Monitoring an Information Source under a Politeness Constraint

Jonathan Eckstein
Department of Management Science and Information Systems and RUTCOR, Rutgers University, 640 Bartholomew Road, Piscataway, NJ 08854 USA, jeckste@rutcor.rutgers.edu

Avigdor Gal, Sarit Reiner
Faculty of Industrial Engineering and Management, Technion - Israel Institute of Technology, Technion City, Haifa 32000, Israel, {avigal@ie,reiner@technion}.technion.ac.il

We describe scheduling algorithms for monitoring a single information source whose contents change at times modeled by a nonhomogeneous Poisson process. In a given time period of length $T$, we enforce a server-side politeness constraint that we may only probe the source at most $n$ times. This constraint, along with an optional constraint that no two probes may be spaced less than $\delta$ time units apart, is intended to prevent the monitor from being classified as a nuisance to be “locked out” of the information source.

To develop our algorithms, we use a portion of the cost model developed in our earlier work (Gal and Eckstein, 2001). Our first algorithm assumes a discrete set of $N > n$ possible update times, and uses dynamic programming to identify a provably optimal subset of $n$ of these times at which to probe the server. Our second algorithm is a simple direct search procedure for locally improving any continuous-time schedule with respect to the same cost model. In particular, this improvement procedure may be applied to the schedule obtained from our first algorithm. We evaluate our algorithms using real-world data feeds.

*Key words:* database management systems; Web monitoring, dynamic programming

*History:* received June 2005; revised February 2006, July 2006; accepted August 2006

1. Introduction

Consider a client maintaining a “mirror” or copy of a single information source maintained by a server. The information source could be defined, for example, as a mailbox, a database table, a single Web page, an entire Website, or a subset of a Website. We concentrate on “pull” environments in which the client must monitor the server for information changes, as opposed to “push” environments in which the server alerts clients about updates. The client monitors via *probes*, each sufficient to detect any change in the information source.

The tradeoffs between push- and pull-based data delivery have been studied extensively; for example, see Franklin and Zdonik (1998) for a detailed discussion. Push protocols have significant advantages, but unfortunately are not always cost-effective or technically feasible, since information servers are often unable or unwilling to support them. For example, Pandey et al. (2004) argue that Web source monitoring must generally be pull- rather than push-based, due either to server noncooperation or the difficulty of maintaining large numbers of
client profiles. Applying push protocols requires sufficient incentive to service clients and a small enough number of clients to make the task manageable; otherwise, pull protocols, not requiring active server cooperation, are far more likely to be implemented. Pull protocols also have the advantage of not requiring clients to register with the server, and freeing the server from the task of tracking client locations. Applications calling for periodic pull-based monitoring and transcription include continuous queries (Pandey et al., 2003), stream data monitors, Web cache managers, Web crawlers, and pervasive systems. Data monitoring quality measures in such settings may reflect the amount of captured information, termed completeness (Pandey et al., 2004), or the delay in propagating source data updates, which has been variously termed timeliness (Pandey et al., 2004), obsolescence (Gal and Eckstein, 2001), or age (Cho and Garcia-Molina, 2000).

When using pull-based policies, one must either constrain the number of probes, or attach some kind of client cost to each probe. Otherwise, the client will probe continually, wasting network resources and risking shutdown as a presumed network attacker. This paper is essentially a follow-up to the numerical experiments of Gal and Eckstein (2001), proposing more rigorous, higher-performance probe scheduling algorithms that more effectively exploit the structure of the same cost model. In particular, we introduce probe scheduling algorithms that operate under a simple server-side poldeness constraint: over the entire planning period \([0, T]\), we may probe the server at most \(n\) times for a given information source (e.g., a Web page). This kind of constraint is difficult to incorporate into the scheduling heuristics of Gal and Eckstein (2001), but is used in practice by Websites like the British Telecom telephone directory (http://www.bt.com). It may also be used in many other applications, such as Web cache collaboration and peer-to-peer communication. In addition, we introduce an optional second constraint that no two probes can be spaced more closely than \(\delta \geq 0\) time units apart. Under these constraints, probes should be scheduled to minimize obsolescence.

Our cost model is essentially the “insertion” model of Gal and Eckstein (2001), which takes into account two different time-varying factors: update intensity and user importance. By update intensity, we mean natural time-of-day and day-of-week variation in the frequency of changes to the information source. For example, a Website may tend to be updated primarily during working hours, or mainly on weekday evenings. We model this kind of variation via a nonhomogeneous Poisson process with periodic arrival intensity function \(\lambda(\cdot)\). Algorithms that take advantage of update intensity patterns are likely to have better average performance than simple periodic polling algorithms, as proposed for example in Naughton et al. (2001).

“User importance” is meant to reflect that up-to-date information has greater value to the consumer at some times than at others, perhaps because of a user query process that
tends to be more active at particular times, or for other reasons. For example, someone may query a database more frequently or assign higher importance to receiving new e-mails during work hours than after hours. Similarly, timely monitoring of an online auction may become increasingly important towards the end of the bidding period. Such client-side time heterogeneity might follow a similar pattern to update intensity, or it might follow an entirely different pattern.

Section 2 reviews the obsolescence cost model of Gal and Eckstein (2001), shows that it has an alternative interpretation using the notion of a query process, and then formulates our main optimization problem. We develop two principal algorithms for this model. For the first algorithm, developed in Section 3, we suppose that updates may only be performed at a discrete set of times $T$ of cardinality $N > n$. Using dynamic programming, we calculate an optimal solution to this additionally-constrained version of the model in polynomial time. We also prove an $O(1/N)$ bound on how much the discretized problem’s optimal cost can differ from the original problem’s.

Our second algorithm, presented in Section 4, dispenses with the discrete-time constraint, but does not calculate a provable global optimum. Instead, it is a specialized, heuristic direct search procedure, using coordinate descent moves to iteratively improve a given solution. We also experimented with classical nonlinear programming methods for the same purpose, but the peculiar features of our cost function prevented them from being useful.

We claim that very good solutions to the probe scheduling problem can be obtained by applying our first algorithm to a sufficiently fine-grained time grid, and fine-tuning them via our second algorithm. In support of this hypothesis, we present experiments using two real-world data feeds.

Our general approach should be useful not only for the particular cost model we develop, but for any pull-based monitoring system whose obsolescence cost model has a simple additive structure; see equation (3) below. Only one subroutine in our method, Algorithm 3, strongly depends on the particulars of our cost model. For a different application, one could simply replace this subroutine with a different one. In that case, however, the $O(1/N)$ error bound might require a modified analysis or fail to hold.

Section 6 briefly describes some related prior work in the information systems literature, including recent work on data warehousing (Dey et al., 2006) and Web crawling (Wolf et al., 2002). To close the paper, Section 7 then offers conclusions and considers how one might generalize our work to schedule monitoring of multiple sources, where it is not appropriate to simply combine individually optimal schedules for each source.
2. Modeling the Problem

Consider the time period $[0, T]$, and let $\{p_i\}_{i=1}^n$ be the times the client probes the server, synchronizing its replica with the information source. We define $p_0 = 0$, $p_n = T$, and require that $0 \leq p_1 \leq p_2 \leq \cdots \leq p_{n-1} \leq T$.

As in a variety of prior work (Cho and Garcia-Molina, 2000; Gal and Eckstein, 2001; Lee et al., 2002), we model updates to an information source at the server via a Poisson process. The independent and memoryless properties of Poisson processes seem plausible for servers drawing from widely scattered sources, such as e.g., incoming emails, postings to newsgroups, or posting of orders from independent customers. However, we do need to capture natural time-variability trends, such as more emails arriving during work hours and more bids being posted towards the end of an auction. Like Gal and Eckstein (2001) and Gal et al. (2003), we use a nonhomogeneous Poisson process with instantaneous arrival rate $\lambda : \mathbb{R} \rightarrow [0, \infty)$; see for example Ross (1980) or Taylor and Karlin (1994). The number of update events occurring in any interval $(s, f]$ is a Poisson random variable with expected value $\Lambda(s, f) = \int_s^f \lambda(t) \, dt$. We assume that $\lambda(\cdot)$ is a nonnegative integrable function bounded above on $[0, T]$. To simplify some of the analysis below, we extend the definition of $\Lambda(s, f)$ to the case $s > f$ through the standard convention for reversed integrals: $\Lambda(s, f) = -\Lambda(f, s)$.

Let $C(s, f)$ denote the obsolescence cost attributable to updates occurring at the server during the time interval $(s, f]$, assuming that probes occur at $s$ and $f$, and no intervening times. Then the cost incurred between probes $i - 1$ and $i$ is $C(p_{i-1}, p_i)$, and the total obsolescence cost of the probe schedule $(p_0, p_1, \ldots, p_n)$ is $C(p_0, p_1, \ldots, p_n) = \sum_{i=1}^n C(p_{i-1}, p_i)$. Define $R(s, f)$ to be the (random) set of times updates occur during $(s, f]$, and let $c(r, f)$ denote the cost of an update occurring at time $r$ and transmitted at time $f$. Then it is natural to define $C(s, f) = \sum_{r \in R(s, f)} c(r, f)$. The particular form we select for $c(r, f)$ is $c(r, f) = \int_r^f a(\tau) \, d\tau$, where $a : [0, T] \rightarrow [0, \infty)$ is an arbitrary nonnegative importance function: an update accumulates cost at rate $a(\tau)$ from inception until transcription to the client. In particular, if $a(\cdot)$ is constant, so $a(\tau) = a_1$ for all $\tau$, then $c(r, f) = a_1 \cdot (f - r)$ is simply proportional to the time until transcription.

One way to conceptualize the role of $a(\cdot)$ is to suppose that, in addition to the update stream generated by a nonhomogeneous Poisson process at the client, there is also a query stream of discrete times information is used at the client, much as in traditional database modeling research — for example Chandy et al. (1975) and Hoffer and Severance (1975), as well as more recent investigations such as Dey et al. (2006). In particular, if we assume that the query process is nonhomogeneous Poisson with intensity $a(\cdot)$, then $c(r, f)$ is precisely the expected number of queries that will occur during $(r, f]$. Thus, we may think of $c(r, f)$
as being the expected number of queries in which the update is present at the server but missing at the client.

Alternately, as originally presented in Gal and Eckstein (2001), one can take the viewpoint that the data are in continuous use at the client, as for example in the case of a continuous query (Pandey et al., 2003). In this case, one might want to leave \( a(\cdot) \) constant, or one might select a nonconstant \( a(\cdot) \) to reflect that obsolescent data are more costly at some times than at others. For example, in this paper’s computational experiments, we set

\[
a(t) = \begin{cases} 
  a_1, & \text{if } t \text{ is during work hours} \\
  a_2, & \text{otherwise},
\end{cases}
\]

where \( a_1 > a_2 \). In this case, \( c(r, f) \) is \( a_1 \) times the amount of work time accumulated until transcription, plus \( a_2 \) times the amount of non-work time. In the simple case of (1), we may refer to \( a_1/a_2 \) as the preference ratio, rather than specifying \( a_1 \) and \( a_2 \) individually. We stress, however, that we use the specific form (1) for demonstration purposes only. As far as our algorithms are concerned, the only restrictions on \( a(\cdot) \) are that it be nonnegative, integrable, and bounded above on \([0, T]\).

In accordance with the terminology in Gal and Eckstein (2001), we refer to \( C(s, f) = \sum_{r \in R(s, f)} c(r, f) \) as the “actual” cost of the probe interval \((s, f]\). However, as each \( c(r, f) \) may be considered to be an expected cost with respect to a query process, we may also think of \( C(s, f) \) as being an expectation with respect to the query process, conditioned on the set of update times \( R(s, f) \) in \((s, f]\).

Since the updates are generated by a stochastic process, the \( C(p_{i-1}, p_i) \) and thus their sum \( C(p_0, p_1, \ldots, p_n) \) are random variables. For any \( s \leq f \), we define \( \overline{C}(s, f) \) to be the expected value of \( C(s, f) \), and note that Gal and Eckstein (2001) prove that

\[
\overline{C}(s, f) = \int_s^f \lambda(t) \left( \int_t^f a(\tau) \, d\tau \right) \, dt. 
\]

Note that if \( a(\cdot) \) is a constant function, then \( \overline{C}(s, f) \) reduces up to a constant factor to \( \int_s^f \lambda(t)(f - t) \, dt \), which is equivalent to a standard “delay” cost model; see Sia and Cho (2005) for an example. If \( \lambda(\cdot) \) is also constant, \( \overline{C}(s, f) \) further reduces within a constant factor to \( (1/2)(f - s)^2 \), which is equivalent to the standard cost model of numerous prior authors, including Dey et al. (2006). Defining \( \overline{C}(p_0, p_1, \ldots, p_n) \) to be the expected value of \( C(p_0, p_1, \ldots, p_n) \), the definition of \( C(p_0, p_1, \ldots, p_n) \) leads to

\[
\overline{C}(p_0, p_1, \ldots, p_n) = \sum_{i=1}^n \overline{C}(p_{i-1}, p_i). 
\]

We now present a new, alternative way of obtaining (2) through the notion of a query process. Consider two consecutive probe times \( s < f \) and imagine a query occurring at some
fixed time $q \in (s, f]$. Define the cost $u(q, s)$ of such a query to be the number of updates that have occurred since $s$, but have not yet been transcribed to the client by time $q$, that is, $u(q, s) = |R(s, q)| = |R(s, f) \cap (s, q]|$. Now, $u(q, s)$ is a random variable, but its expectation is simply the expected number of updates in $(s, q]$, that is, $E_{R(s, f)}[u(q, s)] = \Lambda(s, q)$. Now, denote the set of times at which queries occur during $(s, f]$ by $Q(s, f)$, and suppose we are interested in the total cost $U(s, f) = \sum_{q \in Q(s, f)} u(q, s)$ attributable to these queries. This total cost is a random variable because $Q(s, f)$ and $R(s, f)$ are generated by stochastic processes. The following proposition states that counting the cost by query as embodied by $U(s, f)$ is equivalent in the expected value sense to counting cost by update, as embodied in $C(s, f)$.

**Proposition 1** If the query and update processes are independent, then for any $s \leq f$,

$$E[U(s, f)] = \int_s^f a(\tau)\left(\int_t^\tau \lambda(t) \, dt\right) \, d\tau = \int_s^f \lambda(t) \left(\int_t^f a(\tau) \, d\tau\right) \, dt = \overline{C}(s, f).$$

All necessary proofs of our lemmas and propositions may be found in the online supplement. Proposition 1 implies that the continuous query $\overline{C}(s, f)$ cost model of Gal and Eckstein (2001) is equivalent under a natural assumption to the $E[U(s, f)]$ cost model derived from a nonhomogeneous Poisson query process with intensity $a(\cdot)$, the cost of each query being the number of untranscribed updates when it occurs. In our computational experiments below, we will take the $C(s, f)$ point of view, and refer to $C(p_{i-1}, p_i)$ as the “actual” cost. From the other viewpoint, $C(p_{i-1}, p_i)$ is an expected cost with regard to the query process, conditioned on an observed sample path for the update process. From either viewpoint, the overall expected cost $\overline{C}(p_{i-1}, p_i)$ for each probe interval $(p_{i-1}, p_i]$ is identical.

An optimal probing schedule is a choice of $p_i$ that minimizes the expected obsolescence cost $\overline{C}(p_0, p_1, \ldots, p_n)$, that is, a solution to the optimization problem

$$\begin{align*}
\min & \quad \sum_{i=1}^n \overline{C}(p_{i-1}, p_i) \\
\text{S.T.} & \quad 0 = p_0 \leq p_1 \leq p_2 \leq \cdots \leq p_{n-1} \leq p_n = T,
\end{align*}$$

(4)

where $\overline{C}(p_{i-1}, p_i)$ is defined by (2). We call (4) the polite probe scheduling problem.

We can also imagine adding to (4) the constraint that no two probes can be closer than $\delta$ time units apart, where $\delta \geq 0$ is some given constant. We then obtain the model

$$\begin{align*}
\min & \quad \sum_{i=1}^n \overline{C}(p_{i-1}, p_i) \\
\text{S.T.} & \quad p_0 = 0 \\
& \quad p_i - p_{i-1} \geq \delta \quad i = 1, \ldots, n \\
& \quad p_n = T,
\end{align*}$$

(5)

which is identical to (4) when $\delta = 0$. 

6
3. Dynamic Programming Algorithm in Discrete Time

Unfortunately, the cost function in (4) and (5) has large numbers of local minima, and for simple, nonsmooth forms of \( a(\cdot) \) and \( \lambda(\cdot) \) is also nondifferentiable, having many points where derivatives with respect to the \( p_i \) do not exist. Figure 1 illustrates the case \( n = 2 \), plotting \( \overline{C}(p_0, p_1, p_2) \) (on the vertical axis) as a function of \( p_1 \in [0, T] \) (on the horizontal axis), where we fix \( p_0 = 0 \) and \( p_2 = T \). Here, \( T = 28 \) days, \( a(\cdot) \) has the form (1) with \( a_1/a_2 = 3 \), and \( \lambda(\cdot) \) is estimated from the DBWorld dataset we will describe in Section 5 (see Table 1). Standard continuous optimization techniques do not apply to problems of this form.

![Figure 1: Sample graph of \( \overline{C}(p_0, p_1, p_2) \) as a function of \( p_1 \).](image)

To avoid nondifferentiability, one can choose \( \lambda(\cdot) \) to be a smooth function, and adopt a smooth form of \( a(\cdot) \), as opposed to the discontinuous example given in (1). These adjustments result in \( \overline{C}(s, f) \) having formal derivatives with respect to \( s \) and \( f \), and give (4) a differentiable objective function. This approach will “smooth out” the sharp corners of the objective function \( \overline{C}(p_0, p_1, \ldots, p_n) \) as illustrated in Figure 1, but only locally. The large numbers of local minima and the general “wiggly” shape of the cost function remain. As we have empirically verified, this structure makes it hard for standard nonlinear optimization methods to identify even local minima of the problem.

To find an approximate solution to the problem, we suggest the following strategy. First, we identify a large but finite set \( N \gg n \) of possible times \( t_0 = 0 < t_1 < t_2 < \cdots < t_{N-1} < t_N = T \) at which we might want to place our \( n \) probes. Let \( \mathcal{T} = \{t_0, t_1, \ldots, t_N\} \). If we
constrain our scheduled probe times to lie within $\mathcal{T}$, we obtain the problem

$$
\min \sum_{i=1}^n \overline{C}(p_{i-1}, p_i) \\
\text{S.T.} \quad p_0 = 0 \\
p_i - p_{i-1} \geq \delta \quad i = 1, \ldots, n \\
p_n = T \\
p_1, p_2, \ldots, p_{n-1} \in \mathcal{T}.
$$

(6)

Since it has additional constraints, this problem clearly has a larger optimal objective value than (4). However, if $N$ is large and the points of $\mathcal{T}$ “cover” $[0, T]$ sufficiently thickly, the difference should be slight; we provide a bound on the difference in optimal value between the two problems later in this section. The advantage of (6) is that it can be solved to global optimality by dynamic programming in polynomial time with respect to $n$ and $N$.

Our dynamic programming approach to (6) proceeds as follows: for $m = 0, \ldots, n$ and $i = 0, \ldots, N - 1$, define $D(m, i)$ to be the minimum feasible expected cost that can be accrued over the time interval $[t_i, T]$, given that a probe occurs at $t_i$, one may perform at most $m$ additional probes, and probes may only occur at times in $\mathcal{T}$, with the last probe falling at $T$. We define $D(m, N) = 0$ for $m = 0, \ldots, M$, and letting a value of $+\infty$ denote that no feasible solution is possible, we have

$$
D(1, i) = \begin{cases} 
\overline{C}(t_i, T) & t_i \leq T - \delta \\
+\infty & t_i > T - \delta,
\end{cases} \quad i = 0, \ldots, N - 1.
$$

(7)

Clearly, $D(n, 0)$ is the optimal value of problem (6). Next, take any $m \in \{0, \ldots, n\}$ and $i \in \{0, \ldots, N - 1\}$. Considering that the first update after time $t_i$ must occur at some time $t_j$, where $t_j \geq t_i + \delta$ and $i < j \leq N$, we obtain the recursion

$$
D(m, i) = \min_{j : i < j \leq N \atop t_j \geq t_i + \delta} \{\overline{C}(t_i, t_j) + D(m - 1, j)\}.
$$

(8)

Define $j(m, i)$ to be some value of $j$ attaining the minimum in (8). If the values of $D(m-1, i)$ are known for all $i$, one can use (8) to calculate $D(m, i)$ for all $i$. Since the $D(0, i)$ are known, we can recursively use (8) to calculate all the $D(m, i)$ and $j(m, i)$, as follows:

**Algorithm 1**

For all $0 \leq i < j \leq N$, let $K[i, j] \leftarrow \overline{C}(t_i, t_j)$

for $i = 0, 1, \ldots, N$

if $t_i \leq T - \delta$

$D(1, i) \leftarrow \overline{C}(t_i, t_N)$

$j(0, i) \leftarrow N$

else


\[ D(1, i) \leftarrow +\infty \]
\[ \text{end} \]
\[ \text{for } m = 2, 3, \ldots, n \]
\[ \text{for } i = 0, 1, \ldots, N \]
\[ \text{Choose } j(m, i) \text{ minimizing} \]
\[ K[i, j] + D(m - 1, j) \text{ over } j = i + 1, \ldots, N \text{ with } t_j \geq t_i + \delta \]
\[ D(m, i) \leftarrow K[i, j(m, i)] + D(m - 1, j(m, i)) \]
\[ \text{end} \]
\[ \text{end.} \]

\(K[i, j]\) is a cache matrix saving values of \(\overline{C}(t_i, t_j)\), so that \(\overline{C}(\cdot, \cdot)\) does not have to be called repeatedly with the same arguments. Note that since we initialize \(K[i, j]\) to \(\overline{C}(t_i, t_j)\), the calculations of \(D(m, i)\) and \(j(m, i)\) agree with their definitions and (8). Since the calculation of \(j(m, i)\) takes \(O(N)\) time, the complexity of the algorithm, except for the initialization of \(K[\cdot, \cdot]\), is \(O(nN^2)\). Thus, the algorithm’s complexity is a low-degree polynomial.

At the end of the above calculation, \(D(n, 0)\) is known (it is not actually necessary to calculate \(D(n, i)\) for \(i > 0\)). However, recall that our goal is to identify the optimal schedule itself. To obtain this schedule, note that it is optimal to perform the first update at time \(t_{j(n, 0)}\), so we set \(p_1 = t_{j(n, 0)}\). Setting \(k \leftarrow j(n, 0)\), it is optimal to perform the second update at time \(t_{j(n-1, k)}\). We then set \(k \leftarrow j(n - 1, k)\), and proceed similarly to the third and subsequent updates. Formally, we construct the full optimal solution \(p_0, p_1, \ldots, p_n\) in \(O(n)\) time as follows:

**Algorithm 2**

\[ k \leftarrow 0 \]
\[ p_0 \leftarrow 0 \]
\[ \text{for } \ell = 1, \ldots, n \]
\[ k \leftarrow j(n + 1 - \ell, k) \]
\[ p_\ell \leftarrow t_k \]
\[ \text{end.} \]

Note that running Algorithm 1 for a given number of updates \(n\) has the side effect of also computing optimal schedules for all lesser numbers of updates \(n' < n\). For example, the optimal cost for a schedule with \(n - 1\) updates is \(D(n - 1, 0)\). One may extract such schedules via a slight modification to Algorithm 2.

Up to this point, the algorithm is applicable not only to our specific cost function, but to any scalar cost function \(\overline{C}(p_0, p_1, \ldots, p_n)\) having the interval-wise additive structure expressed by (3). Consider now the implementation of the first line of Algorithm 1, the construction of the cost cache matrix \(K[i, j]\). In practice, we found that brute force calculation of \(K[i, j]\), could consume as much as 85% of Algorithm 1’s total run time. We now exploit the particular properties of the cost formula (2) to remove this bottleneck.
First, to simplify some algebraic manipulations, we also formally define $\overline{C}(s, f)$ via (2) in the case $s > f$. Using the standard negation rule for reversed integrals, we obtain for $s > f$ that

$$
\overline{C}(s, f) = -\int_{f}^{s} \lambda(t) \left( \int_{t}^{s} a(\tau) \, d\tau \right) \, dt = -\int_{f}^{s} \lambda(t) \left( -\int_{t}^{s} a(\tau) \, d\tau \right) \, dt = \int_{f}^{s} \lambda(t) \left( \int_{t}^{s} a(\tau) \, d\tau \right) \, dt.
$$

**Lemma 2** For any times $s, v, f \in [0, T]$,

$$
\overline{C}(s, f) = \overline{C}(s, v) + \Lambda(s, v) A(v, f) + \overline{C}(v, f),
$$

where $A(v, f) = \int_{v}^{f} a(\tau) \, d\tau$, and $A(v, f) = -A(f, v)$ when $v > f$.

Applying Lemma 2, we have

$$
K[i, j] = \overline{C}(t_i, t_j) = \overline{C}(t_i, t_{i+1}) + \Lambda(t_i, t_{i+1}) A(t_{i+1}, t_j) + \overline{C}(t_{i+1}, t_j)
$$

$$
= K[i, i + 1] + \Lambda(t_i, t_{i+1}) A(t_{i+1}, t_j) + K[i + 1, j].
$$

We may thus populate the $K$ matrix by first filling it with all zeroes, and then executing the following:

**Algorithm 3**

for $i = 0, 1, \ldots, N - 1$

  $K[i, i + 1] \leftarrow \overline{C}(t_i, t_{i+1})$

  $A[i] \leftarrow A(t_i, t_{i+1})$

  $L[i] \leftarrow \Lambda(t_i, t_{i+1})$

end

for $j = N, N - 1, \ldots, 1$

  $A' \leftarrow 0$

  for $i = j, j - 1, \ldots, 0$

    $A' \leftarrow A' + A[i + 1]$

    $K[i, j] \leftarrow K[i, i + 1] + L[i] A' + K[i + 1, j]$

  end

end

The first for loop caches the values of $\overline{C}(t_i, t_{i+1})$, $A(t_i, t_{i+1})$, and $\Lambda(t_i, t_{i+1})$ for all $i$. In the inner of the next two for loops, we update $A'$ so that it contains $A(t_{i+1}, t_j)$ at each iteration. Thus, the final assignment is identical to (10).

Despite the somewhat complex logic of Algorithms 1-3, their worst-case execution complexity is not high. Suppose that $E$ is some upper bound on the time to compute $\Lambda(t_i, t_{i+1})$, $A(t_i, t_{i+1})$, and $\overline{C}(t_i, t_{i+1})$ for any $i = 0, \ldots, N - 1$. Then the calculation of $K$ above executes in $O(NE + N^2)$ time, and the entire dynamic programming algorithm requires $O(NE + N^2) + O(nN^2) = O(NE + nN^2)$ time. To summarize our complexity analysis,
Proposition 3 Algorithms 1-3 together calculate an optimal solution to problem (6) in O(NE+nN^2) time, where E is an upper bound on the time to compute \( A(t_i, t_{i+1}) \), \( A(t_i, t_{i+1}) \), and \( \overline{C}(t_i, t_{i+1}) \).

If we choose the functions \( \lambda(\cdot) \) and \( a(\cdot) \) to be piecewise-polynomial with some fixed degree (in the examples and experiments in this paper, we choose the special case of piecewise-constant functions), then \( E \) has a bound proportional to the number of breakpoints of \( \lambda(\cdot) \) and \( a(\cdot) \) contained in any \([t_{i-1}, t_i]\). If we choose the \( \{t_i\} \) and the breakpoints of \( \lambda(\cdot) \), \( a(\cdot) \) in a non-pathological way — for example, \( \{t_i\} \) evenly spaced in \([0, T]\) — then we will have \( NE = O(N) \) and the overall complexity of Algorithms 1-3 simplifies to \( O(nN^2) \). This complexity should allow fairly large values of \( N \).

We next consider how accurately the discretized optimal schedule computed by Algorithms 1 and 2 approximates the true optimal cost of the continuous problems (4) and (5). To this end, we require some bounding and sensitivity results for \( \overline{C}(\cdot, \cdot) \).

Lemma 4 For any times \( s, f \in [0, T] \), we have \( 0 \leq \overline{C}(s, f) \leq \overline{\alpha}(f - s)^2 \), where \( \overline{\alpha}, \overline{\lambda} > 0 \) are upper bounds over \([0, T]\) on the functions \( \alpha(\cdot) \) and \( \lambda(\cdot) \), respectively.

Lemma 5 Suppose \( \varepsilon > 0 \) and \( s, f, s', f' \in [0, T] \) are such that \( |s - s'|, |f - f'| \leq \varepsilon \). Then

\[
|\overline{C}(s, f) - \overline{C}(s', f')| \leq 2\overline{\alpha}\overline{\lambda}(\varepsilon^2 + |f - s|\varepsilon). \tag{11}
\]

Lemma 5 bounds the changes in \( \overline{C}(s, f) \) that can result from perturbations to \( s \) and \( f \). This result makes it possible to bound the cost of appending the discretization constraint \( p_1, \ldots, p_{n-1} \in \mathcal{T} \) to the continuous problems (4) or (5):

Proposition 6 Let \( C^* \) be the optimal value of (5) and \( \overline{C} \) be the optimal value of the discretized problem (6), as computed by Algorithm 1. Suppose that either of the following holds:

(i) \( \delta = 0 \), in which case we define \( \gamma = \max_{j=1, \ldots, N}\{t_j - t_{j-1}\} \). In this case, (5) reduces to (4).

(ii) \( \delta > 0 \), and the \( \{t_j\} \) are evenly spaced \( \gamma \) time units apart, where \( \delta \) is a multiple of \( \gamma \).

Specifically, for some integer \( k \geq 1 \) and all \( j = 1, \ldots, N \), \( t_j - t_{j-1} = T/N = \gamma = \delta/k \). Then \( \overline{C} \leq C^* + \overline{\alpha}\overline{\lambda}((n/2)\gamma^2 + T\gamma) \).

Applying this result in the case of evenly-spaced \( \{t_j\} \) and thus \( \gamma = T/N \), we immediately obtain the following corollary. Note that the restriction between \( N \) and \( \delta \) is only operative when \( \delta > 0 \); when \( \delta = 0 \), it always holds.

Corollary 7 Suppose \( \mathcal{T} \) consists of evenly spaced points, that is, \( t_j = j(T/N) \) for \( j = 0, \ldots, N \), and \( \delta \) is a nonnegative integer multiple of \( T/N \). Then \( \overline{C} - C^* = O(1/N) \), that is, (6)’s optimal value approximates (5)’s by an additive \( O(1/N) \) factor.
4. Improving a Probe Schedule in Continuous Time

Algorithm 1 provides a probe schedule \( \{p_i\}_{i=1}^n \) which is optimal subject to the constraint \( p_0, p_1, \ldots, p_n \in T = \{t_0, t_1, \ldots, t_N\} \). For large \( N \) and reasonably distributed \( \{t_i\} \), we have just proven that the resulting solution should be nearly optimal for the original probe scheduling problem (5).

We next present a method for taking any probe schedule \( \{p_i\}_{i=1}^n \) and improving its expected cost \( \overline{C}(p_0, \ldots, p_n) \). The method is a simple direct search procedure, and does not guarantee either global or true local optimality in the classic differential sense. However, it strictly improves the expected cost of the solution, and is reliable even for the ill-behaved cost functions generated by our model. Our main purpose is to apply this improvement procedure to the output of Algorithms 1-3, but it can be applied from any starting point \( \{p_i\}_{i=1}^n \) feasible for (5).

Suppose three consecutive probes occur at times \( s, t, u \), where \( s + \delta \leq t \leq u - \delta \). First, define \( \overline{C}(s, t, u) = \overline{C}(s, t) + \overline{C}(t, u) \). Next, take any \( \mu \in (0, 1] \), we define \( \Delta^-_\mu(s, t, u) \) to be the improvement in expected cost that would result from moving \( t \) a fraction \( \mu \) of the way towards \( s + \delta \), that is, replacing \( t \) with \( \mu(s + \delta) + (1 - \mu)t \). We define \( \Delta^+_\mu(s, t, u) \) similarly, but moving \( t \) a fraction \( \mu \) of the way towards \( u - \delta \). Formally, we set

\[
\Delta^-_\mu(s, t, u) = \overline{C}(s, t, u) - \overline{C}(s, \mu(s + \delta) + (1 - \mu)t, u)
\]

\[
\Delta^+_\mu(s, t, u) = \overline{C}(s, t, u) - \overline{C}(s, (1 - \mu)t + \mu(u - \delta), u).
\]

If \( \Delta^-_\mu(s, t, u) > 0 \), then the expected cost incurred in \([s, u]\) will be improved by replacing \( t \leftarrow \mu(s + \delta) + (1 - \mu)t \); similarly, if \( \Delta^+_\mu(s, t, u) > 0 \), taking \( t \leftarrow (1 - \mu)t + \mu(u - \delta) \) will lower the expected cost on \([s, u]\). We call these operations a left \( \mu \)-move and a right \( \mu \)-move on \((s, t, u)\), respectively.

We propose improving the cost of the schedule \( p_0, \ldots, p_n \) by looking at all possible left or right \( \mu \)-moves involving consecutive probes \( p_{i-1}, p_i, p_{i+1} \), where \( i = 1, \ldots, n - 1 \). We start by setting \( \mu \) to some starting value \( \mu_0 \). If \( \delta > 0 \), we would typically choose \( \mu_0 = 1 \), whereas if \( \delta = 0 \), we would typically select a smaller value like \( \mu_0 = 1/2 \), since placing two probes at exactly the same time can only increase a schedule’s cost. We then iterate in a greedy manner by choosing the left or right \( \mu \)-move that yields the greatest cost decrease. We repeatedly choose this most improving \( \mu \)-move until no such move improves the cost significantly, that is, \( \Delta^-_\mu(p_{i-1}, p_i, p_{i+1}) < \nu \) for all \( \sigma = +, - \) and \( i = 1, \ldots, n - 1 \), where \( \nu > 0 \) is some small constant. When this occurs, we reduce \( \mu \) by taking \( \mu \leftarrow \rho \mu \), where \( \rho \in (0, 1) \) is another tuning parameter. We iterate in this manner until \( \mu \) drops below some stopping threshold \( \mu_s \), and then halt.
Algorithm 4 Given an initial schedule $p_0, \ldots, p_n$, and parameters $\mu_0 \in (0,1]$, $\mu_* \in (0, \mu]$, $\rho \in (0, 1)$, and $i > 0$, modify $p_1, \ldots, p_{n-1}$ as follows:

$$
\mu \leftarrow \mu_0 \\
\text{while} \ (\mu \geq \mu_*) \\
\quad \text{while} \ \left( \max_{\sigma = +, -} \frac{\Delta^\sigma_\mu(p_{i-1}, p_i, p_{i+1})}{\max_{\sigma = +, -} \Delta^\sigma_\mu(p_{i-1}, p_i, p_{i+1})} \geq \iota \right) \\
\quad \quad \text{Choose } \sigma \in \{+, -\}, i \in \{1, \ldots, n-1\} \text{ minimizing } \Delta^\sigma_\mu(p_{i-1}, p_i, p_{i+1}) \\
\quad \quad \text{if } (\sigma = -) \\
\quad \quad \quad p_i \leftarrow \mu (p_{i-1} + \delta) + (1 - \mu) p_i \\
\quad \quad \text{else if } (\sigma = +) \\
\quad \quad \quad p_i \leftarrow (1 - \mu) p_i + \mu (p_{i+1} - \delta) \\
\quad \end{while} \\
\mu \leftarrow \rho \mu
\end{while}

Suppose now that we implement Algorithm 4 by keeping the values $\{\Delta^\sigma_\mu(p_{i-1}, p_i, p_{i+1})\}$ in a heap of size $2(n-1)$. With appropriate data structures, the maximum element can then be identified in $O(1)$ time. Furthermore, each $\mu$-move only alters the value of $p_i$, which means the only elements in the heap that can change value are

\[
\Delta^-, (p_{i-2}, p_{i-1}, p_i) \quad \quad \Delta^+ (p_{i-2}, p_{i-1}, p_i) \\
\Delta^- (p_{i-1}, p_i, p_{i+1}) \quad \quad \Delta^+ (p_{i-1}, p_i, p_{i+1}) \\
\Delta^- (p_i, p_{i+1}, p_{i+2}) \quad \quad \Delta^+ (p_i, p_{i+1}, p_{i+2}),
\]

two of which will not exist in the boundary cases $i = 1$ and $i = n - 1$. Thus, at most six elements of the heap will be out of position after $p_i$ is changed, and they may be repositioned correctly in $O(\log n)$ time. Thus, the inner loop of Algorithm 4 can be executed in $O(\log n)$ time. At the outset, and whenever $\mu$ is changed, the entire heap needs to be recalculated, taking $O(n \log n)$ time.

While the number of iterations of Algorithm 4’s outer loop can be easily predicted, the number of iterations of the inner loop cannot. However, since $\bar{C}(p_0, \ldots, p_n)$ is always nonnegative and each iteration reduces $\bar{C}(p_0, \ldots, p_n)$ by at least $\iota > 0$, it follows that the algorithm must terminate in a finite number of steps. The empirical tests below suggest that Algorithm 4 has acceptable iteration counts for realistic input data.

It would be desirable to prove that if Algorithm 4 were to be run indefinitely by setting $\iota = \mu_* = 0$, it would asymptotically approach a classically defined local minimum of problem (5). Unfortunately, this is not the case; however, Section A.6 in the online supplement discusses enhancements of Algorithm 4 that would have this property. It is also possible that, instead of applying Algorithm 4’s simple greedy method, one could apply any number
of popular metaheuristics, such as tabu search or simulated annealing, to the basic neighborhood structure defined by left and right \( \mu \)-moves. In view of the near-optimality of our dynamic programming algorithm, we have not investigated such approaches.

5. Numerical Experiments

5.1. Test data and problem parameters

We experimented with two real-world datasets, DBWorld and MicroNet. The DBWorld dataset is an 831-data-point trace of postings to the DBWorld bulletin board between November 9, 2000 and June 15, 2001, partially illustrated on the left side of Figure 2. The horizontal axis represents dates ranging from November 9, 2000 to March 29, 2001, while the vertical axis indicates the times of day updates occurred. The right side of Figure 2 illustrates a similar sample from MicroNet, a University of California Berkeley forum, collected in 2000-2001. For MicroNet, we only recorded thread initiators, i.e., those messages that start a new thread of discussion. Other messages (recognized by an initial “Re:”) were discarded.

We divided each trace into two portions, a training set and a testing set. For DBWorld, the training set contains the first 580 data points and the testing set contains the remaining 251 data points. For MicroNet, we used the 174 data points collected in the second half of 2000 as the training set. The testing set contains 68 data points collected in the first nine weeks of 2001. For both traces, we set \( T \) to be one week.

For both data traces, we modeled \( \lambda(\cdot) \) in a simple manner similar to the experiments in Gal and Eckstein (2001), taking \( \lambda(\cdot) \) to be a periodic piecewise-constant function with a period of one week. We divided each week into fixed intervals, within which \( \lambda(t) \) is assumed constant, and then applied the standard maximum likelihood arrival rate estimate within each interval. Our general approach was to start with three-hour intervals, combine adjacent intervals with similar arrival rates, and then group together days with similar arrival patterns — for example, all weekdays. We continued this combining process as long as there was no
significant impact on the statistical quality of the fit, as measured by the Kolmogorov-Smirnov test used in Gal and Eckstein (2001). For MicroNet, at the end of this process, extending “work” time by one additional hour improved the fit. Tables 1 and 2 show the $\lambda(t)$ levels fitted to the DBWorld and MicroNet training sets, respectively.

One may estimate $\lambda(\cdot)$ by a variety of methods, ranging from manual to totally automated, and from ad hoc estimation to sophisticated statistical methods. A detailed discussion of this topic is out of the scope of this paper; Alizadeh et al. (2004) present some related ongoing research on estimating smooth $\lambda(\cdot)$ functions and adapting the number of breakpoints in a piecewise-polynomial estimate to the available data. The details of the fitting procedure are unlikely to be critical to the results here, since earlier experiments we have conducted with variety of data sets showed that the statistical validity of the update model was resilient to minor changes in the structure of $\lambda(\cdot)$.

Note that it is not necessary to know a history of precise update times, as we have used here, to estimate $\lambda(\cdot)$. Given a past set of probe times, a maximum likelihood estimator for $\lambda(\cdot)$ may be constructed simply from the numbers of updates $|R(p_{i-1}, p_i)|$ observed by the client between probes; for an example, see Alizadeh et al. (2004). Thus, it is possible to estimate $\lambda(\cdot)$ even for an uncooperative server. Of course, more detailed information about update times may be helpful in estimating $\lambda(\cdot)$ more quickly and accurately. In some cases, the updates are intrinsically time-stamped in some manner, so either a full or approximate update history is readily available; examples include e-mails (time-stamped by their headers),

<table>
<thead>
<tr>
<th></th>
<th>Weekdays</th>
<th>Saturday</th>
<th>Sunday</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0:00, 3:00]</td>
<td>2.40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[3:00, 6:00]</td>
<td>5.96</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[6:00, 9:00]</td>
<td>6.04</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[9:00, 18:00]</td>
<td>7.50</td>
<td>1.50</td>
<td>1.15</td>
</tr>
<tr>
<td>[18:00, 21:00]</td>
<td>3.03</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[21:00, 24:00]</td>
<td>2.41</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 1: $\lambda(t)$ pattern for the DBWorld trace.

<table>
<thead>
<tr>
<th></th>
<th>Weekdays</th>
<th>Weekend</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0:00, 9:00]</td>
<td>0.25</td>
<td></td>
</tr>
<tr>
<td>[9:00, 19:00]</td>
<td>2.60</td>
<td>0.08</td>
</tr>
<tr>
<td>(19:00, 24:00)</td>
<td>0.14</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: $\lambda(t)$ pattern for the MicroNet trace.
RSS feeds, and certain kinds of transaction records. In other cases, partially cooperating
servers might be willing to add time stamp information, since it is less burdensome than
implementing a full push protocol. Distributing such information would be cheap, and could
benefit the server by reducing its load through elimination of unnecessary polling by clients;
see Bright et al. (2006) for relevant discussion and examples.

We modeled \( a(\cdot) \) using (1), with work hours defined to be [9:00, 18:00] on weekdays for
the DBWorld trace, and [9:00, 19:00] on weekdays for the MicroNet trace. In situations with
an identifiable discrete query or usage process, one could estimate \( a(\cdot) \) by any of the methods
applicable to \( \lambda(\cdot) \). Here, we took the continuous query viewpoint and used (1), inferring the
typical work hour profile from our \( \lambda(\cdot) \) estimates.

5.2. Alternative methods, performance metrics, and parameters

We evaluated the performance of probing schedules with respect to the following criteria:
(i) the expected cost \( \overline{C}(p_0, \ldots, p_n) \) over the planning period, (ii) the actual obsolescence
cost \( C(p_0, \ldots, p_n) \) over the testing set, and (iii) the number of iterations of Algorithm 4.
Note that the actual cost \( C(p_0, \ldots, p_n) \) is subject to random fluctuations arising from the
testing set sample, but has the advantage of measuring the combined utility of our model
and scheduling procedure on a real data stream. On the other hand, \( \overline{C}(p_0, \ldots, p_n) \) measures
only the desirability of the schedule \( p_0, \ldots, p_n \) under the assumption that the update model
embodied in \( \lambda(\cdot) \) is accurate.

In our Algorithm 1 experiments, we varied the number of probes \( n \) while holding \( N \)
constant at 168. We set the possible probe times \( t_i \) to be evenly spaced, one hour apart.
We tried four different preference ratios, specifically \( a_1/a_2 = 1, 2, 3, 4 \), and also experimented
with varying \( \delta \) values, ranging from \( \delta = 0 \) (meaning no constraints) to \( \delta = 10 \) hours.

Our experiments compare the relative performance of four different policies: uniform,
threshold, first arrival (FA), and the schedule collectively generated by Algorithms 1-3, which
we call Discrete Optimal Expected, or DOE for short. The uniform policy simply spaces \( n \)
probes evenly throughout \([0, T]\). Such uniform allocation schedules were proposed for Web
crawling in Cho and Garcia-Molina (2000). In the threshold policy, proposed in Gal and
Eckstein (2001), the user provides a cost parameter \( \Pi \). Once the expected cost exceeds \( \Pi \), a
probe takes place. Finally, the first arrival policy, proposed in both Gal and Eckstein (2001)
and Lee et al. (2002), probes whenever the probability of having any new data exceeds some
threshold \( \pi \), regardless of the current importance level or the total number of data updates
that might be pending transcription. This approach thus uses only the \( \lambda(\cdot) \) part of the cost
model, and ignores \( a(\cdot) \).
Of the three alternatives, only the uniform policy can be properly compared to DOE, because it is the only one for which one can directly constrain the number of probes. In the threshold and FA policies, the number of probes is influenced by the parameter $\Pi$ or $\pi$, respectively, but in practice it is difficult to manipulate these parameters to obtain a precise number of probes. Furthermore, varying $\Pi$ or $\pi$ can easily result in a different schedule, but with the same total number of probes. Thus, neither of these policies is well-suited to environments where exact satisfaction of politeness constraints is important, and the uniform policy is the only method which should, strictly speaking, be considered a “prior” approach to problem (6).

However, for the sake of thoroughly investigating probe schedule quality, we compared the four policies as follows: we began by generating schedules with the threshold and first arrival policies for various values of $\Pi$ and $\pi$. For each such schedule, we recorded $n$, the number of actual probes. Then, we ran the DOE and uniform policies with all the $n$ values obtained from the threshold and first arrival policies, and compared the results.

In all the experiments with Algorithm 4, we took the initial shifting parameter to be $\mu_0 = 1/2$, the final shifting factor to be $\mu_* = 1/512$, and the reduction parameter to be $\rho = 1/2$. Here, we also varied $\delta$, and used three different $\iota$ values. We tried $\iota = 0$, meaning that the algorithm only halts once no further improvement in the expected cost can be achieved. We also experimented with less strict $\iota$ values, namely $\iota = 0.002$ and $\iota = 0.004$.

5.3. Experiments without the improver

For both the DBWorld (left) and MicroNet (right) datasets, Figure 3 presents a comparison of the expected cost $\mathcal{C}(p_0, \ldots, p_n)$ obtained from each of the four policies, without applying Algorithm 4, based on the model constructed from the training data, with a preference ratio of 3. To avoid cluttering the figures, we present only the middle portion of the expected cost axis; our discussion of the results holds for the whole spectrum.

The expected cost for all policies falls as $n$ increases, which accords with intuition, although DOE is the only policy for which it is theoretically guaranteed (if we hold $T$ fixed). Also, as one would expect from its derivation, the DOE policy yields the lowest expected cost for any given $n$. The uniform policy seems to perform significantly worse than the other policies, but as $n$ grows and probes become more densely packed, the differences between all the policies shrink. While the two traces are qualitatively similar, DOE performs better for the MicroNet data.

Figure 4 also shows expected costs, but for a preference ratio of 1, that is, $a(t) \equiv 1$. The pattern is qualitatively similar to Figure 3, although the separation between policies is more clearly apparent: DOE still outperforms the other policies. We observed this phenomenon
Figure 3: Comparison of the expected costs of the four policies, for a preference ratio of 3.

Figure 4: Comparison of the expected costs of the four policies for a preference ratio of 1.

for all preference ratios. From this point on, we therefore focus on just a single value of the preference ratio, $a_1/a_2 = 3$.

Figure 5 presents a comparison of the actual cost (from the continuous query point of view) realized by each of the four policies, evaluated on the testing set. For each dataset, we took nine weeks of the testing set, calculated the actual cost for each week (recall that we have defined $T$ to be one week) and then averaged the resulting nine results. These data are affected both by any inaccuracies in the update model embodied by $\lambda(\cdot)$, and by random variations in the testing set sample. For DBWorld, DOE outperforms all other policies for all but three cases: for $n = 126$, FA has a slightly lower cost than DOE, and for $n = 10$ and $n = 13$, threshold performed better than DOE. We consider these instances to be natural random deviations from a dominant pattern. For MicroNet, while DOE outperforms the other policies in most cases, there are exceptions: threshold performed best for $n = 4, 5, 7, 9, 18$, and FA performed best for $n = 3, 4, 5, 13, 33$. We explain this reduced effectiveness of DOE by the lower arrival intensity and smaller training set size for MicroNet, which cause random
Figure 5: Comparison of the actual costs of the four policies.

variations in the data stream to have higher relative impact on both the $\lambda(\cdot)$ estimate and the actual cost. Even for MicroNet, however, DOE still performs better than any other single policy for the majority of instances. From the figure, it is also clear that DOE has the most consistently desirable performance over the observed range of $n$.

Figure 6 presents the expected and actual cost for the DBWorld data set of two policies, DOE and threshold, using two different $\delta$ values. Setting $\delta = 0$ sets no restrictions on the spacing of consecutive probes, while $\delta = 10$ forces probes to be set at least 10 hours apart. For the threshold heuristic, we enforced this spacing constraint by probing after waiting at least $\delta$ hours to repeat a probe, even if the expected threshold cost had already been attained. As in the $\delta = 0$ case, it often remains impractical to tailor the parameter $\pi$ of the threshold algorithm to obtain a specific number of probes in a given interval, and so the threshold procedure is not necessarily a true competitor algorithm for solving (4) or (6).

The FA policy could be similarly modified to account for $\delta > 0$, and the same general comments would apply. For the uniform policy, a $\delta$ probe spacing constraint simply means that $T/n \geq \delta$, or equivalently $n \leq T/\delta$. For $n > T/\delta$, it is impossible to make a schedule with all probes at least $\delta$ apart, so depending on $n$, considering $\delta > 0$ either makes an entire uniform policy infeasible, or leaves it unchanged.

For DOE, both the expected and actual costs rise as one increases $\delta$. For small $n$ values, the distance between consecutive probes is so wide that the spacing constraint has no impact on the optimal schedule; however, as $n$ increases, the spacing constraints become binding and the cost difference becomes gradually more pronounced. By comparison, the behavior of the threshold policy is erratic, and in some cases restricting the problem by increasing $\delta$ leads to better expected or actual performance. These experiments suggest that while DOE seems the most desirable approach for $\delta = 0$, its advantages are even greater for $\delta > 0$. 

19
Figure 6: Actual and expected cost comparison of $\delta = 0$ and $\delta = 10$ for the DOE and threshold polices, DBWorld dataset.

5.4. Experiments with the improver

We next consider how our “improver,” Algorithm 4, performs when starting from schedules created by each of the four algorithms of the last section. In each case, we compare the resulting expected and actual obsolescence costs, as well as the number of iterations required by the improver. We also varied $\delta$ and $\iota$.

Figure 7 shows the expected costs of improved schedules seeded by each of the four policies, with $\delta = 0$ and $\iota = 0.004$. Comparing Figure 7 with Figure 3, the improver indeed reduces the expected cost for all policies. The improvement method seems quite effective, in that the differences between expected costs are far less pronounced than those prior to improvement: our improver appears to be able to arrive at schedules of roughly similar cost from starting points of varying quality. Still, different initial schedules do lead to different improved costs, reflecting that the cost function has many closely-spaced local minima, and the improver cannot guarantee a global optimum. Second, close inspection shows that the DOE policy still performs the best, probably by providing better initial schedules. We observed similar results for all other combinations of $\delta$ and $\iota$ with which we experimented.
Figure 7: Comparison of the expected cost of the four policies, after improvement, with \( \delta = 0 \) and \( \iota = 0.004 \).

Figure 8: Comparison of the actual cost of the four policies, after improvement, with \( \delta = 0 \) and \( \iota = 0.004 \).

Figure 8 shows the actual costs of the improved schedules on the testing sets with \( \delta = 0 \) and \( \iota = 0.004 \). The results are similar to Figure 7’s, with DOE being the best policy by a narrow margin in the majority of cases. When DOE does not do best, the FA policy seems to yield good results, especially for the MicroNet data set. As we will soon see, such gains do come at a high cost in terms of number of improver iterations. Again, similar results were observed for other combinations of \( \delta \) and \( \iota \).

We also measured the efficiency of the four starting policies in terms of the number of \( \mu \)-move iterations performed by Algorithm 4 until it terminated with \( \mu < \mu_t \). Figure 9 shows the results for \( \delta = 0 \) and \( \iota = 0.004 \): for any given \( n \), seeding the improver with DOE yields far fewer iterations. Similar results were observed for \( \iota = 0.002 \). In Figure 10, we present results with \( \iota = 0 \). This setting requires that the algorithm continue as long as it obtains any improvement, no matter how small. Technically, the algorithm is not guaranteed to terminate in this case, but in practice we encountered no such difficulty. Here, all policies,
Figure 9: Comparison of the number of \(\mu\)-move improver iterations, with \(\delta = 0\) and \(\iota = 0.004\).

Figure 10: Comparison of the number of \(\mu\)-move improver iterations, with \(\delta = 0\) and \(\iota = 0\).

 including DOE, have their number of iterations increasing roughly exponentially as \(n\) grows. DOE is no longer dominant with respect to the number of improver iterations.

To check the utility of small \(\iota\), we measured the relative increase in expected cost of the final schedule as one raises \(\iota\). Figure 11 compares the expected costs obtained for DBWorld with \(\iota = 0.002\) and \(\iota = 0.004\) to an \(\iota = 0\) baseline. Experiments with MicroNet were qualitatively similar. For both \(\iota = 0.002\) and \(\iota = 0.004\), the increase in expected cost tends to be small, and is less than 1\% for the DOE policy. The uniform policy shows the greatest expected cost impact from increasing \(\iota\), as high as 4\% of 5\% for some values of \(n\).

5.5. Conclusions from numerical experiments

Based on the empirical evaluation presented in this section, we conclude that:

1. DOE is best with respect to the expected cost and also, measured against each policy individually, provides in most cases better schedules in terms of actual cost.
Figure 11: Cost increase as compared to \( t = 0 \), DBWorld, with \( \delta = 0 \).

<table>
<thead>
<tr>
<th></th>
<th>DBWorld</th>
<th>MicroNet</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Uniform</td>
<td>DOE</td>
</tr>
<tr>
<td><strong>Total run time (minutes)</strong></td>
<td>3.26</td>
<td>3.66</td>
</tr>
<tr>
<td><strong>Number of improver iterations</strong></td>
<td>73</td>
<td>0</td>
</tr>
<tr>
<td><strong>Expected cost without improver</strong></td>
<td>4.36</td>
<td>3.18</td>
</tr>
<tr>
<td><strong>Actual cost without improver</strong></td>
<td>4.02</td>
<td>3.14</td>
</tr>
<tr>
<td><strong>Expected cost with improver</strong></td>
<td>3.49</td>
<td>3.18</td>
</tr>
<tr>
<td><strong>Actual cost with improver</strong></td>
<td>3.16</td>
<td>3.14</td>
</tr>
</tbody>
</table>

Table 3: Run time and solution quality for \( n = 44 \), \( \delta = 0 \), and \( t = 0.004 \).

2. Algorithm 4 is effective in reducing the costs of initial schedules generated by all the policies considered. After improvement by Algorithm 4, DOE schedules are still the best overall, although the margin is often narrow.

3. For Algorithm 4, larger \( t \) values increase costs for all policies, but the impact of \( t \) on the DOE policy is marginal, due to the near-optimality of the starting schedules. For DOE, \( t = 0 \) causes a large increase in the number of iterations, with little cost benefit.

4. For strictly positive \( t \), DOE schedules result in by far the fewest number of improver iterations.

Taken together, these observations provide a strong motivation for using DOE to initialize Algorithm 4: DOE provides the best schedules with respect to both expected cost and number of improver iterations. Another motivation is that Proposition 6 and Corollary 7 provide near-optimality guarantees for DOE schedules. Since Algorithm 4 only reduces the schedule’s expected cost, these guarantees of course carry over to improved DOE schedules.

The disadvantage of using DOE is that it requires more computation than other initial schedules. In many cases, simply applying the improver — which is also part of the contribu-
tion of this paper — to a uniform starting schedule can produce a schedule with very similar expected cost in somewhat less time. However, this approach lacks DOE’s near-optimality guarantee, and the time required to run the improver is relatively unpredictable. In general, we thus consider it generally more advisable to use DOE. By way of illustration, Table 3 shows run times for $n = 44$, $\delta = 0$, and $\iota = 0.004$, on a 1.6 GHz Mobile Pentium 4 with 256MB of RAM. Although uniform schedules can be computed essentially instantaneously, they cause the improver to take far more iterations, mitigating their time advantage. Starting with a uniform schedule saves 20-30 seconds out of 3-4 minutes of total run time, but the resulting expected cost is about 10% higher for DBworld, and about 17% higher for MicroNet. In other cases, the trade-off may be less extreme, and if low run time is extremely critical, one might choose to simply apply the improver to a uniform schedule, limiting the total number of iterations. Normally, however, we would not expect computing resources to be at such a high premium, and would recommend using DOE: basically, while Algorithm 4 generally performs well, combining it with DOE performs better and provides a guarantee, at modest computational cost.

Finally, if one is evaluating a range of possible values of $n$, DOE becomes more attractive, since it can generate near-optimal schedules for a range of $n$ values simultaneously, and the improver, which must be run separately for each value of $n$, executes much more rapidly when seeded with DOE schedules.

6. The Information Systems Literature: Related Work

This paper’s analysis is applicable to pull-based environments, in which the client probes the server for new information. Both push-based methods — for example, Cao and Liu (1998) and Yin et al. (2001) — and pull-based methods — e.g., TTL (Gwertzman and Seltzer, 1996) with its extensions (Cohen and Kaplan, 2001; Bright and Raszid, 2002) — are common in applications such as Web caching and publish/subscribe, along with profile-driven cache management (Cherniack et al., 2003) and cache synchronization (Cho and Garcia-Molina, 2000; Carney et al., 2003). Combined push-pull methods also exist (Deolasee et al., 2001). Push-based methods may well be preferable when they are available — see for example Dey et al. (2006) — but they are impractical in many applications.

Methods for optimizing multiple-source Web information monitoring have recently been presented in Pandey et al. (2003, 2004) and Sia and Cho (2005). Pandey et al. (2003) give an offline algorithm implementing a system constraint of at most $C$ probes per unit of time over all resources, which is similar but not identical to our politeness constraint. Pandey et al. (2004) extend the analysis in Pandey et al. (2003), generalizing the cost function, and providing an online algorithm which is 2-approximate of optimality.
Our constraints — $n$ probes over the period $[0, T]$ separated by at least $\delta \geq 0$ — are different from the constraint in Pandey et al. (2004). Our approach uses a time-heterogeneous stochastic update model, which is not explicitly present in Pandey et al. (2003, 2004). Also, our formulation is parameterized by the user importance function $a(\cdot)$, generalizing the weighting scheme in Pandey et al. (2003, 2004), allowing one to model a nonhomogeneous Poisson query process, and permitting more customization of the probe schedule.

All the algorithms proposed here fall into the offline category — once one has specified or estimated $a(\cdot)$ and $\lambda(\cdot)$, the scheduling algorithms treat the arrival model as fixed, and do not attempt to modify it or gather additional data. Nevertheless, our empirical results show that the algorithms yield good results even for real-world traces that may not conform perfectly to the update model assumptions. We also note that if a Poisson update process is really an accurate model, an online algorithm should not outperform an offline one, since the arrival process is memoryless.

A variety of prior work, including Cho and Garcia-Molina (2000), Gal and Eckstein (2001), Lee et al. (2002), and Dey et al. (2006), has suggested using a Poisson update model. An early effort, using a simple homogeneous Poisson process in the context of Web resource monitoring, is due to Cho and Garcia-Molina (2000), who suggest two different policies. In their uniform approach, probes are performed at evenly-spaced time intervals, while the approach they term adaptive, probes are spread according to update frequency, somewhat similarly to the threshold policy of Gal and Eckstein (2001). Threshold policies may be difficult to combine with a politeness constraint, and our current work also demonstrates both uniform and threshold policies to be suboptimal with respect to our cost model.

One way to capture varying update intensity, proposed in Lee et al. (2002), is to use a shifting time window. In that work, the arrival rate estimate $\lambda$ is updated in an online manner based on the number of updates observed in the last $K$ time units, $K$ being a tunable parameter. This approach has the advantage of being able to capture update intensity variations not predictable from historical patterns, but has the disadvantage of being error-prone during predictable “shoulder” periods when $\lambda$ either sharply increases or decreases. The probe scheduling technique suggested in Lee et al. (2002) is identical to the first arrival policy suggested in Gal and Eckstein (2001) and tested here. Again, first arrival policies may be hard to combine with a politeness constraint, and in the context of our cost model, our experiments indicate that they are suboptimal.

In relation to Gal and Eckstein (2001), we use an equivalent cost model, but Algorithms 1-4 exploit it more fully. Thus, we obtain better policies with respect to the same update and cost models. To the best of our knowledge, no other information monitoring research employs a model with a time-varying user importance level. For example, in Cho and Garcia-Molina
(2000), the cost of a missed update is its age, equivalent in our model to taking \( a(t) = 1 \). Also, in Bright and Raschid (2002), the cost of any missed update was set to 1, regardless of its age or the current time. Our importance function may also be viewed as arising from a nonhomogeneous Poisson query process.

Wolf et al. (2002) present an interesting algorithm-driven approach monitoring multiple time-homogeneous sources, either Poisson or non-memoryless, approaching the problem in three stages: first, they consider, separately for each source, the form of an optimal schedule with \( n \) probes. By time homogeneity, this schedule must be uniform, and if source \( i \) is Poisson, they show that the resulting optimal expected cost \( F_i(n) \) is a convex function of \( n \). In the next stage, they allocate a total of \( R \) probes among the multiple sources. If all sources are Poisson, this allocation is a single-constraint convex discrete resource allocation problem which may be solved optimally by a greedy algorithm or refinement thereof, resulting in an idealized probe schedule. Finally, they use a transportation problem formulation to construct a multi-source crawler schedule that is as close as possible to the idealized schedule, but does not overcommit crawling resources at any single time. This last step should be efficient to implement but is heuristic, since its objective is the rectilinear distance to the ideal schedule, rather than the true schedule cost. They only consider one simple form of time-heterogeneous arrival model — their quasi-deterministic case — and suggest that the single-source subproblem for this case be solved heuristically or via an \( 1/e \)-approximate algorithm. In our notation, they also consider only \( \delta = 0 \).

Dey et al. (2006) analyze various methods for updating a data warehouse from a single source. They also consider multiple sources, but since they assume probes of multiple sources are synchronized, the multiple-source case essentially reduces to a single source. Except for one brief section, they use a cost model equivalent to ours when \( a(\cdot) \) and \( \lambda(\cdot) \) are constant functions. The numerical experiments also include non-memoryless processes, but they are time homogeneous. By contrast, the intention of our research is to take advantage of predictable cyclic variations in arrival intensity, and possibly also user interest or query intensity. The analysis in Dey et al. (2006) suggests that push-based refresh policies such as their update-based policy, when practical, are likely to outperform pull-based ones. It would be interesting to test whether this result still holds in a time-heterogeneous setting.

Sia and Cho (2005), in a working paper roughly contemporaneous with our own work, consider monitoring multiple sources using a nonhomogeneous Poisson update model and a “delay” objective function equivalent to our cost model with constant \( a(\cdot) \). Their optimization formulation imposes a single constraint on the total number or probes, taken over all resources. They approach the problem in two stages: first, each resource is allocated a share of the total probes, and then they determine a separate retrieval schedule for each source.
The resource allocation formula is an approximation based on assuming that the update processes are homogeneous, and ignoring integrality constraints. The retrieval scheduling phase involves separate single-source problems similar to this paper’s, but with a constant importance function. The authors propose two possible retrieval scheduling methods based on a standard first-order necessary local optimality condition for the continuous probe scheduling problem. The first method is essentially a “shooting” method which uses the continuous local optimality conditions to reduce the problem to single variable on which it conducts an exhaustive search. Our own unpublished experiments with a similar method, but allowing non-constant importance functions, yielded numerically unstable results, but we did not test exhaustive search procedures. The second method, iterative refinement, appears similar to our improver, but based on satisfying the Lagrangian necessary optimality conditions instead of reducing the expected cost. Since the cost function has many local minima, as demonstrated in Figure 1, it would be more prudent to use an improver based on cost than on local optimality conditions; the local optimality approach also effectively requires one to use a continuous $\lambda(\cdot)$ model. Sia and Cho (2005) do not consider detailed client loading constraints, and therefore do not need a third algorithmic stage like Wolf et al. (2002). While our present work does not consider multiple sources, it does provide alternative algorithms and optimality guarantees absent from the single-source subproblems in Sia and Cho (2005).

7. Conclusions and Possible Extensions

In summary, we have more fully exploited the cost model introduced in Gal and Eckstein (2001) to obtain provably near-optimal update schedules in situations where updates occur randomly but have some known or estimated time variation in intensity. These schedules can account for time variations in user importance, or equivalently, time heterogeneity of a query process.

Although exact solution of the optimization problem (4) we have formulated is very difficult, our dynamic programming algorithm exactly optimizes the approximate, discretized variant (6), with an additive error bound of $O(1/N)$. Our second algorithm can then fine-tune the resulting schedule. Our computational work shows that good schedules do appear to result from this procedure. Furthermore, it appears that our improver algorithm can be very beneficial when applied to other initial schedules. In this case, however, there is no $O(1/N)$ guarantee, and the improver typically takes significantly more iterations.

Our results here concern a single information source. It is also important to consider monitoring multiple sources, with constraints linking the sources’ probe scheduling problems. For instance, one might want to monitor multiple Web pages or sites with dissimilar
update patterns, but hosted by a single server. In this case, one might want to probe the
sources separately, but apply some form of aggregate politeness constraint at the server level.
There may also be sets of sources that are independent with regard to politeness, but for
which the client lacks sufficient network bandwidth or processing resources to probe simulta-
neously. If neither of these situations apply, one should be able to monitor multiple sources
by simply constructing an optimal probing schedule for each one. Otherwise, our results
need to be generalized to cover the case of multiple sources, with constraints regarding the
total number or probes, processing or network utilization at each point in time, politeness
to groups of related sources, or combinations thereof. We reserve such extensions for fu-
ture research, but now briefly discuss the prospects for various solution approaches. The
well-known “curse of dimensionality” phenomenon would likely make a direct dynamic pro-
gramming approach to the entire problem ineffective beyond very small numbers of sources.
Two possible alternatives seems worth considering: an algorithmically motivated approach
essentially combining our work with the ideas of Wolf et al. (2002), or a more model-driven
integer programming approach.

In the former case, one would first apply our DOE method separately to each source,
resulting in a two-dimensional table of values $F_i(n)$, the optimal expected costs obtainable
by applying $n$ discrete-time probes to each source $i$. Next, as in Wolf et al. (2002), we
would use a greedy-related method to decide how many probes to allocate to each source.
Analyzing our experimental data reveals that while $F_i(n)$ is in general not convex in $n$, it
tends to be nearly so; thus, the resource allocation would be heuristic, but probably quite
accurate. Finally, the resulting individual schedules would need to be reconciled similarly to
the third step in Wolf et al. (2002).

The integer programming alternative would involve using a slightly enhanced version
of our dynamic programming method as a column generation procedure within a branch-
and-price algorithm employing linear programming relaxations; see for example Barnhart
et al. (1998). This strategy would sacrifice polynomial complexity, but would be able to pro-
vide problem-specific optimality guarantee information. The algorithm would start with a
heuristically-generated solution, and the longer allowed to run, would produce progressively
better solutions and tighter guarantees on their distance from optimality. It is unclear in
advance whether its performance would be attractive, but the key ingredients for implemen-
tation — a column generation procedure, a way of generating integer-feasible solutions from
fractional ones, and a branching rule compatible with the column generator — all appear
tractable. Parallel computing might also be applicable to the column generation step, which
might present a bottleneck in serial computation, but partitions naturally by information
source.
In either approach, a possible additional last step might involve applying a generalized, multi-source version of our continuous-time improver algorithm to reduce the expected cost of the discrete-time schedule.

Acknowledgments

We thank Michael Rothkopf for the suggestions that helped initiate this work. We also thank Yulia Turchin for her assistance in writing the code and performing the experiments.

References


