A Gradient-Based Reinforcement Learning Approach to Dynamic Pricing in Partially-Observable Environments

David Vengerov

A Gradient-Based Reinforcement Learning Approach to Dynamic Pricing in Partially-Observable Environments

David Vengerov

SMLI TR-2007-169 August 2007

Abstract:

As more companies are beginning to adopt the e-business model, it becomes easier for buyers to compare prices at multiple sellers and choose the one that charges the best price for the same item or service. As a result, the demand for the goods of a particular seller is becoming more unstable, since other sellers are regularly offering discounts that attract large fractions of buyers. Therefore, it becomes more important for each seller to switch from static to dynamic pricing policies that take into account observable characteristics of the current demand and the state of the seller's resources. This paper presents a Reinforcement Learning algorithm that can tune parameters of a seller's dynamic pricing policy in a gradient direction (thus converging to the optimal parameter values that maximize the revenue obtained by the seller) even when the seller's environment is not fully observable. This algorithm is evaluated using a simulated Grid market environment, where customers choose a Grid Service Provider (GSP) to which they want to submit a computing job based on the posted price and expected delay information at each GSP.

Sun Labs
16 Network Circle
Menlo Park, CA 94025

email address:
david.vengerov@sun.com

For information regarding the SML Technical Report Series, contact Jeanie Treichel, Editor-in-Chief <jeanie.treichel@sun.com>.All technical reports are available online on our website, http://research.sun.com/techrep/.

A Gradient-Based Reinforcement Learning Approach to Dynamic Pricing in Partially-Observable Environments

David Vengerov
Sun Microsystems Laboratories
UMPK16-160
16 Network Circle
Menlo Park, CA 94025
david.vengerov@sun.com

Abstract

As more companies are beginning to adopt the e-business model, it becomes easier for buyers to compare prices at multiple sellers and choose the one that charges the best price for the same item or service. As a result, the demand for the goods of a particular seller is becoming more unstable, since other sellers are regularly offering discounts that attract large fractions of buyers. Therefore, it becomes more important for each seller to switch from static to dynamic pricing policies that take into account observable characteristics of the current demand and the state of the seller's resources. This paper presents a Reinforcement Learning algorithm that can tune parameters of a seller's dynamic pricing policy in a gradient direction (thus converging to the optimal parameter values that maximize the revenue obtained by the seller) even when the seller's environment is not fully observable. This algorithm is evaluated using a simulated Grid market environment, where customers choose a Grid Service Provider(GSP) to which they want to submit a computing job based on the posted price and expected delay information at each GSP.

.

1 Introduction

With a rise in Grid computing offerings, a Grid infrastructure and economy are emerging. The Sun Grid Compute Utility (www.network.com) is one of the first examples of a Grid Service Provider (GSP), which allows customers to submit their jobs and get them executed for $\$1$ per CPU-hour. A competitive marketplace will soon emerge, where GSPs accept and execute customer jobs. Customers will have the means to compare rates and submit jobs to the GSP that offers them the best value for the money. Meanwhile, the GSPs will be trying to maximize their profits. If the price charged by a GSP is too low, its profit will be suboptimal. If the price is too high, the GSP will get very few customers and will have a suboptimal profit once again.

Rather than finding an optimal static price to be charged by GSPs, we focus on *learning* an *optimal* dynamic job pricing policy for profit maximization. Dynamic pricing was difficult before the age of the Internet, since collecting all necessary information and disseminating the new prices to all distributors, retailers and sales people could take several months for a large company. The advent of the e-business capabilities have reduced the information collection and price dissemination costs to zero in many cases, and many Internet sellers are beginning to compute the right price for each customer at any time. As a result, the prices charged by the sellers are approaching the true market value of the goods/services being offered. At the same time, the market price itself is becoming more dynamic as it adjusts to new market information affecting both buyers and sellers. Therefore, the static pricing models are now becoming even less efficient than they were before, and the need for adjusting the prices to changes in the business environment and the internal state of the business is steadily increasing.

There are two broad categories of dynamic pricing approaches: the ones using *posted price* and the ones based on *price discovery* [2]. In the posted price approach a seller posts a price for an item/service (which can be changed over time), and a buyer either purchases the item/service for that price or leaves. In the price discovery approach the buyers use some kind of an auction to determine the price at which the item/service will be purchased. The posted price approach is more directly applicable to determining the price at which each arriving customer can submit a job to a computing facility, and so we use the

term *dynamic pricing* within the posted price paradigm.

A recent taxonomy of market-based resource management systems in utility computing clusters/Grids [11] suggests that only the Libra [13] grid architecture uses a dynamic job pricing approach based on the state of readily observable information (i.e., status of the Grid resources). The price charged for a job i on a Grid node is a weighted sum of a static "base" price for using that node and of the expected resource utilization on that node if job i were assigned to it. However, the authors in [13] do not specify how the weights are set in the formula described above. Appropriate context-dependent setting of these weights is crucial to the success of a dynamic pricing policy, and the focus of the current paper is on *learning* the *optimal* parameters in a pricing policy that maps observable information into the price that should be charged for a particular job.

Reinforcement Learning (RL) is a recently developed methodology (e.g., [1, 7]) for dynamically tuning decision policies that specify what action should be taken by an agent (e.g., what price should be posted by a GSP) in each observed state of the environment (a state can be described by a particular reading of relevant observed variables, such as the current resource utilization at a GSP, the current backlog of customers, etc.). The agent is assumed to receive periodic feedback related to its actions (e.g., average revenue since the last policy action), and the agent's policy is tuned so as to maximize the expected future value of the feedback signal.

Some researchers have tried applying RL to dynamic pricing in a general retail market. A recent survey of dynamic pricing approaches [4] cites the author's own work (e.g., [5]) and that of the Information Economies project at IBM (e.g., [6, 9]) as the published work that has investigated the use of RL for dynamic pricing in the posted-price paradigm.

Upon a closer investigation of the market model used in [6, 9], we find that it consists of only two sellers, who take turns setting their prices, and each seller is perfectly aware of the other seller's price. This assumption greatly simplifies the dynamics of the model and makes it inapplicable to the Grid market problem addressed in the current paper. A more recent paper [3] extends this model by assuming that the market has an underlying hidden state, which evolves in a Markov fashion conditional on the actions taken by the sellers. However, the RL approach used in that paper assumes that each seller

observes the previous actions of other sellers and uses that information to predict the probability of each competing seller taking a particular action in every state of the market. This assumption is also unrealistic, and even if it were possible to observe previous actions of other sellers, the solution approach would not be scalable to a true market environment with multiple competing sellers.

The authors in [5] use a market model that is closer to ours. Even though they still assume a market with only two sellers, they do not assume that each seller knows anything about the other seller. Each seller in their model has two queues for customers, with one dedicated to "shoppers" who choose the seller in the market with the best price-delay tradeoff, and the other dedicated to "captives" who randomly choose a seller. Each seller can have at most I_{max} items in stock, and when the seller's inventory drops below some level r, the seller orders more items from the warehouse. The authors then use an RL algorithm commonly called *Q-learning* to tune the pricing policies that make decisions based on the seller's state information, which includes the queue lengths for both types of consumers as well as the current amount of inventory. The Q-learning algorithm used by the authors is restricted to discrete state and actions paces, and learns the best discrete price to be used in each discrete state.

We extend the discrete state-action problem formulation by combining RL with a function approximation architecture that generalizes learning experience across similar states and prices, thereby significantly speeding up the learning process in large state-action spaces. Even though Q-learning can also be combined with function approximation architectures, it loses its convergence guarantees in this case even when only one seller is present in the market. We derive a different RL algorithm that can tune the policy parameters in a gradient direction (thus converging to the optimal parameter values) even when the seller's environment is not fully observable.

We are not aware of any gradient-based RL approaches applied to pricing goods/services in a market with multiple sellers. We formulate this problem in a very general form of a multi-agent Partially Observable Markov Decision Process (POMDP) in continuous time and continuous state-action space (the POMDP framework will be described in Section 3.1). The steps we go through in order to solve this hard problem offer a good learning experience for those who wish to tackle similar problems using the RL methodology. Finally, we present experimental results that demonstrate the benefits of RL-tuned

dynamic pricing using the example of a Grid computing market.

2 Problem Formulation

Consider a Grid Service Provider (GSP) facing the market where customers submit requests to execute their jobs at the GSP's computing facility (which has a finite number of multi-CPU servers). There may be more than one GSP present in the market. When a customer arrives with some job k, it specifies the required execution time τ_k for this job and the number of CPUs n_k the job needs. In return, the GSP tells the customer the expected delay for this job and the price P per CPU-hour it wants to charge. If the customer decides to submit the job, it pays $P\tau_k n_k$.

Each customer has an associated utility function $U_k(p, d)$ for each job k (a curve will be randomly selected for each customer during the simulations described in Section 4). After receiving price offers and expected delay information from existing GSPs (the next section explains how a GSP can approximate its expected delay), the customer submits the job to the GSP offering the highest utility (best price-delay tradeoff). This framework allows the customer to try several requests for a given job, changing the requested execution time or the number of CPUs, and then submit the job in the configuration that best matches the customer's understanding of the job's requirements and its price-delay utility tradeoff. The goal of each GSP is to come up with a policy for scheduling the incoming jobs and setting the posted price P so as to maximize its long-term revenue.

A complementary view of utility functions is taken in a job scheduling framework presented in [10], where customers submit jobs to a computing facility and derive a utility from completed jobs that is a decreasing function of the total job processing time (the customers can enter a service level agreement with the provider where the utility function is specified upfront and the customer pays the price that is a product of the obtained utility and the number of CPU-hours taken by the job). The pricing model used in the current paper, where each GSP estimates the total processing time of each job, reduces the uncertainty to which both the customer and the GSP are exposed and is preferable when such estimations can be performed.

3 Solution Approach

This paper presents a simple but effective approach to scheduling incoming jobs and determining the price each job should pay. The basic idea is that of differentiated pricing, which is used in many of today's commercial environments: each GSP defines several service levels for incoming jobs (that differ in the average processing speed for submitted jobs) and charges more money from jobs requesting a higher service level. This allows the GSP to extract more consumer surplus from the market and expand its customer base, as it will get both the "premium" customers who need expedited processing and are willing to pay more for it and the "budget" customers who just need to get their jobs done at the lowest possible cost.

As a simple demonstration of the above idea, we present simulation results for the case when only two service levels are used at each GSP: standard (at $1 per CPU-hour) and premium (at $X per CPU-hour). The jobs requesting each service level form a queue and wait for their turn to be scheduled onto the available CPUs. The premium jobs are scheduled before any of the standard jobs, and if no CPUs are currently available to start executing a premium job, a reservation is made for the nearest time in the future when enough CPUs become available (since customers specify how long each job needs to be executed, it is easy to compute the exact time when a reservation should start and end). Once all premium jobs are either scheduled or assigned some reservations, the standard jobs are scheduled onto the available CPUs so as not to interfere with any existing reservations (this scheduling strategy is often called "backfilling").

The following simple approach (used in our experiments in Section 4) can be used by a GSP to calculate the expected delay for the premium job service class. If each job k in the queue for this class requires n_k CPUs and needs to run for τ_k units of time, then the delay for a new job can be approximated as $(\sum_k n_k \tau_k)/(N \cdot \bar{u})$, where N is the total number of CPUs available at the GSP and $\bar{u}$ is the average utilization of these CPUs when some jobs are waiting. A similar formula was used to compute the expected delay in the standard service class, except that the summation was performed over jobs waiting for both the premium and the standard service.

A very important consideration for a GSP when setting the price X charged for premium jobs is the relative wait time between the standard and the premium job queues. As this time increases, the potential premium customers will be willing to pay a higher price X in order to avoid the increased waiting time in the standard queue. Hence, the GSP should increase X as the relative wait time increases and decrease X otherwise. The exact price X charged for any given value of the relative wait time depends on the level of demand for computing services in the marketplace and on the statistical distribution of the customer profiles (their price-delay utility curves, the number of CPUs and the execution time required by each job).

We propose to use a Reinforcement Learning (RL) methodology for *learning* the optimal pricing policy in a given market environment. That is, the pricing policy of each GSP is represented by a function that maps the current GSP state (as described by the current delays in the premium and in the standard queues, etc.) and some job characteristics (the number of CPUs it requires, the requested execution time, etc.) into the price that should be charged for that job. That function has parameters that could be tuned to adjust the output obtained for a given input. The GSP starts with any given pricing function, and then in the course of learning varies slightly the recommended output (the premium price X) and observes the revenue obtained after each variation. In this manner, the GSP learns the states in which a higher price is more beneficial and those in which a lower price is more beneficial, and then adjusts the parameters of the pricing function so as to minimize the difference between the current output of the pricing function and the desired output. For the benefit of the interested readers, the next section provides a mathematical description of the particular RL methodology used in this paper. That section can be skipped by those interested only in the experimental results.

3.1 Gradient-Based Reinforcement Learning

When several GSPs are present in the market, the probability of a new customer accepting the price offered by a GSP depends on the offers made by other GSPs. Since we cannot assume that GSPs have full information about each other, their decision process is most appropriately modeled using the framework of a Partially Observable Markov Decision Process (POMDP). Formally, a POMDP is defined by:

- *State space S (defining the states of the Grid market)*
- *Action space A (defining the range of prices that each GSP can set in any given state)*
- *Observation space Y (defining the possible information each GSP can observe about the Grid market)*
- *A stochastic real-valued reward signal $r(x, a)$ received by each GSP after it takes action a in state x (the price paid by the customer if the job is submitted and 0 otherwise).*

When an agent (GSP) takes an action a in state x, the probability of transferring to the next state $x' \in S$ is assumed to depend only on a and x (Markov property). The objective of each agent is to learn a policy that specifies an action a for every observation vector y and that maximizes the average reward per time step received by the agent. If the agent's policy is represented by some parameterized function $\mu_\theta(a, y)$ giving the probability of selecting action a when the observation vector y is perceived in state x, then it is possible for the agent to dynamically tune the parameter vector θ so as to maximize the average reward received per time step.

We derive the algorithm for dynamically updating θ from the policy-gradient theorem [8], which gives an expression for the gradient of policy performance with respect to policy parameters in a fully observable MDP (the one where the agent can observe the state x). Before presenting this theorem, we need to present some basic definitions.

The long-term expected reward per time step of a policy specified by $\mu_\theta(a, x)$ is given by:

$$\rho(\theta) = \lim_{T\to\infty} \frac{1}{T} E[\sum_{t=0}^{T-1} r(x_t, a_t)] = \sum_x d^\theta(x) \sum_a \mu_\theta(a, x) r(x, a), \tag{1}$$

where $d^\theta(x)$ is the steady-state probability (assuming it exists) of the Markov process being in state x under policy μ_θ. The value of a particular state-action pair is given by:

$$Q^\theta(x, a) = E[\sum_{t=0}^{\infty} (r(x_t, a_t) - \rho(\theta)) | x_0 = x, a_0 = a]. \tag{2}$$

The policy-gradient theorem then states that

$$\nabla\rho(\theta) = \sum_x d^\theta(x) \sum_a Q^\theta(x,a)\nabla\mu_\theta(a,x) = E_\theta[\sum_a Q^\theta(x,a)\nabla\mu_\theta(a,x)], \tag{3}$$

where ∇ denotes the vector of partial derivatives with respect to components of the parameter vector θ and $E_\theta[.]$ denotes expected value when the state x is sampled from the distribution $d^\theta(x)$, which can be achieved by sampling x in the course of following the policy μ_θ. This theorem also applies to the discounted reward case, with a suitable modification to the definitions of ρ and Q [8]. The gradient expression in the above theorem implies that the following parameter updating procedure is guaranteed to converge to the optimal value of θ:

$$\theta_{t+1} = \theta_t + \gamma_t \sum_a \hat{Q}^\theta(x_t,a)\nabla\mu_\theta(a,x_t), \tag{4}$$

where $\hat{Q}^\theta(x,a)$ is an unbiased estimate of $Q^\theta(x,a)$ and γ_t is a "learning rate" that is decreased over time. The above updating procedure can be simplified by considering the effect of only the action actually taken as opposed to that of all actions in each state x_t:

$$\theta_{t+1} = \theta_t + \gamma_t \hat{Q}^\theta(x_t,a_t)\frac{\nabla\mu_\theta(a_t,x_t)}{\mu_\theta(a_t,x_t)}, \tag{5}$$

where the division by $\mu_\theta(a_t,x_t)$ is used to account for the fact that a particular action a is taken in state x with probability $\mu_\theta(a,x)$, while the update in (4) assumes an equal effect of all actions. The remaining question is how to form $\hat{Q}^\theta(x,a)$, and different gradient-based algorithms can be obtained by using different estimates of $Q^\theta(x,a)$. For example, a well-known William's REINFORCE algorithm [12] can be obtained by using $\hat{Q}^\theta(x_t,a_t) = \sum_{k=t}^T r(x_k,a_k)$, an actually observed sum of reinforcements starting from state x_t until the end of the episode (statistically independent episodes can be defined as a sequences of states between visits to some recurrent state x^*). In that case, the cumulative update to θ at the end of the episode will be $\sum_{t=0}^T \frac{\nabla\mu_\theta(a_t,x_t)}{\mu_\theta(a_t,x_t)} \sum_{k=t}^T r(x_k,a_k)$, exactly what is suggested in [12]. One can re-write this update as $\sum_{t=0}^T r(x_t,a_t) \sum_{k=0}^t \frac{\nabla\mu_\theta(a_k,x_k)}{\mu_\theta(a_k,x_k)}$, which implies that at every time t the parameter vector θ

can be updated as:

$$z_{t+1} = z_t + \frac{\nabla \mu_\theta(a_t, x_t)}{\mu_\theta(a_t, x_t)}, \qquad \theta_{t+1} = \theta_t + \gamma_t r(x_t, a_t) z_{t+1}, \tag{6}$$

where z_{t+1} is set to 0 whenever the recurrent state x^* is visited. When the state space grows large, visits to the recurrent state become very infrequent, and in a realistic time-limited simulation they may not occur at all. Therefore, one needs to use a *biased* estimate of $Q^\theta(x, a)$ in order to make the updates more frequent and decrease their variance. A reasonable approach is to exponentially discount information about previous state-action pairs by letting

$$z_{t+1} = \beta z_t + \frac{\nabla \mu_\theta(a_t, x_t)}{\mu_\theta(a_t, x_t)}, \qquad \theta_{t+1} = \theta_t + \gamma_t r(x_t, a_t) z_{t+1}, \tag{7}$$

where $\beta < 1$ is a positive "discounting factor." This algorithm can also be used in a POMDP environment by substituting y_t instead of x_t in the update of z_t. Under the assumption that the observation y_t in state x_t is drawn from some probability distribution $\xi_y(x_t)$, this algorithm has been proven to converge in [1] to the optimal vector θ^* maximizing the average reward per time step $\rho(\theta)$ as $\beta \to 1$. However, the variance of updates increases as $\beta \to 1$, and in practice it has been observed that $\beta < 1$ leads to the best vector θ^* obtained after a finite number of learning steps. Note that when $\beta = 0$, the above algorithm learns a policy that maximizes the immediate reward received in every state, and as β increases, the learned policy becomes more and more forward-looking, attempting to maximize the discounted sum of future rewards, with rewards further out into the future receiving geometrically less importance.

Some researchers have observed that convergence rate of the algorithm (7) can be increased significantly if $(r(x_t, a_t) - \bar{r}_t)$ is used instead of $r(x_t, a_t)$, where $\bar{r}_t$ is the average reward per time step observed so far. The RL experiments described in Section 4 make use of this idea. Another interesting suggestion is to use $(r(x_t, a_t) - \hat{V}(x_t))$, where $\hat{V}(x_t)$ is an estimated *value* of the state x_t under the current policy, which can be dynamically updated as $\hat{V}(x_t) = \hat{V}(x_{t-1}) + \alpha_t[r(x_{t-1}, a_{t-1}) - \bar{r}_{t-1} + \hat{V}(x_t) - \hat{V}(x_{t-1})]$, where α_t is a decreasing step-size parameter. Note that if the reward function $(r(x_t, a_t) - b(x))$ is used

with $b(x)$ being any baseline function, the convergence properties of the above RL algorithm are not affected. This can be seen by starting with $\sum_a \mu_\theta(a,x) = 1$ and differentiating both sides with respect to θ to obtain $\sum_a \nabla\mu_\theta(a,x) = 0$, which implies that equation (4) still holds if $\hat{Q}^\theta(x_t,a)$ is changed to $[\hat{Q}^\theta(x_t,a) - b(x_t)]$, with $b(x_t)$ carrying through until equation (7).

3.2 Instantiation of RL Methodology

In order to apply equation (7) to a particular problem, one needs to make important design decisions such as: figure out what situations qualify as decision points for the agent, choose the variables that comprise the observation vector y, specify the space of available actions A, decide on the structure of the probabilistic mapping $\mu_\theta(a,y)$ and decide what to use as the reinforcement signal $r(x,a)$.

In the Grid market problem, an agent (GSP) can make a pricing decision when a customer inquires about a price for a particular job. After some experimentation, the following observation variables were found to have a good predictive power for the GSP agent:

- y_1 – the number of CPUs requested by the job multiplied by the requested execution time for the job.
- y_2 – current expected delay in the premium service class
- y_3 – current expected delay in the standard service class

The action space for a GSP agent is the set of available prices it can charge for the job. Therefore, $\mu_\theta(a,y)$ should be a parameterized function that maps two real-valued input variables a and y into a real-valued output: the probability of selecting action a when observation vector y is recorded. As an example, this paper demonstrates how the RL methodology for tuning the parameter vector θ works when

$$\mu_\theta(a,y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(a-a_\theta(y))^2}{2\sigma^2}}, \tag{8}$$

where $a_\theta(y) = \sum_{i=1}^{M} \theta^i \phi^i(y)$ with $\phi^i(y)$ being fixed basis functions. That is, the policy first computes a deterministic "suggested" action $a_\theta(y)$ and then samples the actual action from a Gaussian distribution

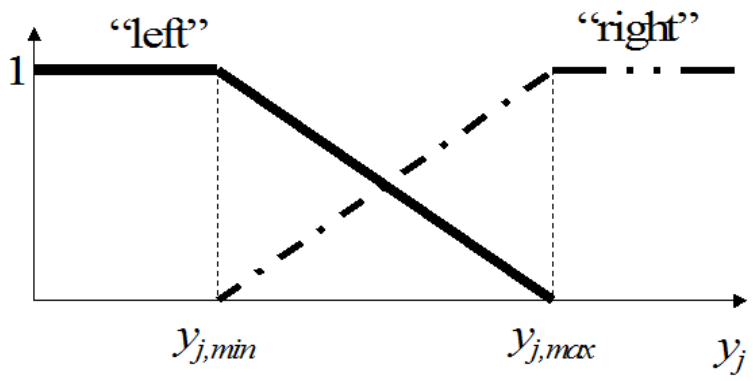

Figure 1. Membership degrees for the "left" and "right" boxes along dimension j.

centered at $a_\theta(y)$ and having the variance of σ. The basis functions are usually chosen to cover the whole state space, and in the simplest case each basis function can be visualized as a "box" in space, so that whenever y belongs to box i, $a_\theta(y) = \theta^i$. In order to make $a_\theta(y)$ a continuous function of y, the boxes need to have "fuzzy" boundaries, so that a given value of y would also belong to some extent to all the neighboring boxes, with the degree of membership (as specified by $\phi^i(y)$) changing continuously as y changes.

Given the choice of components y_j described above, one can expect that the optimal price charged for a given observation vector y should increase monotonically as any particular variable y_j increases while other components of the vector y are kept constant. We can encode this behavior into the structure of $a_\theta(y)$ by specifying that there will be only two boxes for each dimension j, which we will call the "left" and the "right" box. The degree of membership of y_j to the "left" box can then be equal to 1 for $y_j < y_{j,min}$, falling linearly from 1 to 0 over the range $[y_{j,min}, y_{j,max}]$ and staying at 0 for $y_j > y_{j,max}$. Similarly, the degree of membership of y_j to the "right" box can be equal to 0 for $y_j < y_{j,min}$, rising linearly from 0 to 1 over the range $[y_{j,min}, y_{j,max}]$ and staying at 1 for $y_j > y_{j,max}$. These membership degrees are shown graphically in Figure 1. Note that as y_j increases, its degree of membership to the "left" box decreases monotonically from 1 to 0, while its degree of membership to the "right" box increases monotonically from 0 to 1. If we now set $\phi^i(y) = \phi^i_1(y_1)\phi^i_2(y_2)\phi^i_3(y_3)$, where $\phi^i_j(y_j)$ is the degree of membership of y to box i along dimension j, then $a_\theta(y)$ becomes a continuous function of y, as desired.

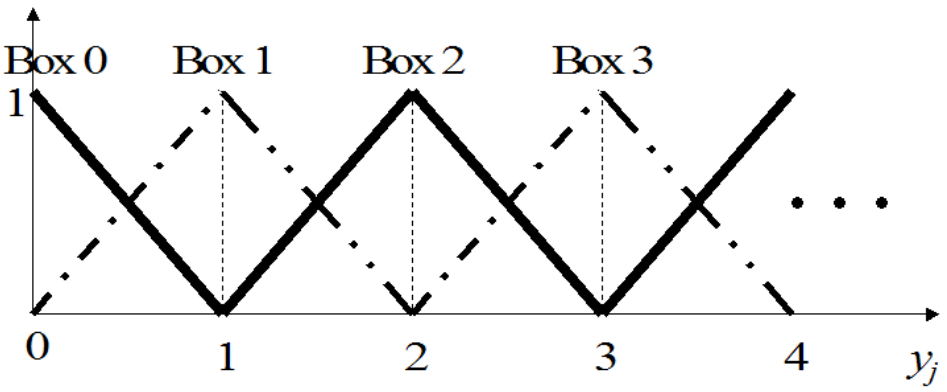

Figure 2. Membership degrees for the multiple boxes along dimensions 2 and 3.

In order for the above approach to work well, the range of possible values of y_j should be covered almost completely by $[y_{j,min}, y_{j,max}]$. Note that if y_j falls outside of this interval and is increased or decreased slightly, then $a_\theta(y)$ would not change. Hence, the interval $[y_{j,min}, y_{j,max}]$ cannot be too narrow. On the other hand, if this interval is too wide, then the degree of membership of y_j to each box (and hence $a_\theta(y)$) will change very little in response to changes in y_j.

Given the definition of y_1, we know that $y_{1,min}$ should be 0. The value of $y_{1,max}$ can be estimated statistically by observing y_1 for some time (we used 1000 time steps for this purpose) and then setting $y_{1,max}$ to be the 95th percentile of the observed values (we found it to be beneficial to ignore some very large outliers). The situation with y_2 and y_3 is more difficult, since the delays in a queuing system can occasionally be very large. As such, the distributions of y_2 and y_3 are skewed to the left but with a heavy tail of outliers to the right. In order to address this situation, we use multiple "boxes" for each of these dimensions, as shown in Figure 2. For example, as y_2 increases from 0 to 2, its degree of membership to "box 1" first increases from 0 to 1 and then decreases back to 0, while its degree of membership to "box 0" decreases from 1 to 0 and then stays at 0 for $y_2 > 1$. In order to ensure that $a_\theta(y)$ increases monotonically with y_j, we can restrict the parameters θ^i so that if box i lies to the right of box k along any particular dimension (while being at the same level with it along the other dimensions), then $\theta^i \geq \theta^k$.

The final important decision that needs to be made is the choice of the reinforcement signal $r(x, a)$ received by a GSP, where x is the unobserved state of the Grid market. At the first glance, it would seem reasonable to let $r(x, a)$ be equal to the total price paid by the submitted job or 0 if the job is not

submitted. However, we don't want the GSP to be rewarded if the asking price for the premium service class was too high and the customer rejected it, but then submitted the job to the standard service class. Therefore, in our experiments a positive reward was given to the reinforcement learning agent managing the premium service class only if the job was submitted to that class.

4 Experimental Results

The Grid market described in Section 2 was simulated so as to evaluate the performance of different job pricing policies. The jobs arrived stochastically with the rate λ, and the arrivals occurring after time t were independent from those occurring before time t (the arrival process was Poisson). The number of CPUs required by each job was sampled from a uniform distribution on [1,24]. Since the total number N of CPUs available at a GSP is not used anywhere in the reinforcement learning algorithm presented in Section 3.1, $N = 24$ was sufficient to test all major features of the scheduler, such as executing multiple jobs in parallel and making reservations for the premium jobs that could not be scheduled immediately. The job execution time was sampled from an exponential distribution on [0,∞], with the average job execution time being 1. The customer utility curve associated with each job k was of the form $U_k(p, d) = 1/[p(1 + d)^{w_k}]$, where w_k was sampled from a uniform distribution on [0.5,1.5].

Figure 3 plots the average revenue of all GSPs (in thousands of dollars) in the market for the case when each GSP provides only the standard service at \$1/CPU-hour and when each GSP also provides the premium service at \$X/CPU-hour, with the value of X having been tuned by hand to maximize the total revenue of each GSPs. In these experiments and in all experiments below, the job pricing policies were evaluated for 4000 time steps, and the standard deviation of each data point was less than 2% of its mean value. In order to keep the demand level the same as the number G of GSPs in the market was varied, the job arrival rate λ was set to $1.3G$. At this arrival rate, a job inquiring for service at a GSP would find on average 5 jobs waiting in the queue, and in about 15% of the case no jobs were waiting. Under these conditions, the optimal integer value of X was 6 for $G \leq 4$, was 5 for $G = 5$, was 4 for $G = 6$ and was 3 for $G = 7$. As one can see, each GSP receives a noticeably larger revenue if it offers the

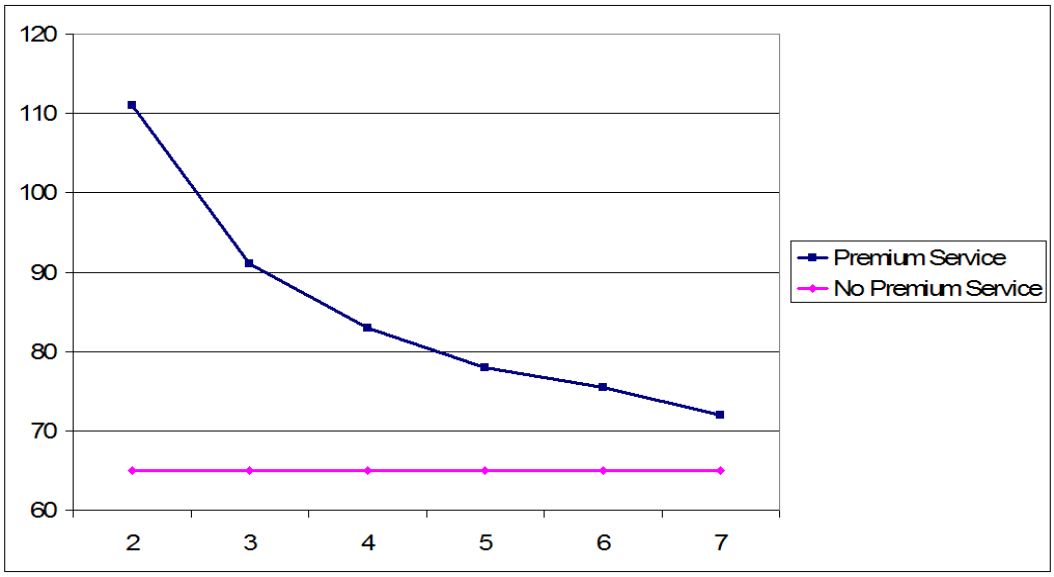

Figure 3. Revenue increase from offering a premium service level

premium service level, consistent with the hypothesis stated at the beginning of Section 3. The decrease in the optimal static price (and hence in the revenue of each GSP using such a policy) as G increases can be explained by the fact that a customer chooses the premium service level at that GSP only if the waiting time for the standard service at ALL other GSPs is large, which becomes progressively less likely as G increases (this is confirmed by the simulation results that show that the number of premium customers served by each GSP decreases as G increases). When all GSPs are offering only the standard service at \$1/CPU-hour, the average revenue of each GSP does not change as G increases because the number of customers increases proportionally (since $\lambda = 1.3G$).

The second set of experiments examined whether a given GSP can receive an even larger revenue with its fixed set of computing resources if it uses dynamic job pricing instead of the optimal symmetric static price (the one that maximizes GSP revenue when all GSPs use this price for their premium service class). The results are presented in Figure 4, where the data points are in percentage terms relative to the optimal symmetric static pricing policy. That is, it shows how the revenue of all GSPs in the market changes if one of them starts using a dynamic pricing policy obtained using the RL methodology described in Sections 3.1 and 3.2.

The methodology consisted of specifying a policy that uses the following input variables: 1) the size

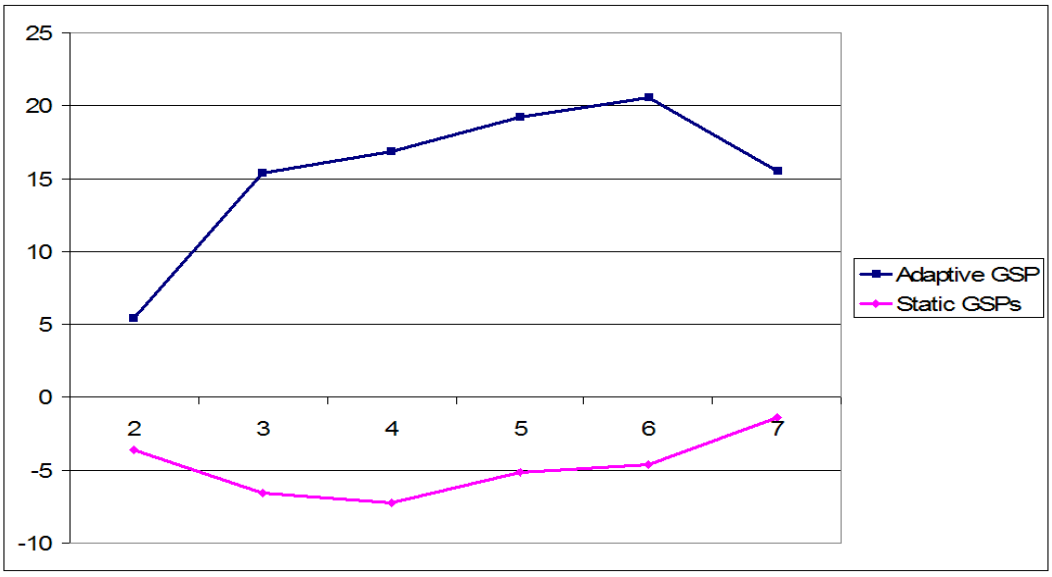

Figure 4. Percentage change in the revenue obtained by the GSP using dynamic pricing and that of the other GSPs relative to symmetric optimal static pricing

of the considered job (the number of CPUs requested multiplied by the requested execution time), 2) the current expected delay in the premium service class, and 3) the current expected delay in the standard service class. The output of the policy is the price to be charged for the considered job. The policy has a tunable parameter vector θ that was dynamically adjusted for 10000 time steps using equation (7) so as to maximize the revenue obtained by each GSP. Each component of the parameter vector θ was initialized to the optimal symmetric static price, so that the adaptive GSP behaved initially just like all other GSPs.

When determining the proper schedule for decreasing the learning rate γ_t (that controls the magnitude of update to vector θ at time t), we found that the standard approach of setting $\gamma_t = 1/t$ decreased γ_t too fast, so that parameters θ_t were not able to reach their final values in 10000 time steps. However, when we used $\gamma_t = 0.01/(1 + t/5000)$, the magnitudes of updates were small enough to make the learning stable but large enough for θ_t to reach the neighborhood of their final values. The variance σ in equation (8) was set to 1. The results in Figure 4 have been obtained over the testing period of 4000 time steps during which no parameter tuning was performed.

Since the learning rate γ_t was always reasonably small, the change in θ was gradual, and correspondingly the average revenue per time step of the adaptive GSP was increasing smoothly during the learning period. In order to have more control over the pricing policy learned by the adaptive GSP, all components of the parameter vector θ were restricted to be greater than 2 (so that the premium price will always be larger than the standard price). After the learning was completed, the smallest component of θ (the one corresponding to the case when no jobs were waiting for service) was always less than 3. However, the largest component (the one corresponding to the case when many jobs were waiting for the standard service) varied from around 12 for 2 GSPs in the market to around 4 for 7 GSPs in the market. This can be explained by the fact that as the number of GSPs increases and the number of customers choosing the premium service level at each GSP decreases (as was noted above), the adaptive GSP has to lower its premium price so as to still attract premium customers. As a result, the adaptive GSP "steals" some jobs from the static GSPs and their revenue decreases, as shown in Figure 4. We should point out, however, that the adaptive GSP learns to lower its premium price only in some cases. For example, we found when $G = 3$ that the adaptive GSP sets its premium price X_t to be less than the optimal static price in about 90% of the cases, but uses the premium price greater than that when the number of queued jobs waiting for service is unusually large.

The hump-shaped form of relative revenues in Figure 4 can be explained by the fact that as G initially increases, the number of *potential* premium customers in the market also increases (since $\lambda = 1.3G$), and thus the adaptive GSP can potentially attract a greater fraction of premium customers by lowering its premium price (as confirmed by the simulation results). As a result, even though each premium customer pays a smaller price per CPU-hour, the increase in the number of premium customers can outweigh the negative effect of the lower premium price. Figure 4 shows that this is indeed the case for $G \leq 7$, but that for $G > 7$ the optimal symmetric price has already become very close to the standard service price, and so the adaptive GSP can no longer undercut other GSPs in price by a significant amount. Also, we have observed experimentally that as G increases, the maximum job queue length in the standard service class for all GSPs decreases (since customers always submit "standard" jobs to the GSP with the smallest wait time in the standard queue, and it becomes more unlikely that ALL GSPs in the market

have long standard queues); thus the premium service class becomes a less attractive alternative relative to the standard service class.

In the third set of experiments all GSPs were using RL to tune the pricing policy. The results showed that when there were 2 GSPs in the market, the revenue of each GSP after learning would surpass the revenue obtained when the optimal static price was used by about 10%. This revenue gain would decrease as the number of GSPs increased, and when 4 or more GSPs were present, their final revenue after learning would be slightly below the revenue obtained when the optimal static price was used. In all cases, GSPs would learn to mostly lower their premium price except for the states corresponding to very large values of the standard queue of customers. The fact that GSPs decreased their revenues somewhat for $G \geq 4$ by jointly lowering their premium price in most of the states does not mean that the RL policy tuning process was not following performance gradient. We confirmed that any GSP significantly increased its revenue if it unilaterally reduced its price below the optimal symmetric static price. This demonstrates the game-theoretic nature of the Grid market, and suggests that the optimal symmetric static price presented in Figure 3 is not a Nash equilibrium, since it is more beneficial for each GSP to "defect" and undercut other GSPs in price. Thus, the observed reduction in revenue of all GSPs has resulted from them sequentially undercutting each other in price.

On the other hand, our experiments suggest that the joint dynamic pricing policy learned by GSPs *is* a Nash equilibrium, since we tried increasing or decreasing the premium price used by any GSP and observed a revenue decrease for that GSP. This hypothesis is further confirmed by the fact that when all GSPs initialized their fuzzy rulebase parameters to 2 (always using the premium price of $2), their final policy after RL tuning (and its performance) was very much the same as the one observed when all GSPs initialized their fuzzy rulebase parameters to the optimal symmetric static price, demonstrating that the RL tuning process is converging to the same policy from different starting points. These results demonstrate the following "theorem" on game-theoretic learning: if each agent adjusts its policy while following the true performance gradient and the multi-agent policy converges, then the resulting multi-agent policy is a Nash equilibrium (a set of policies such that no agent can increase its revenue by unilaterally changing its policy), for if it were not so, then a gradient-following policy-tuning algorithm

would deviate and increase its revenue. The fact that in the case of a "duopoly" (only 2 GSPs in the market) the RL found a Nash equilibrium more profitable than the optimal symmetric static price should not be that surprising, since as we have pointed out earlier, the optimal symmetric static price is not a Nash equilibrium.

5 Conclusions and Future Work

This paper addressed the problem of profit maximization by a seller competing against other unknown sellers in a market environment. A Reinforcement Learning (RL) algorithm for dynamically tuning parameters of a seller's dynamic pricing policy was presented. We showed how this algorithm can be derived from the recently proved "policy-gradient theorem" [8]. After that we showed how this algorithm can actually be used by a Grid Service Provider (GSP), to which customers submit computational jobs for execution. This required us to make the following important decisions: figure out what situations qualify as decision points for the agent (GSP), choose the variables that comprise the observation vector y, specify the space of available actions A, decide on the structure of the mapping $\mu_\theta(a, y)$ that specifies the probability of selecting an action a after receiving the observation vector y, and decide what to use as the reinforcement signal $r(x, a)$. The steps we went through in order to apply the presented RL algorithm to the Grid market problem offer a good learning experience for those who wish to tackle other practical problems using the RL methodology.

A simulated Grid market was used to evaluate the benefit of using the presented RL algorithm for tuning dynamic pricing policies used by GSPs. The experimental results showed that each GSP can significantly increase its profit by offering a premium service level (where a higher price is charged for a smaller expected job delay) in addition to the standard one. The optimal symmetric static price X^* charged by each GSP for its premium service level was shown not to be a Nash equilibrium, since any particular GSP can raise its profits by deviating from it and undercutting other GSPs in price. We have demonstrated this behavior by letting one GSP use a dynamic pricing policy tuned with RL while the other GSPs kept using the optimal static price, and this one GSP obtained a significantly higher profit

than the others. We have also shown that if all GSPs simultaneously use the presented RL algorithm to tune their dynamic pricing policies, then the multi-agent learning process converges to a Nash equilibrium, which is more profitable than the optimal static price for the case of 2 GSPs in the market (duopoly) and slightly less profitable when there were 4 or more GSPs in the market. With just a change of names, the Grid market model considered in this paper can be applied to many other multi-seller environments, such as pricing of goods/services in any retail market.

In order to give a more objective picture of the presented RL methodology for tuning dynamic pricing policies, we should mention that it is expected to converge to the optimal policy only if the environment remains stationary (the probability distributions from which random variables sampled do not change). However, if the customer arrival rate or the distribution of customer utility curves suddenly changes, then parameter tuning should be resumed. Thus, in a real deployment, the RL methodology presented in this paper should be inserted into a meta-framework that observes the important customer characteristics and resumes tuning of the pricing policy if a change in these characteristics is detected. Moreover, the previously learned policies should be saved, so that if the environment returns to the regime similar to one that was previously observed, the GSP can start its tuning from a previously saved policy. Such meta-frameworks have been conceptually analyzed in the field of case-based reasoning and are a subject of our future research.

References

[1] J. Baxter and P.L. Bartlett, "Infinite-Horizon Policy-Gradient Estimation," *Journal of Artificial Intelligence Research*, Vol. 15, pp. 319-350, 2001.

[2] W. Elmaghraby and P. Keskinocak, "Dynamic Pricing in the Presence of Inventory Considerations: Research Overview, Current Practices and Future Directions," *Management Science*, Vol. 49, No. 10, 1287-1309, October 2003.

[3] C. Li, H. Wang and Y. Zhang, "Dynamic Pricing Decision in a Duopolistic Retailing Market," In Proceedings of the Sixth World Congress on Intelligent Control and Automation (WCICA), pp:

6993 - 6997, June 2006.

[4] Y. Narahari, C.V.L. Raju, K. Ravikumar and S. Shah, “Dynamic pricing models for electronic business.” *Sadhana* Vol. 30, Part 2 & 3, pp. 231-256, April/June 2005.

[5] C.V.L. Raju, Y. Narahari, and K. Ravi Kumar. “Learning dynamic prices in multi-seller electronic markets with price sensitive customers, stochastic demands, and inventory replenishments.” *IEEE Transactions on Systems, Man, and Cybernetics, Part C*. Special Issue on Game-theoretic Analysis and Stochastic Simulation of Negotiation Agents, Volume 36, Number 1, pp. 92-106, January 2006.

[6] M. Sridharan and G.J. Tesauro, “Multi-agent Q-Learning and Regression Trees for Automated Pricing Decisions,” In Proceedings of the Seventh International Conference on Machine Learning, pp. 927-934, 2000.

[7] R. Sutton, A.G. Barto, *Reinforcement Learning: An Introduction.* MIT Press, 1998.

[8] R.S. Sutton, D. McAllester, S. Singh, and Y. Mansour, “Policy Gradient Methods for Reinforcement Learning with Function Approximation.” In Advances in Neural Information Processing Systems 12, pp. 1057-1063, 2000.

[9] G.J. Tesauro and J.O. Kephart, “Pricing in Agent Economies with Multi-Agent Q-learning,” In Proceedings of Workshop on Decision Theoretic and Game Theoretic Agents. University College London, London, July 6, 1999.

[10] D. Vengerov, “Adaptive Utility-Based Scheduling in Resource-Constrained Systems.” Presented at the 18th Australian Joint Conference on Artificial Intelligence, December 5-9, Sydney, Australia, 2005. Published in Springer-Verlag Lecture Notes in Computer Science, Vol. 3809, pp. 477-488, 2005.

[11] S. Venugopal, R. Buyya, and K. Ramamohanarao, “A Taxonomy of Data Grids for Distributed Data Sharing, Management and Processing,” *ACM Computing Surveys*, Vol. 38, No. 1, pp. 1-53, March 2006.

[12] R. J. Williams, “Simple statistical gradient-following algorithms for connectionist reinforcement learning,” *Machine Learning*, Vol. 8, pp. 229-256, 1992.

[13] C.S. Yeo and R. Buyya, “Pricing for Utility-driven Resource Management and Allocation in Clusters,” In Proceedings of the 12th International Conference on Advanced Computing and Communication (ADCOM), 2004.

6 About the Author

David Vengerov is a staff engineer at Sun Microsystems Laboratories. He is the principal investigator for the Adaptive Optimization project developing and implementing self-managing and self-optimizing capabilities in computer systems. His primary research interests include Utility and Autonomic Computing, Reinforcement Learning Algorithms, and Multi-Agent Systems. He holds a Ph.D. in Management Science and Engineering from Stanford University, an M.S. in Engineering Economic Systems and Operations Research from Stanford University, an M.S. in Electrical Engineering and Computer Science from MIT and a B.S. in Mathematics from MIT.