Monitoring Anytime Algorithms

Eric A. Hansen and Shlomo Zilberstein
Computer Science Department
University of Massachusetts
Amherst, MA 01003
{hansen,shlomo}@cs.umass.edu

Abstract

Anytime algorithms offer a tradeoff between solution quality and computation time that has proved useful in applying artificial intelligence techniques to time-critical problems. To exploit this tradeoff, a system must be able to determine the best time to stop deliberation and act on the currently available solution. If there is uncertainty about how much solution quality will improve with computation time, or about how the problem state may change after the start of the algorithm, monitoring the algorithm's progress and/or the problem state can make possible a better stopping decision and so improve the utility of the system. This paper analyzes the issues involved in run-time monitoring of anytime algorithms. It reviews previous work and casts the problem in a new framework from which some improved monitoring strategies emerge.

1 Introduction

Anytime algorithms are being used increasingly for time-critical problem-solving in domains such as planning and scheduling [1][5], belief network evaluation [9][18], database query processing [16][17], and others. The defining property of an anytime algorithm is that it can be stopped at any time to provide a solution, where 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. It also introduces a problem of meta-level control: making an optimal time/quality tradeoff requires determining how long to run the algorithm, and when to stop and act on the currently available solution.

Meta-level control of an anytime algorithm can be approached in two different ways. One approach is to allocate an algorithm's running time before it starts, and to let the algorithm run for the predetermined length of time no matter what [1]. If there is no uncertainty (or minimal 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 can determine an optimal stopping time. Very often, however, there is uncertainty about one or both. For AI problem-solving in particular, variance in solution quality is common [11]. Because the best stopping time will vary with fluctuations in the algorithm's performance, and/or with unexpected changes 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 to determine at run-time when to stop deliberation and act on the currently available solution [6][19].

In this paper, we discuss some of the issues involved in run-time monitoring of anytime algorithms. We describe a formal framework for monitoring both the progress of an algorithm and the state of the environment. However, our discussion focuses on issues involved in monitoring an algorithm's progress. This is the more interesting half of the problem because it requires modeling the behavior of the anytime algorithm itself, as opposed to modeling the environment. For a large class of problems, uncertainty about how long to continue deliberation depends only on uncertainty about how much and how fast solution quality will improve. Examples include plannning and optimizing a database query [16], reformulating a belief net before solving it [2], and planning the next move in a chess game [15]. There are other problems, however, for which utility also depends on the state of a dynamic environment and performance can be improved by monitoring the environment as well as the progress of the algorithm. We reserve a place for this in our formal framework, but we discuss the problem of monitoring the environment only briefly.

The outline of the paper is as follows. Section 2 briefly reviews a standard framework for meta-level control of anytime algorithms without monitoring. Section 3 discusses how to extend this framework for run-time monitoring by developing sequential prediction models that can be used to make improved predictions of solution quality at run-time. Section 4 describes how to determine an optimal stopping time in this new framework, which requires sequential decision-making. Section 5 discusses how to minimize the run-time overhead for monitoring. Section 6 describes a simple example that illustrates some of these ideas. Finally, section 7 discusses related work.

2 Background

Meta-level control of an anytime algorithm – deciding how long to run the algorithm and when to stop and act on the currently available solution – requires both a model of how solution quality improves with computation time and a model of the time-dependent utility of a solution. The first model is given by a performance profile of the algorithm. A *probabilistic performance profile* maps computation time to a probability distribution over solution quality, with $P(q|t)$ denoting the probability that running the algorithm for time t produces a solution of quality q. An *expected performance profile* maps computation time to expected solution quality, $E(q|t)$. In the rest of this paper we rely on probabilistic performance profiles, although most of the framework we describe works with expected performance profiles as well.

Although it is sometimes possible to represent a performance profile by a compact parameterized function, a more general representation is a table of discrete values and this is the representation we will assume. It requires discretizing time into a finite number of time steps and discretizing quality into a finite number of levels. The fineness of the discretization fixes a tradeoff between the accuracy of the performance profile and the space needed to store it. The values in 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. If utility only depends on solution quality and computation time, let $U(q,t)$ denote the utility of stopping the algorithm at time t with a solution of quality q. The running time of an anytime algorithm can be determined prior to its execution, without monitoring, by solving

$$\arg\max_t \sum_q P(q|t)U(q,t).$$

For some problems, utility also depends on the state of the environment when the solution is acted on – where a dynamically changing environment introduces time pressure. To model uncertainty about change in the environment after the start of the algorithm, let $P(s|t)$ denote the probability that state s will obtain at time t. Let $U(q,s,t)$ denote the utility of stopping an anytime algorithm after time t with a solution of quality q in state s. In this case, the running time of an anytime algorithm can be determined prior to its execution, without monitoring, by solving

$$\arg\max_t \sum_q \sum_s P(q|t)P(s|t)U(s,q,t).$$

The framework for meta-level control reviewed so far assumes a one-time decision about how long to run an anytime algorithm is made before the algorithm starts. To allow a run-time monitor to determine when to stop an anytime algorithm, this framework must be extended in two ways. Section 3 discusses sequential prediction: how to condition these probabilities on information that can be monitored at run-time. Section 4 discusses sequential decision-making: how a run-time monitor can determine an improved stopping time based on information gathered at run-time.

3 Dynamic performance profiles

A standard 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. We call this a *dynamic performance profile*.

An obvious approach to constructing a dynamic performance profile is to condition a run-time prediction of further improvement on the quality of the currently available solution. Let $P(q'|q,\Delta t)$ denote the probability that resuming an algorithm for time interval Δt when the currently available solution has quality q results in a solution of quality q'. 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 and is sufficient for predicting further improvement in solution quality. (The latter condition is equivalent to assuming that solution quality satisfies the Markov property for this prediction problem.) We believe these conditions can be assumed to hold for many anytime algorithms, but they do not hold for all. 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 even be possible to determine the quality of the currently available solution at run-time. We proceed to discuss some of these complications.

3.1 Estimating/predicting solution quality at run-time

How easy it is to determine solution quality at run-time, or predict improvement in quality based on the quality of the currently available solution, depends to a great extent 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 is very easy to determine at run-time. In the case of the traveling salesman problem, for example, solution quality would simply be the length of the currently available tour. However, solution quality defined in this way may not be the best predictor of subsequent improvement in solution quality. This follows from the observation that 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 simply as tour length will not distinguish these cases, and so will not be as good a predictor of subsequent improvement as a definition of solution quality that takes into account how far the current tour length is from the optimal tour length.

For just this reason, solution quality is often defined as the difference between the current value of an objective function and its optimal value. For combinatorial optimization problems, solution quality is customarily defined as the "approximation ratio" of a solution. For cost-minimization problems, this is $Cost(Approximate\ solution)/Cost(Optimal\ solution)$, and for value maximization problems, it is $Cost(Optimal\ solution)/Cost(Approximate\ solution)$. 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. This is no obstacle to using this measure of quality to construct a performance profile for an anytime algorithm, because a performance profile can be constructed off-line and the quality of approximate solutions measured in terms of the quality of the eventual 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, if quality is measured with reference to the optimal solution. In some cases, it will be possible to bound the degree of approximation to an optimal solution, but a run-time monitor can only estimate (probabilistically) where the optimal solution falls within this bound.

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 [6]. 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 [20].

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 solu-

tion quality at run-time. It is impossible to specify a general solution to this problem – it will vary from algorithm to algorithm. 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 on the cumulative running time of the algorithm, t.

Let $P(q|f,t,\Delta t)$ denote the probability that resuming an algorithm for time Δt when the currently available solution exhibits feature f after running time t results in a solution of quality q. These probabilities can be determined by statistical analysis of the behavior of the algorithm. 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 6, we describe a simple example in which this run-time prediction scheme is used. For that 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 slope 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 [12] 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. But as they note, reliance on nonlinear regression incurs a substantial run-time overhead. This suggests a tradeoff between the accuracy of a run-time estimator/predictor and the overhead it incurs, an issue we will look at more closely in section 5.

3.2 Predicting environment change

At some length, we have discussed the issues involved in conditioning a prediction of improvement in solution quality on a run-time monitor's observation of the progress of an algorithm. Similar issues arise in predicting change in the state of the environment. These have been well-studied in the literature on modeling dynamical systems and we do not review them here. We simply note that most details of the environment can be abstracted away and only those that reflect the degree of time pressure must be modeled. For example, we might model time pressure from the environment as a simple stochastic deadline. As a general notation, we let $P(s'|s,\Delta t)$ denote the probability that the problem state will be s' after time Δt if the current problem state is s. This simple extension of the predictive model of the environment given in section 2 makes possible dynamic prediction based on run-time monitoring. In this respect, it is analogous to our extension of performance profiles.

4 Optimal stopping

Instead of making a single decision about how long to run an anytime algorithm before it starts, run-time monitoring involves making a sequence of decisions. Each time an algorithm's progress is monitored, a decision must be made to continue deliberation or stop. Deliberation should continue as long as the *expected value of computation* (EVC) is positive, where the expected value of computation is defined as the expected utility of the solution obtained by letting the algorithm continue (for at least one more time step and possibly longer) minus the expected utility of acting immediately on the current result.

Because exact determination of the EVC requires considering all possible decisions about whether to continue or stop that could be made at subsequent time steps, a myopic approximation of the EVC is often used. Computed myopically, 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. Formally, the myopic stopping rule is to continue computation as long as this value is positive,

$$\sum_{q'} P(q'|q,\Delta t)U(q',t+\Delta t) - U(q,t),$$

where Δt represents a single time step. 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 very reasonable assumptions. The following theorem gives a sufficient condition for its optimality.

Theorem 1 *Myopic computation of EVC guarantees an optimal stopping time if for every time t and quality level q for which myopic EVC is negative, myopic EVC is also negative for every time $t+\Delta t$ and quality level $q+\Delta q$.*

Proof: If myopic EVC 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 myopic EVC is negative 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 myopic EVC is positive. Because by assumption there is not, stopping the algorithm is the optimal decision.

The comprehensive value of a result of quality q at time t, $U(q,t)$, can sometimes be expressed as the difference between two functions,

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

where $U_I(q)$ is called an *intrinsic utility function* and $Cost(t)$ is the cost of time [14]. Under this separability assumption, the above theorem has the following corollary.

Corollary 1 *Myopic computation of EVC guarantees an optimal stopping time if the intrinsic utility function has a non-increasing slope as a function of quality, and the time cost function has a non-decreasing slope as a function of time.*

The assumption that the intrinsic utility function has a non-increasing slope is identical to the assumption of Dean and Wellman that performance profiles have the property of *diminishing returns*. (See [4] page 364.) The assumption that the time cost function has a non-decreasing slope characterizes most real-time situations, including those with deadlines and those in which the cost of time is constant. Therefore, these conditions for the optimality of the myopic stopping criterion are very reasonable.

Nevertheless, there are cases in which reliance on myopic EVC can lead to a sub-optimal stopping decision. Horvitz [6] describes a bounded conditioning algorithm for probabilistic inference in belief networks for which myopic EVC can mislead because expected improvement is "flat" in the near term but substantial after some number of time steps. Because myopic EVC gives rise to a premature stopping decision in this case, he suggests various degrees of lookahead to compute EVC more reliably.

As we pointed out at the beginning of this section, determining an optimal stopping time by run-time monitoring is a sequential decision problem. This suggests as an alternative the use of dynamic programming to compute a stopping rule that is optimal under all conditions, following the well-established use of 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,

$$V(q,t) = \max_{d} \begin{cases} U(q,t) \text{ if } d = \text{stop}, \\ \sum_{q'} P(q'|q,\Delta t) V(q', t+\Delta t) \\ \text{if } d = \text{continue} \end{cases}$$

to determine the following policy,

$$\pi(q,t) = \arg\max_{d} \begin{cases} U(q,t) \text{ if } d = \text{stop}, \\ \sum_{q'} P(q'|q,\Delta t) V(q', t+\Delta t) \\ \text{if } d = \text{continue} \end{cases}$$

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 time complexity of the dynamic programming algorithm is $O(|q|^2|t|)$. However, the stopping rule can be computed off-line, reducing the run-time overhead of monitoring. (An analogous solution of the stopping problem can be developed for cases in which utility depends not only on time and solution quality, but on the state of a dynamic environment.)

5 Reducing run-time overhead of monitoring

Although run-time monitoring makes it possible to take full advantage of the potential for optimizing the value of computation that anytime algorithms afford, it also incurs an overhead or "cost" – both a design-time overhead for building the predictive models and profiles needed for dynamic control, and a run-time overhead for monitoring itself. The improvement in performance that monitoring makes possible must be weighed against the overhead it incurs to determine whether monitoring is worthwhile. We define the *expected value of monitoring* as the expected utility of a result obtained when a run-time monitor determines an algorithm's stopping time minus the expected utility of a result obtained by allocating a fixed running time without monitoring.

The expected value of monitoring will always be non-negative if monitoring does not incur a cost. However, it may be negative otherwise. (This is analogous to the expected value of information.) Although the run-time overhead for monitoring may be miniscule for some problems, for other problems it can be significant, especially for problems in which it is difficult to determine solution quality at run-time without some complex computation. There are two ways to reduce this run-time overhead without forgoing monitoring. One is to perform as much metareasoning as possible off-line. Horvitz[6] calls this strategy "compilation of metareasoning" and we note that using dynamic programming to compute a stopping rule off-line is one example of this. Not all of the overhead of monitoring can be compiled off-line, however.

A second way to reduce the run-time overhead of monitoring is to adjust the interval at which the algorithm's progress is monitored. Assuming periodic monitoring, there should be an optimal interval for monitoring an algorithm's progress that depends on factors such as the cost of monitoring, the cost of stopping too early or late, and the average overhead incurred for monitoring at this interval. Based on these factors, Russell and Wefald [15] have derived a formula for determining the optimal interval for evaluating a stopping criterion in the context of game tree search, although their derivation makes assumptions (e.g., constant cost for time) that may not suit other problems.

Another approach to finding a cost-effective monitoring interval is to generalize the dynamic programming framework described in the previous section so that it determines an optimal monitoring interval, as well as an optimal stopping time. Assume that monitoring the quality of the currently available solution and deciding whether to continue or stop incurs a cost (or time overhead) of C. Given this cost, an optimal policy should specify two decisions for each time step t and quality level q: how much additional time to run the algorithm, and whether to monitor at the end of this time allocation and re-assess whether to continue, or whether to stop without monitoring. Let Δt and d denote these two decisions: Δt represents the additional interval of time to allocate to the anytime algorithm, and d is a binary variable that represents whether to monitor at the end of this time allocation or to stop without monitoring. The variable Δt makes it possible to control the time interval between one monitoring action and the next. The binary variable d makes it possible to run the algorithm without monitoring; its value is either *stop* or *monitor*. Given this formalization, dynamic programming can be used to find a combined policy for monitoring and stopping by computing the following value function,

$$V(q,t) = \max_{\Delta t, d} \begin{cases} \sum_{q'} P(q'|q,\Delta t) U(q', t+\Delta t) \\ \text{if } d = \text{stop}, \\ \sum_{q'} P(q'|q,\Delta t) V(q', t+\Delta t) - C \\ \text{if } d = \text{monitor} \end{cases}$$

to determine the following policy,

$$\pi(q,t) = \arg\max_{\Delta t, d} \begin{cases} \sum_{q'} P(q'|q,\Delta t) U(q', t+\Delta t) \\ \text{if } d = \text{stop}, \\ \sum_{q'} P(q'|q,\Delta t) V(q', t+\Delta t) - C \\ \text{if } d = \text{monitor} \end{cases}$$

These equations assume that the quality of the currently available solution can be measured accurately by the monitor. If that is not possible, a good (but not necessarily optimal) policy can be found by estimating solution quality based on some run-time feature f and running time t, as discussed in section 3, and by substituting $[f,t]$ for q in the above equations. An example that illustrates this is described in the following section.

The monitoring policies computed by this method have the interesting property that they recommend monitoring more frequently near the expected stopping time of an algorithm, an intuitive strategy. Moreover, the policies are 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. In many cases, 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.

quality level	*approximation ratio*
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	$\infty \rightarrow 1.50$

Table 1: Discretization of solution quality.

Whether the greater accuracy is worth the added time cost can be answered by computing a monitoring policy for each method and comparing their expected values to select the best one. Finally, if monitoring is not cost-effective, the dynamic programming algorithm will recommend allocating a fixed running time without monitoring. Monitoring is only recommended when it has a positive expected value.

6 Example

To illustrate some of the issues discussed in this paper, we use as an example a randomized tour improvement algorithm for solving the traveling salesman problem. This is a local optimization algorithm developed by Lin and Kernighan [10] and also used as an example of an anytime algorithm by Zilberstein [19]. Beginning with an initial tour, the algorithm repeatedly tries to improve the tour by swapping random paths between cities. For a detailed description of the algorithm, see [10] or [19]. All that is important for this example is its representativeness. As an anytime algorithm that has variance in solution quality over time, it can be used as a testbed to see whether monitoring can find an improved stopping time.

We define solution quality as the approximation ratio of a tour, that is, $Length(tour)/Length(optimal\ tour)$, which is discretized into quality levels using Table 1. We define the time-dependent utility of a solution of quality q at time t by the function,

$$U(q,t) = 100q - 20t,$$

where the first term on the right hand side represents the *intrinsic utility* of a solution and the second term represents *time cost*. Making utility a function of solution quality and time only, and not the state of a dynamic environment, allows us to focus on monitoring the algorithm's progress. Letting one time-step of the algorithm correspond to 0.005 CPU seconds, we compiled a dynamic performance profile for the algorithm by generating and solving a thousand random twelve-city traveling salesman problems.

Without monitoring, the optimal running time of the algorithm is eight time-steps, with an expected value of 275.3. Devising a monitoring scheme is difficult because a run-time monitor cannot precisely measure the quality of the currently available solution because it does not yet know the length of an optimal tour. Instead, it must somehow estimate solution quality. We do so based on both the cumulative improvement from the start of the algorithm, measured as $Length(tour)/Length(initial\ tour)$ and discretized using Table 2, and based on the running time of the algorithm. This is the same scheme for estimating/predicting solution quality described in section 3.

For this problem, the myopic stopping rule is optimal and identical to the stopping rule computed by dynamic programming when there is no cost for monitoring. Under this assumption, expected value using this stopping rule is 280.4, making the expected value of monitoring 5.1. When a monitoring cost of 1 is included in the optimization problem, the dynamic programming algorithm computes the combined policy for monitoring and stopping shown in Table 3. The number in each cell represents how much additional time to allocate to the algorithm based on the estimated quality of the currently available solution, as inferred from cumulative improvement and running 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. A zero indicates a decision to stop immediately. This policy has an expected value of 278.5, making the expected value of monitoring 3.2 in this case.

improvement level	*percentage reduction in tour length relative to initial tour*
6	$120\% \rightarrow \infty$
5	$100\% \rightarrow 120\%$
4	$80\% \rightarrow 100\%$
3	$60\% \rightarrow 80\%$
2	$40\% \rightarrow 60\%$
1	$20\% \rightarrow 40\%$
0	$0\% \rightarrow 20\%$

Table 2: Discretization of the cumulative improvement metric for estimating solution quality.

impr.	*time-step*									
level	0	1	2	3	4	5	6	7	8	9
6				1	1	0	0	0	0	0
5			3	3	2	1	1	0	0	0
4			4	4	3	3	2	0	0	0
3		2M	2M	2M	1M	1M	3	2	1	0
2		3M	2M	2M	2M	2M	1M	1M	1	0
1		3M	3M	3M	2M	2M	1M	0	0	0
0	4M	4M	3M	2M	1M	2				

Table 3: Combined policy for monitoring and stopping with solution quality estimated based on cumulative improvement and running time. The cost of monitoring is 1.

For the tour improvement algorithm, variance in solution quality is minor and the improved performance due to run-time monitoring correspondingly small. We plan to apply this technique to other problems for which variance in solution quality is larger and the payoff for run-time monitoring promises to be more significant. However, the fact that this technique improves utility even when variance is small, solution quality is difficult to estimate at run-time, and monitoring incurs a cost, is strong evidence for its validity and potential value.

7 Related work

Interest in anytime algorithms as an approach to managing tradeoffs between solution quality and computation time began with the work of Dean and Boddy [3][1], Russell and Wefald [13] [14], and Horvitz [7] [6]. Dean and Boddy coined the phrases "anytime algorithm" and "performance profile." Instead of using run-time monitoring for meta-level control, they assumed a fixed amount of time is allocated to each anytime algorithm before it starts. Their work focused on the problem of scheduling multiple, independent anytime algorithms, which they called *deliberation scheduling*.

Russell and Wefald [13][14] [15] used run-time monitoring and a myopic value of computation criterion for meta-level control of various search algorithms. They focused as much

on fine-grained control of which search step to take next as on control of when to halt computation and act.

Many aspects of run-time monitoring explored in this paper have been studied by Eric Horvitz [6] [7][8], including probabilistic models of algorithm performance similar to what we call dynamic performance profiles, the use of myopic EVC as a stopping criterion with various degrees of lookahead for problems in which myopic EVC can be misleading, and offline compilation of metareasoning to reduce run-time overhead.

Our primary contribution in this paper is the description of a dynamic programming framework for computing both an optimal stopping rule and a policy for adjusting the monitoring interval to reduce run-time overhead. We have also contributed a discussion of some complications of estimating/predicting solution quality at run time, and a theorem that describes the conditions under which myopic EVC is an optimal stopping rule.

Zilberstein [19] discusses the problem of monitoring a system composed of elementary, anytime components. He describes a scheme for compiling a performance profile for the complete system based on the performance profiles of anytime components. Run-time monitoring can improve performance when the actual quality of a solution generated by an anytime component differs from its expected quality, or when the environment changes unpredictably after the start of the algorithm. In either case, a run-time monitor can re-allocate computation time to anytime components that remain to be executed, improving the expected utility of the system. Rather than describe this work in detail, we simply note an analogy to the problem of monitoring individual anytime algorithms as 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 number IRI-9409827 and in part by Rome Laboratory, USAF, under grant number F30602-95-1-0012. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies of endorsements, either expressed or implied, of Rome Laboratory or the U.S. Government.

References

[1] M. Boddy and T. Dean. Deliberation scheduling for problem solving in time-constrained environments. *Artificial Intelligence* 67:245-285, 1994.

[2] J.S. Breese and E.J. Horvitz. Ideal reformulation of belief networks. In *Proceedings of the Sixth Conference on Uncertainty in Artificial Intelligence*, pp. 129-143, 1990.

[3] T. Dean and M. Boddy. An analysis of time-dependent planning. In *Proceedings AAAI-88*, pp. 49-54, 1988.

[4] T.L. Dean and M.P. Wellman. *Planning and Control.* Morgan Kaufman, 1991.

[5] C. Elkan. Incremental, approximate planning: Abductive default reasoning. In *Proceedings of the AAAI Spring Symposium on Planning in Uncertain Environments*, Palo Alto, California, 1990.

[6] E.J. Horvitz. *Computation and Action Under Bounded Resources.* PhD Thesis, Stanford University, 1990.

[7] E.J. Horvitz; G.F. Cooper; and D.E. Heckerman. Reflection and action under scarce resources: Theoretical principles and empirical study. In *Proceedings of the Eleventh IJCAI*, pp. 1121-1127, 1989.

[8] E. Horvitz and G. Rutledge. Time-dependent utility and action under uncertainty. In *Proceedings of Seventh Conference on Uncertainty in Artificial Intelligence*, pp. 151-158, 1991.

[9] E.J. Horvitz; H.J. Suermondt; and G.F. Cooper. Bounded conditioning: Flexible inference for decisions under scarce resources. In *Proceedings of the Fifth Workshop on Uncertainty in Artificial Intelligence*, 1989.

[10] S. Lin and B.W. Kernighan. An effective heuristic algorithm for the Traveling Salesman problem. *Operations Research* 21:498-516, 1973.

[11] C.J. Paul; A. Acharya; B. Black; and J.K. Strosnider. Reducing problem-solving variance to improve predictability. *CACM* 34(8):80-93, 1991.

[12] S. Raman and B. Wah (1991). Quality-time tradeoffs in simulated annealing for VLSI placement. In *Proceedings of the Fifteenth International Computer Software and Applications Conference.*

[13] S. Russell and E. Wefald. On optimal game-tree search using rational metareasoning. In *Proceedings of the Eleventh IJCAI*, pp. 334-340, 1989.

[14] S. Russell and E. Wefald. Principles of metareasoning. *Artificial Intelligence* 49:361-395, 1991.

[15] S. Russell and E. Wefald. *Do the Right Thing: Studies in Limited Rationality.* The MIT Press, 1991.

[16] S. Shekhar and S. Dutta. Minimizing response times in real time planning and search. In *Proceedings of the Eleventh IJCAI*, pp. 238-242, 1989.

[17] K.P. Smith and J.W.S. Liu. Monotonically improving approximate answers to relational algebra queries. *COMPSAC-89*, Orlando, Florida, 1989.

[18] M.P. Wellman and C.-L. Liu. State-space abstraction for anytime evaluation of probabilistic networks. In *Proceedings of the Tenth Conference on Uncertainty in Artificial Intelligence*, pp. 567-574, 1994.

[19] S. Zilberstein *Operational Rationality through Compilation of Anytime Algorithms.* Ph.D. dissertation, Computer Science Division, University of California at Berkeley, 1993.

[20] S. Zilberstein and S. Russell. Anytime sensing, planning and action: A practical model for robot control. In *Proceedings of the Thirteenth IJCAI*, pp. 1402-1407, 1993.