# Scaling-Up Shopbots - a Dynamic Allocation-Based Approach *

David Sarne
School of Engineering and Applied Sciences
Harvard University
Cambridge MA 02138 USA

Sarit Kraus
Department of Computer Science
Bar-Ilan University
Ramat-Gan, 52900 Israel

Takayuki Ito
Dept. of Techno-Business Administration
Nagoya Institute of Technology
Nagoya 466-8555, Japan

## ABSTRACT

In this paper we consider the problem of eCommerce comparison shopping agents (shopbots) that are limited by capacity constraints. In light of the phenomenal increase both in demand for price comparison services over the internet and in the number of opportunities available in electronic markets, shopbots are nowadays required to improve the utilization of their finite set of querying resources. In this paper we introduce *PlanBot*, an innovative shopbot which uniquely integrates concepts from production management and economic search theory. *PlanBot* aims to maximize its efficiency by dynamically re-planning the allocation of its querying resources according to the results of formerly executed queries and new arriving requests. We detail the design principles that drive the *PlanBot*'s operation and illustrate its improved performance (in comparison to the traditional shopbots' First-Come-First-Served (FCFS) query execution mechanisms) using a simulated environment which is based on price datasets collected over the internet. Our encouraging results suggest that the design principles we apply have the potential of being used as an infrastructure for various implementations of future comparison shopping agents.

## Categories and Subject Descriptors

K.4.4 [ **Electronic Commerce**]: [Autonomous agents]

## General Terms

Economics

## Keywords

Shopbots, Comparison Shopping Agents

## 1. INTRODUCTION

The continuous growth in the number of retailers and virtual stores over the internet, has brought a phenomenal increase in the number of opportunities available for consumers in electronic marketplaces. In an effort to fully exploit this richness of opportunities, small-volume buyers and individuals adopt the use of autonomous agents for enhancing their buying experience. Recent research has suggested many applications in which agents can be used in order to facilitate consumer-related activities over the different stages of the consumer's buying experience [11]. Specific emphasis has been placed on the integration of software agents into the *Merchant brokering* stage. In this stage the buyer searches for sellers who offer a specific desired product [12]. This can be facilitated by many commercial comparison-shopping agents (shopbots) that can be found over the Web (e.g., PriceScan.com, Shopping.com, MySimon.com). The main advantage of comparison-shopping agents (shopbots), in this context, is in their capability to automatically query multiple vendors, searching for price information on desired goods and present it in a consolidated and compact format [10].

While the implementation of shopbots can be based on maintaining a database of prices (i.e., storing the price in which each seller is selling a given product), it is definitely not the preferred option. This is due to the ever-increasing frequency of price updates in electronic markets nowadays. This phenomena has empirical evidence in literature [2] and is also well supported in theory. For example, dynamic pricing theories suggest that sellers can benefit from re-pricing their goods as often as possible based on their observations of the competitors' prices [10]. Alternatively, E-retail managers may use "hit and run" sales strategies undertaking short-term price promotions at unpredictable intervals - a method shown to be effective and widely used [2]. Therefore a reliable shopbot is expected to use real-time querying of electronic merchants (rather than retrieve price data from a formerly collected price database) upon the arrival of price comparison requests from its users.

As any computer based entity, shopbots have a limited querying capacity: the shopbot can maintain a finite number of queries concurrently. The limit derives from the number of communication channels (ports) it can maintain/support simultaneously.[1] Current shopbots that query prices in real-time usually maintain a list of potential electronic merchants for each product. Upon the arrival of a request the shopbot queries all these merchants and returns a list of price quotes. This strategy produces good results when the querying capacity exceeds demands. Alas, the expected increase both in the number of electronic merchants offering any specific product and in the demand for shopbots services[2] suggest that shopbots are continuously proceeding to a point where demand (even temporarily) will exceed capacity. Whenever the latter scenario occurs, some of the requests will remain unanswered or partially

*The research was partially supported by ISF grant 8008.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 2007 IFAAMAS .

[1]Making use of the users' resources (e.g., execute queries from their computers) is infeasible in this business application since it requires the installation of several of the shopbot's core components as an add-on to the clients and partially reveals proprietary information that the shopbot holds.

[2]The comparison shopping market is growing at 30% year-on-year. The estimate is that retail-focused comparison shopping engines made in the UK only, totalled between £120m and £140m in revenue during 2005 [8].

answered (either due to timeouts in the TCP/IP connection to the shopbot or because of impatient users who are not willing to wait so long for results). For example, assume that three requests for comparative price quotes arrive at the same time from 3 different users, for 3 different products: a music CD, a printer tuner cartridge and a scanner (specific brand and model), respectively. Let's assume that the shopbot knows about 20 different merchants selling each of these 3 products, and that its overall querying capacity (until results are required) is 45 queries. The simplest sampling technique according to first come first served (FCFS) is to execute 20 queries for the CD, 20 for the cartridge and 5 for the scanner, resulting in missing 15 potential price quotes for the scanner. Obviously the shopbot's operators can solve this problem by increasing the shopbot's querying capacity (e.g., buy more connection bandwidth and place more servers for generating queries). However this involves substantial costs and often does not justify the added benefit. As an alternative for overcoming temporal overloads by using the resources redundancy approach, we propose in this paper that shopbots can improve their performance by being more selective in terms of the queries they choose to generate, i.e. improving their efficiency. This approach is best illustrated in the communication networks domain, where the key for overcoming timely congestion is not by a continuous increase in bandwidth but rather by improving network efficiency (e.g. using traffic shaping). This is done principally by taking advantage of the shopbot's knowledge of the distribution of prices. Consider the example given above and assume (for the sake of the example) that the CD prices are uniformly distributed in the price range of \$5-\$15, the cartridge prices are uniformly distributed over the interval of \$20-\$40 and the scanner prices are uniformly distributed in the range of \$25-\$55. Table 1 summarizes the performance (in terms of the expected minimum price found by the shopbot for the 3 products)[3] of the FCFS allocation rule and an alternative allocation that takes into consideration the distribution of prices when deciding how many queries to allocate for each request.

| Product | FCFS | | Alternative execution | |
|---|---|---|---|---|
| | queries | E[minPrice] | queries | E[minPrice] |
| CD | 20 | \$5.47 | 11 | \$5.83 |
| Cartridge | 20 | \$20.95 | 15 | \$21.25 |
| Scanner | 5 | \$30 | 19 | \$26.5 |
| Total | 45 | \$56.43 | 45 | \$53.58 |

**Table 1:** Expected minimum price for each requested product according to the different querying schemes

As can be seen from the table, the overall cost of buying the 3 products based on the alternative querying approach (\$53.58) is less than the one achieved when using the FCFS method (\$56.43), while allocating the same number of queries. This improvement might seem moderate at first glimpse. However, even if we had an option to query the 20 prices for each product, the expected overall cost would be \$52.86. Therefore, our small change in the querying scheme brings us very close to the performance that could have been achieved when removing the querying resource constraint.

The improvement given in the above example relates to the fact that each product is associated with a different distribution of prices, thus price-samples of similar sizes taken for each item will result in different price savings. This is one of the key concepts of economic search theory (as detailed in the following section). Our proposed *PlanBot* is designed to take advantage of the above principle, and by partially adopting complementary principles from the production management domain it continuously re-plans the series of its future executed queries in times where demand exceeds capacity.

The main contributions of this paper are threefold: First, we present an innovative resource-allocation based querying methodology for shopbots, driven by expected utilities, rather than a preset querying paradigm of the type characterizing the FCFS method. Furthermore, the proposed mechanism continuously monitors its own performance and uses this data as an input for determining priorities in exploiting future capacity. This enables bypassing the need of modeling the arrival rate and future requests of users. Second, we implement the *PlanBot* based on the proposed mechanism and demonstrate its efficiency using an environment that is based on real data. Last, we suggest several extensions to the model and explain how the mechanism can be modified to support various goals of the shopbot's operators (e.g., supporting different service levels, maximizing the agent's revenue).

In the next section we present the general guidelines adopted from search theory and production management domains that are used for establishing the allocation mechanism of the *PlanBot*, presented in Sections 3-4. The performance achieved by *PlanBot* in a semi-realistic environment is depicted in Section 5. Discussion and methods for extending the applicability of the proposed methodology are given in Section 6. We conclude with a discussion and suggest further research in Section 7.

## 2. APPLICABLE DOMAINS

The comparison shopping agents domain has attracted the focus of researchers and developers for the past 10 years [10, 11, 12]. However, none of the work in this area has considered a scenario where the shopbot needs to operate under limited capacity constraints. The majority of the analysis concern the influence of shopbots on retailers' and consumers' behavior [23, 6, 13], given the premise that shopbots can significantly reduce search costs (associated with obtaining price information).

When adding the resources constraints a new problem arises whenever the shopbot cannot query all the relevant merchants for price quotes, given its capacity and the requests it receives. Here, the shopbot needs to prioritize the different queries, and produce an allocation plan that maximizes some criteria. This problem encountered by shopbots is unique in the sense that though it shares some of the characteristics of both classical economic search problems and scheduling/resource allocation problems in the production management domain, it does not fully resemble any of them.

The production management domain includes many distinctive environments in which scheduling (also referred to as planning or allocation) is used (see [19, 14] for surveys): continuous and repetitive flow lines, batch flow lines, manufacturing cells, job shop, and project (fixed site) processes. Since the scheduling problem is inherent in many computer systems (e.g., scheduling the processing of computer jobs on servers [17]), it also has been extensively studied in computer science, resulting in many theoretical results (optimal and approximation algorithms) for diverse variants. Usually, the goal of production systems is to use a set of resources to accomplish a set of tasks in an order maximizing an optimization criterion. Jobs are usually associated with due-dates and are beneficial if fully satisfied/completed. Many heuristics have been suggested for scheduling problems, most of them aiming to assign some priority index to the incoming jobs waiting to be processed so that the one with the highest priority can be selected to be processed next [19]. Among the general rules and heuristics for determining priorities and building schedules one can find: FCFS (first come, first served); SPT (prioritizing according to the shortest processing time first) ; EDD (prioritizing according the earliest due date) ;

[3]The expected minimum price when querying $N$ merchants (i.e., the minimum of a sample of size $N$) for a uniform distribution function defined over the interval $(a, b)$ is given by: $\frac{b+Na}{N+1}$.

| Domain | Similar | Different |
|---|---|---|
| Search Theory | Tasks are divisible and have deadlines; Uncertain outcomes | Single task at a time; No new tasks arrival; Search resources are not constrained; Search strategy is driven by search costs |
| Resource Allocation | Multiple arriving tasks; Tasks have deadlines; Constrained resources; | Tasks need to be fully satisfied and partial performance of a task has no value |

**Table 2:** Similar and different aspects of the shopbots problem in comparison to Search Theory and Scheduling/Resource allocation problems

the Hodgson Algorithm (a multi-step EDD-based algorithm); the Lawler algorithm (for problems with precedence constraints between jobs) and other various techniques based on one or a mixture of the above for single and multiple processors, aiming to optimize different measures.

In our problem, the system learns about new requests only upon their arrival and needs to react based on partial knowledge of inputs. This pattern can be found in on-line scheduling algorithms designed for one machine or parallel machine environments (see [22] for a survey). Many of these algorithms address problem variants with periodicity and precedence constraints and deadlines (see [7, 1] and references therein). Here the objective function that needs to be minimized is typically the time when the last job is completed. Some variations allow jobs to be rejected at a certain penalty which they then incorporate in the goal function. Other objective functions which have been addressed are the total completion time, the total flow of jobs in the system and the total waiting time. Nevertheless, in the field of on-line scheduling an algorithm typically must schedule a number of jobs or tasks without knowing how long it will take to complete each task. A particular type of on-line algorithms in which each job has to be executed in a precisely given time interval (i.e., fixed start and end times) is known as Interval Scheduling [16]. For example in the on-line scheduling of broadcasts, a server is required to broadcast to clients upon request. Each arriving request defines the requested page, an arrival time, a deadline in which it needs to be received in its entirety, and the weight of the request [20]. However in these problems, if the system does not have available resources for the job within its pre-set execution window (or if it terminates execution before completing the job), the job is rejected. Therefore the common performance measure of these algorithms is the number of accepted jobs.

The main difference between production management scheduling heuristics and the shopbot's problem is that the first takes all arriving jobs (requests) as indivisible entities in terms of their value. In order to gain value, the entire task needs to be eventually processed. The shopbot's problem, on the other hand, concerns jobs that have value even if partially performed (a function of the discrete queries that are executed at any time before approaching the deadline for the specific request). Furthermore, in the shopbot's problem, the value of a task is probabilistic and in particular, the marginal value of each additional portion of task execution depends on the value achieved in the execution of former portions (i.e., the value of any additional query allocated for a task depends on the best price found till that time in formerly executed queries).

Economic search theory, on the other hand, considers the execution of divisible tasks (see [18] for literature review). In its most basic form, the search problem considers an individual interested in locating an opportunity which will minimize its expected cost (or maximize its expected utility), while the search process is associated with a search cost. The three main search models that can be found in literature are the fixed sample size model, the sequential model and the variable sample size model. In the fixed sample size model, introduced in [24], the searcher draws a single sample where all observations are taken simultaneously. In the sequential search strategy [21] the searcher draws one observation at a time, allowing multiple search stages. The last search method [9] suggests a combined approach in which several observations may be made in any period. Attempts to adopt economic search models (where any sample is associated with some search cost) in agent-based electronic trading environments are suggested in [5, 15]. However, economic search models consider a single search activity, whereas the shopbot's problem considers multiple simultaneous search requests. Furthermore, while economic search theory is driven by the tradeoff between the benefit and costs associated with each additional query, the shopbot mainly attempts to use its limited set of resources efficiently. Therefore, the reservation-value based strategies that are usually offered in economic search theory mechanisms are not applicable for the shopbot.

Table 2 describes the similarities and differences between the shopbot's problem and the search theory and production management domains. The shopbot problem is unique in the sense that it combines: (a) dynamic arrival of divisible tasks (i.e., allowing partial performance levels) with deadlines; (b) limited resources/capacities for performing these tasks; and (c) probabilistic payoff, given the amount of resources invested in each task. Despite the inherent differences between the shopbot's problem and the two other domains given above, we can adopt some of the insights discussed in these two areas for constructing our allocation algorithm:

- The expected marginal reduction in price decreases as the number of price queries increases.
- Sequential execution of the queries is always better than Parallel search (assuming there is no advantage for parallel search in terms of the amount of required resources and the decision horizon is not finite). While the shopbot's problem has a finite decision horizon for each request, we will aim to take advantage of sub-sequential scheduling patterns.
- Requests can be delayed given their different due-dates. However, by some means, the mechanism should weight delays against the future probability of arrival of better yielding requests.

In the following sections we formally describe our *PlanBot* shopbot's model and present the mechanism it uses for selecting which requests to process at any time point.

# 3. THE MODEL

We consider a shopbot that offers price comparison services for a large set of well defined products. In order to learn the price by which a product is offered in a given store at a given time, the shopbot executes a real-time query to this store. The shopbot receives from different users requests for obtaining price quotes. We assume requests arrive according to some arrival probabilistic function with which the shopbot is unfamiliar and that the shopbot learns about a new request only upon its arrival. Each arriving request is associated with a deadline by which it must be answered (defined either directly by the user or as part of a Service Level Agreement (SLA) with the user).

We assume that while prices are dynamically set by the stores, the distribution of prices remains constant along time, reflecting the level of competition associated with the said product. This assumption is supported by recent empirical research in well-established

online retail markets, presenting evidence of the persistence of price dispersion [3, 6, 4]. Furthermore, we assume that there is no correlation between a merchant's relative position in the distribution of prices in any two consequent times. The latter assumption is, again, based on empirical research findings of: (a) a considerable turnover in firms' relative positions in the distribution of prices over time; and in particular (b) a significant variation in the identity of the low-price firm for the same product over time [2]. The shopbot is assumed to be familiar with the price distribution of each product, or capable of learning it over time.[4]

We assume the shopbot has a fixed limited set of resources (CPU, communication, etc.) from which a limit can be derived for the amount of price-quote queries it is capable of maintaining in parallel over each time unit. Querying different merchants suggests different querying times. However since the shopbot executes several queries in parallel and since a product is offered by many merchants and any merchant supplies many different products, we assume its querying rate is product-independent (i.e., the number of queries it will be able to fully execute at any given time interval does not depend on the type of products for which it needs to supply price quotes).

Users are assumed to be interested merely in the minimum price found by the shopbot. Thus, the value (as perceived by the user) encapsulated in the results returned for a request is calculated as the difference between the minimum price found and the price initially known to the user. The applicability and extensions of the proposed mechanism to a general utility function are discussed in Section 6. We assume that the shopbot's goal is to maximize the total social welfare of its users, measured as the aggregated price reduction achieved. There are many reasons to maximize the overall social welfare of shopbots' users. For example, if the users repeatedly use the shopbot for different product requests, then increasing overall social welfare will result in improving private welfare. Another scenario is where the shopbot is owned, operated and used by a corporation thus attempts to maximize the corporate welfare rather than individual welfare. Notwithstanding, we supply in Section 6 several extensions that enable the adaptation of the proposed mechanism to other performance measures such as maximizing the shopbot's profit or minimizing deviation from guaranteed SLAs.

We use $P = \{P_1, \ldots, P_N\}$ to denote the set of products for which comparison shopping requests may be received. We use $M_i$ to denote the number of stores in which product $P_i$ can be found. We denote the price probability distribution function of product $P_i$ by $f^i(x)$ and its cumulative distribution function by $F^i(x)$. A request for price quotes is denoted $R_i$, where $P_{R_i}$ is the requested product, $d_{R_i}$ denotes the deadline for this request (relative time), $t_{R_i}$ denotes its arrival time to the shopbot (absolute time), and $c_{R_i}$ denotes the price (i.e., cost) currently known to the user for this product[5]. We denote the shopbot's capacity (the number of queries that can be executed in parallel) by $C^*$, and its immediate free capacity at time $t$ by $C_t$. The average query execution length is denoted by $L$. The shopbot's problem is finding a strategy $S(\{R_i\}, \{m_i\}, \{c_i\}, C_t) \rightarrow \{q_i\}$, that determines the number of queries that should be immediately allocated, $q_i$, for the benefit of each request $R_i$, based on a set of known requests $\{R_i\}$, the best price $\{c_i\}$ found so far for the benefit of each request in $\{R_i\}$, the number of queries already executed $\{m_i\}$ for the benefit of each request in $\{R_i\}$ and the available capacity $C_t$.

[4]There are several methods by which an agent can be acquainted with this distribution function: past experience, Bayesian update, etc. [21].

[5]Usually we will initialize this value with the expected (mean) price of a random quote, e.g., the expected price that could have been obtained if the user had bought the product in one of the stores randomly.

The long term expected overall price reduction from a potential plan can be formulated by taking into consideration the expected immediate gain and the expected weighted aggregated gain from the plans that will follow, based on the results of executing this plan. This should consider all possible combinations of price quote received for the set of queries being executed, for all possible combinations of known prices to start with. The calculation should also include all the different combinations of new requests arrival and their different due date combinations as well as the initial prices reported by the users for each request (known prices). Thus an optimal solution to the problem suggests solving a set of complex equations which is obviously computationally infeasible. Therefore our *PlanBot*, as detailed in the following section, is based on a mechanism that attempts to capture expected performance as a measure of different queries allocation worthiness.

# 4. ALLOCATION PRINCIPLES

In this section we describe the basic principles by which the *PlanBot* was built. These do not necessarily guarantee the optimal performance that theoretically might be achieved. However, they well serve our goal of introducing and demonstrating the notion of economic shopbots that can improve their performance by applying re-allocation techniques when query demands exceed capacity. The core stages of the *PlanBot*'s operational cycle are:

1: **loop**
2: Wait for next time step.
3: Identify the effective planning horizon for current time.
4: Produce an optimal allocation (plan) for the static problem, taking into account expected benefits from future requests.
5: Execute immediate queries according to plan.
6: Update expected performance measures (of future queries, based on local expected performance).
7: **end loop**

As we demonstrate in the following sections, this mechanism outperforms the existing queries allocation schemas (FCFS based) and exhibits exceptional adaptation to changing load scenarios. In the following paragraphs, we explain the above stages in our proposed mechanism, and present the static [6] planning algorithm. The main challenge of the mechanism is to identify opportunities where it might be beneficial to delay queries associated with high immediate benefit however with relatively distant deadlines and instead, execute less beneficial queries associated with requests that are about to expire. The success of the shopbot in doing this is highly correlated with its ability to predict the probability that these delayed queries will actually be executed in the short run, given the fact that new requests are received along time.

## 4.1 Time Steps and Planning Horizon

We begin by defining the time steps used for planning. In principle, we can use any granularity level when setting the time steps length. The time steps length affects the frequency of our planning process execution. More frequent planning results in better performance, however overall it requires more computational resources. Given the length of a time step, $L_{step}$, each planning session will affect the execution of $C = (L_{step}C^*)/L$ queries. In order to keep the exposition of the following paragraph simple we set the time step to be equal to a single time unit over the standard timeline (e.g. one second).

We use $R(t)$ to denote the set of influencing requests at time

[6]The term "static problem" refers to the scheduling problem in which the agent needs to produce a static (i.e., a non-updating) plan/schedule that achieves the maximum expected price reduction, given the known requests and its estimate of its own future performance.

$t$. The set $R(t)$ includes all the requests for which executing additional queries at time $t$ (assuming not exceeding the maximum number of known merchants) is expected to result in an increase in the overall performance. The set $R(t)$ can be extracted using:

$$R(t) = \{R_i | t_{R_i} \leq t \wedge (t_{R_i} + d_{R_i}) \geq t \wedge m_{R_i} < M_{P_{R_i}}\} \quad (1)$$

At each time $t$, we define our planning horizon $H_t$ as: $H_t = max\{ d_{R_i} | R_i \in R(t)\}$. This is the extent of planning that needs to be generated at time $t$, since none of the current requests in $R(t)$ can benefit from queries executed after time $t + H_t$. The goal of the shopbot is to set its immediate execution plan for time $t$. However, for this purpose it needs to maintain a tentative plan for queries execution in the rest of the time steps up to the current planning horizon, $H_t$. We store our plan in a matrix $S$ of size $C \cdot H_t$ (see Figure 1), where each element $S[i][j]$ stores the request for which the $j-th$ query at the $i-th$ time step (measured relative to $t$) should be executed. For each element $S[i][j]$ we use $S[i][j].request$ to denote the request to which this element refers, $S[i][j].value$ to denote the expected marginal benefit from executing this query (assuming that the plan in $\{S[x][y] | (x < i) \vee ((x = i) \wedge (y < j))\}$ is executed) and $S[i][j].due$ to denote the due date associated with the request for which we execute this query. We update the plan stored in $S$ over each time step $t$, thus while the elements stored in $S[1]$ constitute the set of queries that are immediately executed, the other elements in $S$ are subject to changes in subsequent future updates of $S$. For convenience, we assign the elements in $S[i]$ in a descending order (i.e., $S[i][j].value \geq S[i][l].value \forall j < l$).

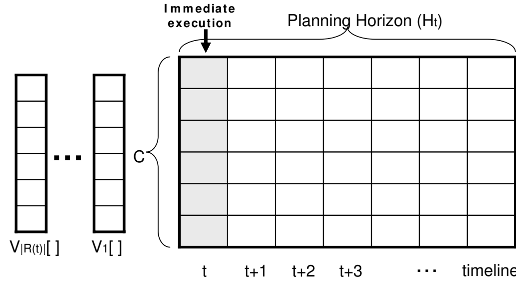


**Figure 1:** The planning matrix and the marginal saving vectors

As explained above, the major difficulty in delaying queries is the estimation of their execution probability, given the uncertainty related to the arrival of "more beneficial" requests in the near future. Therefore, we continuously maintain our estimated expected performance, represented as the expected marginal benefit associated with the $j-th$ query sent at each time step. This is done by using a vector $A$ of size $C$, which is updated every time step $t$ using exponential smoothing:

$$A[j] = \alpha A[j] + (1-\alpha) S[1][j].value \ \ , \ \forall 1 \leq j \leq C \quad (2)$$

where originally $A$ is initialized as $A[j] = 0, \forall 1 \leq j \leq C$. Notice that updates of the elements of $A$ are always based on the expected values of the latest executed queries, which are stored in $S[1]$. Intuitively, vector $A$ represents the expected value associated with future arriving requests that will be scheduled alongside existing requests. Vector $A$ plays an important role in the decision of whether or not to delay queries (free some room for less beneficial ones that have more restrictive deadlines). During temporal overload the values of this vector are expected to rise, indicating extended expected overall benefits due to excessive demands. The parameter $\alpha$ allows us to tune the mechanism for different requests arrival rates. When arrival rate changes are frequent, we will prefer to use a large $\alpha$ value and vice versa.

## 4.2 Local Allocation

For each request $R_i \in R(t)$ the *PlanBot* maintains a vector $V_i$ of size $min(d_{R_i} \cdot C, M_{R_i} - m_{r_i})$ (this is the maximum applicable number of queries for a request in a given planning session). The set of vectors $V_i$ is denoted $V$. Each element $V_i[j]$ in the vector represents the marginal expected saving from obtaining the $j-th$ additional price quote for request $R_i$, given the best price found so far for this request, $c_{R_i}$. The value of $V_i[j]$ can be calculated according to:

$$V_i[j] = E_j[x/c] - E_{j+1}[x/c] = \int_{y=0}^{c_{R_i}} F^i(y)(1-F^i(y))^{j-1} dy \quad (3)$$

where $E_j[x/c]$ is the expected price when sampling $j$ prices, while already knowing a price $c$. As expected, $V_i[j]$ is always positive, i.e., there is always an improvement from increasing the number of queries to be executed for any given request. Furthermore, the first derivative of the integrated term is always negative, thus the magnitude of the expected marginal improvement decreases as the number of queries that are about to be executed for this product increases.

Each re-planning session involves the execution of the following algorithm:

**Algorithm 1** Planning phase

**Input:** $R(t)$ - set of requests; $V = \{V_1, \ldots, V_k\}$ - set of marginal improvement vectors correlated with $R(t)$
**Output:** $S[][]$ - a plan.
1: Set $S[i][j].value = A[j] \ \forall 2 \leq i \leq H_t, 1 \leq j \leq C$
2: Set $S[1][i].value = 0 \ \forall 1 \leq i \leq C$
3: **while** ($\exists i, j, k$ where $V_k[1] > S[i][j].value \wedge d_{R_k} > i$) **do**
4: Set $i = argmax_i\{V_i[1] \mid V_i \in V\}$
5: Set $position = argmax_x\{x \mid S[x][y].value < V_i[1] \wedge d_{R_i} > x, 1 \leq y \leq C\}$
6: **if** (position==null) **then**
7: Remove $V_i$ from the set $V$
8: **else**
9: Set $num = argmin_y\{S[position][y] | 1 \leq y \leq C\}$
10: $S[position][num].value = V_i[0]$
11: $S[position][num].request = R_i$
12: $S[position][num].due = d_{R_i}$
13: Remove first element from $V_i$
14: **end if**
15: **end while**
16: **return** $S[][]$

The algorithm starts by setting the expected price reductions at each future time (possibly from future requests) in the planning horizon to the appropriate value in vector $A$ (step 1). Then, at each stage the algorithm selects the query that yields the maximum expected marginal price reduction (step 4) and schedules it in matrix $S$ in the latest slot that is currently associated with a smaller value (steps 5,9-12), given the deadline of the request associated with this query. Then it updates the value of additional potential queries for this request by removing the first element in the vector $V_i$ associated with that request. If the query cannot be scheduled according to the above rule, then that vector is removed from set $V$. Notice that at the end of the process, some of the elements in $S$ remain unallocated (since the value with which they were initialized (from vector $A$) does not allow allocation). These can intuitively be seen as queries "reserved" for future allocation for the benefit of incoming requests.

THEOREM 1. *Algorithm 1 results in the best allocation for the static allocation problem (in terms of the expected price reduction associated with it), with a complexity of* $O(H \cdot C)$.

**Proof:** Consider the matrix $S$ after executing Algorithm 1. If this is not the best allocation, then the matrix $S$ can be re-arranged in

a way that a set of queries $Q$ from the different vectors in $V$ are now included in the matrix (and a similar number $|Q|$ of queries, formerly scheduled in $S$, are excluded). This can be seen as a set of $Q$ sequences of replacements in $S$ where each sequence results in the inclusion of a query with a value greater than the value of the excluded query[7]. Assume the entering query in any of these sequences is for request $R_v$, has a value $v$ and that it replaces the query currently scheduled in $S[i_1][j_1]$. The replaced query can now replace any element $S[i_2][j_2]$ and the process can continue until the final replacement is set, and the query originally scheduled in $S[i_k][j_k]$ is removed from the planning matrix. Now notice that given the structure of the allocation protocol given in Algorithm 1 and the fact that $V_i[l]$ is a decreasing function of $l$: (a) $S[i][j].value > v \; \forall i < d_{R_v}$; and (b) $S[i][j].value < S[x][y].value$ $\forall\{i, j | S[i][j].due > x \wedge x > i\}$. Therefore, in order to have a sequence of replacements in $S$ that results in the inclusion of a query associated with $R_v$ and the exclusion of element $S[i_k][j_k]$ that has a value smaller than $v$, one of the replacements in the sequence should be $S[a_1][b_1]$ replaces $S[a_2][b_2]$, where $a_1 \leq i_1 \wedge a_2 > i_1$ (otherwise, the excluded element will have a value smaller than $v$ according to (a)). However, according to (b) the replaced element, $S[a_2][b_2]$, has a value greater than the replacing element, $S[a_1][b_1]$ (according to (b)), and the latter element has a value greater than $v$ (according to (a)). Thus the value of the leaving element is inevitably greater than the value of the entering query ($v$). The main loop (steps 3-15) executes at most $H \cdot C$ times since once an element in matrix $S$ is set, it is never replaced. □

Prior to execution we replace the elements in the first vector ($S[1]$) with higher value elements in $S$ without changing the value of the plan. This is solvable as a simple matching problem (with a polynomial complexity) and can only improve performance (since the static plan remains optimal). After finalizing the plan, vector $A$ is updated as detailed earlier in this section. Based on the execution of the queries scheduled in $S[1]$, the planBot updates the values in vectors $V_i$ according to the new $c_i$ values (i.e., the best known prices) associated with each request. Summary data for requests reaching their deadline is sent back to the originating clients, and a new plan (starting at time $t + 1$, for the new horizon $H_{t+1}$) is formed for the requests in $R(t + 1)$ (which now includes also newly arriving requests from time $t$), using the same mechanism.

# 5. SIMULATION

In order to demonstrate the advantages of the *PlanBot* operational mechanism, we constructed a synthetic environment which is based on empirical price distributions found over the internet, as described below.

ENVIRONMENT 1. *The shopbot offers its comparison shopping services for 100 different (real) products (mostly digital cameras, accessories, computer hardware and peripherals and office supplies) sold over the internet. The price distribution associated with each product was set according to a set of prices collected over the internet (using the shopbot pricegrabber.com) from real electronic merchants (at least 30 price quotes were collected for each product). For simplicity, we fitted each empirical distribution function to a continuous uniform distribution (the correlation[8] between the two was at least 0.98 for each product). The number of electronic merchants in the market, the shopbot's capacity and the requests arrival rate were set to be the controlled variables as follows. The number of electronic merchants for each product was initially set to 50. Requests for comparison shopping are received by the shopbot on a timely basis, according to a Poisson arrival rate with the parameter $\lambda = 20$ (i.e., the probability of $i$ requests arriving within a single time step is $\frac{e^{-20}20^i}{i!}$). Each incoming request is defined over a product randomly drawn from the 100 products defined above. The deadline for each request was set arbitrarily to 10 time units. The value of the parameter $\alpha$ used by the shopbot was set to 0.05. Lastly, the shopbot capacity was set to 50 queries per time unit.*

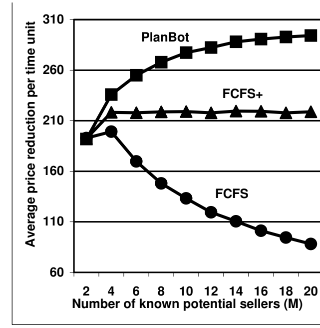


**Figure 2:** Average performance per time unit as a function of the number of stores known, $M_i$

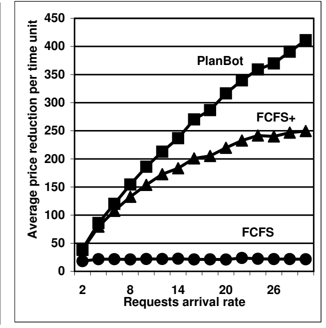


**Figure 3:** Average performance per time unit as a function of the requests' arrival rate $\lambda$

The figures we present in this section are the results of 10000 time units simulation runs for each specific environment setting as described below. Performance was measured in terms of the average cost reduction per time unit achieved throughout the simulation. For comparison purposes, we also simulated the performance of *FCFS* and *FCFS+* shopbots in each run:

• *FCFS* - samples all the electronic merchants with whom it is familiar for any specific product, based on an *FCFS* rule (as long as the deadline has not elapsed). This is actually the method used by shopbots found on the World-Wide-Web today.

• *FCFS+* - calculates the expected number of queries that can be dedicated for each query: $Q = (capacity/\lambda)$. Then executes queries based on an *FCFS* rule, with reference to $Q$ and the specified deadlines (i.e., if in total $Q$ queries are executed for request $R_i$ then it moves to the next request $R_{i+1}$ according to the order of arrival). Finally, if the temporal capacity allows more queries, then the additional capacity is equally divided among existing requests.

Figure 2 illustrates the effect the parameter $M_i$ (the number of stores in which a product can be found) has on the performance of the shopbots. From the graph we observe that for a relatively small number of information sources all three shopbots exhibit similar performance. This can be explained by the fact that for small $M_i$ values (e.g. $M_i = 2$), the agents do not reach full utilization of their querying capacity (i.e., the agents fully query all potential stores using only part of their available $C$ queries at each time unit). Then, as the number of querying options increases, capacity becomes a constraint and *PlanBot* performs significantly better (both in comparison to the $FCFS$ and the $FCFS+$ shopbots). This is mainly due to the selective nature of our proposed mechanism and its replanning capabilities. It is notable that unlike the $FCFS$ mechanism in which performance significantly drops (due to its obligation to cover all $M_i$ stores for each request), the performance of the $FCFS+$ mechanism remains steady. This is because the latter method manages to equally divide the capacity among the incoming queries thus benefiting from the relatively greater improvement that is associated with the initial queries of each request (see Equation 3).

Figure 3 depicts the performance of the two mechanisms as a

[7]Obviously elements initially assigned with elements from vector $A$ cannot participate in these sequences as these are merely expected values that are not correlated with actual requests.

[8]Measured as the correlation between the cumulative distribution function of both the empirical and the fitted function.

function of the requests' arrival rate $\lambda$. As expected, *PlanBot* outperforms both other shopbots, and the difference between them increases as the value of $\lambda$ increases. This again can be explained intuitively. Any increase in $\lambda$ increases the relative load on the shopbot, thus improves the benefit encapsulated in the re-planning mechanism (enabling an efficient utilization of the querying resources) that is inherent in our mechanism.

Recall that our planning mechanism relies on the vector $A$ (in which each element $A[k]$ represents the estimated expected performance to be achieved by the $k-th$ query in future time). Figure 4 depicts the value of selected elements in the vector $A$ along time (the horizontal axis). Notice that the vector converges to steady values for its different elements in a relatively short time from the beginning of its operation. Once stability is reached, the values remain stable along time, with small fluctuations representing the self adjustment process for local overloads.

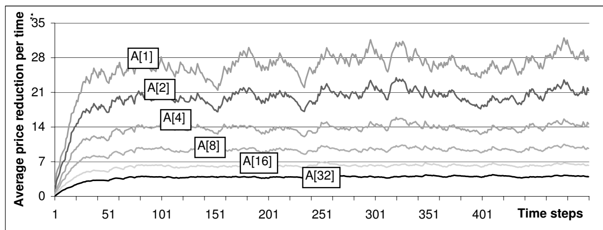


**Figure 4:** The values in the vector $A$ along time (the box on each curve indicates which element in the vector is described)

Finally, we measured the *PlanBot*'s querying capacity required for supporting each level of service that is currently achieved with the *FCFS+* mechanism as a function of the querying capacity used in the latter method. Table 3 depicts the results of this experiment. As we discuss in Section 6 this can serve as an efficient planning tool for the shopbot's operators, which can be used for determining the amount of computational resources needed to operate the shopbot. For example, in order to reach the same level of performance that can be obtained when using 120 requests per time unit in the *FCFS+* mechanism, we only need to support 54 requests per time unit in our proposed mechanism. This can be easily translated into hardware and communication lines monetary equivalent savings.

| FCFS+ Capacity | Performance | *PlanBot*'s Required Capacity |
|---|---|---|
| 20 | 115.05 | 5 |
| 30 | 153.69 | 9 |
| 40 | 193.10 | 14 |
| 50 | 218.12 | 18 |
| 60 | 245.61 | 23 |
| 70 | 264.07 | 28 |
| 80 | 281.04 | 33 |
| 90 | 296.89 | 38 |
| 100 | 309.43 | 43 |
| 110 | 319.39 | 48 |
| 120 | 330.15 | 54 |

**Table 3:** Capacity-based comparison

Additional simulations, executed over randomly generated environments, revealed similar improvement patterns as in the above. In particular, the variance in the initial due-date of the different requests was found to have an insignificant affect on the results (in comparison to using a fixed value that equals the mean).

# 6. EXTENDING APPLICABILITY

In this section we discuss several important extensions of the *PlanBot*'s mechanism. Our mechanism assumes that the shopbot is familiar with (or can learn) the distribution of prices for each requested product at any specific time. This assumption derives from the strong empirical evidence of the persistence of price dispersion, as reviewed in Section 3. Nevertheless, even if the distribution function dynamically changes, the agent can apply simple mechanisms in order to learn this distribution function (e.g. using density and interpolation or Bayesian update techniques [21]). It is notable that the mechanism we use is not affected by the alignment of the distributed prices along the price scale, but rather by the way they are relatively distributed. Thus, the shopbot's performance remains unaffected even if the distribution of prices suddenly shifts on the price scale but maintains its initial structure (e.g., due to a taxation change).

The mechanism can be extended to guarantee any specific level of service (in addition to due date) for every user. For example, a user can be guaranteed a minimum number of price quotes, leaving the decision of how many more quotes to obtain for the specific request to the shopbot. This merely requires a complementary planning phase after completing the routine defined in Section 4 in which we enforce the allocation of the required additional queries for each request according to the same planning logic given in Algorithm 1 (however while removing the condition that the entering query must be associated with a higher marginal benefit in comparison to the removed query).

Similarly, the algorithm can be adapted to maximize any shopbot's revenue function that depends on its performance. For example, consider the scenario where each arriving request specifies, along with the product description and the due-date, the payment the user is willing to pay for the information received (e.g., the user is willing to pay the shopbot 2 percent of any additional price reduction below a price of $20 for the product). If the function defined by the user is linear (or if the marginal payment is a non-increasing function of the marginal price reduction found) then the necessary change is straightforward: we simply need to apply the function specified by the user to the values stored in the vectors of marginal benefit ($V_i$s, taking the initial known price for the product to be fixed - $20 in the example above) and follow the same routine specified in Section 4. For other functions (i.e., non-linear or marginal increasing), we should adjust the assignment algorithm to include assignments of "bulks" of queries that have a single value (that cannot be decomposed) for the shopbot.

Another possible extension of the proposed mechanism relates to caching techniques. Despite the empirical evidence given in Section 3 for the significant variation in firms' relative positions in the distribution of prices over time (and in particular of the low-price firm for the same product over time) we do see room for re-querying cached merchants that offered a low price for the same product in requests processed within the last few time steps (rather than randomly selecting the queried merchants). This requires maintaining a set of beliefs concerning the distribution of prices offered by these merchants given the time elapsed since the last query for this product. Then, the values of the marginal benefit vectors ($V_i$'s) can be re-calculated according to the adjusted distribution functions and the same algorithm suggested in Section 4 can be applied. We estimate that these modifications will result in small changes in the shopbot's performance, since in a world of dynamic prices and pricebots, an electronic merchant's prices are expected to continuously change [10].

Last, our mechanism assumes that the shopbot's users are mainly interested in minimizing their costs thus their utility is not affected by other product attributes (e.g., shipping time, merchant's reputation, branding, reputation, trust, etc.). Assuming an elicitation of the user's multi-attribute utility function (based on these product/merchant attributes) is plausible (a necessary assumption for any algorithm/heuristic aiming to improve performance in this do-

main), a simple adaptation of the algorithm can be suggested: the shopbot would construct the joint distribution of opportunities in the market for any specific product according to the specific user's utility function and would calculate the vectors of marginal benefit ($V_i$s) in terms of utility. All the remaining components of the proposed mechanism would remain unchanged. Obviously there are many complementary aspects that need to be considered (such as enforcing the revelation of the true utility function of each user). Nevertheless, since the purpose of our mechanism is to maximize performance given some expected utility function (either multi-attribute utility or merely cost) these considerations are external to this research. It is notable, however, that while this multi-attribute utility variant is a straightforward and elegant solution, we doubt the relevance of such adaptation, since empirical research has shown that these factors only partially account for the variation in prices charged for homogeneous products [3].

# 7. DISCUSSION AND CONCLUSIONS

Despite the vast interest in shopbots research in recent years, to date the operation of these agents with capacity constraints has not been investigated. As we show in this paper, this problem is unique in the sense that it incorporates similar characteristics of both classical economic search and resource allocation theories, and yet at the same time it does not fully resemble any of them. Our simulation results suggest that there is much room for improvement in shopbots' performance (in comparison to the use of the traditional *FCFS* strategy) in environments where comparison shopping demands may exceed the shopbots' querying capacity. Our proposed *PlanBot*'s implementation is just one example of how performance can be improved by adopting re-planning techniques throughout the shopbot's operational cycle.

The assumption we use, concerning the existence and stationarity of the distribution of prices of each product are well grounded and are supported by empirical research investigating prices in ecommerce markets. Using the distribution of prices, the *PlanBot* manages to estimate the marginal expected price reduction for each query it adds to any specific request in its execution plan. This, along with the complementary allocation tuning mechanism, used for identifying opportunities to delay the execution of beneficial queries for the benefit of immediate execution of others, enables significantly improved performance by the *PlanBot*. The improved performance enables shopbots' operators to supply better levels of service to their users using the same capacity of querying resources. Furthermore, as discussed in the previous section, the mechanism can be adapted to fit various performance measures set by shopbot operators. The performance difference is expected to grow as the number of opportunities found in future markets increases (i.e., as the number of potential merchants for each request grows) and as the demands for shopbot services increases (i.e., as the ratio between requests and capacity grows).

An important advantage of the *PlanBot* is that it constantly updates its beliefs concerning the expected performance associated with future queries. This enables it to bypass the need for modeling the stream of incoming requests and significantly simplifies its re-planning routine.

The extensions suggested in Section 6 improve the applicability of the proposed method. In particular, the ability to maximize some payoff function for the shopbot, can influence and hopefully change the business models upon which comparison shopping agents are currently based.

# 8. REFERENCES


[1] S. Baruah, A. Mok, and L. Rosier. Algorithms and complexity concerning the preemptively scheduling of periodic, real-time tasks on one processor. *Real-Time Systems Journal*, 2:301–324, 1990.

[2] M. Baye, J. Morgan, and P. Scholten. Temporal price dispersion: Evidence from an online consumer electronics market. *J. of Interactive Marketing*, 18(4):101–115, 2004.

[3] M. Baye, J. Morgan, and P. Scholten. Persistent price dispersion in online markets. In D. Jansen, editor, *The New Economy and Beyond*. Edward Elgar Press, 2006.

[4] E. Brynjolfsson, E. Hu, and M. Smith. Consumer surplus in the digital economy: Estimating the value of increased product variety at online bookseller. *Management Science*, 49(11):1580–1596, 2003.

[5] S. Choi and J. Liu. Optimal time-constrained trading strategies for autonomous agents. In *Proc. of MAMA'00*, 2000.

[6] K. Clay, R. Krishnan, E. Wolff, and D. Fernandes. Retail strategies on the web: Price and non-price competition in the online book industry. *Journal of Industrial Economics*, 50:351–367, 2002.

[7] L. Cucu, R. Kocik, and Y. Sorel. Real-time scheduling for systems with precedence, periodicity and latency constraints. In *Proc. of RTS'02*, 2002.

[8] E-consultancy. Shopping comparison engines - a buyer's guide. http://www.e-consultancy.com/, 2006.

[9] S. Gal, M. Landsberger, and B. Levykson. A compound strategy for search in the labor market. *International Economic Review*, 22(3):597–608, 1981.

[10] A. Greenwald and J. Kephart. Shopbots and pricebots. In *IJCAI '99*, pages 506–511, 1999.

[11] R. Guttman, A. Moukas, and P. Maes. Agent-mediated electronic commerce: A survey. *Knowledge Engineering Review*, 13(2):147–159, 1998.

[12] M. He, N. R. Jennings, and H. Leung. On agent-mediated electronic commerce. *IEEE Trans. on Knowledge and Data Engineering*, 15(4):985–1003, 2003.

[13] E. Johnson, W. Moe, P. Fader, S. Bellman, and G. Lohse. On the depth and dynamics of online search behavior. *Manage. Sci.*, 50(3):299–308, 2004.

[14] D. Karger, C. Stein, and J. Wein. Scheduling algorithms. In M. J. Atallah, editor, *Handbook of Algorithms and Theory of Computation*. CRC Press, 1997.

[15] J. Kephart and A. Greenwald. Shopbot economics. *JAAMAS*, 5(3):255–287, 2002.

[16] R. Lipton and A. Tomkins. Online interval scheduling. In *SODA*, pages 302-311, 1994.

[17] C. Martel. Preemptive scheduling with release times, deadlines, and due times. *J. ACM*, 29(3), 1982.

[18] J. McMillan and M. Rothschild. Search. In *Handbook of Game Theory with Economic App.*, pages 905–927. 1994.

[19] S. Nahmias. *Production and Operations Analysis*. The McGraw-Hill Companies, Inc., $3^{rd}$ edition, 1997.

[20] C. Poon, F. Zheng, and Y. Xu. On-demand bounded broadcast scheduling with tight deadlines. In Proc. of *CRPITS'51*, pages 139–143, 2006.

[21] M. Rothschild. Searching for the lowest price when the distribution of prices is unknown. *Journal of Political Economy*, 82:689–711, 1974.

[22] J. Sgall. On-line scheduling - a survey. Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1997.

[23] M. Smith. The impact of shopbots on electronic markets. *Journal of the Academy of Marketing Science*, 30(4):442–450, 2002.

[24] G. Stigler. The economics of information. *Journal of Political Economy*, 69(3):213–225, 1961.