
Journal of Intelligent Information Systems, 23:1, 67–97, 2004

Dynamic Composition of Information Retrieval Techniques*

ANDREW ARNT arnt@cs.umass.edu
SHLOMO ZILBERSTEIN shlomo@cs.umass.edu
JAMES ALLAN allan@cs.umass.edu
Department of Computer Science, University of Massachusetts, Amherst, MA 01003, USA

ABDEL-ILLAH MOUADDIB mouaddib@info.unicaen.fr
Département d'informatique, Université de Caen, 14032 Caen Cedex, France

Received May 29, 2002; Revised May 30, 2003; Accepted July 31, 2003

Abstract. This paper presents a new approach to information retrieval (IR) based on run-time selection of the best set of techniques to respond to a given query. A technique is selected based on its projected effectiveness with respect to the specific query, the load on the system, and a time-dependent utility function. The paper examines two fundamental questions: (1) can the selection of the best IR techniques be performed at run-time with minimal computational overhead? and (2) is it possible to construct a reliable probabilistic model of the performance of an IR technique that is conditioned on the characteristics of the query? We show that both of these questions can be answered positively. These results suggest a new system design that carries a great potential to improve the quality of service of future IR systems.

Keywords: progressive processing, information retrieval, opportunity cost, meta-level control

1. Introduction

Information retrieval (IR) systems such as web search engines are becoming increasingly prominent as the Internet grows in both amount of information indexed and the number of users. While gains are being made to improve the performance and effectiveness of these systems, the majority share a common characteristic: they are static. These IR systems are constructed by chaining together a fixed set of IR techniques such as query formation, query evaluation, precision improvement, recall improvement, clustering, and results visualization. Once the system is completed and put into operation, the same sequence of operations is performed on every query processed by it. This static nature of IR systems reflects the methods used to evaluate new IR techniques. An IR technique is typically tested offline to determine if the technique is "good" enough to be included in an IR system. In these experiments a test collection consisting of a set of queries and a set of documents is used. Each document has been tagged with relevance judgments for each query in the test

*This work was supported in part by the National Science Foundation under grants IIS-9907331 and INT-9612092, and by the Library of Congress and Department of Commerce under cooperative agreement number EEC-9209623.

collection. A document is tagged "relevant" if it satisfies the information need posed by the query.

To quantify how good the retrieval performance of an IR system is, two basic measures of retrieval performance are used in the IR community. *Recall* is the fraction of relevant documents that are retrieved by the system out of the entire collection, while *precision* is the fraction of documents retrieved by the system that are relevant. It is desirable, however, to have a single number that describes the performance. One of the most used evaluation measures is average precision: the mean of the precision ratios obtained after each relevant document is retrieved, using zero as the precision for relevant documents that are not retrieved. It has been shown to be both stable and a good indicator of overall system performance (Buckley and Voorhees, 2000).

If the new technique is found *on average* to substantially improve a quality metric such as average precision (i.e. it improves the *mean average precision*), and is computationally reasonable, then it is considered to be a good technique and is added to the system (Buckley and Voorhees, 2000). Unfortunately, there are inherent problems with this methodology. First, although the technique improves *overall* retrieved document quality, there are likely to be queries where the quality of the retrieved documents would be unchanged or even decrease with use of the technique. Secondly, people may disagree on whether a document is relevant to a given query. Studies have shown that relevance judgments made by different people can vary substantially (Swanson, 1988). Therefore the judgments tagged in the test collection may not necessarily correspond to those of the user who actually issues a query to the system.

We describe a new, dynamic approach to the design of IR systems. In this approach, IR techniques are chosen at run-time on a per-query or even a per-user basis, with the goal of optimizing the overall quality of service of the system, where quality of service is defined in terms of both the quality (average precision) of the retrieved documents and the timeliness with which they are returned. This approach involves picking techniques to use based on a probabilistic description of each technique's expected performance in terms of both processing time and change in quality of the result. There are several factors that can influence what modules are chosen to be executed:

Module characteristics: If the expected execution time of each of the IR modules is known, then when the IR system is very busy and has a long queue of queries to process, it can conserve system resources by running less computationally intensive tasks on the query. This amounts to a trade off between quality of the retrieved documents and system resources. If the expected contribution of that module to the quality of the final result is known, we can consider that as well.

Query characteristics: Some IR operations may not be suitable for queries that have certain properties. For example some expansion techniques are designed to be used with shorter queries (Allan and Raghavan, 2002), and other general purpose techniques been shown to have harmful effects on longer queries (Singhal and Pereira, 1999). In a dynamic system, each query can be examined as it is received to determine which IR tasks are likely to give the best results.

Intermediate results: Intermediate results generated as part of the IR process can be used to help decide which modules to select for the rest of the retrieval. An example of

intermediate results that would be useful to examine are the initial set of documents returned by LCA (as described below) or some other local feedback technique (Croft and Harper, 1979; Attar and Fraenkel, 1977). Furthermore, we can observe how much time has been used in computing these intermediate results, which can also influence our decision.

User characteristics: The dynamic IR system can use information that is provided by or collected from the user to decide which IR methods will produce the best results for queries issued by *that user*. One example of this is estimating a cost model that reflects how much value the current user places on the speed of retrieval versus the quality of retrieval.

A dynamic IR system provides several clear benefits over static IR systems. It can be more flexible and robust while at the same time providing a greater quality of service. Primarily, we can increase the overall quality of the documents returned by the system. Incoming queries can be quickly characterized and retrieval techniques can be chosen that have been empirically determined to work best on queries with same or similar characteristics. Above all, techniques that are likely to not change or even reduce the quality of retrieved documents can be avoided. Current static IR systems cannot offer such flexibility.

Furthermore, a dynamic system has the ability to regulate the trade off between retrieved document quality and system resources consumed. For example, cost-benefit analysis can be done at run time to determine whether it is worthwhile to invest system resources in applying a technique that may result in only a small improvement in retrieved document quality. This leads to savings in computational resources. Such a system can also reason about techniques that have both large variance in expected execution time and retrieved document quality.

The major question is then how to actually pick which IR modules to execute on an incoming query. This decision must be made in such a way as to be able to adapt quickly to any feedback that can be obtained from the system as the query is processed. This feedback can consist of such things as the quality of intermediate results, running time of modules already executed, and number of queries waiting to be serviced. An additional issue is how to gather feedback from the system; specifically how to determine the quality of intermediate results.

In this paper, we develop a computational model necessary for fast run-time selection of the best IR techniques and provide a case study of building a predictive probabilistic model of the performance of an IR technique suitable for the model. In Section 2 we develop effective mechanisms for run-time selection of best IR techniques based on the progressive processing model. In Section 3 we detail a method for training artificial neural networks that take query characteristics as input and output whether a certain IR technique is expected to improve or degrade the quality of retrieved documents if run on the query. The IR technique examined is Local Context Analysis (LCA), a method for query expansion (Xu and Croft, 2000). We show that predictions can be made accurately enough to give a small improvement to overall average precision. Related work is discussed in Section 4. We conclude with a discussion of the implications of this research and examine possible future strategies for meta-level control of IR systems.

2. Progressive processing of information retrieval tasks

In this section we develop a task representation and corresponding meta-level control mechanisms that make it possible to dynamically compose information retrieval techniques. We use the progressive processing approach, an extremely general framework to solving problems under time constraints (Mouaddib, 1993). The approach relies on providing the system with a task structure composed of a set of problem solving methods and a set of constraints on execution. The constraints allow the system to produce useful solutions by executing a subset of the entire task structure. Additional information included in the task structure describes the duration of each method and how the quality of the result depends on the selected methods.

The most simple example of a task structure is a sequence of methods, each of which can increase the expected quality of the result. This approach has been used to develop a progressive processing solution to train scheduling (Mouaddib, 1993). Different topologies of task structures offer more flexibility in designing the system, but they also increase the complexity of selecting the best subset at run-time.

Among existing resource-bounded reasoning techniques, the progressive processing model resembles most the *design-to-time* approach (Garvey and Lesser, 1993). But unlike other techniques for run-time method selection, we develop an adaptive approach that is suitable for dynamic environments. An information retrieval search engine is a particularly good application because it is characterized by high level of uncertainty regarding the duration of the process and the quality of the result. In addition, there may be large variability in the number of queries that require a response at any given time. By taking a context dependent, dynamic approach to the problem we aim at significantly improving the average quality of service provided by such systems.

The progressive processing approach offers a natural framework to describe the set of information retrieval techniques available to the system. Figure 1 shows a simple task structure also known as a *progressive processing unit* (PRU). The input of a PRU is a *query* composed of a list of keywords. The task structure has three processing *levels*. The first level includes three alternative techniques to improve the initial query: (a) scan the query using concept recognizers to identify company names, dates, locations, personal names, and so on; (b) examine the query for pairs of words that have high statistical likelihood of being related and enhance the query with that information; and (c) perform part-of-speech analysis to identify noun phrases within the query. The second level includes two alternative techniques that can improve the query's recall ability by expanding it to include related words and phrases: (d) use of Local Context Analysis (LCA), a statistical method for expanding queries that relies upon in-context analysis of word co-occurrence (Xu and Croft, 1996); and (e) use of InFinder, an association thesaurus that is faster than LCA and does not capture context as well (Conrad and Utt, 1994). Finally, the third level performs the actual query evaluation and returns the results. Quality in this application is measured by the number of relevant documents within the top n documents retrieved (i.e., precision in the retrieved set).

Because the task structure is known in advance, our objective is to solve as much of the problem as we can off-line and minimize the overhead of meta-level control. This is

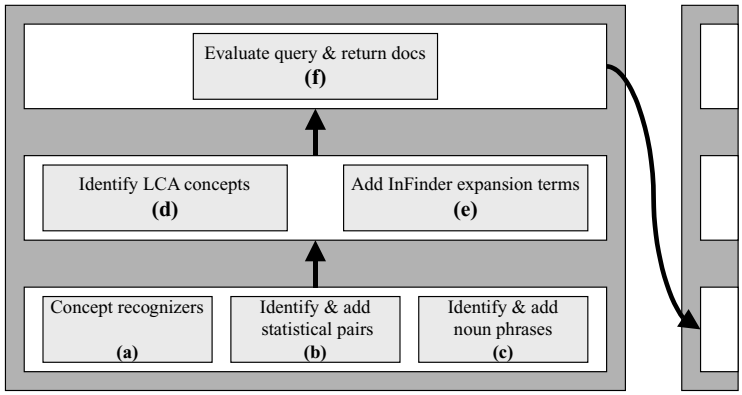

Figure 1. Illustration of a progressive processing task for an information retrieval search engine.

particularly challenging because we operate in a dynamic environment and cannot predict the set of queries waiting for execution at each point.

2.1. The meta-level control problem

This section defines the specific type of progressive processing task structure used in our application and the resulting meta-level control problem. By meta-level control, we refer to the process that monitors the base-level computation and selects the particular subset of methods to be used in response to each query. The selection of methods is done at run-time based on the progress being made with the particular query and the remaining queries waiting for execution. While previous approaches to meta-level control of progressive processing could handle either rapid domain change (Mouaddib and Zilberstein, 1997) or a high level of duration uncertainty (Mouaddib and Zilberstein, 1998), the technique described in this paper addresses effectively the combination of high level of uncertainty and dynamic environments.

Each information retrieval request is handled by a fixed progressive processing task structure. A formal definition of the components of the task structure is given below.

Definition 1. A *progressive processing unit* (PRU) is composed of a sequence of processing levels, $(l_1, l_2, \ldots, l_L)$. The first level receives the input query and the last one produces the result.

Note that unlike other progressive processing models, the intermediate results in this application have no value. Intermediate levels can in some cases be skipped, but only the final level produces the desired result.

Definition 2. Each processing level, l_i, is composed of a set of p_i *alternative modules*, $\{m_i^1, m_i^2, \ldots, m_i^{p_i}\}$, and a binary variable that indicates if that level is able to be skipped during processing. We use $skip_i = 1$ when level l_i is skippable and 0 otherwise. The last level is unskippable, so $skip_L = 0$ always.

Each module can perform the logical function of level l_i, but each has different computational characteristics defined below.

Definition 3. The *module descriptor*, $P_i^j((q', \delta) \mid q, l_{i-1})$, of module m_i^j is the probability distribution of output quality and duration for a *given* input quality and previous level executed.

Note that q is a discrete variable representing estimated quality and δ is a discrete variable representing duration. The module descriptor specifies the probability that module m_i^j takes δ time units and returns a result of quality q' when the quality achieved by the previously executed level is q. Module descriptors are similar to *conditional performance profiles* of anytime algorithms (Zilberstein and Russell, 1996). They are constructed empirically by collecting performance data for a sample set of inputs.

In the IR setting described here, quality is a measure of how well the retrieved documents satisfy the information need expressed in the query. Thus we use the average precision measure as q. Note that for intermediate levels the definition is not as clear, as there may not yet be any set retrieved documents. In these cases, q is a rough estimate of the expected final quality given the current state of the retrieval process. We discuss later how to use machine learning techniques to estimate this otherwise unobservable value.

When the system responds to a particular query, it receives an immediate reward defined as follows.

Definition 4. A *time-dependent utility function*, $U(q, t)$, measures the utility of a solution of quality q if it is returned t time units after the arrival time of the query.

We assume that there is a given constant T such that $\forall q, t > T : U(q, t) = 0$. That is, responding to a request more than T time units after its arrival has no value. The time-dependent utility function is an arbitrary description of user preferences. It can measure, for example, the amount of money a user is willing to pay for a response to a query given its quality and waiting time. Although we assume that the utility function is fixed, the approach we develop can be generalized to allow different utility functions to be attached to each query. However, the set of functions must be known in advance. The utility function provides a means of quantifying the quality of service of the overall retrieval system. It reflects the preferences of the users of system with respect to the desirability of good results versus the time it takes to get them. It is possible to model users so as to fit them into a predefined set of categories, each corresponding to a different utility function.

Suppose that the system maintains a set of information retrieval queries, W, with arrival times $\{a_1, a_2, \ldots, a_n\}$. The set of queries is updated dynamically as new requests arrive. The system processes the queries in a first-in-first-out order using a progressive processing unit to handle each query.

Given a set of queries, the module descriptors of all the components of the progressive processing unit, and a time-dependent utility function, we define the following control problem.

Definition 5. The *reactive control problem* is the problem of selecting a set of alternative modules so as to maximize the expected utility over the set of information retrieval queries.

Maximizing the expected utility directly corresponds to providing the optimal quality of service to users of the system. The meta-level control is "reactive" in the sense that we assume that the module selection mechanism is very fast, largely based on off-line analysis of the problem.

2.2. *Optimal control of a single PRU*

We begin with the problem of meta-level control of a single progressive processing unit corresponding to a single task. This problem can be formulated as a simple form of a Markov decision process (MDP). MDPs have been studied in operations research as a model of sequential decision-making. They have been adopted as a framework for artificial intelligence research in decision-theoretic planning (Dean et al., 1995) and reinforcement learning (Barto et al., 1995). MDPs allow reasoning about actions with uncertain outcomes in order to determine a course of action that maximizes expected utility. The defining property of a Markov process is that state transitions depend only on the current state and the control action.

In our case, the states of the MDP represent the current state of a computational process. The state includes the current level of the PRU, the quality produced so far, and the elapsed time since the arrival of the request. The rewards are defined by the utility of the solution which depends on both quality and time. There are two possible actions: to *execute* a module of the next processing level or to *skip* that processing level if allowed. The transition model is defined by the descriptor of the module selected for execution. The rest of this section gives a formal definition of the MDP and the reactive controller produced by solving it.

2.2.1. State representation. The execution of a single progressive processing unit, u, can be seen as an MDP with a finite set of states $\mathcal{S} = \{[l_i, q, t]\} \bigcup \{[\text{fail}, t]\}$ where $0 \leq i \leq L$ indicates the last executed or skipped level, $0 \leq q \leq 1$ is the quality produced by the last executed module, and $0 \leq t \leq T$ is the elapsed time since the arrival time, a_u, of the query. Note that quality is discretized and normalized to be in the range $[0 \ldots 100]$. All the intermediate modules use a uniform representation of input and output (a "query" in our application). Note also that T is the maximum delay after which we consider the response to be useless. When the system is in state $[l_i, q, t]$, the i-th level has been skipped or executed. The states $[\text{fail}, t]$ represent termination at time t without any useful result. We distinguish between different failure states because failure can occur before the deadline; the earlier it occurs, the more valuable the state because it leaves more time for executing the remaining queries in the queue.

2.2.2. Transition model. The initial state of the MDP is $[l_0, q_{\text{init}}, t]$, where t is the elapsed time since the arrival of the request ($t = \text{currenttime} - a_u$), q_{init} is the initial quality of the query (0 in our application), and l_0 is a dummy variable indicating that no levels have been executed or skipped yet. The initial state indicates that the system is ready to start executing a module of the first level of the PRU. The terminal states are all the states of the form $[l_L, q, t]$ or $[\mathit{fail}, t]$. The former set represents successful completion of the last level and the latter set represents failure. Other states such as $[l_i, q_{\max}, t]$ (reaching maximal intermediate quality)

or $[l_i, q, T]$ (reaching the deadline before the execution of the last level) are not considered terminal states. A terminal state can be reached from state $[l_i, q, T]$ by executing a series of *skip* actions until a failure state is reached. Similarly *skip* actions can take the automaton from state $[l_i, q_{\max}, t]$ to the last level executing the fewest number of modules possible because no *execute* action can improve the intermediate quality. In practice, the *skip* action is implemented as a module with no duration that causes no change in quality.

In every nonterminal state the possible actions are: $\mathbf{E}_{i+1}^{j}$ (execute the j-th module of the next level) and $\mathbf{S}_{i+1}$ (skip the next level). To complete the transition model, we need to specify the probabilistic outcome of these actions. Equations (1)–(4) define the transition probabilities for any nonterminal state $[l_i, q, t]$.

The $\mathbf{S}_{i+1}$ action is deterministic. It skips the next level without affecting the quality or elapsed time. As noted earlier, it can be implemented as an additional "dummy" module whose execution takes no time and has no effect on quality.

$$\Pr([l_{i+1}, q, t] \mid [l_i, q, t], \mathbf{S}_{i+1}, skip_{i+1} = 1) = 1 \quad \text{when } 0 \le i < L - 1 \tag{1}$$

Skipping when a level marked as unskippable results in failure. Therefore, skipping the last level always results in failure.

$$\Pr([fail, t] \mid [l_i, q, t], \mathbf{S}_{i+1}, skip_{i+1} = 0) = 1 \tag{2}$$

The $\mathbf{E}_{i+1}^{j}$ action is probabilistic. Duration and quality uncertainties define the new state. Equation (3) determines the transitions following successful execution and Eq. (4) determines the transition to the failure state when the deadline, T, is reached.

$$\Pr\left([l_{i+1}, q', t + \delta] \,\middle|\, [l_i, q, t], \mathbf{E}_{i+1}^{j}\right) = P_{i+1}^{j}((q', \delta) \mid q) \quad \text{when } t + \delta \le T \tag{3}$$

$$\Pr\left([fail, T] \,\middle|\, [l_i, q, t], \mathbf{E}_{i+1}^{j}\right) = \sum_{q', \delta > T - t} P_{i+1}^{j}((q', \delta) \mid q) \tag{4}$$

It is easy to see that this process satisfies the Markov assumption because the probabilistic outcome of each action (executing a particular module) depends only of that module's descriptor.

2.2.3. Rewards and the value function. Rewards are determined by the given time-dependent utility function applied to the final result (produced by the last level of the PRU). The utility depends on the quality of the result and the elapsed time. Keep in mind that in our application the intermediate results are useless and therefore have no direct rewards associated with them. We now define a value function (expected reward-to-go) over all states. The value of terminal states is defined as follows:

$$V([l_L, q, t]) = R(q, t) = U(q, t) \tag{5}$$

$$V([fail, t]) = R(0, t) = U(0, t) \tag{6}$$

The value of nonterminal states of the MDP is defined as follows.

$$V([l_i, q, t]) = \max_a \begin{cases} V([l_{i+1}, q, t]) & \text{If } a = \mathbf{S}_{i+1}, skip_{i+1} = 1 \\ V([fail, t]) & \text{If } a = \mathbf{S}_{i+1}, skip_{i+1} = 0 \\ EV\left([l_i, q, t] \,\middle|\, \mathbf{E}^j_{i+1}\right) & \text{If } a = \mathbf{E}^j_{i+1}, 0 < j \leq p_i \end{cases} \tag{7}$$

Such that the expected value

$$EV\left([l_i, q, t] \,\middle|\, \mathbf{E}^j_{i+1}\right) = \sum_{q', \delta > T-t} P^j_{i+1}((q', \delta) \,|\, q) V([fail, T]) + \sum_{q', \delta \leq T-t} P^j_{i+1}((q', \delta) \,|\, q) V([l_{i+1}, q', t + \delta])$$

The value function is defined as the maximum over all actions with the top expression representing the value of a skip action for any skippable level l_i, the middle expression representing the value of a skip action for an unskippable level, and the bottom expression representing the value of an execute action.

This concludes the definition of an MDP. This MDP has a finite-horizon (determined by the number of levels) and no cycles (because time, level number or both are incremented by each action). Therefore, the MDP can be solved easily using standard dynamic programming algorithms or using search algorithms such as AO*.

Proposition 1. *Given a progressive processing unit u and a time-dependent utility function $U(q, t)$, the optimal policy for the corresponding MDP provides an optimal reactive controller for u.*

Proof: There is a one-to-one correspondence between the reactive control problem and the Markov decision process. We also know that the PRU transition model satisfies the Markov assumption. From the optimality of the resulting policy for the MDP, we conclude that it provides optimal reactive control for the progressive processing problem. □

2.2.4. The effect of discretization. The above proof of optimality assumes that the computational model is a precise description of the progressive processing task structure. In practice however, it is necessary to discretize the representation of both time and quality. The number of states of the resulting MDP is proportional to the product of the number of the number of discrete quality levels and the maximum execution time. Both of these are system parameters that must be determined with care. The maximum execution time, in particular, can be quite large. A small time unit leads to a more effective controller at the expense of a larger state-space. The choice of a unit of quality has a similar effect. These units introduce a tradeoff between the size of the policy and its effectiveness.

In this application we have only a rough approximation of quality (as described in Section 3), so no more than a very coarse discretization of quality is necessary. The discretization for time (which is easily observed) can be determined empirically to find a good trade-off between policy size, execution time, and accuracy of control.

2.3. *Optimal control of multiple PRUs using opportunity cost*

Suppose now that we need to schedule the execution of multiple PRUs. We assume that there are $n + 1$ queries whose arrival times are $a_0 \leq a_1 \leq \cdots \leq a_n$. One approach to construct an optimal schedule is to generalize the solution presented in the previous section. We can simply construct a larger MDP for the combined sequential decision problem including the entire set of $n + 1$ PRUs. To do that, each state must also include i, the request number, leading to a general state represented as $[i, l, q, t]$. Note that t is measured relative to the arrival time of the *first* request in the queue.

This rather complex MDP is still a finite-horizon MDP with no loops. Moreover, the only possible transitions between different PRUs are from a terminal state of one PRU to an initial state of a succeeding PRU. Therefore, we can solve this MDP by computing an optimal policy for the *last* PRU for any starting time between 0 and $T + a_{n+1} - a_0$, then use the value of its initial states to compute an optimal policy for the previous PRU and so on. The time interval for the starting time is defined by the constraint that all the requests must be processed between the arrival of the first request, a_0, and the deadline of processing the last request, $a_{n+1} + T$. To further simplify and unify all the control policies, we measure time relative to the arrival of the first request, always starting from zero.

Proposition 2. *Given a set, W, of progressive processing units and a time-dependent utility function $U(q, t)$, the optimal policy for the corresponding MDP is an optimal reactive control for W.*

This is an obvious generalization of Proposition 1. The complete proof, by induction on the number of PRUs, is omitted.

We now show how to reformulate the effect of the remaining n requests on the execution of the first query. This reformulation preserves the optimality of the solution, but it suggests a more efficient control structure that will be developed below.

Definition 6. Let $V_i^*(t) = V([i, l_0, q_0, t])$ denote the expected value of the optimal policy for the last $n - i + 1$ PRUs.

To compute the optimal policy for the i-th PRU, we can simply use the following modified reward function.

$$R_i(q, t) = U(q, t + a_0 - a_i) + V([i + 1, l_0, q_0, t]) \tag{8}$$

The new reward for responding to the i-th query is composed of the immediate reward (defined by the time-dependent utility function) and the reward-to-go (defined by the remaining PRUs). In contrast, Eq. (5) refers only to immediate reward, ignoring the reward that could be gained from executing the remaining PRUs. Alternatively, the new reward function can be represented as follows.

$$R_i(q, t) = U(q, t + a_0 - a_i) + V_{i+1}^*(t) \tag{9}$$

Therefore, the best policy for the first PRU can be calculated if we use the following reward function for its terminal states:

$$R_0(q,t) = U(q,t) + V_1^*(t) \tag{10}$$

Definition 7. Let $OC(t) = V_1^*(0) - V_1^*(t)$ be the *opportunity cost* at time t.

The term opportunity cost is borrowed from economics where it normally represents the highest value alternative that must be foregone when a choice is made. In our case, this alternative refers to starting to process the remaining queries immediately. Hence, the opportunity cost measures the loss of expected value due to delay in the starting point of executing the remaining n queries.

Definition 8. Let the *OC-policy* be a control policy for the first PRU computed with the following reward function:

$$R(q,t) = U(q,t) - OC(t)$$

The OC-policy is the policy computed by deducting from the actual reward for the first task the opportunity cost of its execution time. Unlike the reward function used in Section 2.2.3, this definition captures both the reward for responding to the current query as well as the loss of value due to delay of the remaining queries.

Proposition 3. *Controlling the first PRU using the OC-policy is optimal.*

Proof: From the definition of $OC(t)$ we get:

$$V_1^*(t) = V_1^*(0) - OC(t) \tag{11}$$

To compute the optimal schedule we need to use the reward function defined in Eq. (9) that can be rewritten as follows:

$$R_0(q,t) = U(q,t) + V_1^*(0) - OC(t) \tag{12}$$

But this reward function is the same as the one used to construct the OC-policy, except for the added constant $V_1^*(0)$. Because adding a constant to a reward function does not affect the policy, the conditions of Proposition 2 are met and the resulting policy is optimal. □

Proposition 3 suggests an optimal approach to scheduling the entire $n + 1$ requests by first using an OC-policy for the first query that takes into account the opportunity cost of the remaining n queries. Then the OC-policy for the second query is used taking into account the opportunity cost of the remaining $n - 1$ queries and so on. To be able to implement this approach we need to have the control policies readily available. This issue is addressed in the following section.

2.4. *Reactive control based on estimated opportunity cost*

In the previous section, we presented an optimal solution to the control problem of multiple progressive processing units without accounting for its computational complexity. In particular, the opportunity cost must be computed and revised quickly each time a new request arrives. Once the opportunity cost is revised, a new policy for the current PRU must be constructed. Finding the exact opportunity cost requires the construction of an optimal policy for the entire set of queries. In practice, this may slow down substantially the operation of the system.

In order to provide an effective reactive controller for dynamic progressive processing, it is necessary to:

1. use a fast approximation scheme to estimate the opportunity cost; and
2. use pre-compiled policies for different levels of opportunity cost.

The rest of this section explains this method in detail and evaluates it with respect to four types of task structures.

2.4.1. Estimating the opportunity cost. The opportunity cost is defined in terms of the function V_1^* which represents the value of an optimal policy for the remaining tasks in the queue. Thus, it can be estimated by approximating this function. We have examined two approximation schemes: a naïve approach based on fast run-time approximation and an off-line learning approach.

2.4.1.1. Naïve approximation. A naïve approach to approximating the cumulative value of the remaining tasks is to quickly add the expected values of the queries in the queue *without* taking into account the opportunity cost. In this calculation, the start time of executing a query is the *expected* end time of the previous one. The following set of equations summarizes this approximation scheme.

$$\begin{aligned}
&V_1^*(t) \simeq V([l_0, q_0, t + a_0 - a_1]) + V_2^*(t + \tau_1)\\
&\vdots\\
&V_i^*(t) \simeq V([l_0, q_0, t + a_0 - a_i]) + V_{i+1}^*\left(t + \sum_{j=1}^{j=i} \tau_j\right)\\
&\vdots\\
&V_n^*(t) = V([l_0, q_0, t + a_0 - a_n])
\end{aligned} \tag{13}$$

where $V[l, q, t]$ is the value function defined in Section 2.2.3 for a single PRU. Therefore, V_1^* can be approximated as follows:

$$V_1^*(t) \simeq \sum_{i=1}^{i=n} V\left[l_0, q_0, t + a_0 + \left(\sum_{j<i} \tau_j\right) - a_i\right] \tag{14}$$

In these equations, τ_i is the expected duration of processing query i. Note that τ_i depends on the duration of the previous tasks. Let $\tau(d)$ be the expected duration of the optimal single-PRU policy when starting at time d relative to the arrival time of the query. Then τ_i is computed using τ with the expected starting time of task i relative to *its* arrival time. The starting time of task i relative to a_0 is $t + \sum_{j<i} \tau_j$. This time relative to the arrival time of query i (a_i) is $d = t + a_0 + (\sum_{j<i} \tau_j) - a_i$. Therefore,

$$\begin{aligned} &\tau_0 = 0, \\ &\tau_i = \tau\left(t + a_0 + \left(\sum_{j<i} \tau_j\right) - a_i\right). \end{aligned} \tag{15}$$

The function τ (expected duration) can be computed for any finite-horizon MDP once the optimal policy is available by simply evaluating the optimal policy with respect to duration instead of reward. The function can be computed once off-line, making it easy to revise the opportunity cost when a new request is added. By ignoring the opportunity cost, the naïve approximation consistently allocates more time to the first queries and leaves less time for the last queries in the queue. It is definitely not optimal, but it provides a simple, fast approximation.

2.4.1.2. Learning the opportunity cost function. Another approach is to estimate the opportunity cost using some features that characterize the remaining PRUs in the queue. Using a data set composed of randomly generated queues of one of the synthetic PRUs as described in Table 1 for which the precise opportunity cost is known (calculated off-line), we can learn a feature-based opportunity cost function. Then, this function can be used at run-time to quickly approximate the opportunity cost for any given set of queries.

The synthetic PRUs are constructed by randomly specifying the module descriptors $P_i^j((q', \delta) \mid q)$ for each module in the PRU. Type **A** is representative of the characteristics of an actual information retrieval application, while the others are used to test scalability.

The features used in our experiment are the total number of PRUs in the queue and the average waiting time of the PRUs in the queue.

We evaluated the the effectiveness of this approach by conducting the following experiments. A dataset of queues was generated using a simple model of query arrival time (a random number between 0 and 3 requests arrive over a period of ten time units). The exact opportunity cost was computed at each of the time units for 100 randomly generated queues.

Table 1. Four types of PRUs used in evaluation.

PRU type	L	Modules per level	T
A	3	6	300
B	3	15	300
C	3	6	1000
D	3	15	1000

A simple 1-Nearest-Neighbor algorithm is used, where a Euclidean distance metric on the features determines the closest data queue to the test queue. The estimated opportunity cost for each time unit is then determined by the exact opportunity cost of the closest data queues for that time unit.

2.4.1.3. Comparison of opportunity cost approximation methods. We now compare the performance of the naïve method for opportunity cost approximation against the 1-Nearest-Neighbor function approximation technique described above. Performance using no opportunity cost is given for comparison. This is based on an optimal policy for a single task, which ignores the entire queue of requests (using Eq. (5) to define the reward). Unlike the naïve method, in this case the reward function does not take into account the reward-to-go as defined by Eqs. (8) and (9).

To perform this comparison, 50 different PRU arrival queues were randomly generated for each of the four PRU types described in Table 1. They were generated by having a random number between 0 and 3 requests arrive at each time unit over a period of ten time units (using a unit size of 10 seconds). Note that all the PRUs in a given queue correspond to the same task structure. For each arrival queue, we estimated $OC(t)$ at each of the possible arrival times using both estimation methods. We then computed the exact opportunity cost. Table 2 gives the average relative error for each of the methods. As expected, 1NN generally outperforms the Naïve method.

We also observed how often actions chosen by the estimated OC policy differed from those specified by the optimal policy. These values are also given in Table 2. We see that both of the estimation methods perform very well, with the 1NN method actually generating a policy identical to the optimal for PRUs of type A. Ignoring the opportunity cost leads to a large action error (up to 35%). It is interesting to note that in PRUs of type D, the action error is small for all three approaches. The large number of alternatives and high level of uncertainty about duration make the value of the second-best action closer to the value of

Table 2. Comparison of OC approximation methods.

PRU type	OC Est method	Est OC error	Action error
A	None	N/A	20.345
	Naïve	34.102	2.014
	1NN	7.178	0.0
B	None	N/A	18.422
	Naïve	10.550	5.586
	1NN	2.813	4.678
C	None	N/A	35.185
	Naïve	6.042	0.971
	1NN	2.668	0.233
D	None	N/A	1.453
	Naïve	1.142	1.302
	1NN	2.255	0.376

the best action. Note also that in this case the naïve method provides the more accurate estimate of OC, but it also leads to larger error in action selection. A possible explanation is that while the estimate is more accurate in general, it is less accurate for some critical cases in which a small error makes a difference in action selection.

These results are encouraging, and imply that such estimation techniques should provide adequate results in non-synthetic applications.

2.4.2. Pre-compiled control policies. To make the meta-level control truly reactive for large task structures, one may want to avoid computing a new policy (for a single PRU) each time the opportunity cost is revised. To avoid this, the space of opportunity cost can be divided into a small set of regions representing typical situations. For example, there could be just three regions that capture *low*, *medium*, and *high* loads. For each region, an optimal policy would be computed off-line and stored in a library. At run-time, the system will first estimate the opportunity cost and then use the appropriate pre-compiled policy from the library. These policies remain valid as long as the overall task structure and the utility function are fixed. Because the dependency of the control decisions on the opportunity cost is monotonic (higher costs imply less time for execution), we anticipate that a small set of classes that correspond to *qualitatively* different action selection policies will be sufficient.

Another advantage of the use of pre-compiled policies is the ability to react to dynamic changes. Control policies can be switched *during* the execution of a single request if the opportunity cost changes. This is possible because the policies share the same state space.

3. Probabilistic modeling of local context analysis

The above reasoning techniques have relied on the notion of observable quality of both intermediate results and final returned documents of an information retrieval system, where we define quality as the average precision of the returned documents. This is problematic for a few reasons. First, the quality of the final returned documents is unknown, since there is no way of knowing a priori what documents are relevant to the query at hand. Secondly, intermediate results are often in the form of a single bag-of-words query. It is even more difficult to define quality in this case as there are no documents on which to base quality, much less documents with known relevance judgments. Therefore, to effectively implement a progressive processing IR system, we must be able to quickly estimate quality of both intermediate queries and final returned document sets.

One obvious benefit of a dynamic IR system is that we can try to avoid using techniques that may hurt retrieval quality for a given query. There are few IR methods that improve retrieval on all queries, all of the time.

IR researchers typically publish only results of their systems on collections of queries rather than on a query-by-query basis. Nonetheless, there are a few occasions when this information is presented, and gives evidence to the fact that otherwise "helpful" IR techniques can hurt performance on some queries.

Xu and Croft (1996) show that a local feedback-based query expansion technique improves the average precision on the 49 TREC4 collection queries from 25.2% to 27.9%. Similarly, LCA is shown to improve the average precision from 25.2% to 31.1%. However,

the average precision is hurt on 21 of the 49 queries by the local feedback technique, and on 11 of 49 by LCA .

Crouch et al. (2001) describe an algorithm for query expansion that performs reranking of the initial document set based on a comparison with unstemmed terms in the original query. Stemming refers to removing obvious prefixes and suffixes from terms in documents and queries, prior to any indexing or retrieval. They state that 31 of the 149 queries tested were hurt by their technique, while overall average precision was increased.

Singhal and Pereira (1999) present a method for document expansion in which the document corpus is composed of documents created from speech recognition programs. Since speech recognition is far from perfect, the documents in the corpus containing many transcription errors (on the scale of 24% to 60%). Speech documents are augmented with terms from documents in a non-speech text corpus in hopes of restoring terms that were lost in transcription. They show when performing document expansion based a text corpus not closely related to the speech corpus that their method drops the mean average precision 0.7% on long queries, while the mean average on *all* queries is increased 12%.

In this section we give a simple example of this estimation using artificial neural networks (ANNs) to estimate quality for the modules of a single IR technique, Local Context Analysis. We show that ANNs trained on a select subset of available data can be used to attain modest gains in retrieval performance by running LCA only on those queries that are predicted to improve retrieval performance. In Section 3.5 we show how this online prediction can be expressed in the framework of a simple PRU. Lastly, in Section 3.6, we demonstrate a full progressive processing system based on this PRU. We show that using progressive processing allows the retrieval system to perform at a high level of service over a wide range of server loads.

3.1. Local context analysis

One of the major obstacles IR systems must overcome is the word mismatch problem (Xu, 1997), which refers to the fact that users of IR systems often use different words in their queries to describe the same information need. This can lead to many relevant documents not being retrieved by the system. For example, the user's query may use the word "car", but some documents might only use the word "automobile", and therefore would not be retrieved. One of the most effective ways to overcome the word mismatch problem is through query expansion, where a user's query is expanded by adding new terms to it before performing the retrieval step.

Query expansion methods have been studied for many years. One of the first attempts, in Sparck Jones (1971), clustered words based on co-occurrence frequencies in the document collection, and expanded queries based on those clusters. This is known as a global technique, as the entire document corpus is analyzed to obtain the word clusters. In local analysis, only the top ranked documents retrieved for the original query are examined to obtain expansion terms and was first studied in Attar and Fraenkel (1977) and Croft and Harper (1979). This method is known as local feedback or pseudo relevance feedback, and forms of it have become common in IR literature (Yang et al., 1998; Robertson and Walker, 1997; Davis and Dunning, 1996; Allan, 1995).

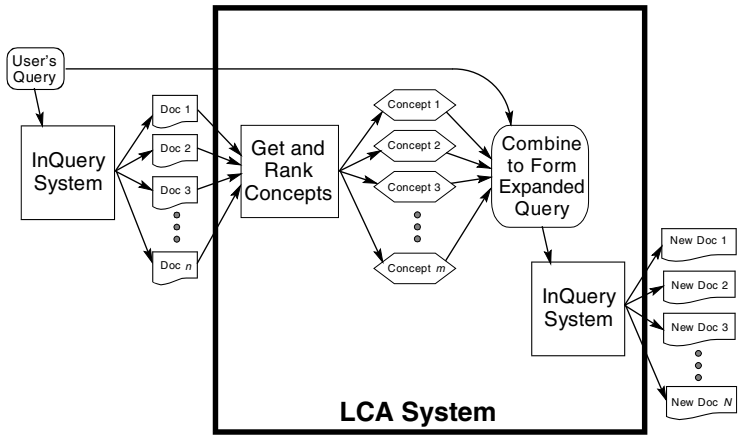

Figure 2. Diagram of the LCA process.

LCA is currently one of the more successful query expansion techniques. It applies ideas from global analysis methods, such as phrase identification and use of term context, to the set of top ranked documents used in local feedback (Xu and Croft, 1996). We choose to perform our experiments on LCA not only because its overall good performance, but because an implementation was easily accessible.

The LCA process is detailed below and in figure 2:

– First, the user's query is run through InQuery, an IR system developed at the University of Massachusetts (Callan et al., 1995). InQuery returns a list of documents ranked in order of estimated relevance to the query.
– The top n documents are selected from the list, and are analyzed to find "concepts." A concept is either a noun or a sequence of nouns. Each concept is given a weight based on co-occurrence of query terms with that concept and the IDF (see Section 3.2.1 for a definition) values of the concept. Essentially, LCA is finding those noun phrases that occur very often near query terms in the top ranked documents, while not occurring with high frequency in the overall document collection. The concepts are ranked in order of decreasing weight.
– The top m concepts are merged into the user's original query, creating a new expanded query.
– The expanded query is run through InQuery, and the documents retrieved are returned to the user.

We use $n = 100$ and $m = 26$, as these values have produced good empirical results. An example expansion using LCA is shown in figure 3 (Xu and Croft, 1996).

On average, LCA has been shown to improve retrieval performance. But as shown above, LCA can also hurt retrieval on some queries. For example, if very few of the documents returned in the initial pass of InQuery are relevant to the original query, then the concepts that are added to the query will be mostly noise, causing the retrieval performance to drop

hypnosis	brain-wave	ms.-burns	technique	pulse
reed	brain	ms.-olness	trance	hallucination
process	circuit	van-dyck	behavior	suggestion
case	spiegel	finding	hypnotizables	subject
van-dyke	patient	memory	application	katie
muncie	approach	study	point	contrast

Figure 3. Example LCA concepts for the query "What are the different techniques used to create self induced hypnosis?"

significantly. Furthermore, if a query is initially quite long, there may be little advantage to including additional search terms. Thus, it is advantageous to be able to use LCA on a per-query basis.

3.2. Methodology

We want to predict, given a query, if using LCA will improve or hurt the retrieval performance for that query. This prediction must be very fast, since it happens at runtime. A sensible thing to do is to first create a set of training queries. The actual average precision obtained by just InQuery and by InQuery with LCA query expansion on these queries can be computed by using the relevance-tagged document sets as described above. The training data then can be fed into a machine learning algorithm, training it to classify if LCA will improve the average precision for the input query. Artificial neural networks (ANNs) are well suited for this problem, as they are robust with respect to complex, noisy training data and have near-instantaneous evaluation of the learned target function (Mitchell, 1996).

3.2.1. The data set. For our queries and tagged document sets, we made use of the TREC queries, with corresponding TREC document collections and relevance judgments as described in Table 3 (Voorhees, 1999). The shortest formulations of the TREC queries (the title fields) were chosen, as they are most typical of queries given to a web-based search engine (Croft et al., 1995). Furthermore, shorter queries have a higher potential for word

Table 3. Queries, document collections, and relevance judgments used in our experiments.

Query numbers	Doc. collection	TREC
51–100	disk 3	TREC-2
101–150	disk 3	TREC-3
151–200	disks 1 & 2	TREC-3
201–250	disks 2 & 3	TREC-4
251–300	disks 2 & 4	TREC-5
301–350	disks 4 & 5	TREC-6
351–400	disks 4 & 5	TREC-7

mismatch, so they should allow LCA expansion to make more of an impact on retrieval performance.

Because ANNs require numerical inputs, we must use statistics derived from each query in lieu of the actual query. These are the 'query features'. Those features considered are as follows:

Query Length: The length of the query after stopwords have been removed. A stopword is a very common word adding no meaning to the query: e.g. the, with, but, said. Since shorter queries have more potential for useful terms to be added via expansion, it seems logical that LCA would be more likely to result in increased performance in shorter queries as opposed to longer queries. This feature requires negligible online computation time to determine.

IDF Statistics: The Inverse Document Frequency (IDF) of a given term is defined as $\log(\frac{D}{d})$ where D is the total number of documents in the collection, and d is the number of documents in the collection that contain that term. The IDF is generally low for commonly occurring terms, and high for rare ones. The IDF is determined for each term in the query (with stopwords removed), and the Mean IDF, the Max IDF, and the Min IDF over the query terms are computed. If the IDF values for each term in the vocabulary are cached, then computing these features also takes negligible online computation time.

LCA Weights: The $m = 26$ weights computed by the "Get and Rank Concepts" phase of LCA reflect the co-occurrence of the top concepts with the query terms. If the concept weights are low that indicates that these concepts may not be closely related to the original terms, and thus retrieval performance may be hurt by adding them to the query. Determining these features takes significant computation time, as an entire InQuery retrieval is performed, noun phrases are identified and tagged, and weights are computed. However, if the system decides not to use LCA, the returned documents from InQuery are already available, and therefore it is not necessary to run InQuery again.

LCA Weight Statistics: Once the $m = 26$ LCA weights are obtained, various statistics based on them are computed in an attempt to facilitate the ANN training. We consider the Mean, Median, and Standard Deviation of the following: All weights, top 5 weights, top 10 weights, middle 6 weights, middle 12 weights, bottom 10 weights, bottom 5 weights. Computing all of these requires little computation time beyond computing the weights themselves.

3.3. Training method

The NevProp4 neural network simulator was used in our experiments (Goodman, 1998). Many networks were created and trained using various combinations of the inputs described above. Each network was fully connected with a single layer of symmetric logistic hidden units (with range -0.5 to 0.5). For each combination of inputs, three networks were created with the number of hidden units set to be equal to either the number of inputs, half the number of inputs, or a quarter of the number of inputs. The output of the network is dichotomous: the output should be 0 if the network predicts that LCA will not improve the average precision, and 1 if it will. Therefore, a single asymmetric logistic output unit with range 0.0 to 1.0 is used.

Half of the 350 queries were randomly selected to use as training data. The quickprop algorithm was used to update the network weights (Fahlman, 1988). To avoid overfitting of the training data, we used cross-validation with a 10% holdout. This means that 17 queries randomly picked from the training set are set aside to use as 'temporary' testing data, and the neural net is trained on the remaining 158 until the classification error on the 'temporary' testing queries begins to increase. This step is repeated 10 times (randomly picking 17 new queries to hold out each time) for each network to reduce the variance usually associated with cross-validation. The mean testing error over the 10 cross-validation runs is calculated. The network is then reset and retrained using all 175 training queries, stopping training when the mean testing error is achieved.

The best network is then tested against the 175 queries not used in training as an indication of overall network performance.

3.3.1. Results. The two best performing networks are presented here. The first network (PredAll) uses the concept weight median statistics, query length, and IDF mean as inputs with 10 hidden units. The second (PredFast) uses just the IDF min and max, with 2 hidden units. Notice that the PredFast network does not use any of the query features that involve LCA, so it is not necessary to perform the "Get and Rank Concepts" portion of LCA to use this network to make the prediction. The results are presented in Table 4. "Num Times LCA Used" shows how many times LCA was chosen to be executed on the queries. "Num Hurt by LCA" shows how many times the decision to use LCA hurts retrieval performance on a query. The "InQuery" column shows the results for just plain InQuery retrieval, while the "Inq+LCA" column shows the results if LCA is used on every query. As a metric for comparison, the "Perfect" column shows the results if the predictions were made perfectly, using LCA if and only if retrieval quality is increased.

Table 4. Results for PredAll and PredFast.

	InQuery	Inq+LCA	PredAll	PredFast	Perfect
Train (175)					
Num LCA used	0 (0%)	175 (100%)	106 (61%)	52 (30%)	73 (42%)
Num LCA hurt	0 (0%)	102 (58%)	50 (47%)	26 (50%)	0 (0%)
Avg precision	0.241	0.255	0.255	0.246	0.264
% over InQuery		+5.77	+6.02	+2.38	+9.68
% over Inq+LCA			+0.24	−3.20	+3.70
Avg CPU time	5.240	16.921	10.448	8.096	8.364
Test (175)					
Num LCA used	0 (0%)	175 (100%)	93 (53%)	53 (30%)	61 (35%)
Num LCA hurt	0 (0%)	114 (65%)	60 (65%)	40 (75%)	0 (0%)
Avg precision	0.231	0.246	0.238	0.234	0.254
% over InQuery		+6.51	+2.83	+1.08	+9.61
% over Inq+LCA			−3.46	−5.09	+2.91
Avg CPU time	4.587	15.668	8.308	6.829	7.387

PredAll shows a small improvement over Inq+LCA on the training data, but does not improve over Inq+LCA on the testing data. PredFast does not improve over Inq+LCA on either data set. However, both networks used much less CPU time than Inq+LCA, while providing a modest performance gain over using just InQuery. Not surprisingly, PredAll outperforms PredFast, but also uses almost 2 seconds more CPU time per query.

Note that as a research-oriented IR system, InQuery is not designed for speed. This explains why the average running times in all cases are much larger than what one would expect on a commercial system. The great majority of CPU time is spent in the InQuery itself, so plugging in a faster retrieval engine would uniformly scale all CPU times listed above, resulting in almost no relative change between the average running times for different methods.

3.4. Threshold training

For many of the queries it is observed that using LCA changes the average precision very little. These queries basically amount to noise when training an ANN. Furthermore, (Sparck Jones, 1974) suggests that small changes in precision (i.e. less than 5%) are not noticeable to users of an IR system. Perhaps performance can be improved by focusing attention on those queries where LCA changes the average precision greatly. We will train the network only on those queries where the average precision changes more than some percentage (we use a 5% threshold). A network trained in such a way should perform very well on test queries that are above the precision change threshold (or would be, if the actual precisions were known). Intuitively, it is anticipated that the network should predict somewhat well those new queries that are just under the threshold, and that the predictive ability degrades smoothly as the precision change decreases.

The same network parameters as described above are used in these experiments. The training set is created by first finding all queries where using LCA produces an above-threshold change in precision relative to using just InQuery. Then half of those above-threshold queries are randomly selected to be in the training set. The networks are trained using the cross-validation with early stopping method described in Section 3.3. The best network is then tested on *all* remaining queries, regardless of whether they are above the threshold.

3.4.1. Threshold training results. Results are presented in Table 5. We present testing results for both the entire test set and just those test queries above the threshold used for training. The best network for the 5% threshold case (PredAll5) used all concept weight statistics, the query length, and the IDF mean as inputs, with 7 hidden units. Without using LCA-derived features, the best network (PredFast5) used the IDF min and max, with 2 hidden units.

These results are more encouraging, with both networks showing modest performance improvements over Inq+LCA on both train and test data. However, notice that by restricting the training data, the resulting training set is very small (just 40 queries). Furthermore, there are only eight training instances where LCA does not improve the precision. This lack of negative training examples makes prediction more difficult. Presumably adding more queries above the threshold to the training set would further improve prediction accuracy.

Table 5. Results for PredAll5 and PredFast5.

	InQuery	Inq+LCA	PredAll5	PredFast5	Perfect
$>\Delta 5\%$ train (40)					
Num LCA used	0 (0%)	40 (100%)	33 (82%)	38 (95%)	32 (80%)
Num LCA hurt	0 (0%)	8 (20%)	1 (3%)	6 (16%)	0 (0%)
Avg precision	0.338	0.409	0.428	0.416	0.431
% over InQuery		+21.26	+26.72	+23.16	+27.76
% over Inq+LCA			+4.50	+1.56	+5.35
Avg CPU time	5.267	15.401	14.168	14.245	10.994
Test (310)					
Num LCA used	0 (0%)	310 (100%)	267 (86%)	299 (96%)	102 (33%)
Num LCA hurt	0 (0%)	208 (67%)	172 (64%)	198 (66%)	0 (0%)
Avg precision	0.223	0.230	0.231	0.230	0.237
% over InQuery		+3.17	+3.77	+3.18	+6.10
% over Inq+LCA			+0.58	+0.01	+2.84
Avg CPU time	4.868	16.410	15.410	15.867	7.473
$>\Delta 5\%$ test (39)					
Num LCA used	0 (0%)	39 (100%)	34 (87%)	38 (97%)	26 (67%)
Num LCA hurt	0 (0%)	13 (33%)	9 (26%)	13 (34%)	0 (0%)
Avg precision	0.313	0.352	0.360	0.350	0.381
% over InQuery		+12.67	+15.05	+11.83	+21.89
% over Inq+LCA			+2.18	−0.74	+8.18
Avg CPU time	4.577	11.957	10.818	11.811	9.567

Also notice that the 'fast' prediction network does not save much CPU time in this case. The computation time saved in not computing the LCA concept weights was negated by the fact that those networks predict that LCA improves nearly every query, so the concept weights are computed nevertheless as part of the LCA process. We conclude that the information obtained from the intermediate results is crucial in making accurate predictions.

3.5. LCA prediction as a progressive processing task

Given that we have successfully trained a classifier, how do we express this as a PRU? One possibility is shown in figure 4. On the first level the system must generate LCA concepts. On the second level, it can choose to either create the LCA-expanded query or continue with the original query (the *skip* action, shown in the diagram for clarity). The final level does the actual retrieval on the query output by the second level. We use the PredAll5 classifier described in Section 3.4 to estimate the query quality after execution of the first level.

We must specify module descriptors $P_i^j((q', \delta) \mid q)$ for each of the modules described above (Recall that q represents query quality, and δ the query duration). We make the

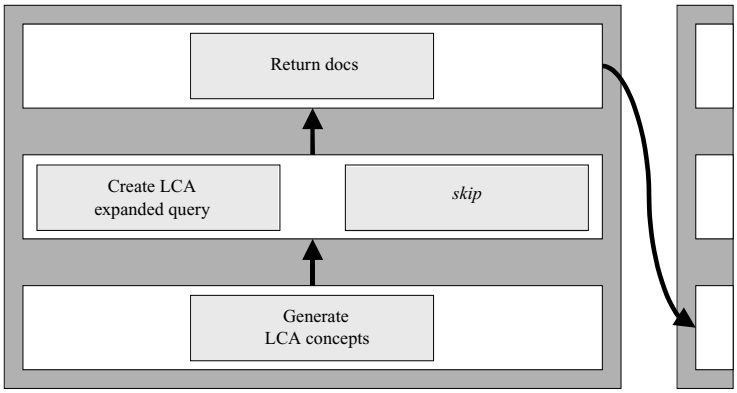

Figure 4. LCA prediction as a PRU.

simplifying assumption that q' and δ are conditionally independent, so it suffices to show $P_i^j(q' \mid q)$ and $P_i^j(\delta \mid q)$ for each of the modules listed above. We can estimate the duration distributions by empirical analysis of the modules on a group of queries. We use the training subset of the TREC queries discussed above and assume that duration is independent of quality.

Generate LCA concepts. The distribution of running time for this module is $P_1^1(\delta) = \mathcal{N}'(\delta \mid 11.34, 11.67)$ where $\mathcal{N}'(x \mid \mu, \sigma)$ here represents the probability of x under a normal distribution with mean μ and standard deviation σ, with the probability mass redistributed so that $P(x \leq 0) = 0$. There is no dependence on q as all queries start with $q = 0$.

We model the quality distribution as an approximation of the output of the PredAll5 classifier, with the adjustment that instead of using the output value x of the network, we will use $100 * (1 - x)$. This gives us qualities on a nice 0 to 100 scale, where queries that are predicted to not be improved by LCA will be given a high quality, and those that may be helped by LCA are given a lower quality. We can use a mixture of two normal distributions to fit the output of the network for all training queries.

$$P_1^1(q) = \frac{3}{4}\mathcal{N}'(q \mid 55.9, 35.6) + \frac{1}{4}\mathcal{N}'(q \mid 7.9, 10.1)$$

Create LCA expanded query. Once the concepts have been generated, creating the new query is near instantaneous, independent of the value of q. We must then do retrieval on this new query, which we empirically observe to be: $P_2^1(\delta) = \mathcal{N}'(\delta \mid 4.955, 3.768)$
For quality, the incoming quality is 'inverted':
$P_2^1(q' = (100 - q) \mid q) = 1$. This encourages the use of the expanded query in cases when the classifier has said that LCA is likely to improve the query, and lessens the incentive to use LCA when it is predicted to not be helpful.

Return docs. In this step, nothing has to be done except to return the documents to the user. The duration will be zero, and the quality will be unchanged.

Although this example was quite simple, it gives an idea of how to formulate a progressive processing task when the application domain has semi-observable quality.

3.6. Progressive processing using online LCA prediction

With the PRU defined we can now examine how a progressive processing controller will behave in a simulated information retrieval system. We compare the progressive processing controller with three other policies in an environment that simulates queries arriving over time at a server which must respond to them.

3.6.1. Query arrival model. To simulate queries arriving over time we use a simple model. At each time t, there is probability p_a of any queries being added to the queue. When queries do arrive, the number arriving is 0 with probability 0.25, 1 with probability 0.4, 2 with probability 0.25, and 3 with probability 0.1. During our experiments, we vary p_a to simulate low and high load conditions. When $p_a = 0.025$, the load on the server is very light, while at $p_a = 0.2$ the load is very heavy.

In order to do any evaluation of this experiment, we need to use queries which have relevance judgments available; Therefore the actual queries arriving at the server are randomly selected, with replacement, from the testing set described in Table 5.

3.6.2. Control policies. We examine the behavior of four different control policies in this environment.

Neighbor. We approximate the optimal control policy using the 1NN strategy described in Section 2.4.1. To do this, we create a set of 100 precompiled policies offline for various queue states encountered over the range of arrival probabilities.

Greedy. This is the single PRU control policy discussed in Section 2.2. Recall that this policy maximizes the utility for the task at the head of the queue, but ignores any waiting queries.

LCA All. This handcoded policy will chose to run LCA on every query. It will also abort the head query in either of two situations: First if the maximum waiting time T has been exceeded for that query, or also if the queue has grown to length greater than 5, and no modules have yet been executed. The value 5 was chosen because it gave the best empirical results. This kind of fixed aborting policy is a more traditional way for a server to cope with an overload of work.

LCA None. This policy will always skip the LCA expansion, and will abort the head query under the same conditions as LCA All.

3.6.3. Experimental setup and parameters. For every arrival rate p_a, we randomly generate a query arrival sequence for time 0 to 1000. We then use each of the above control policies to process the incoming stream of queries over time. We record the quality q (average precision) of the documents returned for each query. We use this quantity and the total processing time t for the query to compute the total utility value of the query using the utility function defined below. Processing ends when the time is greater than 1000 and

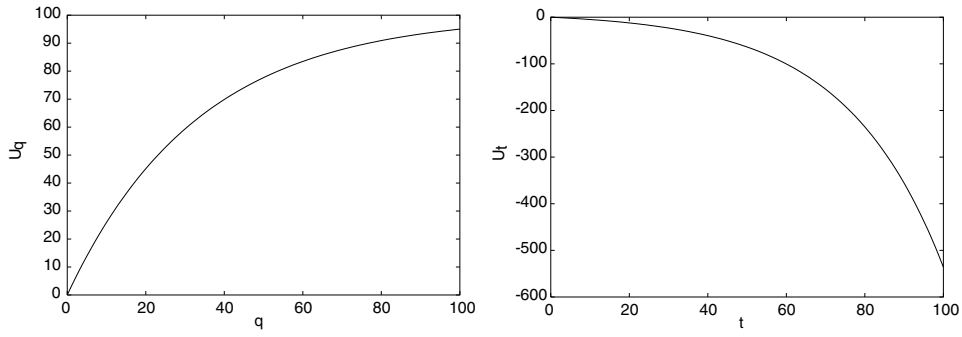

Figure 5. Plots of U_t and U_q.

all queries have been completed or aborted. We then compute average quality and average utility metrics over all queries that arrived during the experiment.

We repeat the above procedure 20 times and record the mean of the average quality and average utility metrics for each control policy and arrival rate.

We also need to define the time-dependent utility function $U(q, t)$, as this function defines the overall quality of service that we wish to maximize. We choose to define U as the sum of the two functions $U_q(q)$ and $U_t(t)$ where

$$U_q(q) = -100e^{(-0.03q)} + 100$$
$$U_t(t) = -10e^{(0.04t)} + 10$$

These two functions are shown in figure 5. U_q demonstrates the law of diminishing returns, where as quality increases, the same unit increase in quality gives less increase in utility. Similarly, U_t shows an increasing penalty as time progresses, reflecting the fact that the value to the user of the final output drops with increasing slope as time progresses.

Quality is discretized to tens, while time discretized into units. We set the maximum possible wait time after which queries must be aborted to $T = 100$.

3.6.4. Results. In figure 6 we observe that for higher loads, all policies experience a drop in average utility per query. For the Greedy policy this drop is extreme (continuing downward until reaching -110 average utility per query for $p_a = 0.2$), due to the fact that the system just tries to maximize utility in the current query, regardless of the state of the queue. Therefore, this policy never "pre-emptively" aborts a query, as the other three policies do in some form, and it will also never skip the LCA process unless doing so will provide greater utility for the current query. The LCA All policy suffers from the inability to skip as well: the system must invest the time to perform the LCA expansion for every query, even though doing so may have adverse effect of increasing the waiting time for all those remaining in the queue. The LCA None policy turns out to be fairly competitive with the Neighbor policy, but recall that the length of the queue above which aborts would automatically occur was optimized empirically.

Figure 7 demonstrates how for all policies the average quality of the returned documents decays as the load on the server increases (we say that an aborted query has quality 0). The

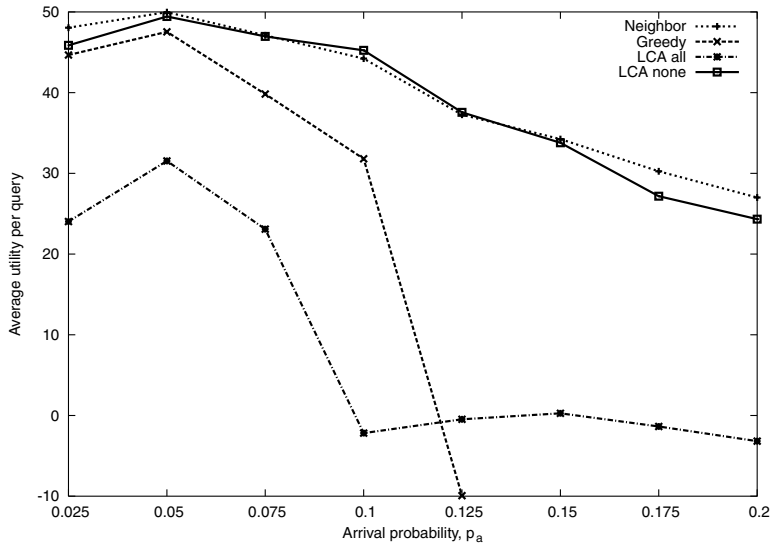

Figure 6. The average utility achieved per query for various server loads.

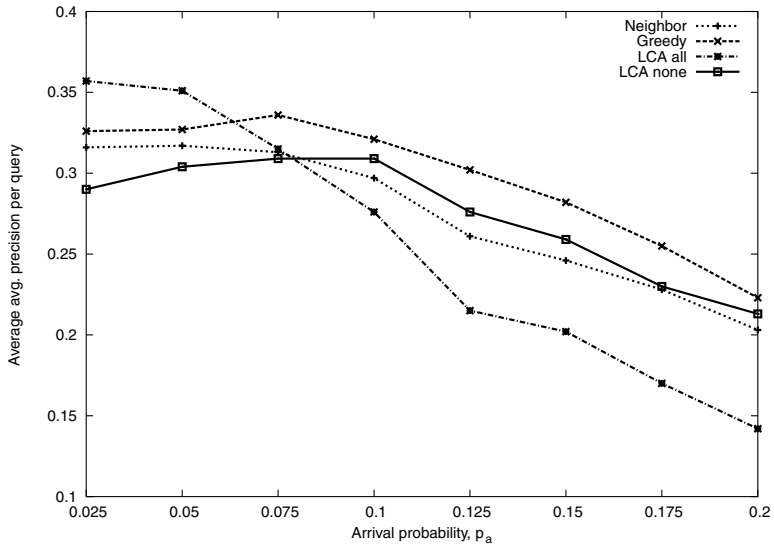

Figure 7. The quality of the returned documents decays as server load increases.

LCA All policy has the most extreme drop, due to the fact that many queries must be aborted due to excessive queue length or exceeding T. Note that the Neighbor policy has a higher average quality than LCA None when the load is light, due to the fact that it can often use LCA to boost quality on queries which are predicted to be helped by the technique, while at higher loads, this opportunity becomes increasingly rare.

3.7. Discussion

Although the results on predicting the effectiveness of LCA are somewhat weak due to a scarcity of training data, and perhaps too simple a feature set, we see that such prediction is still valuable when built into the progressive processing system. Training the ANN classifier on only 'extreme' examples provides us the ability to predict with confidence when LCA will greatly improve or greatly harm a query. The rough quality estimates provided by the classifier allow for a robust policy to be computed that will use LCA only when the classifier is confident that the query will be improved, while taking into account the ramifications to the rest of the queue if that computation time is so invested. As server load increases, and the queue grows, the LCA prediction must be increasingly confident if the system is to choose to run LCA.

4. Related work

There has been much research into the application of machine learning techniques to the creation and configuration of IR systems. One of the earliest examples is the AIR system of Belew (1986). It used relevance feedback from users to train a three layer neural network of authors, index terms, and documents. In Leuski (2001), reinforcement learning was used to develop a system to help guide users through a collection of retrieved documents. A good survey of other applications of machine learning techniques to the retrieval process itself is given in Chen (1995). These can be called "adaptive" IR systems in that their parameters are slowly learned and tuned over time. However, the work presented here is fundamentally different and novel, in that the IR system is dynamically reconfigured to use best techniques for a given query, user, and server load at runtime.

The ability to dynamically adjust computational effort based on the availability of computational resources and the projected benefit from continued computation has been studied extensively by the AI community since the mid 1980's. These efforts have led to the development of a variety of techniques such as *anytime algorithms* (Dean and Boddy, 1988; Zilberstein and Russell, 1996), *design-to-time* (Garvey and Lesser, 1993), *flexible computation* (Horvitz, 1988), *imprecise computation* (Liu et al., 1991), and *progressive reasoning* (Mouaddib, 1993; Mouaddib and Zilberstein, 1997). Each resource-bounded reasoning approach addresses two fundamental questions: how to introduce computational tradeoffs into the base-level problem-solving process and how to control these tradeoffs so as to optimize the utility of the system. For example, in the case of anytime algorithms, the base-level problem solving technique is an interruptible, iterative improvement process that provides multiple solutions to a given problem whose quality improves with computation time. Using a variety of representations of the performance of the algorithm, researchers have proposed several techniques to control anytime algorithms such as a static allocation of the algorithm's running time before it starts (Horvitz, 1987; Boddy and Dean, 1994), a myopic stopping criterion based on the marginal value of computation (Horvitz, 1990; Russell and Wefald, 1991), and a non-myopic approach that treats the stopping criterion as a sequential decision problem (Hansen and Zilberstein, 1996).

Resource-bounded reasoning techniques offer a disciplined approach to managing computational resources in complex systems. Currently, the primary method for achieving real-time performance is based in many cases on speeding up individual algorithms in a generate-and-test manner until an acceptable performance is reached. In contrast, resource-bounded reasoning techniques allow a system to perform a computation based on a well-defined notion of the *value of computation* (Horvitz, 1990; Russell and Wefald, 1991) which measures the net gain in performance. The gain is calculated by projecting the improvement in response quality on the one hand and the cost of computation on the other hand.

From a resource-bounded reasoning perspective, the dynamic selection of information retrieval techniques present several unique challenges. First, the set of queries waiting for processing is changing rapidly, making it necessary to revise the value of computation. Second, the effect of a technique on the precision of the response may not be immediate. It may depend on further processing, making it impossible to use the more classical *myopic* approach to estimating the value of computation (Horvitz, 1990; Russell and Wefald, 1991). A myopic estimate of the value of a computation is based on the assumption that the final system response will be returned at the end of the computation. We treat the problem as a sequential decision problem, factoring into the value function the ability of the system to make further decisions before returning the result. These two characteristics, a rapidly changing set of queries and a non-myopic approach, distinguish our solution from previous work in resource-bounded reasoning.

The progressive processing framework used in this paper is related to a large body of work within the systems community on *imprecise computation* (Liu et al., 1991). Each task in that model is decomposed into a *mandatory* subtask and an *optional* subtask. The mandatory subtask must be executed to produce results of some initial value; the optional subtask may be executed to increase the value of the results. With few exceptions, tasks in this model are assumed to be independent and to have individual deadlines. A variety of scheduling algorithms have been developed for imprecise computation under different assumptions about the optional part. Our model allows for a richer representation of quality and duration uncertainty and quality dependency. Unlike imprecise computation, the schedule constructed in this paper is a *conditional schedule*; the selection of IR techniques is conditioned on the *actual* execution time and outcome of previous techniques. As a result, the system can handle effectively a high level of uncertainty.

The application of dynamic programming to solve the problem of meta-level control of computation has been previously used by Hansen and Zilberstein (1996) to control interruptible anytime algorithms. Optimal monitoring of progressive processing tasks using a corresponding MDP has been studied by Mouaddib and Zilberstein (1998) with respect to a simpler task structure and without the notion of quality uncertainty and quality dependency.

The use of pre-compiled control policies to construct a highly reactive real-time system has been studied by several researchers. For example, Greenwald and Dean (1998) show how a real-time avionics control system can use a library of schedules that cover all possible situations. Each schedule is conditioned on the state of the flight operation. Another advantage of the use of pre-compiled control policies is the reduction of communications in interleaved planning/execution systems since the approach determines which module to select given the current state of execution.

The notion of *opportunity cost* that we use to measure the effect of the delay in processing existing queries is borrowed from economics. It has been used previously in meta-level reasoning by Russell and Wefald (1991). Horvitz (1997) uses a similar notion to develop a model of *continual computation* in which idle time is used to solve anticipated future problems.

To review, the work presented in this paper differs from previous uses of machine learning in IR in that other methods learn a fixed system, but we use the learned classifiers to dynamically reconfigure the IR system at runtime. The progressive processing method for control of computation differs from previously explored approaches in that we use a non-myopic approach that can effectively handle the high levels of uncertainty resulting from the rapidly changing set of waiting queries.

5. Conclusions

This paper examined the possibility of run-time selection of the best information retrieval techniques for a given query. We first developed a new approach to representing the problem within the progressive processing framework. The resulting meta-level control was solved by reformulating it as a Markov decision problem. It was shown that an optimal policy for a set of tasks can be constructed by controlling a single progressive processing unit, taking into account the opportunity cost associated with the remaining queries. A fast approximation of the opportunity cost was developed that allows a reactive controller to select the best IR techniques using a library of pre-compiled control policies.

We then examined the ability to predict the performance of IR techniques using LCA as a case study. We showed that online query prediction using threshold training can provide an improvement in retrieval performance. Although the results were only modestly good, they demonstrate that is possible to predict success well enough to help retrieval effectiveness and improve efficiency on average. A simple example PRU structure based on this prediction was given, and a full progressive processing system based on this structure was built and compared to other meta-level control strategies.

We have shown that time-consuming IR techniques can be integrated in a robust way into systems, even when it is not beneficial to use them on every query, or in situations where the system is under high-load. The overhead introduced by the run-time selection mechanism is minimal because it is based on off-line learning methods.

Future work should address a few issues: More training queries are needed, especially more negative instances. Creating training queries is hardly a trivial task, as hundreds of manual relevance judgments have to be made. At the very least, there is the possibility that future TREC conferences might provide more queries. Secondly, the query features used were fairly arbitrary. We simply used features that were easy to collect and compute, that were not 'tuned' to the task of online query prediction. Perhaps with more carefully engineered features, prediction performance could be increased. However, we have found no obvious candidates thus far. Additionally, new IR modules can be designed with this framework in mind, allowing for techniques that may only be beneficial to a only a specific kind of query, or that may take a larger amount of computional resources than would otherwise be tolerable in a static IR system. As long as the success rate is somewhat

predictable, this dynamic selection system can benefit from them. Finally, the methods for computing the optimal policy presented here do not consider query arrivals that may occur while a module is running. If the query arrival rate can be modeled and future events anticipated, there may be an opportunity to increase overall quality of service.

Although we investigate only LCA here, online query prediction could be a useful tool for many different IR techniques. Online query prediction is an important step in developing a truly dynamic information retrieval system. While the results presented here are modest, we feel that dynamic information retrieval systems present a previously unexplored method for improving retrieval performance in terms of both quality and speed. Furthermore, the progressive processing framework allows us to take into account such things as system load and query priority when deciding to use some IR technique.

References

Allan, J. (1995). Relevance Feedback With Too Much Data. In *Research and Development in Information Retrieval* (pp. 337–343).

Allan, J. and Raghavan, H. (2002). Using Part-of-Speech Patterns to Reduce Query Ambiguity. In *Proceedings of the 25th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval*.

Attar, R. and Fraenkel, A.S. (1977). Local Feedback in Full-Text Retrieval Systems. *Journal of the ACM*, 24(3), 397–417.

Barto, A., Bradtke, S.J., and Singh, S.P. (1995). Learning to Act Using Real-Time Dynamic Programming. *Artificial Intelligence*, 72, 81–138.

Belew, R.K. (1986). Adaptive Information Retrieval: Machine Learing in Associative Networks. Ph.D. thesis, University of Michigan.

Boddy, M. and Dean, T. (1994). Decision-Theoretic Deliberation Scheduling for Problem Solving in Time-Constrained Environments. *Artificial Intelligence*, 67, 245–285.

Buckley, C. and Voorhees, E.M. (2000). Evaluating Evaluation Measure Stability. In *Proceedings of the 23rd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 33–40).

Callan, J.P., Croft, W.B., and Broglio, J. (1995). TREC and Tipster Experiments with InQuery. *Information Processing and Management*, 31(3), 327–343.

Chen, H. (1995). Machine Learning for Information Retrieval: Neural Networks, Symbolic Learning, and Genetic Algorithms. *Journal of the American Society for Information Science*, 46(3), 194–216.

Conrad, J.G. and Utt, M.H. (1994). A System for Discovering Relationships by Feature Extraction from Text Databases. In W.B. Croft and C.J. van Rijsbergen (Eds.), *Proceedings of the 17th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval* (pp. 260–270).

Croft, W.B., Cook, R., and Wilder, D. (1995). Providing Government Information on the Internet: Experiences with THOMAS. In *Proceedings of the Second Annual Conference on the Theory and Practice of Digital Libraries*.

Croft, W.B. and Harper, D.J. (1979). Using Probabilistic Models of Document Retrieval Without Relevance Information. *Journal of Documentation*, 35, 285–295.

Crouch, C.J., Crouch, D.B., and Chen, Q. (2001). Initial Experiments in Short Query Retrieval. Technical Report TR-00-01, University of Minnesota Duluth.

Davis, M. and Dunning, T. (1996). A TREC Evaluation of Query Translation Methods for Multi-Lingual Text Retrieval. In *Proceedings of TREC-4*.

Dean, T. and Boddy, M. (1988). An Analysis of Time-Dependent Planning. In *Proceeedings of the Seventh National Conference on Artificial Intelligence* (pp. 49–54).

Dean, T., Kaelbling, L.P., Kirman, J., and Nicholson, A. (1995). Planning Under Time Constraints in Stochastic Domains. *Artificial Intelligence*, 76, 35–74.

Fahlman, S.E. (1988). An Empirical Study of Learning Speed in Back-Propagation Networks. Technical Report, Carnegie Mellon University.

Garvey, A. and Lesser, V. (1993). Design-to-Time Real-Time Scheduling. *IEEE Transactions on Systems, Man, and Cybernetics*, 23(6), 1491–1502.

Goodman, P.H. (1998). *NevProp Software, Version 4*. University of Nevada, Reno.

Greenwald, L. and Dean, T. (1998). A Conditional Scheduling Approach to Designing Real-Time Systems. In *Artificial Intelligence Planning Systems* (pp. 224–231).

Hansen, E.A. and Zilberstein, S. (1996). Monitoring the Progress of Anytime Problem-Solving. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence* (pp. 1229–1234).

Horvitz, E.J. (1987). Reasoning About Beliefs and Actions Under Computational Resource Constraints. In *Proceedings of the Workshop on Uncertainty in Artificial Intelligence*.

Horvitz, E.J. (1988). Reasoning Under Varying and Uncertain Resource Constraints. In *National Conference on Artificial Intelligence* (pp. 111–116).

Horvitz, E.J. (1990). Computation and Action Under Bounded Resources. Ph.D. thesis, Stanford University.

Horvitz, E.J. (1997). Models of Continual Computation. In *Fourteenth National Conference on Artificial Intelligence* (pp. 286–293).

Leuski, A. (2001). Interactive Information Organization: Techniques and Evaluation. Ph.D. thesis, University of Massachusetts at Amherst.

Liu, J., Lin, K., Shih, W., Yu, A., Chung, J., and Zao, W. (1991). Algortihms for Scheduling Imprecise Computations. *IEEE Transactions on Computers*, 24(5), 58–68.

Mitchell, T.M. (1996). *Machine Learning*. New York, US: McGraw Hill.

Mouaddib, A.I. (1993). Contribution au Raisonnement Progressif et Temps rel dans un Univers Multi-Agents. Ph.D. thesis, Univeristy of Nancy I.

Mouaddib, A.I. and Zilberstein, S. (1997). Handling Duration Uncertainty in Meta-Level Control of Progressive Processing. In *Fifteenth International Joint Conference on Artificial Intelligence* (pp. 1201–1206).

Mouaddib, A.I. and Zilberstein, S. (1998). Optimal Scheduling of Dynamic Progressive Processing. In *Thirteenth Biennial European Conference on Artificial Intelligence* (pp. 449–503).

Robertson, S.E. and Walker, S. (1997). On Relevance Weights with Little Relevance Information. In *Proceedings of the 20th Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval* (pp. 16–23).

Russell, S. and Wefald, E. (1991). *Do the Right Thing: Studies in Limited Rationality*. Cambridge, MA: MIT Press.

Singhal, A. and Pereira, F. (1999). Document Expansion for Speech Retrieval. In *Research and Development in Information Retrieval* (pp. 34–41).

Sparck Jones, K. (1971). *Automatic Keyword Classification for Information Retrieval*. Butterworths, London.

Sparck Jones, K. (1974). Automatic Indexing. *Journal of Documentation*, 30, 393–432.

Swanson, D.R. (1988). Historical Note: Information Retrieval and the Future of an Illusion. *Journal of the American Society for Information Science*, 39, 92–98.

Voorhees, E.M. (1999). Overview of the Eighth Text REtrieval Conference. In *Proceedings of TREC-8*.

Xu, J. (1997). Solving the Word Mismatch Problem through Automatic Text Analysis. Ph.D. thesis, University of Massachusetts at Amherst.

Xu, J. and Croft, W.B. (1996). Query Expansion Using Local and Global Document Analysis. In *Proceedings of the 19th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval* (pp. 4–11).

Xu, J. and Croft, W.B. (2000). Improving the Effectiveness of Information Retrieval with Local Context Analysis. *ACM Transactions on Information Systems*, 18(1), 79–112.

Yang, Y., Carbonell, J.G., Brown, R.D., and Frederking, R.E. (1998). Translingual Information Retrieval: Learning from Bilingual Corpora. *Artificial Intelligence*, 103(1/2), 323–345.

Zilberstein, S. and Russell, S.J. (1996). Optimal Composition of Real-Time Systems. *Artificial Intelligence*, 82(1/2), 181–213.