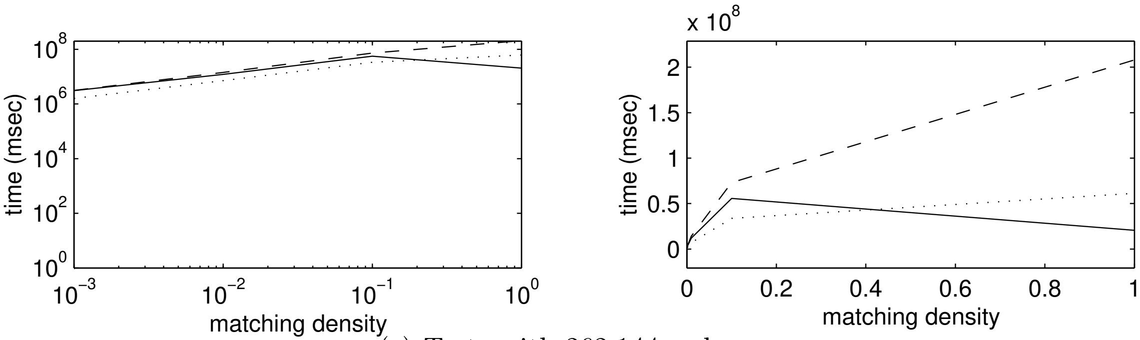
Figure 6.19: Dependency of the throughput on the number of orders. The legend is the same as in Figure 6.17.



(a) Tests with 512 orders.

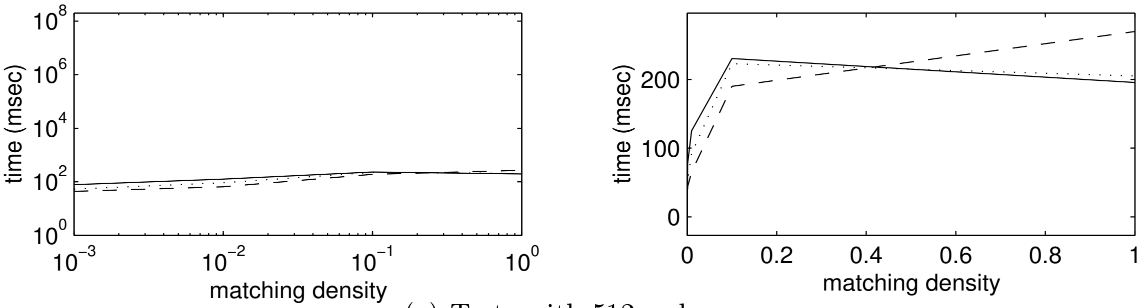


(b) Tests with 16,384 orders.

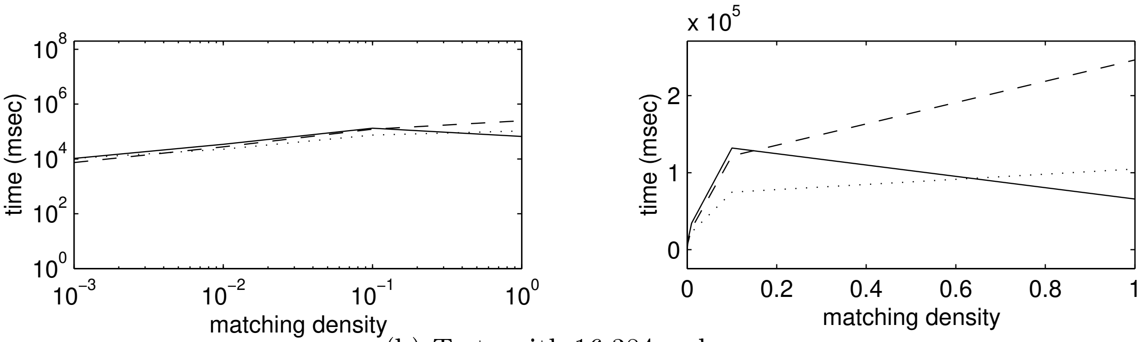


(c) Tests with 262,144 orders.

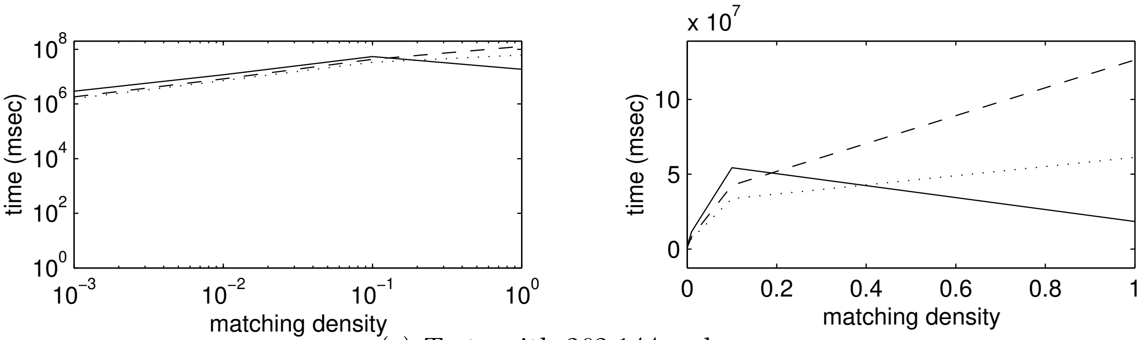
Figure 6.20: Dependency of the total matching time on the matching density. The legend is the same as in Figure 6.17.



(a) Tests with 512 orders.

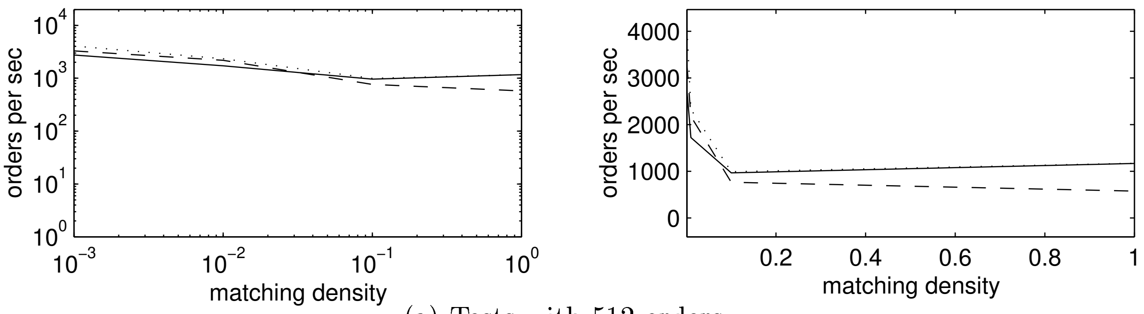


(b) Tests with 16,384 orders.

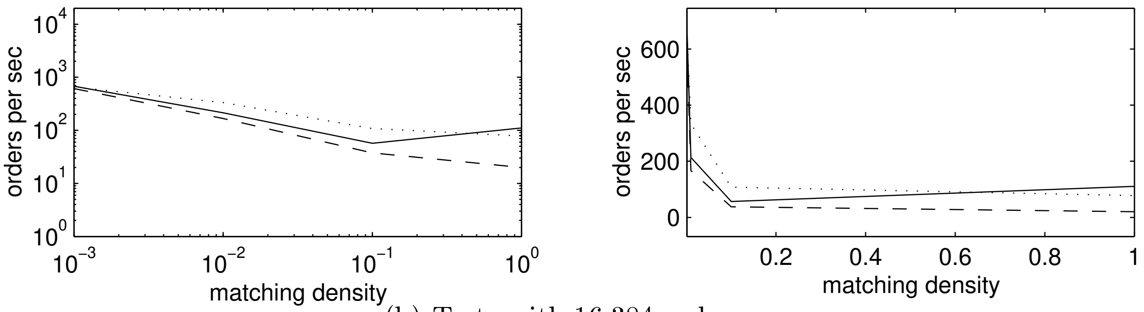


(c) Tests with 262,144 orders.

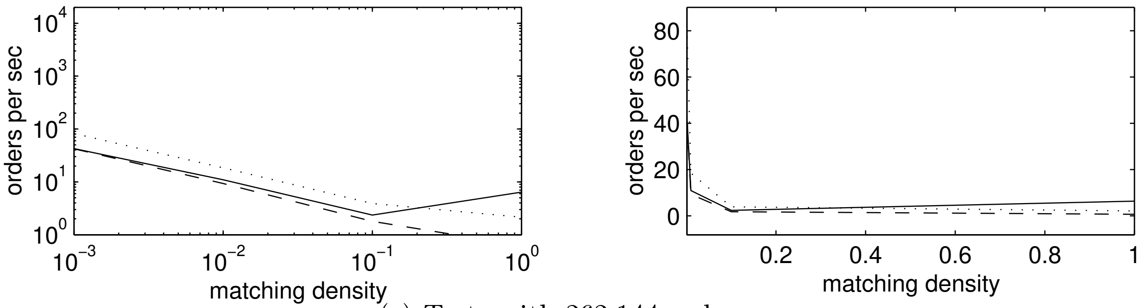
Figure 6.21: Dependency of the response time on the matching density. The legend is the same as in Figure 6.17.



(a) Tests with 512 orders.



(b) Tests with 16,384 orders.



(c) Tests with 262,144 orders.

Figure 6.22: Dependency of the throughput on the matching density. The legend is the same as in Figure 6.17.

6.2.2 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 along 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.

Investors usually describe the income from commercial paper in terms of the annual interest rate. For example, suppose that a company has issued a seven-day commercial paper, in the amount of \$10,000,000, with annual interest 10%. Then, the daily interest is $10\%/365 = 0.0274\%$, and the total amount of interest is about $\$10{,}000{,}000 \cdot 0.0274\% \cdot 7 = \$19{,}180$. The exact amount is slightly higher, \$19,197, because of compound interest.

The appropriate interest depends on the current rate of US Treasury bonds, and on the chance of a company's bankruptcy before the maturity date. The estimated chance of bankruptcy depends on the company's reputation and the paper's time until maturity. The investors expect less reliable companies to pay higher interest; furthermore, they expect that, the more time to maturity, the higher the interest. For example, a risky software provider should pay higher interest than a reliable utilities company, and a nine-month paper should carry higher interest than a one-month paper of the same company.

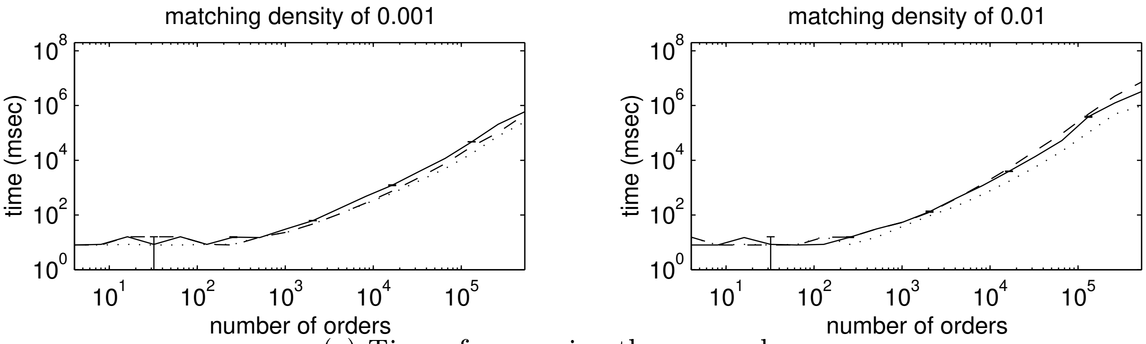
After investors buy a commercial paper, they may resell it on a secondary market, before the maturity date. For example, suppose that a buyer has bought a three-month paper in May, and then decided that she needs money in June. Then, she may resell the paper and keep part of the interest; if the rate 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 a buyer 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:

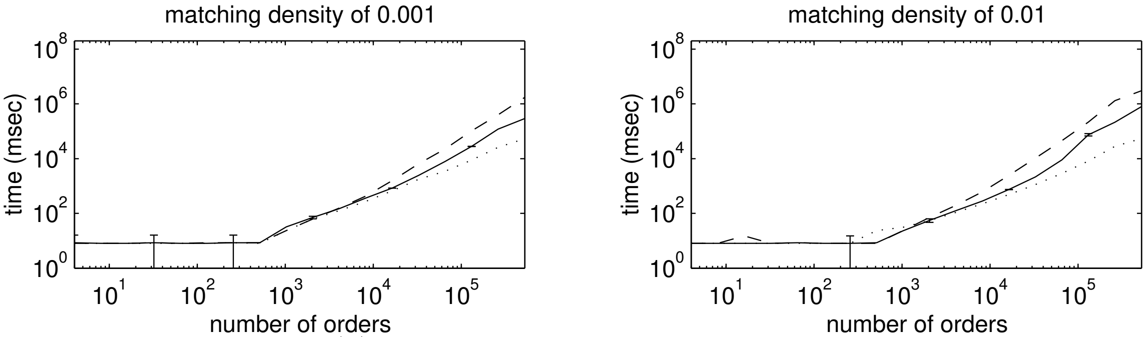
- Company (5000 values): 5000 US companies.
- Maturity date (2550 values): Business days from year 2001 to 2010.

We have run experiments with up to 524,288 orders; we show the results in Figures 6.23 and 6.24, and plot the dependency of the system's performance on the control variables in Figures 6.25–6.30. 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.

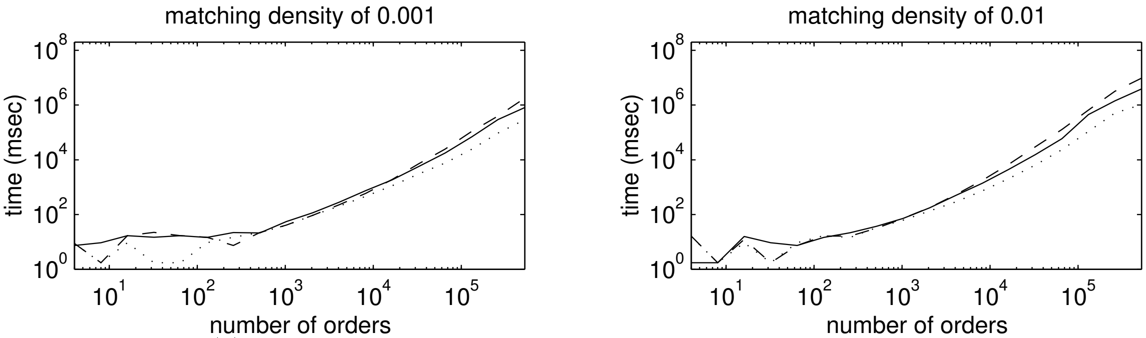
The best-first search system usually processes 100 to 10,000 new orders per second; it outperforms the depth-first search that identifies all matches by a factor of 2.3 to 8.8, with mean at 4.5. On the other hand, it is slower than the limited depth-first search; thus, the search for optimal matches 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.

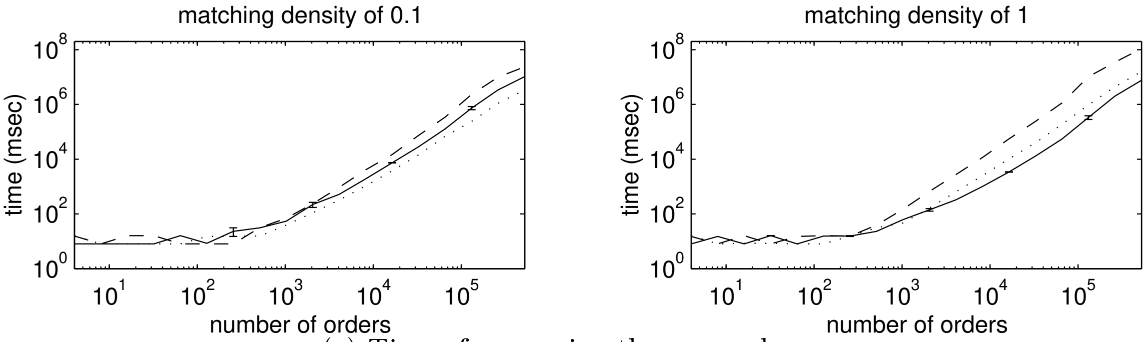


(b) Time of matching the old orders.

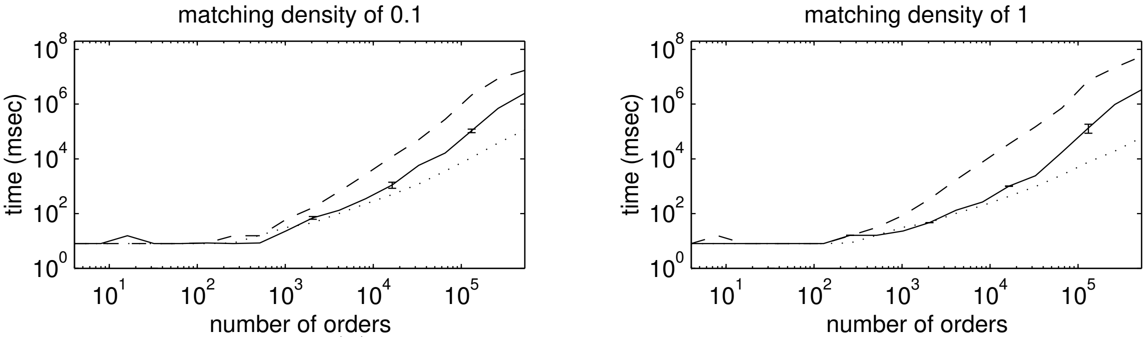


(c) Time between order placement and system response.

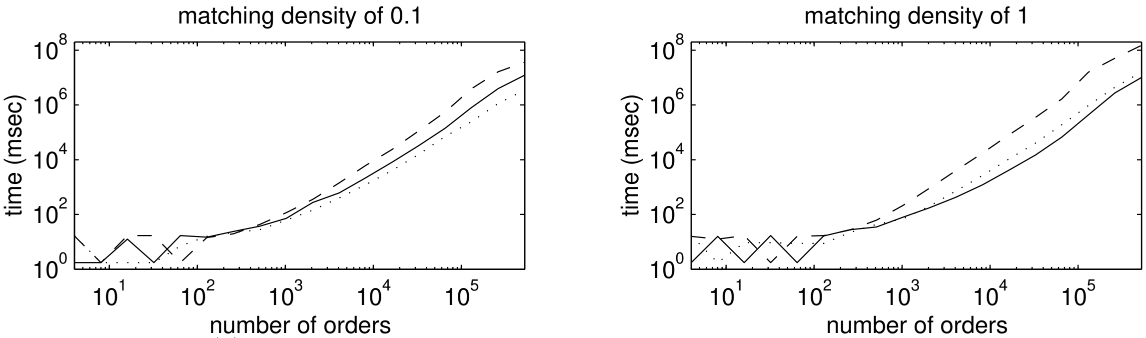
Figure 6.23: Performance in the commercial-paper market for matching density of 0.001 (left) and 0.01 (right). We show the performance of the best-first search (solid lines), depth-first search that identifies all matches (dashed lines), and depth-first search with a limit on the number of matches (dotted lines).



(a) Time of processing the new orders.

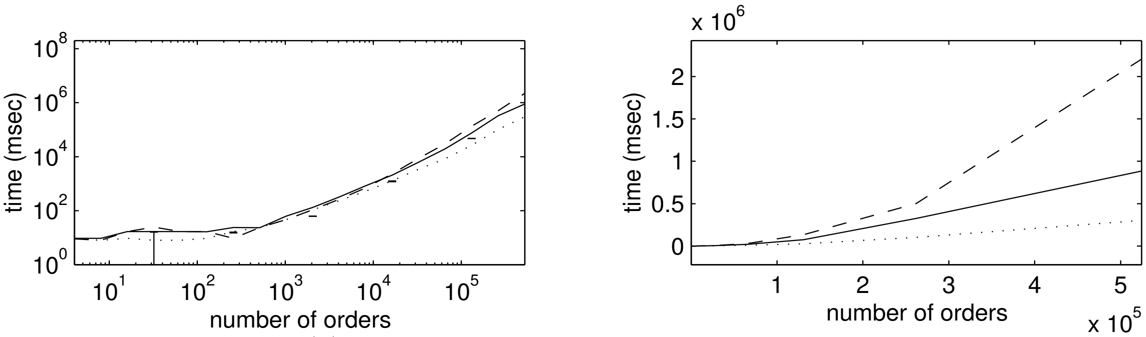


(b) Time of matching the old orders.

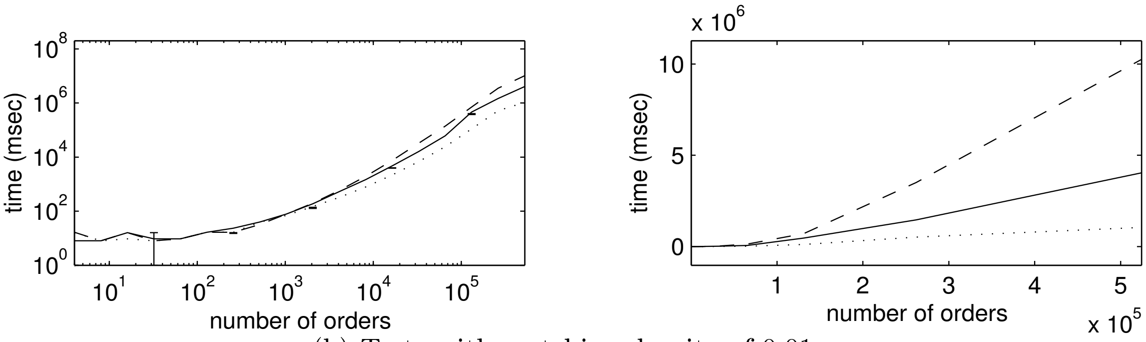


(c) Time between order placement and system response.

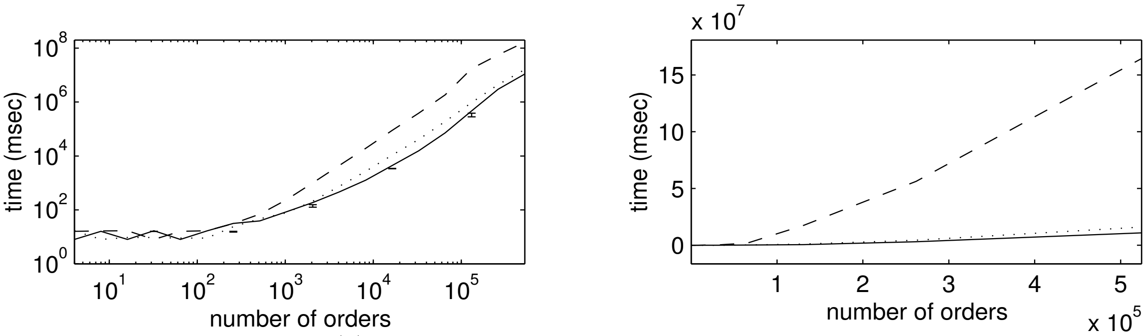
Figure 6.24: Performance in the commercial-paper market for matching density of 0.1 (left) and 1 (right). The legend is the same as in Figure 6.23.



(a) Tests with matching density of 0.001.

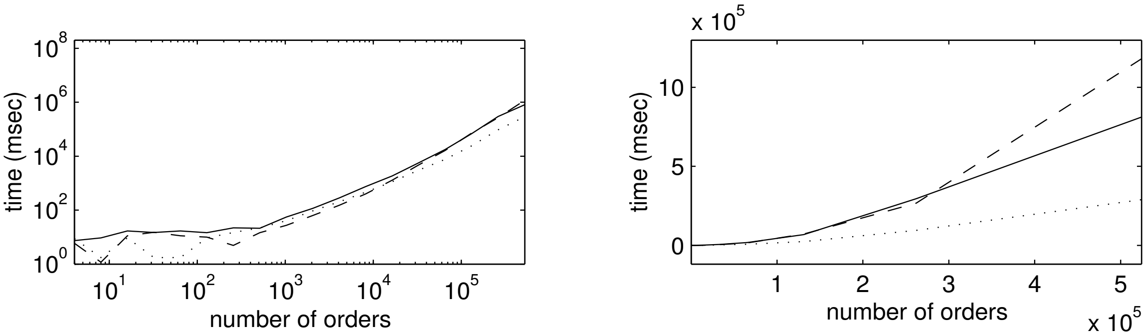


(b) Tests with matching density of 0.01.

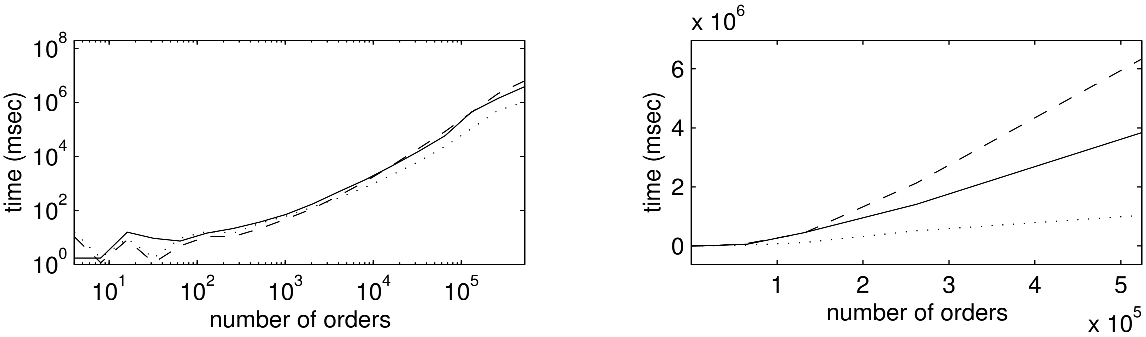


(c) Tests with matching density of 1.

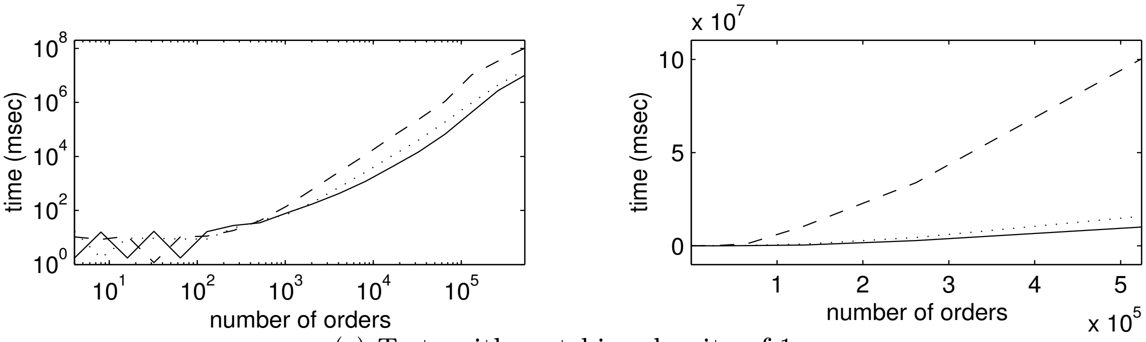
Figure 6.25: Dependency of the total matching time on the number of orders. We give the results for the best-first search (solid lines), depth-first search that identifies all matches (dashed lines), and depth-first search with a limit on the number of matches (dotted lines). The graphs on the left are in logarithmic scale, whereas the graphs on the right are in linear scale.



(a) Tests with matching density of 0.001.

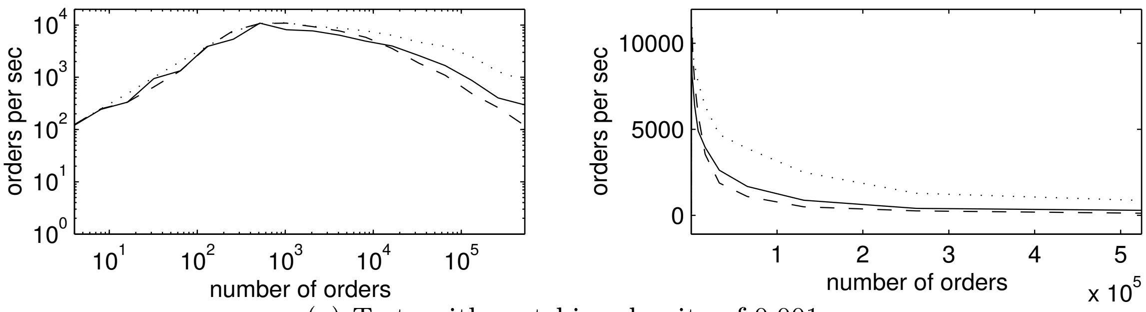


(b) Tests with matching density of 0.01.

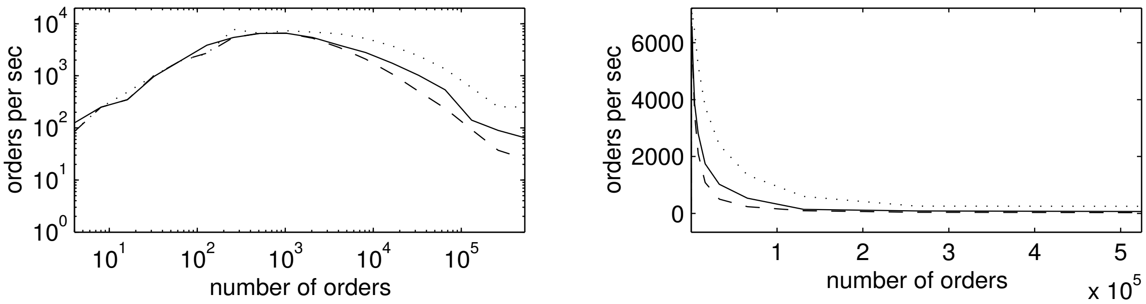


(c) Tests with matching density of 1.

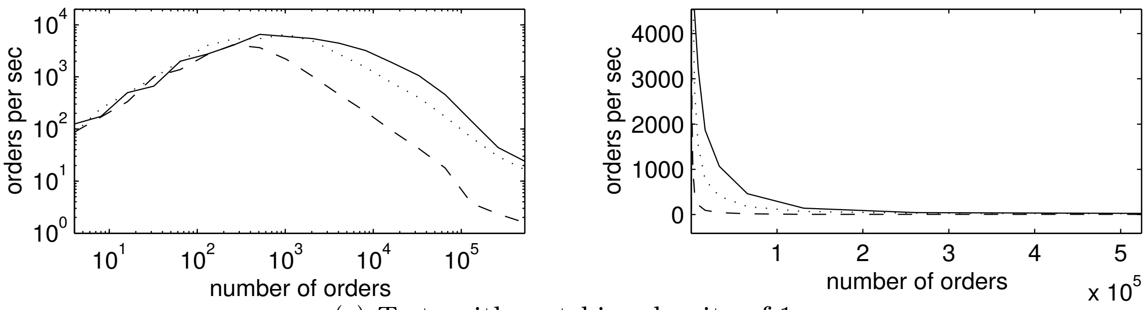
Figure 6.26: Dependency of the response time on the number of orders. The legend is the same as in Figure 6.25.



(a) Tests with matching density of 0.001.

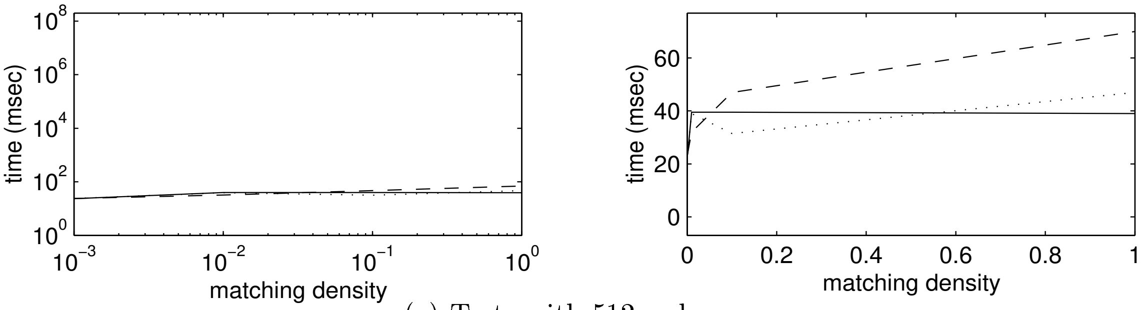


(b) Tests with matching density of 0.01.

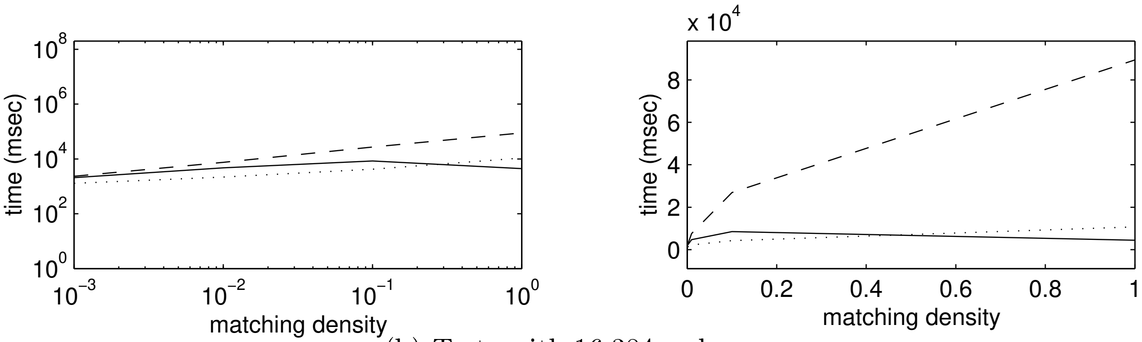


(c) Tests with matching density of 1.

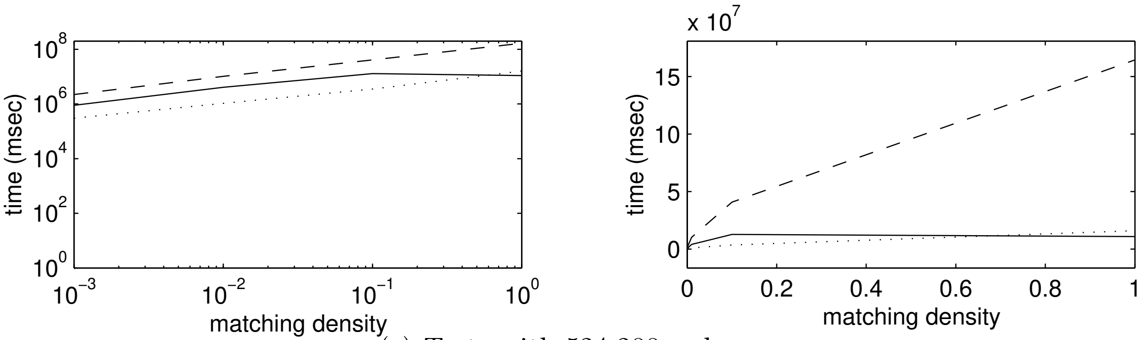
Figure 6.27: Dependency of the throughput on the number of orders. The legend is the same as in Figure 6.25.



(a) Tests with 512 orders.

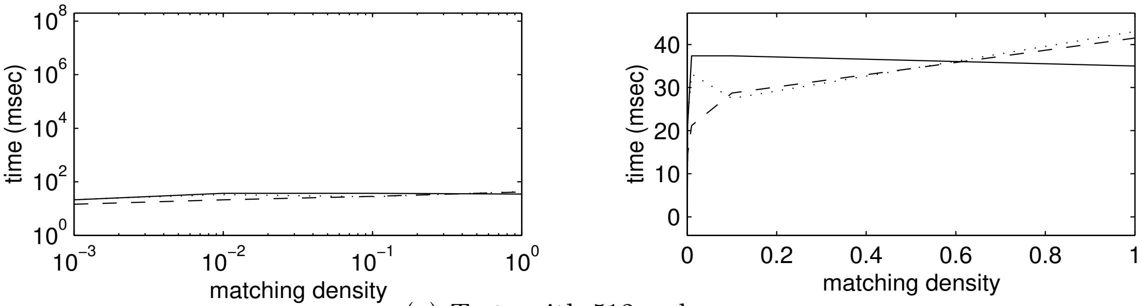


(b) Tests with 16,384 orders.

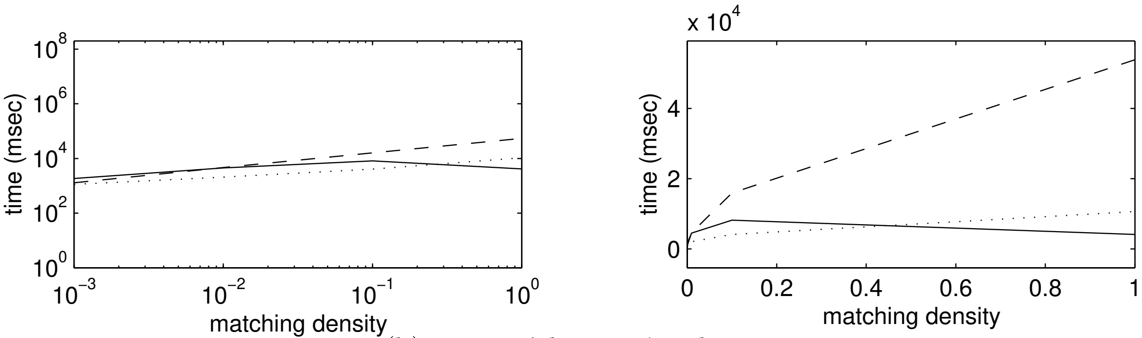


(c) Tests with 524,288 orders.

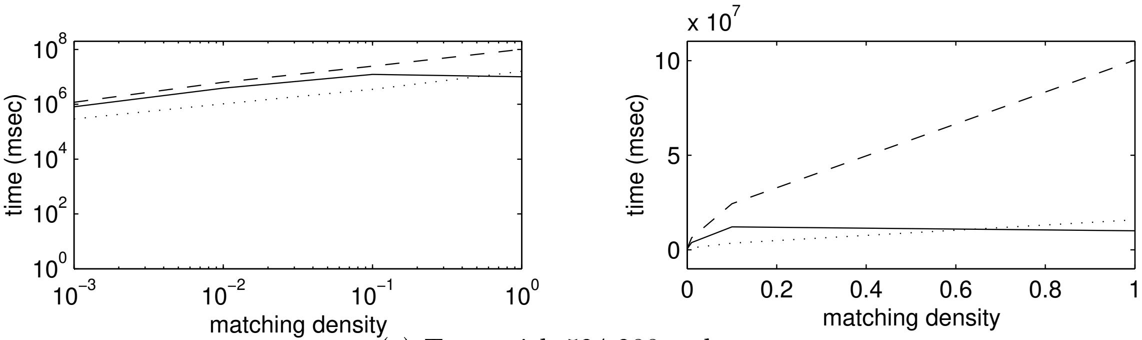
Figure 6.28: Dependency of the total matching time on the matching density. The legend is the same as in Figure 6.25.



(a) Tests with 512 orders.

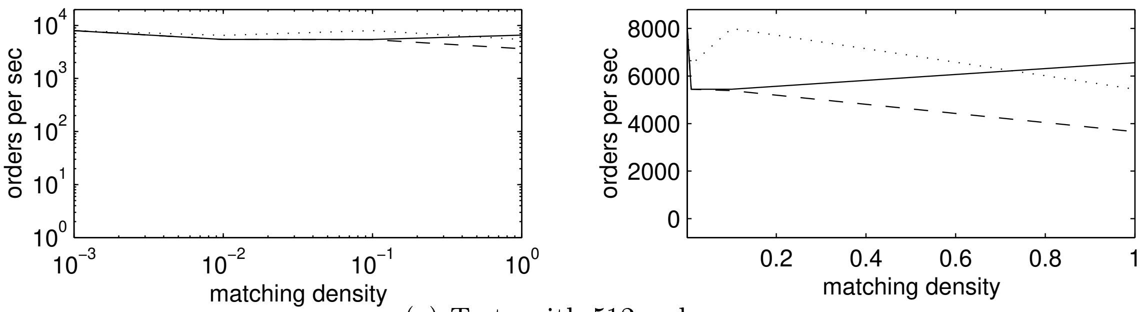


(b) Tests with 16,384 orders.

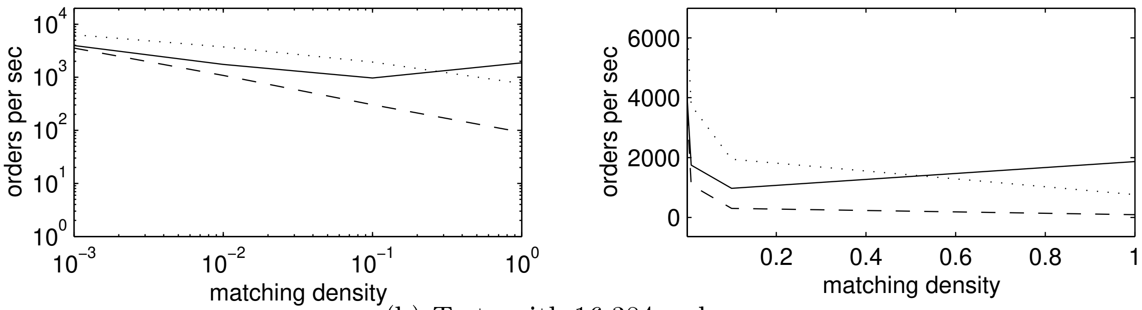


(c) Tests with 524,288 orders.

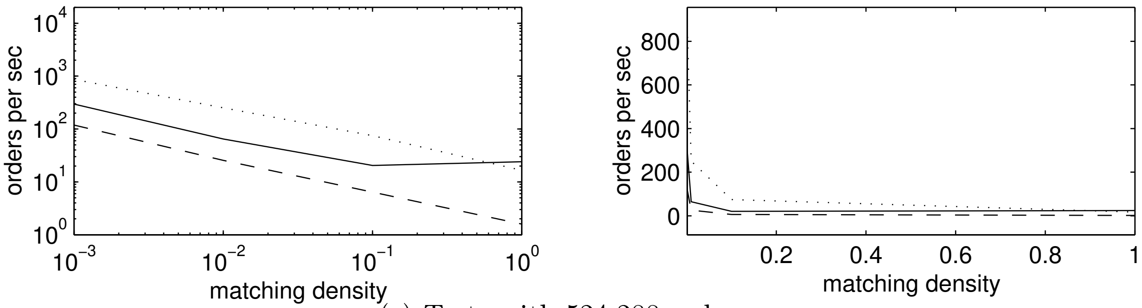
Figure 6.29: Dependency of the response time on the matching density. The legend is the same as in Figure 6.25.



(a) Tests with 512 orders.



(b) Tests with 16,384 orders.



(c) Tests with 524,288 orders.

Figure 6.30: Dependency of the throughput on the matching density. The legend is the same as in Figure 6.25.

Chapter 7

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], Hu [2002], and Gong [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.

On the negative side, the system does not allow for bulk discounts, barter trading, or matches between two nonindex orders. These concepts are commonly found in today’s markets, and we plan to address them as part of future research. In addition, we are currently working on a distributed version of the exchange, which will improve scalability; it will include a central matcher and multiple preprocessing modules, whose roles are similar to that of stock brokers on Wall Street.

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.

[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 Kaukal, 1999] Martin Bichler and Marion Kaukal. Design and implementation of a brokerage service for electronic procurement. In *Proceedings of the Tenth International Workshop on Database and Expert Systems Applications*, pages 618–622, 1999.

[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 and Werthner, 2000] Martin Bichler and Hannes Werthner. A classification framework of multidimensional, multi-unit procurement negotiations. In *Proceedings of the 11th International Workshop on Database and Expert Systems Applications*, pages 1003–1009, 2000.

[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 *et al.*, 2001] Martin Bichler, Juhnyoung Lee, Ho Soo Lee, and Jen-Yao Chung. Absolute: An intelligent decision making framework for e-sourcing. In *Proceedings of the Third International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems*, pages 195–201, 2001.

[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.

[Burmeister *et al.*, 2002] Birgit Burmeister, Tobias Ihde, Thomas Kittsteiner, Benny Moldovanu, and Jorg Nikutta. A practical approach to multi-attribute auctions. In *Proceedings of the 13th International Workshop on Database and Expert Systems Applications*, pages 573–577, 2002.

[Cagno *et al.*, 2001] Enrico Cagno, Franco Caron, and Alessandro Perego. Multi-criteria assessment of the probability of winning in the competitive bidding process. *International Journal of Project Management*, 19:313–324, 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.

[Cliff, 1998] Dave Cliff. Genetic optimization of adaptive trading agents for double-auction markets. In *Proceedings of the* IEEE/IAFE/INFORMS *1998 Conference on Computational Intelligence for Financial Engineering*, pages 252–258, 1998.

[Cliff, 2002] Dave Cliff. Evolution of market mechanism through a continuous space of auction-types. In *Proceedings of the Third* ACM *Conference on Electronic Commerce*, pages 206–209, 2002.

[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.

[Contreras *et al.*, 2001] Javier Contreras, Arturo Losi, and Mario Russo. A JAVA/MATLAB simulator for power exchange markets. In *Proceedings of the 22nd* IEEE *Power Engineering Society International Conference on Power Industry Computer Applications. Innovative Computing for Power-Electric Energy Meets the Market*, pages 106–111, 2001.

[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.

[Dumas *et al.*, 2002] Marlon Dumas, Guido Governatori, Arthur Hofstede, and Phillipa Oaks. A formal approach to negotiating agents development. *Electronic Commerce Research and Applications*, 1:193–207, 2002.

[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.

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

[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.

[He and Leung, 2001] Minghua He and Ho-Fang Leung. An agent-bidding strategy based on fuzzy logic in a continuous double auction. In *Proceedings of the 2001* IEEE *International Conference on Machine Learning and Cybernetics*, pages 583–588, 2001.

[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 Shi, 2002] Shan-Li Hu and Chun-Yi Shi. The computational complexity of dynamic programming algorithm for combinatorial auctions. In *Proceedings of the 2002 International Conference on Machine Learning and Cybernetics*, pages 266–268, 2002.

[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.

[Huhns and Vidal, 1999] Michael N. Huhns and Jose M. Vidal. Online auctions. *Internet Computing*, 3:103–105, 1999.

[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 and Koehler, 2002] Joni L. Jones and Gary J. Koehler. Combinatorial auctions using rule-based bids. *Decision Support Systems*, 34:59–74, 2002.

[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.

[Kastner *et al.*, 2002] Ryan Kastner, Christina Hsieh, Miodrag Potkonjak, and Majid Sarrafzadeh. On the sensitivity of incremental algorithms for combinatorial auctions. In *Proceedings of the Fourth* IEEE *International Workshop on Advanced Issues of E-Commerce and Web-based Information Systems*, pages 81–88, 2002.

[Keever and Alcorn, 2000] David Keever and Walter Alcorn. Development and growth of internet environmental exchange services. In *Proceedings of the 2000* IEEE *Engineering Management Society*, pages 308–312, 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.

[Leyton-Brown *et al.*, 2002] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Learning the empirical hardness of optimization problems: The case of the combinatorial auctions, 2002. Unpublished manuscript.

[Maamar, 2002] Zakaria Maamar. Association of users with software agents in e-commerce. *Electronic Commerce Research and Applications*, 1:104–112, 2002.

[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.

[Moore *et al.*, 1990] James Moore, William Richmond, and Andrew Whinston. Optimal decision processes and algorithms. *Journal of Economic Dynamics and Control*, 14:375–417, 1990.

[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.

[Piccinelli *et al.*, 2001] Giacomo Piccinelli, Chris Priest, and Claudio Bartolini. E-service composition: Supporting dynamic definition of process-oriented negotiation. In *Proceedings of the 12th International Workshop on Database and Expert Systems Applications*, pages 727–731, 2001.

[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.

[Rahwan *et al.*, 2002] Iyad Rahwan, Ryszard Kowalczyk, and Ha Hai Pham. Intelligent agents for automated one-to-many e-commerce negotiation. In *Proceedings of the Twenty-Fifth Australian Conference on Computer Science*, pages 197–204, 2002.

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

[Richter and Sheble, 1998] Charles W. Richter and Gerald B. Sheble. Genetic algorithm evolution of utility bidding strategies for the competitive marketplace. IEEE *Transactions on Power Systems*, 13:256–261, 1998.

[Ronen and Saberi, 2002] Amir Ronen and Amin Saberi. On the hardness of optimal auctions. In *Proceedings of the 43rd Annual* IEEE *Symposium on Foundations of Computer Science*, pages 396–405, 2002.

[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 and Suri, 2003] Tuomas Sandholm and Subhash Suri. BOB: Improved winner determination in combinatorial auctions and generalizations. *Artificial Intelligence*, 145:33–58, 2003.

[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.

[Sandholm, 2002] Tuomas Sandholm. Algorithm for optimal winner determination in combinatorial auctions. *Artificial Intelligence*, 135:1–54, 2002.

[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.

[Tennenholtz, 2002] Moshe Tennenholtz. Tractable combinatorial auctions and b-matching. *Artificial Intelligence*, 140:231–243, 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.

[Tewari *et al.*, 2003] Gaurav Tewari, Jim Youll, and Pattie Maes. Personalized location-based brokering using an agent-based intermediary architecture. *Decision Support Systems*, 34:127–137, 2003.

[Trenton, 1964] Rudolf W. Trenton. *Basic Economics.* Meredith Publishing Company, New York, NY, 1964.

[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.

[Weinhardt and Gomber, 1999] Christof Weinhardt and Peter Gomber. Agent-mediated off-exchange trading. In *Proceedings of the 32nd Annual Hawaii International Conference on System Sciences*, pages 1–6, 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.

[Xia *et al.*, 2003] Mu Xia, Gary J. Koehler, and Andrew B. Whinston. Pricing combinatorial auctions. *European Journal of Operational Research*, 2003. To appear.

[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.