ELSEVIER

Artificial Intelligence 126 (2001) 139–157

Artificial Intelligence

www.elsevier.com/locate/artint

Monitoring and control of anytime algorithms: A dynamic programming approach

Eric A. Hansen [a,*], Shlomo Zilberstein [b]

[a] *Computer Science Department, Mississippi State University, Mississippi, MS 39762, USA*
[b] *Computer Science Department, University of Massachusetts, Amherst, MA 01002, USA*

Abstract

Anytime algorithms offer a tradeoff between solution quality and computation time that has proved useful in solving time-critical problems such as planning and scheduling, belief network evaluation, and information gathering. To exploit this tradeoff, a system must be able to decide when to stop deliberation and act on the currently available solution. This paper analyzes the characteristics of existing techniques for meta-level control of anytime algorithms and develops a new framework for monitoring and control. The new framework handles effectively the uncertainty associated with the algorithm's performance profile, the uncertainty associated with the domain of operation, and the cost of monitoring progress. The result is an efficient non-myopic solution to the meta-level control problem for anytime algorithms. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Anytime algorithms; Flexible computation; Meta-level control; Monitoring; Resource-bounded reasoning; Real-time deliberation

1. Introduction

From its early days, the field of artificial intelligence (AI) has searched for useful techniques for coping with the computational complexity of decision making. An early advocate of the view that decision makers (both human and artificial) should forgo perfect rationality in favor of limited, economical reasoning was Herbert Simon. In 1958, he claimed that "the global optimization problem is to find the least-cost or best-return decision, net of computational costs" [32]. The statistician I.J. Good advocated a similar approach to decision making that takes deliberation costs into account [9].

[*] Corresponding author.
E-mail address: hansen@cs.msstate.edu (E.A. Hansen).

By the mid 1980s, AI researchers began to formalize these ideas and produce effective models for trading off computational resources for quality of results (e.g., [19]). At the forefront of these efforts were two, essentially identical, models called "anytime algorithms" (developed by Dean and Boddy [2,4]) and "flexible computation" (developed by Horvitz [13,15,17]). We adopt the phrase "anytime algorithm" in this paper, although our work is equally influenced by both models.

The defining property of an anytime algorithm is that it can be stopped at any time to provide a solution, and the quality of the solution increases with computation time. This property allows a tradeoff between computation time and solution quality, making it possible to compute approximate solutions to complex problems under time constraints. Anytime algorithms are being used increasingly in a range of practical domains that include planning and scheduling [2,38], belief network and influence diagram evaluation [12,16, 36], database query processing [31,33], and information gathering [10]. By itself, however, an anytime algorithm does not provide a complete solution to Simon's challenge to make the best-return decision, net of computational costs. To achieve this, a meta-level control procedure in needed that determines how long to run the anytime algorithm, and when to stop and act on the currently available solution.

Horvitz developed an approach to meta-level control of computation based on the *expected value of computation* (EVC) [17]. The expected value of computation is defined as the expected improvement in decision quality that results from performing a computation, taking into account computational costs. (The concept of the expected value of computation is related to the concept of the expected value of information, although the two are not identical [22].) The meta-level control problem for anytime algorithms is the problem of determining the stopping time for an anytime algorithm that optimizes the expected value of computation.

Meta-level control of an anytime algorithm can be approached in two different ways. One approach is to allocate the algorithm's running time before it starts [2,13]. If there is little or no uncertainty about the rate of improvement of solution quality, or about how the urgency for a solution might change after the start of the algorithm, then this approach works well. Very often, however, there is uncertainty about one or both. For AI problem-solving in particular, variance in the rate at which solution quality improves is common [24]. Because the best stopping time will vary with fluctuations in the algorithm's performance (and/or the state of the environment), a second approach to meta-level control is to monitor the progress of the algorithm (and/or the state of the environment) and determine at run-time when to stop deliberation and act on the currently available solution [3,17,37].

The optimal time allocation for an anytime algorithm depends on several factors: the quality of the available solution, the prospect for further improvement in solution quality, the current time, the cost of delay in action, the current state of the environment, and the prospect for further change in the environment. In this paper, we develop a framework for run-time monitoring and control of anytime algorithms that takes into account these various factors. The solution differs from previously developed approaches to this problem that rely on a myopic estimate of the value of computation. We formalize the meta-level control problem as a sequential decision problem that can be solved by dynamic programming, in order to construct a non-myopic solution. Dynamic programming has been used before for meta-level control of computation. Einav and Fehling [7] and Russell and Subramanian [27] use dynamic programming to schedule a sequence of solution methods for a real-time decision-making problem; and Zilberstein, Charpillet and Chassaing [40] use dynamic programming to schedule a sequence of of contract algorithms to create the best interruptible system given a stochastic deadline. The framework developed in this paper uses dynamic programming to determine the stopping time of anytime algorithm, by monitoring its progress.

The paper is organized as follows. Section 2 briefly reviews a framework for meta-level control of anytime algorithms that does not use monitoring. The rest of the paper extends this framework to include run-time monitoring. Section 3 begins with a review of a myopic approach to run-time monitoring and meta-level control that is based on the work of Horvitz [17] and Russell and Wefald [30]. Then it introduces a non-myopic framework that determines not only when to stop an anytime algorithm, but at what intervals to monitor its progress and re-assess whether to continue or stop. Section 4 extends this framework to include situations in which the state of the environment must be monitored. The paper concludes with a discussion of some extensions of the framework and future work.

2. Meta-level control without monitoring

We begin by briefly reviewing a framework for meta-level control of anytime algorithms that does not use run-time monitoring. This framework provides a foundation for the extensions introduced in the rest of the paper.

A framework for meta-level control of anytime algorithms requires a model of how the quality of a solution produced by the algorithm increases with computation time, as well as a model of the time-dependent utility of a solution. The phrase *performance profile* was introduced by Dean and Boddy [4] to refer to a model of the performance of an anytime algorithm that specifies expected quality as a function of computation time. Horvitz [13,17] introduced a model that characterizes uncertainty about an algorithm's performance using probabilities, and we follow Zilberstein and Russell [39] in calling this a *probabilistic performance profile*.

Definition 1. A *probabilistic performance profile* of an anytime algorithm, $Pr(q_j|t)$, denotes the probability of getting a solution of quality q_j by running the algorithm for t time units.

Although it is sometimes possible to represent a performance profile by a parameterized continuous function [4,15], a convenient and widely-used representation is a table of discrete values and this is the representation we assume in this paper. [1] Computation time is discretized into a finite number of time steps, $t_0 \dots t_n$, where t_0 represents the starting time of the algorithm and t_n its maximum running time. Similarly, solution quality is discretized

[1] A potential advantage of generalizing our results to a functional (rather than tabular) representation of a performance profile is that a functional representation may be more compact, and may be constructed more simply.

into a finite number of levels, $q_0 \ldots q_m$, where q_0 is the lowest quality level and q_m is the highest quality level. The fineness of the discretization fixes a tradeoff between the accuracy of the performance profile and the space needed to store it. The values defining the performance profile are collected by statistical analysis of the performance of the algorithm.

In addition to a performance profile that models the behavior of an algorithm, meta-level control requires a model of the time-dependent utility of a solution [2,18]. For now, we assume that utility depends only on the quality of a solution and computation time. (In Section 4, we consider a more general model in which utility also depends on the state of a dynamic environment.)

Definition 2. A *time-dependent utility function*, $U(q,t)$, represents the utility of a solution of quality q at time t.

It is often possible to simplify this function by treating it as the sum of two functions that Horvitz [14] calls *object-level utility* and *inference-related utility*. Object-level utility represents the utility of a solution without regard to the costs associated with its computation, and inference-related utility represents these computational costs. We adopt the following terminology of Russell and Wefald [29] to refer to this distinction.

Definition 3. A time-dependent utility function is called *time-separable* if the utility of a solution of quality q at time t, $U(q,t)$, can be expressed as the difference between two functions,

$$U(q,t) = U_I(q) - U_C(t),$$

where $U_I(q)$ is called the *intrinsic value function* and $U_C(t)$ is called the *cost of time*.

Given an anytime algorithm, its performance profile, and a time-dependent utility function, the meta-level control problem is the problem of deciding when to stop the algorithm and act on the currently available solution. One approach to this problem is to determine the algorithm's running time before it starts [2,13]. Because the time allocation is not revised after the start of the algorithm, we call this a fixed allocation approach.

Definition 4. Given a time-dependent utility function and a probabilistic performance profile, an *optimal fixed allocation*, t^*, is defined by:

$$t^* = \arg\max_t \sum_i Pr(q_i|t)U(q_i,t).$$

Fig. 1 illustrates the selection of an optimal, fixed time allocation for the meta-level control problem. The figure does not illustrate uncertainty about the rate of improvement of solution quality or about the cost of time, however. This framework for meta-level control is best suited for problems for which there is little or no uncertainty about either. The rest of the paper discusses how to extend this framework to compensate for uncertainty by using run-time monitoring.

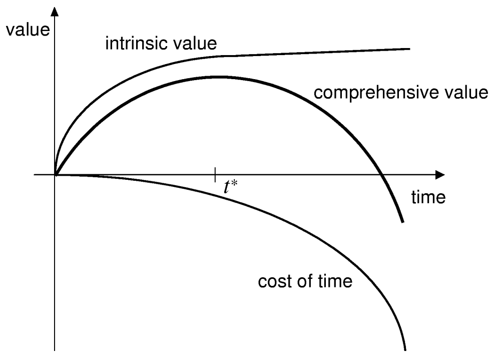

Fig. 1. A typical view of anytime algorithms [5,17,30].

3. Monitoring the progress of computation

In this section, we describe how to generalize the framework introduced in the previous section to incorporate run-time monitoring of the progress of computation. If there is uncertainty about the progress of an anytime algorithm, monitoring the algorithm's progress and revising the initial time allocation when progress is faster or slower than expected can improve the utility of the system. This generalization requires a more complex performance profile that allows prediction of the progress of an algorithm to be revised based on intermediate results. It also presents a more complex optimization problem; instead of a single once-and-for-all decision about how long to run an algorithm, a sequence of decisions about whether the continue or stop the algorithm is required.

3.1. Dynamic performance profiles

A performance profile that predicts solution quality as a function of an algorithm's overall running time is suitable for making a one-time decision about how long to run an algorithm, before the algorithm starts. To take advantage of information gathered by monitoring the progress of an anytime algorithm, a more informative performance profile is needed that conditions predicted improvement of solution quality on features of the currently available solution that can be monitored at run time. These features can be defined broadly enough to include the entire state of the computation. However, it is useful (and, we claim, usually sufficient) to condition a run-time prediction of further improvement on the quality of the currently available solution. We define a dynamic performance profile accordingly.

Definition 5. A *dynamic performance profile* of an anytime algorithm, $Pr(q_j|q_i, \Delta t)$, denotes the probability of getting a solution of quality q_j by continuing the algorithm for time interval Δt when the currently available solution has quality q_i.

The assumption that further improvement of solution quality depends only on the quality of the currently available solution is equivalent to assuming that solution quality satisfies the Markov property for this prediction problem. For many anytime algorithms, conditioning a prediction of future improvement on the quality of the currently available solution works well—particularly if solution quality is easily determined at run-time. But in some cases, the quality of the currently available solution may not be the best (or only) predictor of likely improvement. In other cases, it may not be possible to determine the quality of the currently available solution at run-time.

Whether it is easy to determine solution quality at run-time, or predict improvement in quality based on the quality of the currently available solution, depends on how solution quality is defined. For example, if solution quality is defined as the value of an objective function that is iteratively improved by an optimization algorithm, solution quality can be easily determined at run-time. In the case of the traveling salesman problem, for example, solution quality defined in this way is simply the length of the currently available tour. However, this may not be the best predictor of subsequent improvement in solution quality. The optimal value of an objective function usually differs from one problem instance to the next. For example, the minimum tour length for one instance of the traveling salesman problem may be 100, while for another instance of the traveling salesman problem it may be 250. Solution quality defined as tour length will not distinguish these cases, and thus will not be a good predictor of subsequent improvement. A better approach would take into account how far the current tour length is from the optimal tour length.

For this reason, solution quality is often defined as the difference between the current value of an objective function and its optimal value. For cost-minimization problems, solution quality is customarily defined as the "approximation ratio":

$$\mathit{Cost}(\mathit{Approximate\ solution})/\mathit{Cost}(\mathit{Optimal\ solution})$$

The inverse ratio is used for value-maximization problems. This definition of solution quality makes possible general claims about the performance of an algorithm on a class of problem instances. However, defining solution quality in this way poses a problem for run-time monitoring: assessing solution quality requires knowing the optimal solution. But a run-time monitor needs to make a decision based on the approximate solution currently available, without knowing what the optimal solution will eventually be. As a result, it cannot know, with certainty, the actual quality of the approximate solution.

This observation holds for other classes of problems besides combinatorial optimization problems. For problems that involve estimating a point value, the difference between the estimated point value and the true point value can't be known until the algorithm has converged to an exact value [17]. For anytime problem-solvers that rely on abstraction to create approximate solutions, solution quality may be difficult to assess for other reasons. For example, it may be difficult for a run-time monitor to predict the extent of reactive planning needed to fill in the details of an abstract plan [38].

For many problems, it is possible to bound the degree of approximation to an optimal solution. Because a run-time monitor can only estimate where the optimal solution falls within this bound, it is sometimes useful to treat the error bound itself as a measure of solution quality. For example, Horvitz [17] uses an error bound to measure solution quality in meta-level control of an approximation algorithm for belief network evaluation. Similarly, de Givry and Verfaillie [6] use an error bound to measure solution quality in meta-level control of an anytime algorithm for solving constraint optimization problems.

For now, we assume that solution quality is defined in such a way that it can be observed at run time and used to improve prediction of the subsequent progress of the algorithm. When a run-time monitor cannot easily determine the quality of the currently available solution, or when solution quality itself is not the best predictor of subsequent improvement, the success of run-time monitoring depends on being able to design a reliable scheme for estimating/predicting solution quality at run-time. We discuss this issue at greater length in Section 3.5.

3.2. A myopic stopping criterion

Instead of making a single decision about how long to run an anytime algorithm before it starts, a run-time monitor must make a sequence of decisions. Each time it monitors an algorithm's progress, it must decide whether to continue deliberation or stop. It is easy to state an optimal stopping rule: computation should stop as soon as the expected value of computation is no longer positive [17]. However, exact determination of the EVC requires considering all possible decisions about whether to continue or stop that could be made at subsequent time steps. Therefore, a myopic estimate of the EVC is often used. A myopic estimate of the EVC is the expected utility of acting on the result that will be available after continuing the algorithm for exactly one more time step minus the expected value of acting immediately on the result currently available. Using our representation of performance profiles, a myopic estimate of the expected value of computation is defined as follows.

Definition 6. Suppose that an anytime algorithm produces a solution of quality q_i at time t. Then a *myopic estimate of the expected value of computation* (MEVC) is:

$$\mathrm{MEVC}(\Delta t) = \sum_j Pr(q_j|q_i, \Delta t)U(q_j, t + \Delta t) - U(q_i, t),$$

where Δt denotes an additional time step.

A myopic estimate of the value of computation is used by a run-time monitor that periodically checks the progress of the algorithm and decides whether to continue computation or stop.

Definition 7. The *myopic monitoring approach* is to continue the computation as long as $\mathrm{MEVC}(\Delta t) > 0$.

Although a stopping rule that relies on myopic computation of EVC is not optimal under all circumstances, it can be shown to perform optimally under some assumptions. The following theorem gives sufficient conditions for its optimality.

Theorem 1. *Given a time-dependent utility function, the myopic monitoring approach is optimal when*:

(1) *evaluating the MEVC takes a negligible amount of time*; *and*
(2) *for every time t and quality level q for which MEVC is non-positive, MEVC is also non-positive for every time $t + \Delta t$ and quality level $q + \Delta q$.*

Proof. If MEVC is positive when the currently available solution has quality level q at time t, continuing the algorithm for another time step has positive expected value and is the optimal decision. If MEVC is non-positive when the currently available solution has quality level q at time t, then any policy for continuing cannot improve expected value unless there is some time $t + \Delta t$ and quality level $q + \Delta q$ for which MEVC is positive. The second condition guarantees that this cannot happen. Therefore, stopping the algorithm is the optimal decision. □

The second condition of Theorem 1 takes a simpler and more intuitive form for time-separable utility functions.

Corollary 1. *The second condition in Theorem* 1 *is met when the expected marginal increase in the intrinsic value of a solution is a non-increasing function of quality and the marginal cost of time is a non-decreasing function of time.*

Proof. Under the separability assumption, a myopic estimate of the expected value of computation can be expressed as:

$$\mathrm{MEVC}(q_i, \Delta t) = \left[\sum_j Pr(q_j|q_i, \Delta t)U_I(q_j) - U_I(q_i)\right] - \left[U_C(t+\Delta t) - U_C(t)\right].$$

The first term in the expression on the right-hand side of this equation represents the expected marginal increase in the intrinsic value of a solution. The assumption that this is a non-increasing function of quality, combined with monotonic improvement of quality with computation time, means this quantity cannot increase from one time step to the next. The second term in the expression on the right-hand side of this equation represents the marginal cost of time. By assumption, it is a non-decreasing function of time. Therefore, the future marginal cost can only become larger. As a consequence, the EVC can only become smaller as time and quality progress. In other words, if it is negative at one point it cannot be positive in any future situation. □

Because the assumptions on which they rest are reasonably intuitive, Theorem 1 and its corollary suggest that there is a large class of applications for which the myopic approach to meta-level control is optimal. Nevertheless, there are cases in which reliance on MEVC can lead to a sub-optimal stopping decision. Horvitz and Breese [3,17] describe a bounded conditioning algorithm for probabilistic inference in belief networks for which MEVC can mislead because expected improvement is “flat” in the near term but substantial after some number of time steps. Because this can lead to a premature stopping decision, Horvitz suggests various degrees of lookahead to compute EVC more reliably. Horsch and Poole [12] address a similar problem in considering meta-level control of an anytime algorithm for influence diagram evaluation. To counter it, they describe a non-myopic approach to estimating the EVC based on a linear model constructed from empirical data. The approach to meta-level control described in the following section addresses this same problem by extending the framework for meta-level control introduced earlier.

3.3. A non-myopic stopping criterion

Using run-time monitoring to determine when to stop an anytime algorithm presents a sequence of decisions; at each step, the decision is whether to continue the algorithm or stop and act on the currently available solution. In this section, we formalize and solve this sequential decision problem. The solution to the problem is a non-myopic monitoring policy that takes into account the effect of future monitoring decisions.

Definition 8. A *monitoring policy*, $\pi(q_i, t_k)$, is a mapping from time step t_k and quality level q_i to a decision whether to continue the algorithm or stop and act on the currently available solution.

To construct a non-myopic monitoring policy, we follow the well-established literature on using dynamic programming for solving optimal stopping problems. If utility only depends on solution quality and computation time, a stopping rule can be found by optimizing the following value function,

$$V(q_i, t) = \max_d \begin{cases} U(q_i, t) & \text{if } d = \text{stop}, \\ \sum_j Pr(q_j|q_i, \Delta t)V(q_j, t + \Delta t) & \text{if } d = \text{continue} \end{cases}$$

to determine the following policy,

$$\pi(q_i, t) = \arg\max_d \begin{cases} U(q_i, t) & \text{if } d = \text{stop}, \\ \sum_j Pr(q_j|q_i, \Delta t)V(q_j, t + \Delta t) & \text{if } d = \text{continue}, \end{cases}$$

where Δt represents a single time step and d is a binary variable that represents the decision to either stop or continue the algorithm. The optimal policy can be computed off-line using a dynamic programming algorithm with complexity $O(|m|^2|n|)$, where m is the number of quality levels and n is the number of time steps. Because computation is off-line, this is an example of what Horvitz calls compilation of metareasoning.

Once an optimal monitoring policy is computed, meta-level control is straightforward. Every time step, the current solution quality and the current time are used to determine the best action (stop or continue), by indexing the policy. This leads to a globally optimal (non-myopic) stopping criterion.

Theorem 2. *A monitoring policy that maximizes the above value function is optimal when quality improvement satisfies the Markov property and monitoring has no cost.*

Proof. This is an immediate outcome of the application of dynamic programming under the Markov assumption [1]. The Markov assumption requires that the probability distribution of future quality depends only on the current "state" of the anytime algorithm, which is taken to be the quality of the currently available solution. $\square$

We note that the same monitoring policy may also be computed using the state-space search algorithm AO* [21], if an appropriate heuristic is available. AO* finds an optimal solution for an initial state, which for this problem is $(q_{start}, 0)$, and may find the same monitoring policy more efficiently than dynamic programming if many states are not reachable from the initial state under an optimal policy.

3.4. Cost-sensitive monitoring

Many existing approaches to monitoring assume continuous or periodic monitoring that incurs negligible cost [30,34,38]. In many situations, however, run-time monitoring may have a significant overhead because of the need to determine the current solution quality (and sometimes also the current state of the environment). If that information is available at no cost, monitoring every time step may be reasonable. But suppose that monitoring incurs a constant cost, C. In this case, we need to construct a more complex type of monitoring policy.

Definition 9. A *cost-sensitive monitoring policy*, $\pi_c(q_i, t_k)$, is a mapping from time step t_k and quality level q_i into a monitoring decision $(\Delta t, m)$ such that Δt represents the additional amount of time to allocate to the anytime algorithm, and m is a binary variable that represents whether to monitor at the end of this time allocation or to stop without monitoring.

In other words, a cost-sensitive monitoring policy specifies, for each time step t_k and quality level q_i, the following two decisions:

(1) how much additional time to run the algorithm; and
(2) whether to monitor at the end of this time allocation and re-assess whether to continue, or to stop without monitoring.

An initial decision, $\pi_c(q_{start}, t_0)$, specifies how much time to allocate to the algorithm before monitoring for the first time or stopping without monitoring. Note that the variable Δt makes it possible to control the time interval between one monitoring action and the next; its value can range from 0 to $t_n - t_i$, where t_n is the maximum running time of the algorithm and t_i is how long it has already run. The binary variable m makes it possible to allocate time to the algorithm without necessarily monitoring at the end of the time interval; its value is either *stop* or *monitor*.

Given this formalization, it is possible to use dynamic programming to compute a combined policy for monitoring and stopping. Although dynamic programming is often used to solve optimal stopping problems, the novel aspect of this solution is that dynamic programming is used to determine when to monitor, as well as when to stop and act on the currently available solution. A cost-sensitive monitoring policy is found by optimizing the following value function:

$$V_c(q_i, t_k) = \max_{\Delta t, m} \begin{cases} \sum_j Pr(q_j | q_i, \Delta t) U(q_j, t_k + \Delta t) & \text{if } m = \text{stop}, \\ \sum_j Pr(q_j | q_i, \Delta t) V_c(q_j, t_k + \Delta t) - C & \text{if } m = \text{monitor}. \end{cases}$$

Theorem 3. *A cost-sensitive monitoring policy that maximizes the above value function is optimal when quality improvement satisfies the Markov property.*

The proof is identical to that of Theorem 2 except that the cost of monitoring is taken into account.

An advantage of this framework is that it makes it possible to find an intermediate strategy between continuous monitoring and not monitoring at all. It can recognize whether or not monitoring is cost-effective, and when it is, it can adjust the frequency of monitoring to optimize utility. The resulting policy often recommends monitoring more frequently near the expected stopping time of an algorithm, which is an intuitive strategy.

3.5. Estimating solution quality

We have described a framework for computing a cost-sensitive monitoring policy for an anytime algorithm. Besides the assumption that quality improvement satisfies the Markov property, the optimality of the policy depends on the assumption that the quality of the currently available solution can be determined accurately by a run-time monitor. As argued in Section 3.1, there are classes of problems for which determining the precise quality of a solution at run-time is not feasible.

When the quality of approximate solutions cannot be accurately determined at run-time, a run-time monitor must rely on some method for estimating solution quality. A monitoring policy must be conditioned on this estimate of solution quality rather than on solution quality itself. It is impossible to specify a general solution to this problem. How solution quality is estimated will vary from algorithm to algorithm, and how a monitoring policy is conditioned on such an estimate may also vary. However, we can adopt the following general notation. Let f denote some "feature" of the currently available solution that is either highly correlated with solution quality, or else a good predictor of improvement. If improvement in solution quality does not depend exclusively on this feature (i.e., we cannot assume the Markov property), run-time prediction can be further improved by conditioning the prediction on the cumulative running time of the algorithm, t. Accordingly, we let $Pr(q \mid f, t, \Delta t)$ denote the probability that continuing an algorithm for time Δt when the currently available solution exhibits feature f after running time t results in a solution of quality q. We note that an algorithm's running time provides probabilistic evidence about the quality of the currently available solution that supplements the evidence provided by the observed feature. It also makes possible an important guarantee: conditioning a performance profile on both running time and some feature of the currently available solution ensures a prediction that is at least as good as a prediction based on running time alone.

In Section 3.6, we describe a simple example in which this run-time prediction scheme is used. In this example, the "feature" that is monitored is the cumulative improvement in solution quality from the start of the algorithm. Conditioning an estimate/prediction of quality on both cumulative improvement and running time means that both the amount and the average rate of improvement are considered, but not the actual trajectory of improvement. A more accurate estimator/predictor might be achieved by conditioning on the trajectory also, or on other run-time information. For example, Raman and Wah [25] predict future improvement in solution quality based on the observed trajectory of improvement from the start of the algorithm, using nonlinear regression to make the prediction. However, they note that use of nonlinear regression incurs a substantial run-time overhead and this suggests a tradeoff between the accuracy of a run-time estimator/predictor and the overhead it incurs.

We can construct a dynamic performance profile that is conditioned on an estimate of solution quality, as follows. Let $Pr(q_i \mid f_r, t_k)$ denote the probability that the currently available solution has quality q_i when the run-time monitor observes feature f_r after running time t_k. In addition, let $Pr(f_r|q_i, t_k)$ denote the probability that the run-time monitor will observe feature f_r if the currently available solution has quality q_i after running time t_k. These probabilities can be determined from statistical analysis of the behavior of the algorithm. Together with the dynamic performance profile defined earlier, they can be used to calculate the following probabilities for use in predicting the improvement of solution quality after additional time allocation Δt when the quality of the currently available solution can only be estimated.

$$Pr(q_j|f_r, t_k, \Delta t) = \sum_i Pr(q_i|f_r, t_k) Pr(q_j|q_i, \Delta t),$$

$$Pr(f_s|f_r, t_k, \Delta t) = \sum_i Pr(q_i|f_r, t_k) \sum_j Pr(q_j|q_i, \Delta t) Pr(f_s|q_j, t_k + \Delta t).$$

These probabilities can also be determined directly from statistical analysis of the behavior of the algorithm, without the intermediate calculations. In either case, these probabilities make it possible to find a monitoring policy that is conditioned on an estimate of solution quality. The policy is found by using dynamic programming to optimize the following value function.

$$V_c(f_r, t_k) = \max_{\Delta t, m} \begin{cases} \sum_j Pr(q_j|f_r, t_k, \Delta t) U(q_j, t_k + \Delta t) & \text{if } m = \text{stop}, \\ \sum_s Pr(f_s|f_r, t_k, \Delta t) V_c(f_s, t_k + \Delta t) - C & \text{if } m = \text{monitor}. \end{cases}$$

The resulting policy may not be optimal in the sense that it may not take advantage of all possible run-time evidence about solution quality; for example, it may ignore the trajectory of observed improvement. However, the approach we have described is simple and it provides an important guarantee: it only recommends monitoring if it results in a higher expected value than allocating a fixed running time without monitoring. This makes it possible to distinguish cases in which monitoring is cost-effective from cases in which it is not. Whether monitoring is cost-effective will depend on the variance of the performance profile, the time-dependent utility of the solution, how well the quality of the currently available solution can be estimated by the run-time monitor, and the cost of monitoring. All of these factors are weighed in computing this monitoring policy.

3.6. Example

As an example of how this framework can be used to determine a combined policy for monitoring and stopping, we apply it to a tour improvement algorithm for the traveling salesman problem developed by Lin and Kernighan [20]. This local optimization algorithm begins with an initial tour, then repeatedly tries to improve the tour by swapping random paths between cities. This algorithm is representative of anytime algorithms that have variance in solution quality as a function of computation time.

In our experiment, solution quality is defined as the approximation ratio of a tour, *Length*(*Current tour*)/*Length*(*Optimal tour*), and discretized using Table 1(a). The maximum running time of the algorithm is discretized into twelve time-steps, with one time-step corresponding to approximately 0.005 CPU seconds. A dynamic performance profile is compiled by generating and solving a thousand random twelve-city traveling

Table 1
Discretization of: (a) solution quality, and (b) feature correlated with quality

Quality	$\frac{Length(Current\ tour)}{Length(Optimal\ tour)}$
5	$1.05 \rightarrow 1.00$
4	$1.10 \rightarrow 1.05$
3	$1.20 \rightarrow 1.10$
2	$1.35 \rightarrow 1.20$
1	$1.50 \rightarrow 1.35$
0	$\infty \rightarrow 1.50$

(a)

Feature	$\frac{Length(Current\ tour)}{Length(Lower\ bound)}$
6	$1.3 \rightarrow 1.0$
5	$1.4 \rightarrow 1.3$
4	$1.5 \rightarrow 1.4$
3	$1.6 \rightarrow 1.5$
2	$1.7 \rightarrow 1.6$
1	$2.0 \rightarrow 1.7$
0	$\infty \rightarrow 2.0$

(b)

Table 2
Optimal cost-sensitive monitoring policy based on actual solution quality

	Time-step											
Quality	Start	1	2	3	4	5	6	7	8	9	10	11
5	–	0	0	0	0	0	0	0	0	0	0	0
4	–	1M	1M	1M	1M	1M	1M	1M	1M	1M	1	0
3	–	1M	1M	1M	1M	1M	1M	1M	1M	1M	1	0
2	–	3M	3M	3M	3M	3M	3M	3M	3M	2	1	0
1	–	4M	4M	4M	4M	4M	5	4	3	2	1	0
0	5M	5M	5M	5M	5M	6	5	4	3	2	1	0

salesman problems. The time-dependent utility of a solution of quality q_i at time t_k is arbitrarily defined by the function

$$U(q_i, t_k) = 100q_i - 20t_k.$$

Note that the first term of the utility function can be regarded as the *intrinsic value* of a solution and the second term as the *time cost*, as defined in Section 2.

Without monitoring, the optimal running time of the algorithm is eight time-steps, with an expected value of 269.2. Assuming solution quality can be measured accurately by the run-time monitor (an unrealistic assumption in this case) and assuming a monitoring cost of 1, the dynamic programming algorithm described earlier computes the monitoring policy shown in Table 2. The number in each cell of Table 2 represents how much additional time to allocate to the algorithm based on the observed quality of the solution and the current time. The letter M next to a number indicates a decision to monitor at the end of this time allocation, and possibly allocate additional running time; if no M is present, the decision is to stop at the end of this time allocation without monitoring. The policy has an expected value of 303.3. This is better than the expected value of allocating a fixed running time, despite the added cost of monitoring. The improved performance is due to the fact that the run-time monitor can stop the algorithm after anywhere from 5 to 11 time steps, depending on how quickly the algorithm finds a good result. (If there is no cost for monitoring, a policy that monitors every time step has an expected value of 309.4.)

The policy shown in Table 2 was constructed by assuming the actual quality of an approximate solution could be determined by the run-time monitor. This is an unrealistic assumption because the quality of the current tour is defined with reference to the length of an optimal tour. To estimate the quality of the current tour, we need to estimate the length of an optimal tour. One approach to estimating the optimal tour length is to compute a lower bound on the optimal length by solving a relaxation of the problem [26]. For a traveling salesman problem that satisfies the triangle inequality, there exist polynomial-time algorithms that can compute a lower bound that is on average within two or three percent of the optimal tour length. For our test, we simply used Prim's minimal spanning tree algorithm that very quickly computes a bound that is less tight, but still correlated with the optimal tour length. The feature, $\mathit{Length}(\mathit{Current\ tour})/\mathit{Length}(\mathit{Lower\ bound})$, used to estimate solution quality was discretized using Table 1(b). The cost overhead of monitoring consists of computing the lower bound at the beginning of the algorithm and monitoring the current tour length at intervals thereafter.

Table 3 shows the monitoring policy given a monitoring cost of 1, when an estimate of solution quality is conditioned on both this feature and the running time of the algorithm. The expected value of the policy is 282.3. This is better than the expected value of allocating a fixed running time without monitoring. But it is worse than the expected value that could be achieved if the run-time monitor could determine the actual quality of an approximate solution. This demonstrates that the less accurately a run-time monitor can measure the quality of an approximate solution, the less valuable it is to monitor.

When an estimate of solution quality is based only on this feature, and not also on running time, the expected value of monitoring is 277.0. This is still an improvement over not monitoring, but the performance is not as good as when an estimate is conditioned on running time as well. Because conditioning a dynamic performance profile on running time significantly increases its size, however, this tradeoff may be acceptable in cases

Table 3
Cost-sensitive monitoring policy when solution quality is estimated

	Time-step											
Feature	Start	1	2	3	4	5	6	7	8	9	10	11
6	–	–	–	0	0	0	0	0	0	0	0	0
5	–	–	1	1	1	1	0	0	0	0	0	0
4	–	–	2M	2M	1M	1M	1M	1M	1	0	0	0
3	–	4M	3M	2M	1M	1M	1M	1M	1M	1	0	0
2	–	4M	3M	2M	2M	2M	2M	1M	3	2	1	0
1	–	4M	3M	3M	3M	3M	3M	2M	2M	1M	1	–
0	5M	4M	3M	3M	3M	3M	3M	–	–	–	–	–

when the feature used to estimate quality is very reliable. For all of these results, the improved performance predicted by dynamic programming was confirmed by simulation experiments.

For this simple example, variance in improvement of solution quality is small and the improved performance with run-time monitoring is correspondingly small. For problems for which variance in improvement of solution quality is larger, the payoff for run-time monitoring will be greater. It is significant that this technique improves performance even when variance is small, solution quality is difficult to estimate at run-time, and monitoring incurs a cost.

4. Monitoring the environment

We have described a framework for meta-level control of anytime algorithms that uses run-time monitoring to compensate for uncertainty about the progress of computation. For a large class of problems, the rate of improvement of solution quality is the only source of uncertainty about how long to continue deliberation. Examples include optimizing a database query [31], reformulating a belief net before solving it [3], and planning the next move in a chess game [30]. For other problems, the utility of a solution may also depend on the state of a dynamic environment that can change unpredictably after the start of the algorithm. Examples include real-time planning [2], mobile robot control [8], medical diagnosis [17], and image tracking [35]. For such problems, meta-level control can be further improved by monitoring the state of the environment as well as the progress of problem-solving.

In this section, we discuss how to extend our framework for meta-level control to allow monitoring the environment as well as monitoring the progress of computation. In this extension, the utility of a solution is a function of the state of the environment as well as of time. We modify our definition of the utility function accordingly.

Definition 10. A *state-dependent utility function*, $U(q,s,t)$, represents the utility of a solution of quality q at time t given that the state of the environment is s.

When utility depends on the state of a dynamic environment, a meta-level controller needs a predictive model of how the state of the environment changes over time. Issues that arise in predicting change in the state of the environment are well-studied in the literature on modeling dynamical systems and we do not review them here. For the purpose of meta-level control, most details of the environment may be abstracted away and only those that reflect the degree of time pressure must be modeled. For example, time pressure from the environment may sometimes simply be modeled as a stochastic deadline. As a general notation, we let $Pr(s_y|s_x,\Delta t)$ denote the probability that the problem state will be s_y after time Δt if the current problem state is s_x.

Given a probabilistic performance profile, a state-dependent utility function and an initial environment state s_0, we can determine an optimal fixed time allocation as follows:

$$t^* = \arg\max_t \sum_i \sum_x Pr(q_i|t)Pr(s_x|s_0,t)U(q_i,s_x,t).$$

When there is uncertainty about the state of the environment, performance can be improved by replacing this fixed allocation approach with a more flexible approach to meta-level control that monitors the state of the environment. In principle, this meta-level control problem can be formulated by generalizing the approach we have used so far. However, there are more options to consider. At any given time, it is possible to monitor the computation only, the environment only, both the computation and the environment, or neither. It is also possible to stop the computation and return the result. To select the best option, the monitor must keep track of the following information,

$$[(q, t_q), (s, t_s), t],$$

where q represents the most recently observed quality (or state of computation), t_q represents the time of this observation, s represents the most recently observed state of the environment, t_s represents the time of this observation, and t is the current time. Given this information, a dynamic performance profile, a model of the environment, and a cost model for monitoring, an optimal monitoring policy can be determined. However, it will be larger and more complex than a policy for monitoring the progress of computation only. Some simplifying assumptions can reduce its complexity.

Synchronous monitoring

One simplification is to synchronize the process of monitoring the environment with the process of monitoring the computation. In other words, the only options considered are monitoring both the state of the environment and the state of the computation, or monitoring neither. Let C represent the combined cost of monitoring. Then a cost-sensitive monitoring policy can be found by optimizing the following value function:

$$V_c(q_i, s_x, t_k) = \max_{\Delta t, m} \begin{cases} \sum_{j,y} Pr(q_j|q_i, \Delta t) Pr(s_y|s_x, \Delta t) U(q_j, s_y, t_k + \Delta t) \\ \quad \text{if } m = \text{stop}, \\ \sum_{j,y} Pr(q_j|q_i, \Delta t) Pr(s_y|s_x, \Delta t) V_c(q_j, s_y, t_k + \Delta t) - C \\ \quad \text{if } m = \text{monitor}. \end{cases}$$

Note that synchronous monitoring simplifies both the state representation and the number of control actions to be considered. Synchronous monitoring is particularly suitable when both of the monitored processes (computation and environment) have roughly the same cost, the same degree of uncertainty, and the same level of influence on the comprehensive value. In such cases, the desired frequencies of monitoring for the two processes are roughly the same, so synchronizing them is a reasonable simplification.

Asynchronous monitoring

In developing asynchronous monitoring policies for the computation and the environment, some simplifying assumptions can reduce the size of the policy and the time it takes to compute it. One way to decouple the decision to monitor the computation from the decision to monitor the environment is to assume that one has a significant cost while the other incurs little or none. In practice, monitoring the environment requires collecting and interpreting sensory data to determine the state of the domain. If the cost of monitoring the environment dominates the cost of monitoring computation, a monitoring policy only needs to determine when to monitor the environment (assuming that the computation is monitored every time step).

If there are substantial costs for monitoring both the environment and the computation, one can simplify the value function by treating the problems of when to monitor the environment and when to monitor the computation as independent. A policy for monitoring the environment may be determined by assuming the algorithm behaves in accordance with its average performance profile. Similarly, a policy for monitoring the computation may be determined by assuming the state of the environment unfolds as expected. Simplifying a control problem by ignoring uncertainty, that is, by assuming a process exhibits its average-case behavior, is called *certainty equivalence control*. For many problems, it serves as a useful approximation [1].

5. Conclusion

We have presented a framework for monitoring and control of anytime algorithms that formalizes the meta-level control problem as an optimal stopping problem and uses dynamic programming to find a non-myopic stopping rule that is sensitive to the variance of the algorithm's performance, the time-dependent utility of a solution, the ability of the run-time monitor to estimate the quality of the currently available solution, and the cost of monitoring. By taking into account all of these factors, this framework makes it possible to evaluate tradeoffs that influence the utility of monitoring. For example, the dynamic programming technique is sensitive to both the cost of monitoring and to how well the quality of the currently available solution can be estimated by the run-time monitor. This makes it possible to evaluate a tradeoff between these two factors. It is likely that there will be more than one method for estimating a solution's quality and the estimate that takes longer to compute will be more accurate. Is the greater accuracy worth the added time cost? This question can be answered by computing a monitoring policy for each method and comparing their expected values to select the best one.

Although we have focused on the meta-level control problem for anytime algorithms, a similar framework may be used to develop non-myopic strategies for other resource-bounded reasoning techniques. For example, Russell and Wefald [28,30] describe an approach to meta-level control of real-time search algorithms that uses an estimate of the value of computation to select which node of the search tree to expand next, as well as when to terminate the search and act on the currently available solution. To simplify this meta-level control problem, they make a *meta-greedy* assumption that is equivalent to estimating the value of computation myopically. Recognizing that a non-myopic approach can lead to better performance, Harada and Russell [11] consider treating this meta-level control problem as a Markov decision problem and propose using reinforcement learning with value function approximation techniques to solve it.

Mouaddib and Zilberstein [23,41] have developed a similar framework for controlling *progressive processing* task structures. In this model, a system satisfies a set of information requests under time pressure by limiting the amount of processing allocated to each task based on a predefined hierarchical task structure. Using dynamic programming, Mouaddib and Zilberstein show how to construct an optimal monitoring policy that results in significant improvement over heuristic scheduling techniques in handling the duration uncertainty associated with the basic computational components. The progressive processing model has recently been extended to allow monitoring of other resources besides computation time [42].

The problem of monitoring a system composed of multiple anytime algorithms has been studied by Zilberstein [37]. He shows how to compile a performance profile for a complete system from the performance profiles of its components. Run-time monitoring can improve the performance of the system when the actual quality of a solution generated by an anytime component differs from its expected quality, or when the environment changes unpredictably. Without describing this work in detail, we note the analogy to the problem of monitoring individual anytime algorithms discussed in this paper. In both cases, run-time monitoring is an option when the problem of allocating computation time can be treated as a sequence of predictions and decisions instead of a single, once-and-for-all decision.

Acknowledgements

Support for this work was provided in part by the National Science Foundation under Grant No. IRI-9409827, Grant No. IRI-9624992, and Grant No. IIS-9984952. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies or endorsements of the US Government. The authors thank the anonymous reviewers for providing valuable feedback on the original manuscript.

References

[1] D.P. Bertsekas, Dynamic Programming: Deterministic and Stochastic Models, Prentice-Hall, Englewood Cliffs, NJ, 1987.
[2] M. Boddy, T. Dean, Deliberation scheduling for problem solving in time-constrained environments, Artificial Intelligence 67 (1994) 245–285.
[3] J.S. Breese, E.J. Horvitz, Ideal reformulation of belief networks, in: Proc. 6th Conference on Uncertainty in Artificial Intelligence, Cambridge, MA, 1990, pp. 129–143.
[4] T. Dean, M. Boddy, An analysis of time-dependent planning, in: Proc. AAAI-88, St. Paul, MN, 1988, pp. 49–54.
[5] T.L. Dean, M.P. Wellman, Planning and Control, Morgan Kaufmann, San Mateo, CA, 1991.
[6] S. de Givry, G. Verfaillie, Optimum anytime bounding for constraint optimization problems, in: AAAI Workshop on Building Resource-Bounded Reasoning Systems, 1997, pp. 37–42.
[7] D. Einav, M.R. Fehling, Computationally-optimal real-resource strategies, in: Proc. IEEE International Conference on Systems, Man and Cybernetics, 1990, pp. 581–586.
[8] E. Gat, Integration planning and reacting in a heterogeneous asynchronous architecture for controlling real-world mobile robots, in: Proc. AAAI-92, San Jose, CA, 1992, pp. 809–815.
[9] I.J. Good, Twenty-seven principles of rationality, in: V.P. Godambe, D. Sprott (Eds.), Foundations of Statistical Inference, Holt, Rinehart, Winston, Toronto, 1971, pp. 108–141.
[10] J. Grass, S. Zilberstein, A value-driven system for autonomous information gathering, Intelligent Information Systems (2000), to appear.
[11] D. Harada, S. Russell, Extended abstract: Learning search strategies, in: AAAI Spring Symposium on Search Techniques for Problem Solving under Uncertainty and Incomplete Information, 1999.
[12] M.C. Horsch, D. Poole, Estimating the value of computation in flexible information refinement, in: Proc. 7th Conference on Uncertainty in Artificial Intelligence, 1999.
[13] E.J. Horvitz, Reasoning about beliefs and actions under computational resource constraints, in: Proc. Workshop on Uncertainty in Artificial Intelligence, 1987.

[14] E.J. Horvitz, Reasoning under varying and uncertain resource constraints, in: Proc. AAAI-88, St. Paul, MN, 1988, pp. 111–116.
[15] E.J. Horvitz, G.F. Cooper, D.E. Heckerman, Reflection and action under scarce resources: Theoretical principles and empirical study, in: Proc. IJCAI-89, Detroit, MI, 1989, pp. 1121–1127.
[16] E.J. Horvitz, H.J. Suermondt, G.F. Cooper, Bounded conditioning: Flexible inference for decisions under scarce resources, in: Proc. 5th Workshop on Uncertainty in Artificial Intelligence, Windsor, Ont., 1989.
[17] E.J. Horvitz, Computation and action under bounded resources, Ph.D. Thesis, Stanford University, 1990.
[18] E.J. Horvitz, G. Rutledge, Time-dependent utility and action under uncertainty, in: Proc. 7th Conference on Uncertainty in Artificial Intelligence, Los Angeles, CA, 1991, pp. 151–158.
[19] V. Lesser, J. Pavlin, E. Durfee, Approximate processing in real-time problem solving, AI Magazine 9 (1988) 49–61.
[20] S. Lin, B.W. Kernighan, An effective heuristic algorithm for the Traveling Salesman problem, Oper. Res. 21 (1973) 498–516.
[21] A. Martelli, U. Montanari, Optimizing decision trees through heuristically guided search, Comm. ACM 21 (1978) 1025–1039.
[22] J. Matheson, The value of analysis and computation, IEEE Transactions on Systems Science, and Cybernetics 4 (1968) 211–219.
[23] A.I. Mouaddib, S. Zilberstein, Optimal scheduling of dynamic progressive processing, in: Proc. 13th Biennial European Conference on Artificial Intelligence (ECAI-98), Brighton, UK, 1998, pp. 449–503.
[24] V.J. Paul, A. Acharya, B. Black, J.K. Strosnider, Reducing problem-solving variance to improve predictability, Comm. ACM 34 (8) (1991) 80–93.
[25] S. Raman, B. Wah, Quality-time tradeoffs in simulated annealing for VLSI placement, in: Proc. 15th International Computer Software and Applications Conference, 1991.
[26] G. Reinelt, The Traveling Salesman: Computational Solutions for TSP Applications, Springer, Berlin, 1994.
[27] S. Russell, D. Subramanian, Provably bounded-optimal agents, J. Artificial Intelligence Res. 1 (1995) 1–36.
[28] S. Russell, E. Wefald, On optimal game-tree search using rational metareasoning, in: Proc. IJCAI-89, Detroit, MI, 1989, pp. 334–340.
[29] S. Russell, E. Wefald, Principles of metareasoning, Artificial Intelligence 49 (1991) 361–395.
[30] S. Russell, E. Wefald, Do the Right Thing: Studies in Limited Rationality, MIT Press, Cambridge, MA, 1991.
[31] S. Shekhar, S. Dutta, Minimizing response times in real time planning and search, in: Proc. IJCAI-89, Detroit, MI, 1989, pp. 238–242.
[32] H.A. Simon, Models of Bounded Rationality, Vol. 2, MIT Press, Cambridge, MA, 1982.
[33] K.P. Smith, J.W.S. Liu, Monotonically improving approximate answers to relational algebra queries, in: Proc. COMPSAC, Orlando, FL, 1989.
[34] L. Steinberg, J.S. Hall, B.D. Davison, Highest utility first search across multiple levels of stochastic design, in: Proc. AAAI-98, Madison, WI, 1998, pp. 477–484.
[35] K. Toyama, Handling tradeoffs between precision and robustness with incremental focus of attention for visual tracking, in: AAAI Fall Symposium on Flexible Computation in Intelligent Systems, 1996.
[36] M.P. Wellman, C.-L. Liu, State-space abstraction for anytime evaluation of probabilistic networks, in: Proc. 10th Conference on Uncertainty in Artificial Intelligence, Seattle, WA, 1994, pp. 567–574.
[37] S. Zilberstein, Operational rationality through compilation of anytime algorithms, Ph.D. Dissertation, Computer Science Division, University of California at Berkeley, 1993.
[38] S. Zilberstein, Resource-bounded sensing and planning in autonomous systems, Autonomous Robots 3 (1996) 31–48.
[39] S. Zilberstein, S.J. Russell, Optimal composition of real-time systems, Artificial Intelligence 82 (1–2) (1996) 181–213.
[40] S. Zilberstein, F. Charpillet, P. Chassaing, Real-time problem-solving with contract algorithms, in: Proc. IJCAI-99, Stockholm, Sweden, 1999, pp. 1008–1013.
[41] S. Zilberstein, A.-I. Mouaddib, Reactive control of dynamic progressive processing, in: Proc. IJCAI-99, Stockholm, Sweden, 1999, pp. 1268–1273.
[42] S. Zilberstein, A.-I. Mouaddib, Optimizing resource utilization in planetary rovers, in: Proc. 2nd NASA International Workshop on Planning and Scheduling for Space, 2000, pp. 163–168.