Decision-Theoretic Deliberation Scheduling for Problem Solving in Time-Constrained Environments

Mark Boddy
Honeywell Systems and Research Center
MN65-2100
3660 Technology Drive
Minneapolis, MN 55418
boddy@src.honeywell.com

Thomas Dean
Department of Computer Science
Brown University
Providence, Rhode Island 02912
tld@cs.brown.edu

March 21, 1993

Abstract

We are interested in the problem faced by an agent with limited computational capabilities, embedded in a complex environment with other agents and processes not under its control. Careful management of computational resources is important for complex problem-solving tasks in which the time spent in decision making affects the quality of the responses generated by a system. This paper describes an approach to designing systems that are capable of taking their own computational resources into consideration during planning and problem solving. In particular, we address the design of systems that manage their computational resources by using expectations about the performance of decision-making procedures and preferences over the outcomes resulting from applying those procedures. The approach we propose is *deliberation scheduling*: explicit manipulation of expectations on the behavior of the environment and the benefits of computation in order to determine how computational resources should be allocated.

1 Introduction

This paper describes an approach to designing systems that are capable of taking their own computational resources into consideration during planning and problem solving. In particular, we are interested in the design of systems that manage their computational resources by using expectations about the performance of decision-making procedures and preferences over the outcomes resulting from applying such procedures.

The work described in this paper can be seen as a response to a movement, started in the early 1980's, away from systems that make use of complex representations and engage in lengthy deliberations, and towards systems capable of making many very simple decisions quickly. This movement brought about the advent of the so-called "reactive systems" [7, 33, 16]. Most reactive systems are essentially programming languages for building systems that must be responsive to their environment.

Many of the researchers building reactive systems were interested in robotics and decision-support applications requiring real-time response. The responsiveness of reactive systems was in stark contrast with the performance of most planning and problem solving systems in use at that time. Most existing planning systems were essentially off-line data processing procedures that accepted as input some initial (and generally complete) description of the current state of the environment, and, after some indeterminate (and generally lengthy) delay, returned a rigid sequence of actions which, if the environment was particularly cooperative, might result in the successful achievement of some goal.

Reactive systems might be seen as an extreme response to the shortcomings of the existing planning systems. Reactive systems provided responsiveness at the cost of shallow and often short-sighted decision making. Since there were no proposals for how to control decision making in time-critical situations, researchers turned away from the traditional approaches to planning and attempted to incorporate more sophisticated decision making into reactive systems. Unwilling to sacrifice response time, the researchers that were trying to improve the decision-making capabilities of reactive systems were forced to trade space for time, often without a great deal of attention to the consequences.

Some of the dissatisfaction with complex representations and complicated deliberation strategies was due to misinterpreting asymptotic complexity results as evi-

dence of the existence of impassable computational barriers. Proofs of NP-hardness certainly indicate that we must be prepared to make concessions to complexity in the form of tradeoffs. The lesson to be learned, however, is that we have to control inference, not that we have to abandon it altogether.

In this paper, we discuss an approach to the explicit control of computation in time-constrained situations. We call this approach *deliberation scheduling*. Deliberation scheduling involves the explicit allocation of computational resources based on the expected effect of those allocations on the system's behavior. Section 2 provides a brief historical background to the current work on deliberation scheduling. In Section 3, we introduce concepts from decision theory that are useful for reasoning about expectations and preferences on the system's interaction with its environment. In Section 4, we discuss two different ways of modelling deliberation and the costs and expected benefits of time allocated to deliberation. One of those models involves the use of *anytime algorithms*, algorithms that can be interrupted at any point to supply an answer whose quality increases with increasing computation time. In Section 5, we demonstrate the application of deliberation scheduling using anytime algorithms to a class of abstract time-dependent problems called *time-dependent planning problems*, providing an optimal deliberation scheduler for a simple subclass and discussing some ways in which this class might be extended.

The rest of the paper concerns some of the practical issues that arise in trying to do deliberation scheduling for more complex problems. Section 6 talks about how to go about finding or constructing useful anytime algorithms and about the problem of constructing useful expectations on the behavior of those algorithms. Section 7 is a discussion of the difficulties of calculating the expected utility of the system's behavior and constructing an appropriate deliberation scheduler.

2 Historical Background

Much of the work on deliberation scheduling has employed decision theory. [1]. I. J. Good's work on Type II rationality laid the foundations for most of the work on decision theoretic deliberation scheduling that has been done since [17].

A line of work running back to [38] views deliberation time as being quantized into chunks of fixed, though not necessarily constant, size. Sproull [39] applies this

[1] Not all of it, however. See, for example [37, 11, 13]

model to a system for planning travel itineraries, though he is more concerned with the use of decision theory to guide planning choices. In [14], the deliberative chunks are called "methods." In [36], the computation to be controlled is modeled as a sequence of "inference steps" leading to the performance of an action. One of the problems with allocating deliberation time in discrete increments is that it is easy to wind up with a combinatorial problem [12].

An alternative approach is to assume that deliberation is performed using anytime decision procedures [10] (also called *flexible computations* in [25]), and that time can be allocated in values drawn from a continuous range (this is inevitably an approximation). Briefly, the advantages of this approach include an ability to make use of any amount of time available, robust behavior in the presence of unexpected interruption, and simplifying the problem of optimal or near-optimal deliberation scheduling. Both of these approaches are discussed in more detail in Section 4.

The use of deliberation scheduling has been explored in several domains, including planning [13, 11], search [35, 19, 28], medical decision making [26], and robot control [10, 5].

3 Decision Theory and the Control of Inference

Probability and decision theory are methods for dealing with uncertain information and outcomes on the basis of a small set of axioms concerning *rational* behavior [31]. If you accept the axioms, decision theory provides a powerful set of tools for modelling and evaluating decision making under uncertainty. Deliberation scheduling for independent agents (i.e., robots) can involve several sources of uncertainty, including the future occurrence of events, sensor limitations, and incomplete knowledge of the effects of actions. In addition, we have only limited knowledge of the effects of allocating time to the decision procedures used for deliberation. Much of the work on deliberation scheduling has employed decision theory.[2]

We begin with the idea of a *decision procedure*: a procedure used by an agent to select an action which, if executed, changes the world. Some actions are purely computational. For our purposes, such computational actions correspond to an agent running a decision procedure, and we refer to such actions as *inferential*. The results of inferential actions have no immediate effect on the world external to the

[2]Not all of it, however. See for example [37].

agent, but they do have an effect on the internal state of the agent. In particular, inferential actions consume computational resources that might be spent otherwise and result in the agent revising its estimations regarding how to act in various circumstances. In addition to the purely computational actions, there are *physical actions* that change the state of the world external to the agent, but that may also require some allocation of computational resources.

Real agents have severely limited computational capabilities, and, since inferential actions take time, an inferential action may end up selecting a physical action that might have been appropriate at some earlier time but is no longer so. Inferential actions are useful only insofar as they enable an agent to select physical actions that lead to desirable outcomes.[3] Decision theory provides us with the language required to talk precisely about what it would mean for an action to lead to a desirable outcome. Before we can proceed further, we will need some precise language for talking about possible outcomes and stating preferences among them. The language that we adopt is borrowed directly from statistical decision theory.

Let Ω correspond to a set of possible states of the world. We assume that the agent has a function,

$$\mathrm{U} : \Omega \rightarrow \mathbf{R},$$

that assigns a real number to each state of the world. This is referred to as the agent's *utility function*.[4] These numbers enable the agent to compare various states of the world that might result as a consequence of its actions. It is assumed that a rational agent will act so as to maximize its utility. The quantity, $\mathrm{U}(\omega)$ where $\omega \in \Omega$, is generally meant to account for both the immediate costs and benefits of being in the state ω and the delayed costs and benefits derived from future possible states. We assume that there is some deterministic or stochastic process governing the transition between states, and that this process is partially determined or biased by the agent's choice of action. In the case of a stochastic process, the agent cannot know what state will result from a given action and must make use of expectations regarding the consequences of its actions. In order to account for these longer-term consequences, it is often useful to think of the agent as having a particular long-term plan or *policy*. In such cases, the agent will generally assign an expected

[3] Inferential actions are also useful for learning purposes, as in learning search strategies, but even here the actions are ultimately in service to selecting physical actions that lead to desirable outcomes.

[4] See Chernoff and Moses [8], Barnett [1], or Pearl [30] for discussions regarding the axioms of utility theory.

utility to a given state based upon the immediate rewards available in that state and expectations about the subsequent states, given that the agent continues to select actions based upon its current policy.

In addition to expectations about the possible future consequences of its physical actions, an agent capable of reasoning about its computational capabilities must also have expectations regarding the potential value of its computations, and estimates of how long those computations are likely to take. In the work discussed in this paper, an agent is assumed to engage in *meta-reasoning*. For our purposes, meta-reasoning consists of running a decision procedure whose purpose it is to determine what other decision procedures should run and when. We prefer the term *deliberation scheduling* [10] to the more general meta-reasoning and will use the two interchangeably in this paper. If the meta-level decision procedure takes a significant amount of time to run, it must be argued that this time is well spent. In some cases, the time spent in meta-reasoning is small enough that it can be safely ignored; in other cases, it may be useful to invoke a meta-meta-level decision procedure to reason about the costs and benefits of meta-reasoning.

Deliberation scheduling is accomplished by a sequence of decisions made by the system as time passes, events happen, and new information becomes available. These allocation decisions influence the system's behavior. Preferences on various outcomes or sequences of events are encoded in the utility function U. We treat the value of U as a random variable dependent on the actual outcome of a decision or a sequence of decisions and calculate its expected value. This is how we evaluate strategies for deliberation scheduling: an optimal deliberation scheduler maximizes the expected utility of its decisions, given what it knows about the environment and the effect of those allocations.

4 Deliberation

Currently, there are two main approaches to modelling the allocation of deliberation and the effects of of that allocation. The first treats deliberation as being divisible into "chunks," such that the allocation of each chunk is a separate decision. The second approach approximates deliberation allocations as being drawn from a continuous range, such that the decisions being made are both what procedure to run and how much time to allocate to it.

4.1 Discrete Deliberation Scheduling

In Russell and Wefald's work [36], the computation to be controlled is modeled as a sequence of inference steps leading to the performance of an action. The utility of a particular inference step is defined in terms of its effect on the eventual choice of action and the state in which the action is executed.[5]

At any given time the agent has a default action $\alpha \in \mathcal{A}$ which is the action that currently appears to be best. In addition, the agent has a set of computational actions $\{S_i\}$ which might cause the agent to revise its default action. The agent is faced with the decision to choose from among the available options: $\alpha, S_1, S_2, \ldots, S_k$. Computational actions only affect the agent's internal state. However, time passes while the agent is deliberating and opportunities are lost, so the net value of computation is defined to be the difference between the utility of the state resulting from the computation minus the utility of the state resulting from executing α:

$$\mathrm{V}(S_j) = \mathrm{U}([S_j]) - \mathrm{U}([\alpha]).$$

In order to simplify reasoning about the utility of combined computational and base-level actions, Russell and Wefald separate the *intrinsic utility*, that is the utility of an action independent of time, from the *time cost* of computational actions, defining the utility of a state as the difference between these two:

$$\hat{\mathrm{U}}([A_i, [S_j]]) = \hat{\mathrm{U}}_I([A_i]) - TC(S_j).$$

It should be noted, however, that determining an appropriate time cost function can become quite complicated in applying Russell and Wefald's approach. In particular, costs concerning hard and soft deadlines must be accounted for by this function. In the game-playing application explored in [35], there are no hard deadlines on a per-move basis, instead there is a per-game time limit that is factored into the time cost. In many time-critical problem solving applications, calculating the time cost can be quite complicated (e.g., consider the sort of medical care applications investigated in [24] and [21]).

There are a number of assumptions that Russell and Wefald make in their analysis. First, it is assumed that the agent considers only single computation steps, estimates their ultimate effect, and then chooses the step appearing to have highest

[5]For a more detailed discussion of the work presented in this section, see [9].

benefit. This is referred to as the *meta-greedy* assumption. Second, the computation of expected utility assumes that the agent will take only a single additional search step. This is referred to as the *single-step* assumption. Finally, it is assumed that a computational action will change the expected utility estimate for exactly one base-level action. In Russell and Wefald's state-space search paradigm, this is referred to as the *subtree-independence* assumption.

The assumptions stated in the previous paragraph may seem overly restrictive, but it is quite difficult to avoid these or similar assumptions in general. Pearl [30] identifies two assumptions that most practical meta-reasoning systems ascribe to: *no competition*, each information source is evaluated in isolation from all the others, and *one-step horizon*, we consult at most one additional information source before committing to a base-level action. Assessments of information sources based on the no-competition and one-step-horizon assumptions are referred to as *myopic*, and most practical systems employ myopic decision policies.

The next piece of research that we consider in this section is due to Etzioni [14], and it borrows from the Russell and Wefald work, and builds on the early work of Simon and Kadane [38] on satisficing search. It is particularly interesting for the fact that it attempts to combine the sort of goal-driven behavior prevalent in artificial intelligence with the decision theoretic view of maximizing expected utility. In Etzioni's model, the agent is given a set of goals, a set of methods, and a deadline. The agent attempts to determine a sequence of methods

$$\sigma = m_{1,1}, m_{2,1}, \ldots, m_{k_1,1}, m_{1,2}, m_{2,2}, \ldots, m_{k_2,2}, \ldots, m_{1,n}, m_{2,n}, \ldots, m_{k_n,n}$$

where $m_{i,j}$ is the ith method to be applied to solving the jth goal. The idea is that the agent will apply each method in turn until it either runs out of methods or achieves the goal, at which point it will turn its attention to the next goal.

For the case in which the agent has a single goal and multiple methods for achieving it, computing a sequence σ of methods which maximizes the probability of achieving the goal by the deadline is straightforward. In the case of an agent faced with multiple goals, even where there is only one method for each goal, optimal choice of a sequence of methods is more difficult. By reducing the *knapsack problem* [15] to the problem of computing the expected opportunity cost, Etzioni shows that computing the expected opportunity cost of a method is NP-complete. [6] The "expected opportunity cost" of a method is a measure of the benefit that might

[6] The knapsack problem involves a set of objects represented by value/volume pairs. The problem

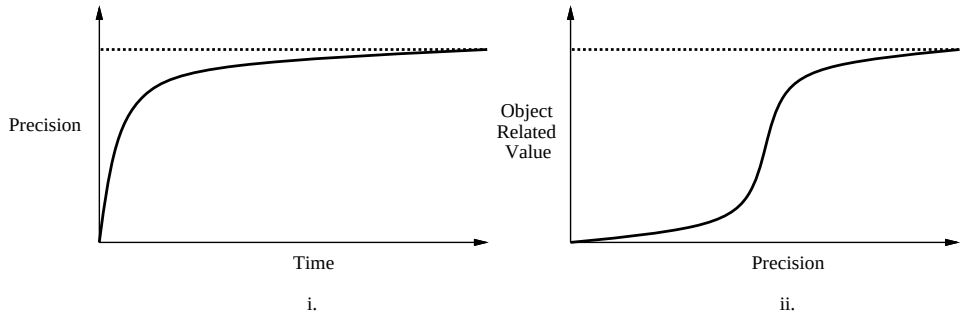

Figure 1: Plots relating (i) time spent in computation to the precision of a probabilistic calculation, and (ii) precision to the value associated with getting the diagnosis correct and treating the patient accordingly (after [25]).

have been gained using other methods (for goals of equal weight, the difference in the probabilities of achieving a goal using this method as opposed to another).

It is not too surprising that there are some hard problems lurking among the deliberation scheduling problems that underlie decision-theoretic control of inference. It should be pointed out, however, that all we should really be concerned with is the expected cost of meta-reasoning, and that in many practical applications approximations are much preferred to even polynomial-time methods for computing exact solutions. In the next section, we see how approximation algorithms for computationally expensive problems can provide us with even greater flexibility in allocating processor time to decision procedures.

4.2 Flexible Computations and Anytime Algorithms

In this section, we consider two independently developed but closely related approaches to decision-theoretic control of problem solving. The two approaches are due to Dean and Boddy [10] and Horvitz [25]. Horvitz refers to his decision procedures as *flexible computations* and Dean and Boddy refer to theirs as *anytime algorithms*, but the basic idea behind the two proposals is the same, and we will use the two terms interchangeably.

In the ideal flexible computation, the object-related value of the result returned by a decision procedure is a continuous function of the time spent in computa-

is to determine whether a given value K can be obtained by summing values for a set of objects whose summed volume is less than a specified N.

tion. The notion of "object-related value" of a result is to be contrasted with the "comprehensive value" of a system's response to a given state. For example, the object-related value of the answer returned by a diagnostics procedure will be some measure of how good the answer is (e.g., the probablility of a correct diagnosis). [7] The comprehensive value of the system's response is related to the current situation: applying therapy based on a correct diagnosis to a patient who has expired while the diagnosis was being determined will not have a high comprehensive value, however good the diagnosis may be.

In some cases, the task of relating a result to the comprehensive value of the overall response can be quite complex. This is especially so in cases in which there are several results from several different decision procedures. In these cases, it is often convenient to make the assumption that the value function is *separable* so that the comprehensive value of the system's response can be computed as the sum of the value of a sequence of outcomes.

We assume that a flexible computation can be interrupted at any point during computation—hence, the name "anytime"—to return an answer whose object-related value increases as it is allocated additional time. Horvitz provides a good example of a flexible computation and an analysis of its object-related value drawn from the health care domain. Suppose that we have an anytime algorithm that computes a posterior distribution for a set of possible diagnoses given certain information regarding a particular patient. Figure 1.i (from [25]) shows a graph that relates the precision of the result returned by this algorithm to the time spent in computation. The object-related value can be determined as a function of precision by considering the expected utility of administering a treatment based on a diagnosis of a given precision ignoring when the treatment is administered (see Figure 1.ii).

The comprehensive value of computation is meant to account for the costs and benefits related to the time at which the results of decision procedures are made use of to initiate physical actions. Figure 2.i (from [25]) indicates how a physician might discount the object-related value of a computation as a function of delay in administering treatment. The comprehensive value of computation is shown in Figure 2.ii and is obtained by combining the information in Figures 1.i, 1.ii, and 2.i. This method of combining information assumes both time-cost separability (object-related value and the cost of delaying a decision are independent functions of time) and a one-step horizon (the value of the physician's response is not dependent on

[7] Object-related value is exactly Russell and Wefald's intrinsic utility.

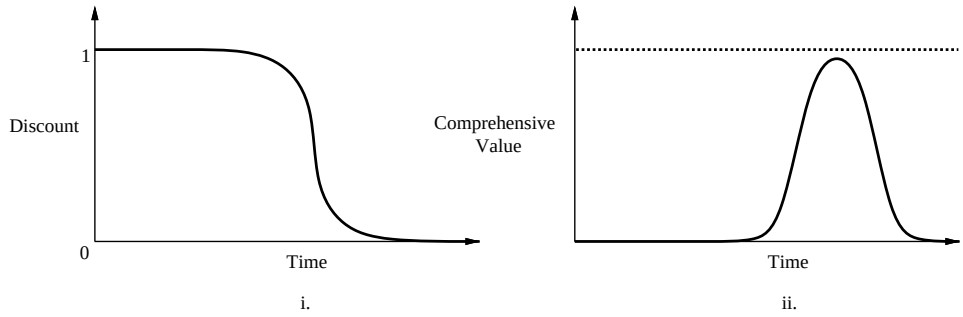

Figure 2: Plots indicating (i) a discounting factor for delayed treatment, and (ii) the comprehensive value of computation as a function of time (after [25]).

any further decision-making or information gathering).

Horvitz has addressed a number of interesting theoretical issues in the use of anytime algorithms. These include the difficulty of providing a single characterization of the "utility" associated with a given procedure and showing how differing time costs can affect the optimal choice of a decision procedure in a particular situation [26]. Horvitz has so far concentrated on the theoretical problems involving deliberating about a single task. In contrast, we have pursued a more empirical course in trying to find approximate solutions to the kind of deliberation-scheduling problems that arise in controlling robots.

Both Horvitz and Dean and Boddy note that the most useful sort of flexible computations are those whose object-related value increases monotonically over some range of computation times. This is difficult to guarantee for certain classes of decision problems: further computation may lead to a worse answer, if your initial guess was good. Instead, we employ algorithms for which the *expected* value increases monotonically. For some algorithms, there is no difference: a probabilistic algorithm for identifying large prime numbers has a "value" that is strictly increasing, in the sense that the probability of a correct identification is guaranteed to improve by a known amount with increasing computational effort. The implications of using algorithms whose answers are only expected to improve is addressed in somewhat more detail in Section 6, and at considerable length in [4].

Dean and Boddy [10] employ decision procedures whose answers are expected to increase in value monotonically throughout the range of computation times, and exploit this fact to expedite deliberation scheduling for a special class of planning

problems referred to as *time-dependent planning problems*. A planning problem is said to be time-dependent if the time available for responding to a given event varies from situation to situation. In this model, a predictive component, whose time cost is not considered, predicts events and their time of occurrence on the basis of observations, and the planning system is given the task of formulating a response to each event and executing it in the time available before the event occurs.

Our model of time-dependent planning generalizes on the multiple-goals/single-method-for-each model of Etzioni described in the previous section by allowing each goal to have a separate deadline. If the responses to the different events are independent, the task of deliberation scheduling can be stated in terms of maximizing the sum

$$\sum_{e \in E} \mathrm{V}(\mathrm{Response}(e)),$$

where E is the set of all events predicted thus far. It is assumed that there is exactly one decision procedure for each type of event likely to be encountered, and that there are statistics on the performance of these decision procedures. The statistics are summarized in what are called *performance profiles* which are essentially the same as the graphs used by Horvitz in his analysis (e.g., see Figure 1.ii).

Performance profiles graph the expected value of some parameter of the answer returned by a given procedure as a function of time. Part of the definition of an anytime decision procedure is that the answer it returns is expected to improve with increasing allocations of deliberation time. The 2-dimensional graphs shown here are a simplification. Knowing the expected value is in general insufficient—there are situations where what is needed is a marginal distribution, not just a sum or product of expected values. For instance, two path planners that produce paths with the same expected length for a given allocation of deliberation time can have very different expected utilities if arrival time has a hard deadline: the planner with greater variance on expected length may have a lower expected utility, because it has a greater probability of missing the deadline. A completely general performance profile is a surface in 3-space, not a 2D graph. Taking a slice across this surface for a given allocation of deliberation time provides a distribution on some parameter of the answer returned. 2D performance profiles graph the means of the distributions represented in the more general form.

The assumption of task independence allows us to employ the two-dimensional versions of performance profiles: maximizing the expected value of the sum of k

independent random variables can be accomplished by maximizing the sum of the expected values. If the various decisions are not independent or if calculating a response requires running more than one procedure (e.g., the output of one procedure is the input of a second, which supplies the final answer), then calculating an optimal allocation of deliberation time may require distributions as well as expected values for the decision procedures involved.

In Section 5, we define the class of time-dependent planning problems precisely, and provide polynomial-time algorithms for deliberation scheduling for particular subclasses. These algorithms use a simple greedy strategy working backward from the last predicted event, choosing the decision procedure whose expected gain computed from the performance profiles is greatest.

The NP-completeness result reported by Etzioni does not apply in this more general case. In job-shop scheduling, if it is possible to suspend and later resume a job, then many otherwise difficult problems become trivial [18, 6]. Such (preemptive) scheduling problems are somewhat rare in real job shops given that there is often significant overhead involved is suspending and resuming jobs (e.g., traveling between workstations or changing tools), but they are considerably more common with regard to purely computational tasks (e.g., suspending and resuming Unix processes). In many scheduling problems, each job has a fixed cost and requires a fixed amount of time to perform; spending any less than the full amount yields no advantage. This is the case in the decision procedures considered by Etzioni. If, however, the decision procedures for computing appropriate actions are preemptible and provide better answers depending upon the time available for deliberation, then the task of deliberation scheduling is considerably simplified. Anytime decision procedures thus provide more flexibility in responding to time-critical situations, and simplify the task of allocating processor time in cases where there is contention among several decision tasks.

The use of flexible computations can also simplify problems in which one decision procedure produces an intermediate result that is used as input to a second decision procedure. Boddy and Dean [5] investigate one such problem involving a robot courier assigned the task of delivering packages to a set of locations. The robot has to determine both the order in which to visit the locations, referred to as a *tour*, and, given a tour, to plan paths to traverse in moving between consecutive locations in the tour. To simplify the analysis, it is assumed that the robot's only concern is time; it seeks to minimize the total amount of time consumed both in sitting idly

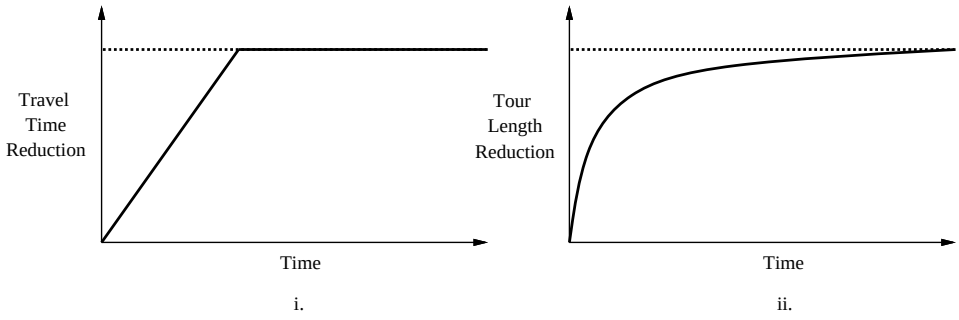

Figure 3: Performance profiles relating (i) the expected savings in travel time to time spent in path planning, and (ii) the expected reduction in the length of a returned tour as a function of time spent in tour improvement.

deliberating about what to do next, and in actually moving about the environment on its errands. Furthermore, it is assumed that there is no advantage to the robot in starting off in some direction until it knows the first location to be visited on its tour. Finally, while the robot can deliberate about any subsequent paths while traversing a path between consecutive locations in the tour, it must complete the planning for a given path before starting to traverse that path.

The two primary components of the decision making process involve generating the tour and planning the paths between consecutive locations in the tour. The first is referred to as *tour improvement* and the second as *path planning*. Boddy and Dean employ iterative refinement approximation routines for solving each of these problems, and gather statistics on their performance to be used at run-time in guiding deliberation scheduling. The statistics are summarized in *performance profiles*. Figure 3 (from [5]) shows experimentally derived profiles for path planning and tour improvement in a domain called the *gridworld*. Figure 3.i shows how the expected savings in travel time increase as a function of time spent in path planning. Figure 3.ii shows how the expected length of the tour decreases as a fraction of the shortest tour for a given amount of time spent in tour improvement. We assume that the robot starts out with an initial randomly selected tour. Given the length of some initial tour, the expected reduction in length as a function of time spent in tour improvement, and some assumptions about the performance of path planning, the robot can figure out how much time to devote to tour improvement in order to minimize the expected time spent in stationary deliberation and combined deliberation and traversal.

There currently is no general theory about how to combine anytime algorithms. For the case of independent decision problems for which anytime decision procedures exist, or for which a pipelined sequence of anytime decision procedures exist, as in the robot courier problem, there is a great deal of interesting research to be done; research that can draw heavily on the scheduling and combinatorial optimization literature. This issue is discussed in more detail in Section 6.3, and in [4].

It is worth pointing out some connections between the Russell and Wefald work and that of Dean and Boddy and Horvitz. The Russell and Wefald work can be seen as trying to construct an optimal anytime algorithm; a single algorithm that operates by calculating a situation-specific estimate of utility using only local information, just as prescribed by information value theory. It should be possible to apply the Russell and Wefald approach to scheduling anytime algorithms. For some purposes (e.g., the game-playing and search applications described in [36]), the monolithic approach of Russell and Wefald seems perfectly suited. For other applications (e.g., the robotic applications described in [5] or the intensive-care applications described in [24]), it is convenient to think in terms of scheduling existing approximation algorithms.

At one level, a unified view of time-dependent problems is straightforward. As time passes, an agent makes a sequence of decisions about what to do. "What to do" covers whether or not to perform certain actions or certain computations. The basis for making these decisions should be the agent's expectations regarding the outcome of a given sequence of decisions. This starts to get complicated only when we attempt to model the decisions to be made by the agent, their effects, and the agent's expectations. The formal specification in [4] provides a language for constructing such models, as does Russell and Wefald's definition and analysis of meta-reasoning [36]. These two views of time-dependent problems are distinct and may not be easy to reconcile. Russell and Wefald's formalism makes representing deadlines difficult and models deliberation in terms of discrete chunks, whereas our work emphasizes events and their temporal relationships (including deadlines of various kinds), and views deliberation time as being allocated in values drawn from a continuous range.

A common language for building models of time-dependent problems will allow us to compare and contrast different models and approaches to providing solutions given those models. A common solution methodology may be much harder to come by, however. Entire disciplines are devoted to solving particular classes

of time-dependent problems (e.g., control theory, real-time systems, and job-shop scheduling). In addition, consider what has happened in the work on scheduling: there are a myriad of separate problems, distinguished by minor differences in problem characteristics, for which the difficulty of providing a scheduler (and the best way of doing so) varies enormously [27]. The most general notion of deliberation scheduling subsumes the scheduling problem, and thus will be no easier to provide a general solution for.[8]

5 Time-Dependent Planning

In this section, we define time-dependent problems more precisely, and explore the abstract structure of deliberation scheduling for selected classes of time-dependent problems. We define a class of time-dependent problems in terms of:

- a set of event (or *condition*) types, $\mathcal{C}$,
- a set of action (or *response*) types, $\mathcal{A}$,
- a set of time points, $\mathcal{T}$,
- a set of decision procedures, $\mathcal{D}$, and
- a value function, V.

We assume that at each point in time the agent knows about some set of pending events that it has to formulate a response to. We are not concerned with how the agent came to know this information; suffice it to say that the agent has some advance notice of their type and time of occurrence. To represent its knowledge regarding future events, we say that the agent knows about a set of *tokens* drawn from the set $\mathcal{C} \times \mathcal{T}$. When we talk about events or conditions, we will be referring to tokens and not types. Each condition, c, has a type, $\text{type}(c) \in \mathcal{C}$, and a time of occurrence, $\text{time}(c) \in \mathcal{T}$. In the following, all conditions are assumed to be instantaneous (i.e., corresponding to point events).

We evaluate the agent's performance entirely on the basis of its responses. Let $\text{Response}(c) \in \mathcal{A}$ be the agent's response to the condition c. Let $\text{V}(a|c)$ be the value

[8] This is one of the advantages of using anytime decision procedures: the resulting scheduling problems are more likely to be easy to solve.

of responding to the condition c with the action $a \in \mathcal{A}$. To simplify the analysis, we make the strong assumption that the value of the agent's response to one condition is completely independent of the agent's response to any other condition. Given this independence assumption, we can determine the total value of the agent's response to a set of conditions C as the sum,

$$\sum_{c \in C} \mathrm{V}(\mathrm{Response}(c)|c).$$

Since we are primarily interested in investigating issues concerning the costs and benefits of computation, we abstract the problem somewhat more in order to simplify the analysis. We require the agent to formulate a response to every condition it is confronted with. We further require that the agent perform all of its deliberations regarding a given event prior to the time of occurrence of that event. There is no benefit to coming up with a response early.

For each condition type, $c \in \mathcal{C}$, there is a single decision procedure $\mathrm{dp}(c) \in \mathcal{D}$ (the converse need not hold). The agent knows how to select an appropriate decision procedure given the type of an event. The decision procedures in $\mathcal{D}$ have the properties of flexible computations that we discussed earlier (monotonically increasing expected value, and some answer available for any allocation of time). In addition, the agent has expectations about the performance of these decision procedures in the form of performance profiles. For each condition type, $c \in \mathcal{C}$, there is a corresponding function $\mu_c : \mathbf{R} \rightarrow \mathbf{R}$ that takes an amount of time, δ, and returns the expected value of the response to c generated by $\mathrm{dp}(c)$ having been run for the specified amount of time:

$$\mu_c(\delta) = \mathrm{E}(\mathrm{V}(\mathrm{Response}(c)|c, \mathrm{alloc}(\delta, \mathrm{dp}(c)))),$$

where $\mathrm{alloc}(\delta, \mathrm{dp}(c))$ denotes the allocation of δ time units to $\mathrm{dp}(c)$. $\mathrm{V}(\mathrm{Response}(c)|c, \mathrm{alloc}(\delta, \mathrm{dp}(c)))$ is the value of the response to c, given the occurrence of c and the allocation of δ time units to $\mathrm{dp}(c)$ to calculate $\mathrm{Response}(c)$.

5.1 A Simple Deliberation Scheduler

In the rest of this section, we consider various restricted classes of decision procedures. We begin by considering decision procedures whose performance profiles can be represented or suitably approximated by piecewise linear monotonic increasing

functions. We add the further restriction that the slopes of consecutive line segments be decreasing. If the functions representing the performance profiles were everywhere differentiable, this restriction would correspond to the first derivative function being monotonic decreasing. In [10], we suggested a similar restriction referred to as *diminishing returns*, and defined as follows: $\forall\, c \in C, \exists\, f, \mu_c(t) = f(t)$ such that f is monotonic increasing, continuous, and piecewise differentiable, and $\forall\, x, y \in \mathbf{R}^+$ such that $f'(x)$ and $f'(y)$ exist, $(x < y) \supset (f'(y) \leq f'(x))$. For this class of problems, we provided an approximation algorithm that uses time-slicing to come within some ϵ of optimal. The algorithm presented here is exact given the restrictions on the form of performance profiles.

These restrictions are perhaps limiting, but by no means debilitating. Empirical evidence reported on in Section 7.2 and discussed in considerably more detail in [4, 3], suggests that a profile exhibiting diminishing returns may be obtained experimentally for decision procedures of a wide variety of types. For example, probabilistic testing of primes and heuristic search in some domains both exhibit this property. Piecewise linear "curves" can be fitted to such profiles with arbitrary precision, depending on the number and length of the segments employed.

Let $C = \{c_1, \ldots, c_n\}$ be the set of conditions that the system has to formulate responses for, and $\hat{t}$ be the present time. Let μ_i be the function describing the performance profile for the decision procedure used to compute responses for the ith condition. We present an algorithm that works backward from the time of occurrence of the last event in C. On every iteration through the main loop, the program allocates some interval of time to deliberating about its response to one of the conditions, c, whose time of occurrence, time(c), lies forward of some particular time t. The set of all conditions whose time of occurrence lies forward of some particular time t is denoted as

$$\Lambda(t) = \{c | (c \in C) \wedge (\text{time}(c) \geq t)\}.$$

The criterion for choosing which decision procedure to allocate processor time to is based on the expected gain in value for those decision procedures associated with the conditions in $\Lambda(t)$. The criterion also has to account for the time already allocated to decision procedures. Let $\gamma_i(x)$ be the slope of the linear segment of μ_i at x unless μ_i is discontinuous at x in which case $\gamma_i(x)$ is the slope of the linear segment on the positive side of x. We refer to $\gamma_i(x)$ as the *gain* of the ith decision procedure having already been allocated x amount of processor time.

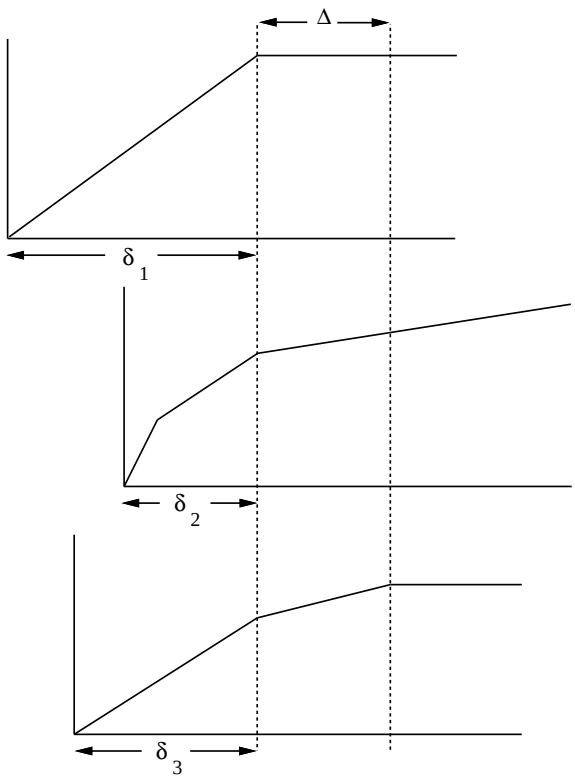

Figure 4: Determining min_alloc($\{\delta_1, \delta_2, \delta_3\}$)

Having allocated all of the time forward of t in previous iterations, figuring out how much time to allocate on the next iteration is a bit tricky. It is certainly bounded by $t - \hat{t}$; we cannot make use of time that is already past. In addition, given that we are using the gain of the decision procedures for conditions in $\Lambda(t)$ as part of our allocation criterion, the criterion only applies over intervals in which $\Lambda(t)$ is unchanged. Let last(t) be the first time prior to t that a condition in C occurs that is not already accounted for in $\Lambda(t)$:

$$\text{last}(t) = \max\{\text{time}(c) | c \in C - \Lambda(t)\}$$

Finally, given that the gains determine the slope of particular line segments characterizing the performance of the decision procedures, we have to be careful not to apply our criterion to an interval longer than that over which the current gains are constant. Let min_alloc($\{\delta_i\}$) be the minimum of the lengths of the intervals of time for the next linear segments for the performance profiles given the time allocated thus far. Figure 4 illustrates min_alloc($\{\delta_i\}$) for a particular case.

Figure 5 lists the procedure for deliberation scheduling for the class of problems under consideration. We prove below that this procedure generates an optimal deliberation schedule. The procedure, `DS`, consists of three iterative loops. The first initializes the allocation variables, the second determines how much time to allocate to each of the decision procedures, and the third determines when the decision procedures will be run. For convenience, we assume that the events in C are sorted so that time(c_i) $\leq$ time(c_j) for all $i < j$. This assumption is only made use of in determining when decision procedures will be run.

Consider the following simple example to illustrate how `DS` works. Suppose that we have two events to contend with, c_1 and c_2. Figure 6.i shows the performance profiles for the decision procedures for c_1 and c_2. `DS` starts by allocating all of the time between c_1 and c_2 to the decision procedure for c_2. The next interval of time to be allocated (Δ) is determined by the first linear segment of μ_1, and this interval is allocated to c_1.

At this point, the slope of the second linear segment of μ_1 is less than the slope of the first segment of μ_2, so the next interval (determined by what is left of the first linear segment of μ_2) is allocated to c_2. The next interval corresponds to the second linear segment of μ_1, and this entire interval is allocated to c_1. Finally, the remainder of μ_1 has slope 0, so the remaining time is allocated to c_2. Figure 6.ii

```
Procedure DS
    ;; Initialize the δ_i's to 0.
    for i = 1 to n,
        δ_i ← 0
    ;; Set t to be the time of the last event in C.
    t ← last(+∞)
    ;; Allocate time working backward from the last event.
    until t = t̂,
        ;; Compute the amount of time to allocate next.
        Δ ← min{t − t̂, t − last(t), min_alloc({δ_i})}
        ;; Find the procedure index with the maximum gain.
        i ← arg_i max{γ_i(δ_i)|c_i ∈ Λ(t)}
        ;; Allocate the time to the appropriate procedure.
        δ_i ← δ_i + Δ
        ;; Decrement time by the amount of allocated time.
        t ← t − Δ
    ;; Set t to be the current time.
    t ← t̂
    ;; Schedule working forwards in contiguous segments.
    for i = 1 to n,
        ;; Assume that time(c_i) ≤ time(c_j) for all i < j.
        run the ith decision procedure from t til t + δ_i
        ;; Increment time by the amount of allocated time.
        t ← t + δ_i
```

Figure 5: Deliberation scheduling procedure

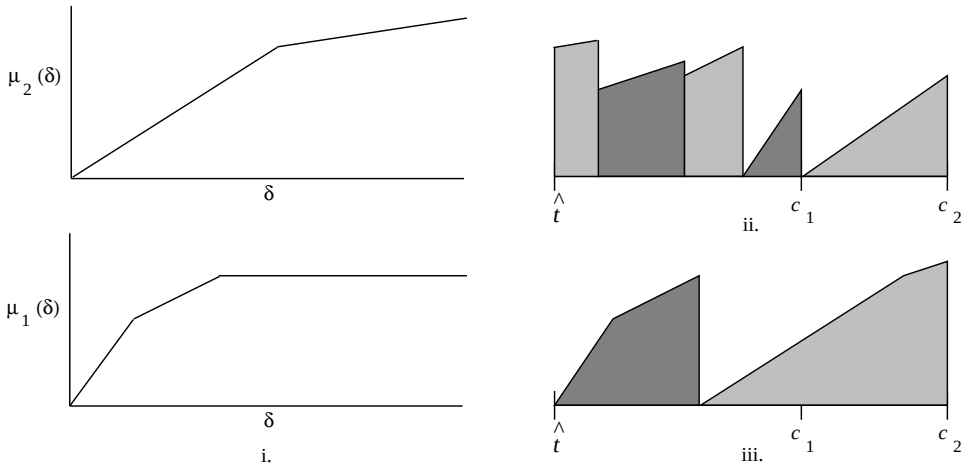

Figure 6: A simple example of deliberation scheduling

shows the complete history of allocations, and Figure 6.iii shows how the decision procedures are scheduled to run. Now we prove that DS is optimal.

Theorem 1 *The procedure* DS *is optimal in the sense that it generates a set of allocations* $\{\delta_i\}$ *maximizing* $\sum_i^n \mu_i(\delta_i)$.

Proof:

We proceed by induction on n, the number of conditions in C. For the basis step, $n = 1$, the algorithm allocates all of the time available to the only event in C, and hence is clearly optimal. For the induction step, we assume that DS is optimal for up to and including $n-1$ events. Our strategy is to prove each of the following:

1. Let $\hat{t}'$ be the time of the earliest event in C. Using $\hat{t}'$ as the starting time, DS optimally allocates processor time to the $n-1$ (or fewer assuming simultaneously occurring events) events in C occurring after $\hat{t}'$.

2. DS optimally allocates processor time to all n events in C over the period from $\hat{t}$ until the time of occurrence of the first event in C, accounting for the processor time already committed to in the allocations described in Step 1.

3. Given Steps 1 and 2, the combined allocations result in optimal allocations for C starting at $\hat{t}$.

Step 1 follows immediately from the induction premise. To prove Step 2, we have to demonstrate that DS solves the simpler problem of maximizing $\sum_i^n \mu_i(\delta_i)$ subject to the constraint that $\sum_i^n \delta_i = l$, where l is the length of time separating $\hat{t}$ and the first event in C. For this demonstration, it is enough to note that, as long as the set of events being considered ($\Lambda(t)$) remains unchanged, during each iteration of the main loop, DS chooses an interval with maximal gain, and, by making this choice, DS in no way restricts its future choices given that all subsequent intervals are bound to have the same or smaller gains. This last point is due to the restriction that the slopes of consecutive line segments for all performance profiles are decreasing. Note that, if we were to relax this restriction, the greedy strategy used by DS would not produce optimal allocations. Figure 7.i provides a pair of performance profiles such that DS will produce suboptimal allocations. Figure 7.ii shows the allocations made by DS, and Figure 7.iii shows the optimal allocations.

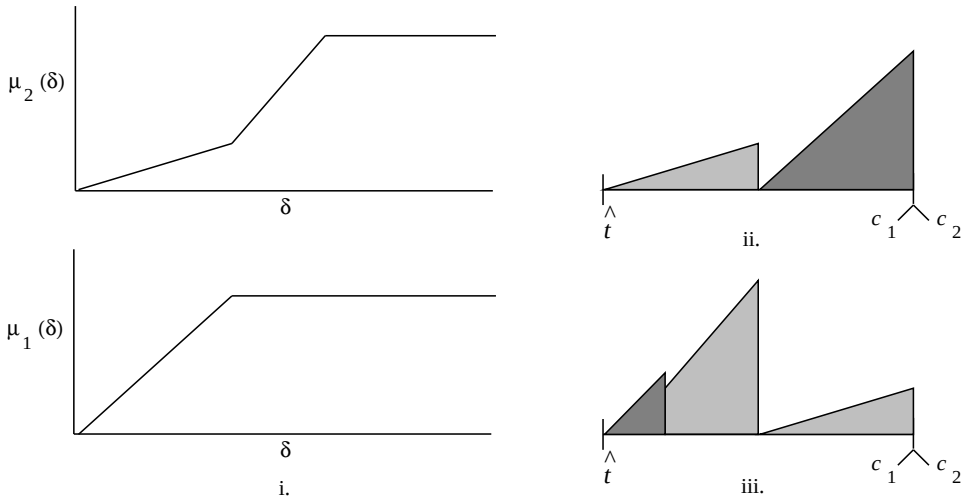

Figure 7: Performance profiles that foil DS

Step 3 follows from the observation that the allocation of the time from the occurrence of the first event to the occurrence of the last is independent of any consideration of the first event or any time available for deliberation prior to the occurrence of the first event. □

Theorem 1 proves that the allocations made by DS are optimal in a well-defined sense. We still have to show that the method for scheduling when to run decision procedures is correct. In particular, we have to show that DS generates a *legal* schedule, where a legal schedule is one such that for all $c \in C$ the time allocated to the decision procedure for c is scheduled prior to the time at which c occurs. To see that DS does generate legal schedules, note that DS ensures that the sum of the time allocated to all conditions that occur prior to t for any $t > \hat{t}$, is less than $t - \hat{t}$. DS is guaranteed to generate a legal schedule since it schedules all of the time for any condition c before any condition occurring later.

5.2 Deliberation Scheduling with Uncertain Occurrence

In the time-dependent planning problems described above, the exact time of occurrence of conditions is known by the deliberation scheduler. One can easily imagine variants in which the scheduler only has probabilistic information about the time of occurrence of events.

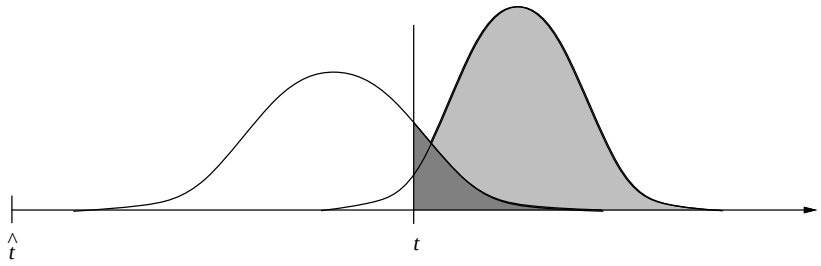

Figure 8: Uncertainty about the occurrence of conditions

For instance, for each condition, c_i, the scheduler might possess a probability distribution

$$p_i(t) = \Pr(\texttt{occurs}(t, c_i)),$$

indicating the probability that a particular condition will occur at time, t. For practical reasons, we will assume that for each condition, c_i, there is a latest time, $\sup(c_i)$, and an earliest time, $\inf(c_i)$, such that

$$p_i(\sup(c_i)) = p_i(\inf(c_i)) = 0.$$

While the scheduler does not know exactly when conditions will occur, we assume that the executor will know when to carry out a given action. For instance, conditions might have precursor events signaling their immediate occurrence. The executor would simply take the best response available at the time the precursor event for a given condition is observed.

Our performance criterion for deliberation scheduling is no longer

$$\sum_{c \in C} \mathrm{V}(\mathrm{Response}(c)|c)),$$

but rather,

$$\sum_{c \in C} \int_{\hat{t}}^{\infty} \Pr(\texttt{occurs}(t, c))\mathrm{V}(\mathrm{Response}(c, t)|c)\, dt,$$

where $\mathrm{Response}(c, t)$ indicates the response generated with respect to condition, c, given that c occurs at t.

In deciding how to allocate an interval of processor time given uncertainty about the occurrence of conditions, we have to account for the possibility that the event may have already occurred. Figure 8 depicts the probability density functions for

the time of occurrence of two conditions. The areas of the shaded regions indicate the probability that each of the corresponding conditions occur in the future of the time marked t.

We extend the δ_i notation to represent processor schedules. Let each δ_i be a function,

$$\delta_i : \mathbf{R} \rightarrow \{0, 1\},$$

where $\delta_i(t) = 1$ if the decision procedure for c_i is allocated the processor at t, and $\delta_i(t) = 0$ otherwise. Let δ be a uniprocessor schedule,

$$\delta : \mathbf{R} \rightarrow \mathbf{Z},$$

where $\delta(t) = i$ just in case $\delta_i(t) = 1$, and the processor is allocated to at most one decision procedure at a time (i.e., if there exists an $i > 0$ such that $\delta_i(t) = 1$ then $\delta_j(t) = 0$ for all $j \neq i$). If $\delta(t) = 0$, then the processor is idle at t.

The expected value of δ_i is the sum over all t following $\hat{t}$ of the expected value of the response generated by the decision procedure for c_i given the processor time scheduled between $\hat{t}$ and t weighted by the probability that c_i occurs at t,

$$\mathrm{E}(\mathrm{V}(\delta_i)) = \int_{\hat{t}}^{\infty} p_i(t)\mu_i(\delta_i(\hat{t}, t))\, dt,$$

where $\delta_i(\hat{t}, t)$ is the total amount of time allocated to c_i between $\hat{t}$ and t. The expected value of a uniprocessor schedule, δ, is the sum of the expected value of the δ_i,

$$\mathrm{E}(\mathrm{V}(\delta)) = \sum_{i=1}^{n} \mathrm{E}(\mathrm{V}(\delta_i)).$$

The procedure DS′ in Figure 9 generates deliberation schedules for the class of problems involving uncertainty in condition times. We represent the time to be scheduled as a window of width $t_{\text{last}} - \hat{t}$, defined by the start time $\hat{t}$ and $t_{\text{last}} = \max\{\inf(c_i) | c_i \in C\}$. This window is divided m times into intervals of width Δ, which are allocated discretely. The assignment $\delta \leftarrow 0$ results in $\delta(t) = 0$ for all $t \in \mathbf{R}$. The assignment $\delta(t, t') \leftarrow i$ results in $\delta_i(\tau) = 1$ and $\delta_j(\tau) = 0$ for $j \neq i$ for all $t \leq \tau < t'$, and, for all $\tau < t$ and $\tau \geq t'$, δ is the same as it was prior to the assignment.

In the procedure DS, any deliberation concerning the ith event is allocated prior to the occurrence of that event. In DS′, the exact time of occurrence of the ith event

```
Procedure DS′
   ;; Initialize the schedule to 0.
   δ ← 0
   ;; Execute m backward passes.
   for i = 0 to m − 1
      ;; Set the latest possible time for an event to occur.
      t ← t̂ + m Δ
      ;; Set the earliest time for the ith backward pass.
      t′ ← t̂ + i Δ
      ;; Allocate time working backward from the latest time.
      until t = t′
         ;; Select the index with the maximum expected gain.
         i ← arg_i max{E(V(δ_i|δ(t − Δ, t) ← i)) − E(V(δ_i))|c_i ∈ C}
         ;; Allocate the time to the appropriate procedure.
         δ(t − Δ, t) ← i
         ;; Decrement time by the amount of allocated time.
         t ← t − Δ
```

Figure 9: Deliberation scheduling with uncertain condition times

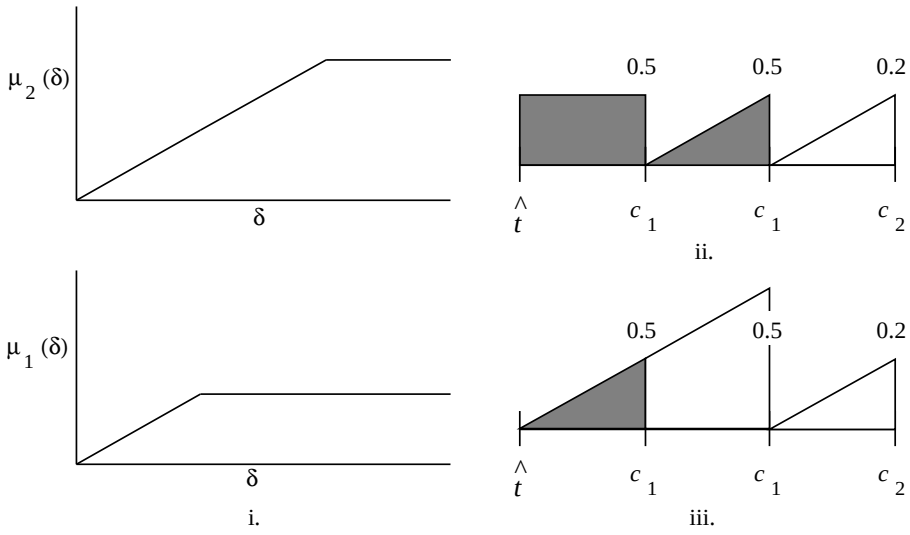

Figure 10: Motivation for a multiple-pass algorithm

is not known; deliberating about the ith event at t will not help if the event occurs prior to t. Figure 10 illustrates why a single-pass approach such as that used in DS is inadequate for the case in which there is uncertainty about the time of occurrence of events. Figure 10.i shows performance profiles for two condition types, c_1 and c_2. Figure 10.ii shows the schedule generated by DS$'$ after a single pass through the `for` loop. In Figures 10.ii and 10.iii, vertical tick marks corresponding to the possible occurrence of events are annotated with the type and probability that the expected instance of that type will occur at the indicated time. Figure 10.iii shows the schedule after a second pass through the `for` loop.

DS$'$ is not optimal. One problem is that DS$'$ allocates each interval of width Δ on the basis of expectations computed for a single point at the boundary of that interval; in general, this method will result in a suboptimal deliberation schedule. We say that δ is a Δ-*approximate schedule* just in case for each $1 \leq j \leq m$ there exists some $0 \leq i \leq n$ such that $\delta(\tau) = i$ for all $(t - (j-1)\Delta) \leq \tau < (t - j\Delta)$. We say that δ is an optimal Δ-approximate schedule if δ is a Δ-approximate schedule and there is no alternative Δ-approximate schedule, δ', such that

$$\mathrm{E}(\mathrm{V}(\delta')) > \mathrm{E}(\mathrm{V}(\delta)).$$

The schedule shown in Figure 10.iii is an optimal Δ-approximate schedule for Δ equal to the distance separating two consecutive tick marks on the time line.

Suppose that the distributions governing the time of occurrence of events are discrete and that events have a non-zero probability of occurring only on the boundaries of Δ-width intervals (i.e., $p_i(t) = 0$ for all $t \notin \{\hat{t} + j\Delta | 1 \leq j \leq m\}$). Suppose further that the functions describing the performance profiles satisfy the property of *diminishing returns* defined earlier. It can be shown that even under these restrictions, DS$'$ does not always find Δ-approximate schedules.[9] At this point, the complexity of finding an optimal Δ-approximate schedule is an open problem (i.e. we have failed thus far to find either a polynomial scheduling algorithm or a suitable reduction for a hard problem).

The schedules produced by DS$'$ are evaluated under the assumption that there will be no further opportunities to modify the schedule. In practice, however, each

[9]For a counter example, consider a modification to the example shown in Figure 10 such that the probability of the latest possible event, c_2, is 0.5, and assume that, whenever DS$'$ has a choice between two equally good options, it chooses disadvantageously.

time that a condition occurs (so that we have precise, rather than stochastic, information regarding the time of that occurrence), it will be useful to compute a new deliberation schedule. For some classes of time-dependent problems, the difference in expected utility between an "optimal" schedule under this assumption and an optimal schedule using expectations on the occurrence of conditions in the future can be bounded and shown to be negligible.

5.3 Dependencies Among Decision Procedure Results

Both DS and DS′ require the assumption that the responses to the different conditions are independent, so that the expected utility of a given deliberation schedule may be computed additively. In this section, we consider a variant of time-dependent planning in which this assumption does not hold.

In this variant, we assume that there are no external conditions requiring responses of the controller; instead, the controller has some number of tasks it is assigned to carry out. The tasks do not have to be completed by any particular time, but the sooner they are completed the better. As in the previous problems, we assume that there is a decision procedure for each task. Generally, the more time the controller deliberates about a given task, the less time it takes to carry out that task. We assume that the outcome of deliberation concerning one task is independent of the outcome of deliberation concerning any other.[10]

The performance profiles relate the time spent in deliberation to the time saved in execution. For instance, suppose that the task is to navigate from one location to another, and the decision procedure is to plan a path to follow between the two locations; up to a certain limit, the more time spent in path planning, the less time spent in navigation.

In the following, we consider a few special instances of this class of problems. In the first instance, all of the deliberations are performed in advance of carrying out any task. This model might be appropriate in the case in which a set of instructions are compiled in advance, and then loaded into a robot that carries out the instructions. Deliberation scheduling is simple. For each task, the scheduler allocates time to deliberation as long as the time spent in deliberation results in a greater reduction in the time spent in execution. All of the deliberation is then performed in advance of any execution.

[10] This is an abstraction of the robot courier problem discussed in Section 4.

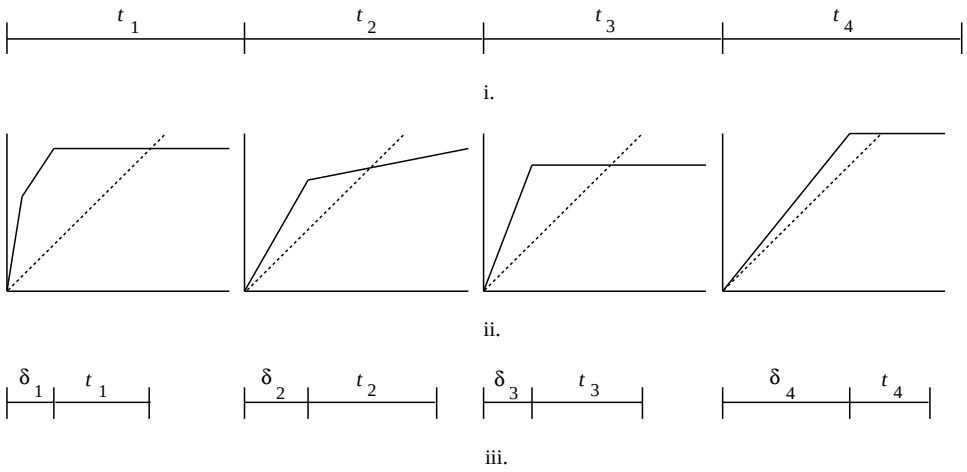

Figure 11: Minimal allocations of processor time

In the second instance, the order in which the tasks are to be carried out is fixed in advance, and all deliberation concerning a given task is performed in advance of carrying out that task, but deliberation concerning one task can be performed while carrying out another. In this instance, deliberation scheduling is somewhat more complicated. We consider deliberation scheduling in terms of three steps, *minimal allocation*, *dead-time reduction*, and *free-time optimization*. In the minimal-allocation step, we proceed as in the previous instance, by determining a minimal allocation for each task, ignoring the possibility of performing additional deliberation during execution.

This minimal allocation for a given task corresponds to that allocation of deliberation time minimizing the sum of deliberation and expected execution time. Figure 11.i shows four tasks and the time they are expected to take, assuming no time spent in deliberation. Figure 11.ii shows the performance profiles for each of the four tasks. The dotted line in each performance profile indicates the minimum slope such that allocating deliberation time will result in a net decrease in the sum of deliberation and expected execution time for the minimal allocation. Figure 11.iii shows the minimal allocations for each of the four tasks, where δ_i indicates the time allocated to deliberation for t_i.

Using the allocations computed in the minimal allocation step, we construct a schedule in which tasks begin as early as possible subject to the constraint that all of the deliberation for a given task occurs in a continuous block immediately preceding

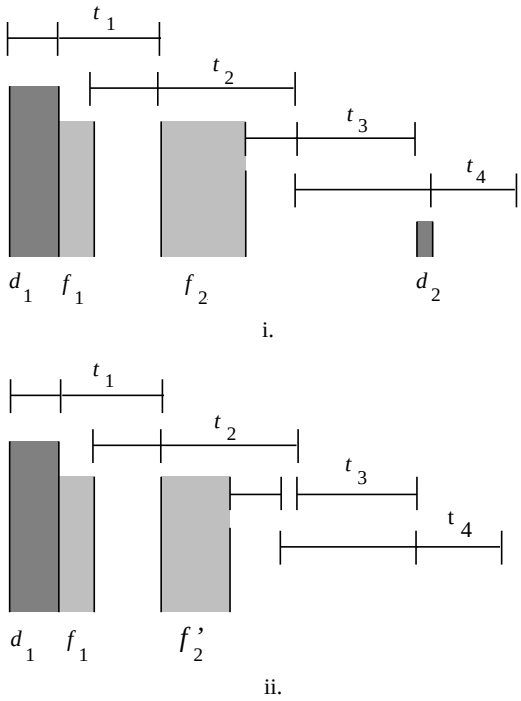

Figure 12: Reducing dead time using available free time

the task and following any deliberation for the previous task. Figure 12.i shows the resulting schedule for the example of Figure 11. Note that there are two additional types of intervals labeled in Figure 12.i. This first type, notated f_i, indicates the *free time* associated with t_i, corresponding to time when the system is performing a task but not deliberating. The second type, notated d_i, indicates the *dead time* associated with t_i, corresponding to time when the system is deliberating but not performing any task.

In the dead-time reduction step, we attempt to reduce the amount of dead time in the minimal-allocations schedule by making use of earlier free time. Where possible, we allocate earlier free time to performing the deliberations previously performed during the dead time, starting with latest dead-time intervals and working backward from the end of the schedule and using the latest possible intervals of free time. Figure 12.ii shows the schedule of Figure 12.i modified to eliminate one of the dead-time intervals through a decrease in the free time f_2, associated with task t_2. It is not always possible to eliminate all dead-time intervals. In particular, any deliberation time allocated for the first task will always correspond to dead time.

Following this process of dead-time reduction, if there is any free time left, we attempt to allocate it for deliberating about other tasks. This is just a bit tricky, since by performing additional deliberation we eliminate previously available free time. Not only do we eliminate the free time we are filling in by scheduling deliberation, but the deliberation reduces execution time thereby eliminating additional free time.

The three-step procedure consisting of minimal allocation, followed by dead-time reduction, followed by free-time optimization does not produce optimal deliberation schedules.[11] The three steps were described to illustrate some of the issues that arise

[11] Suppose that we have n tasks of length L, and each task has a piecewise-linear performance profile, consisting of an initial segment of slope $3/2$ followed by a segment of slope 0. The positive-slope segment ends at $t = L/2$ so that the maximum reduction in task duration is $3L/4$. The minimal allocation for each task is $L/2$ and the resulting execution time is $L/4$. The total time including both deliberation and execution using the three-step procedure outlined above is

$$nL/2 + L/4 = (2n+1)L/4.$$

Instead of using the minimal allocations, suppose that we allocate $L/3$ time units for deliberation to each of the first $n-1$ tasks, and $L/2$ to the last task. The execution time for all but the last task is now $L/2$. In this case, the total time required for $n \geq 2$ is

$$L/3 + (n-1)L/2 + L/4 = (2n-1)L/4 + L/3 < (2n+1)L/4.$$

in problems whose performance criteria depend on minimizing the total time spent in deliberation and execution. Boddy [4] provides a provably optimal, efficient algorithm for a special case in which all of the performance profiles are piecewise linear composed of two linear segments such that the slope of the first segment is the same for all profiles and the slope of the second is 0. This corresponds to the specification of the robot-courier problem described in the previous section, regarding the task of optimally allocating processor time for planning several paths between locations in a tour of such locations to be visited.

There are many variations on the problems described above. This section is meant as a sampler of problems and associated deliberation scheduling techniques. In the next few sections, we discuss how these techniques need to be modified in order to handle more complex problems.

6 Anytime Algorithms

For the time-dependent planning problems discussed in Section 5, we abstract away from the actual decision procedures being scheduled. We manipulate performance profiles that we have been handed. The deliberation-scheduling problem is structured such that using the expected values cached in these profiles (rather than distributions over the answers returned for a given allocation of time, for example) will suffice.

Looking at less abstract problems leads immediately to the realization that employing anytime algorithms involves some complex issues. For example:

- What kinds of anytime algorithms are there?
- What does it mean for an answer to "improve" as more time is allocated?
- How do we construct new anytime algorithms for complex problems?

There is also a question that is more properly addressed in terms of the implemented version of an anytime algorithm (i.e., an *anytime decision procedure*):

This deliberation schedule is still not optimal, but it is better than the schedule obtained using the three-step procedure.

- What kinds of expectations on the behavior of anytime decision procedures will be useful, and what kinds of expectations can we gather (i.e., what kinds of performance profiles can we or should we construct)?

6.1 Existing Anytime Algorithms

One question has been asked repeatedly since we started this line of research: what makes us think that useful anytime algorithms can be found for a sufficiently wide variety of computational tasks to make this approach interesting? In the course of our work and in scanning the the computer science literature, we have turned up a wide variety of algorithms or classes of algorithms that could be employed in anytime procedures. Among the kinds of algorithms we found:

- Numerical approximation – For example, Taylor series approximations (e.g., computing π or e) and iterative finite-element methods [40].
- Heuristic search – Algorithms for heuristic search, in particular those employing variable lookahead and fast evaluation functions, can be cast as anytime algorithms [29, 28].
- Dynamic Programming - Dynamic programming [2] is a general and flexible methodology that can be applied to a wide variety of optimization and control problems, many of them relevant to current AI research (e.g., scheduling, probabilistic reasoning, and controlling machinery). Boddy [3] discusses a method for constructing anytime procedures for solving dynamic programs.
- Probablistic algorithms – One family of probabilistic algorithms that can easily be adapted for anytime use are *Monte Carlo* algorithms [20].
- Probabilistic inference – A wide variety of methods has been developed for approximate evaluation of belief nets (i.e., providing bounds on the posterior distribution, rather than the exact distribution). Several of these methods are anytime algorithms in the sense that the bounds get smaller for additional iterations of the basic method, e.g., [24, 23].
- Discrete or symbolic processing – Symbolic processing can be viewed as the manipulation of finite sets (of bindings, constraints, entities, etc.) [32]. Elements are successively added to or removed from a set representing an ap-

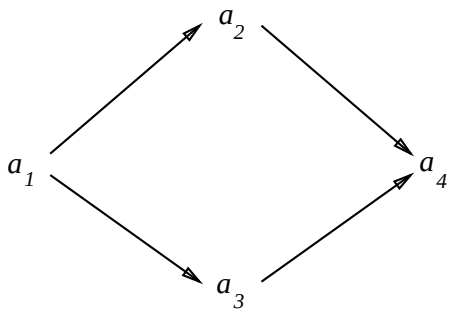

Figure 13: Undirected cycle in a procedural-dependency graph

proximate answer so as to reduce the difference between that set and a set representing the correct answer.

6.2 Constructing Anytime Algorithms

For a given time-dependent problem, deliberation may involve some computation that can be accomplished by a combination of known anytime decision procedures. For example, the system may need to combine the results of several database queries or to perform several different kinds of analysis on a data set in order to come up with an interpretation. A library of anytime decision procedures and performance profiles that can be combined to construct new procedures should simplify the process of generating solutions for time-dependent problems. The end result of this combination should be a procedure for performing a single deliberative task, one of a suite of procedures to which a deliberation scheduler might allocate time.

We can combine anytime decision procedures in various ways. The final result may be the direct output of a component procedure or some simple combination (e.g., addition) of the results of several component procedures. One component anytime decision procedure may take as input the output of several others. It is also possible for several anytime decision procedures to use the result of a single procedure (e.g., a sensor interpretation routine). One approach as to how this combination can be accomplished, including constructing a performance profile for the composite procedure, is given in [4].

Moving beyond simple combinations complicates the situation. We can think of the information dependencies among anytime decision procedures as a directed

graph in which the nodes are anytime decision procedures or combinations of data in some other way (e.g., addition), and the arcs represent the flow of information (i.e., the results of one procedure being used by another). Such a graph provides a partial ordering for the corresponding anytime decision procedures according to the flow of information from one to another. One problem with this formulation is that undirected cycles represent dependencies we need to take into account. Consider the example in Figure 13. If a_1 with probability p produces an output that causes both a_2 and a_3 to produce very bad answers, this fact is accurately represented in the distributions calculated for the outputs of a_2 and a_3. Unfortunately, if in constructing a output distribution for a_4 we only consider a_2 and a_3, the probability of these very bad answers happening at the same time will be calculated as p^2 rather than the correct value of p. We can use simple parallel and serial combinations to construct performance profiles for composite procedures corresponding to arbitrary graphs, but the dependencies represented by undirected cycles will not taken into account.

6.3 Performance Profiles

In order to allocate deliberation time effectively, we gather expectations on the behavior of the anytime decision procedures available. Ultimately, what we want to know is the expected effect of a particular allocation on the utility of the system's behavior. In Section 5, we are given this expectation.

More commonly, what we have are expectations or distributions on how some parameter of the answer returned by an anytime decision procedure changes with increasing allocation. Expectations on the answer returned are simpler to obtain, and not so dependent on the structure of a particular problem, as expectations on how the system's behavior will change. We can compute expectations on the answers returned by an anytime decision procedure given a particular input distribution, and then use that anytime decision procedure and those expectations for any time-dependent problem for which the assumptions made about the input distribution are reasonable. We call such a set of expectations a *performance profile*.

Part of the definition of an anytime decision procedure is that the answer it returns is expected to improve with increasing allocations of deliberation time. Answers might "improve" in the following ways, among others: convergence to an optimal value, narrowing bounds on an estimate, increasing probability that the

answer is correct, or increasing the intersection of or decreasing the difference between the current answer and the correct answer (the "answers" in this case are sets). A single anytime decision procedure may have several performance profiles, each providing information on the distribution of a different parameter of the answer returned or for a different distribution over the possible values of its inputs.

Which of these profiles is most appropriate or useful depends on the task at hand. Horvitz [26] discusses various algorithms for sorting an array viewed as anytime algorithms (he calls them *flexible computations*). Each algorithm makes repeated changes to the array, each one bringing the array closer to being sorted. What is meant by "closer" differs for each algorithm, however. Suppose that we start with an unsorted array, and look at the changes made to that array as the sort progresses. Insertion sort produces a lengthening sorted section at the beginning or end of the array. Bubble sort produces an increasing number of elements in the correct relative position. Shell sort produces a gradually decreasing maximum distance for entries from their true position. Which of these algorithms is most appropriate for a particular application depends in part on what kind of "approximate sort" is most useful.

In previous work [10, 5], we have compiled performance profiles that tracked the expected value of (some parameter of) the answer returned by a given anytime decision procedure as a function of deliberation time. Performance profiles of the type shown in shown in Figure 3, Section 4 are approximations in that anytime procedures are generally iterative, and thus provide answers that improve in a series of discrete steps. Continuous improvement is an assumption we make, and is required for applying many of the techniques we employ (e.g., this continuity is required for the procedure DS in Section 5). If the time required for each iteration is uncertain, a continuous profile may provide the best information available, in the sense that the points at which the answer improves (an interation completes) cannot be specified precisely.

The property of *diminishing returns* defined in Section 5 turns out to be useful, in that it can simplify the optimization problem involved in deliberation scheduling (as in Section 5).[12] This property is shared by many interative algorithms, including most of the examples discussed in Section 6. There are certainly iterative methods

[12] An anytime procedure exhibits diminishing returns if the rate of improvement in the answer for an additional allocation does not increase (and will eventually decrease) with the time already spent.

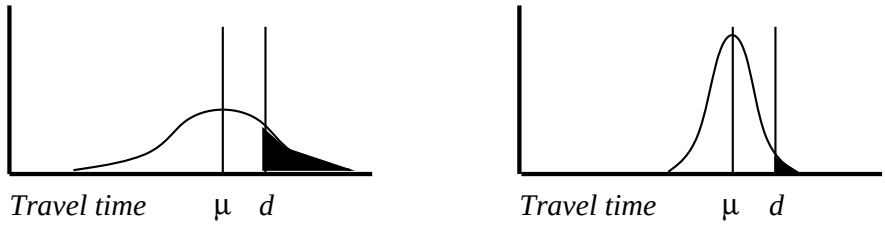

Figure 14: Expected travel time and a deadline

that do not exhibit diminishing returns. Consider the problem of finding a 10-digit combination for a safe, using a procedure that at each step produces the next digit [0-9] in the combination. The probabilities of guessing the correct combination, given the digits already known, at each iteration form the sequence $\langle 10^{-10}, 10^{-9}, \ldots, .1, 1\rangle$. If we take the expected utility to be the utility u of opening the safe times the probability of opening the safe, the gain in expected utility for the last step ($0.9u$) is much greater than for the first step ($0.000000009u$).

Another point that should be made regarding performance profiles is that the 2-dimensional graphs we have shown so far are a simplification. Knowing the expected value is in general insufficient—there are situations where what is needed is a marginal distribution, not just a sum or product of expected values. For instance, two path planners that produce paths with the same expected length for a given allocation of deliberation time can have very different expected utilities if arrival time has a hard deadline: the planner with greater variance on expected length may have a lower expected utility, because it has a greater probability of missing the deadline. Figure 14 shows an example of this. Each graph shows a distribution of travel times and a deadline, and in each case, the mean μ and deadline d are the same. The area of the shaded region is the probability of missing the deadline. The relative cost of using the path planner with the larger variation in arrival times will depend on the cost of being late.

A completely general performance profile is a surface in 3-space, not a 2D graph. Taking a slice across this surface for a given allocation of deliberation time provides a distribution on some parameter of the answer returned. 2D performance profiles graph the means of the distributions represented in the more general form. One problem with this more general form of profile is that it requires more storage space. It also takes longer to gather enough data to compute reasonable distributions for each different allocation, or even for just a few of them. In addition, calculations

using the means are usually much simpler and faster than computing a distribution would be.

We can define the cost of using the simpler performance profiles in terms of the expected utility of the system's performance. Let $E(U_x)$ be the expected utility of using an optimal deliberation scheduler for a decision model that includes the simpler kind of performance profiles, and $E(U_{IU})$ be the expected utility of using an optimal deliberation scheduler for a second decision model that differs only in using the 3D performance profiles. Then the cost of using the simpler profiles is just $E(U_x) - E(U_{IU})$.[13] For time-dependent planning problems (Section 5), this cost is zero, because overall utility is the sum of the utility of independent responses.

As long as we ignore the cost of deliberation scheduling, the cost of using the simpler profiles is at least zero. In situations where the space or time required for deliberation scheduling is significant and must itself be scheduled, the "cost" of calculating using only the expected values may well be negative, because the calculations are so much simpler.

7 Practical Deliberation Scheduling

Decision-theoretic deliberation scheduling requires that we be able to formulate a decision model for making allocation decisions, and to generate a deliberation scheduler that makes allocation decisions (perhaps approximately) in accordance with that decision model. In this section, we address each of these problems in turn.

7.1 Optimal Decisions

We wish to maximize the expected utility resulting from the allocation decisions made by the system. The basic decision to be made is made repeatedly: for each increment of time, which of the available decision procedures will be run? This is a *sequential decision problem* of a form commonly addressed in operations research [34]. An *optimal decision policy* is defined as a policy that at each point makes the

[13] Henrion [22] calls this the *expected value of including uncertainty.* (EVIU)

decision maximizing the expected utility of the current decision, given expectations on the decisions yet to be made.[14]

Consider the simple example of a database server that must respond to queries by fixed deadlines (this is a time-dependent planning problem of the sort discussed in Section 5). Queries cannot be predicted (other than in a statistical sense) before they are made. Each moment of time that passes provides new information in terms of the queries that did (or didn't) happen. For this example, this new information could take any of the following forms:

- A query of some particular type was received.
- No query was received.
- New information on an anytime decision procedure became known:
  - An anytime decision procedure finished, in the sense that there is known to be no benefit to allocating additional time. The system may or may not be able to detect this.
  - The intermediate result provided by an anytime algorithm was better/worse/different than expected. Again, the system may or may not know enough about the anytime algorithm's behavior to detect this.

A deliberation scheduler that does not take this new information into account in making allocation decisions may not do as well as one that does (i.e., the expected utility of the resulting behavior may be lower).

Optimal deliberation scheduling is a sequential decision problem, in which each allocation decision depends on expectations on what will happen in the future. Each instant of time provides new information (a new *state*) and requires a new allocation decision. Following methods used in operations research, we might try using *dynamic programming* [2, 34] to solve our optimization problem. Dynamic programming involves caching the optimal answer to a series of subproblems of increasing size, until finally we can determine the optimal answer to the full problem. For problems involving a sequence of decisions made over time, these subproblems will be subsequences of decisions, starting from a particular state. We could start

[14] Infinite sequences of decisions are handled using *discounting*, in which the expected reward resulting from a particular decision is weighted inversely to how far in the future the decision is.

with the last tick in some history, for example, caching the optimal allocation for that tick for any of the possible states, and then move back to consider the next-to-last tick. Our problem is complicated by the fact that we have only expectations on the schedule of events and the results of decisions. This complicates the computation by requiring us to calculate expected utility based on what *may* happen for each sequence of decisions (typically a set of possibilities), rather than from a single deterministic outcome.

Calculating a policy for a sequential decision model requires reasoning about expectations from a given state of the world. For time-dependent planning problems, the current state is determined by:

- The types and times of occurrence for predicted events that have not occurred yet.
- Any deliberation that has already been done on these events.
- State-specific expectations on the future occurrence of events.

Even if times of occurrence and deliberation allocations are integer-valued, there may be a very large set of possible states. While some of these states may be equivalent from the point of view of allocating deliberation time, determining that may not be easy.

7.2 Using Reduced Decision Models

The discussion in the last section suggests that constructing decision models (and ultimately deliberation schedulers) that make full use of available expectations on the future occurrence of events may be difficult. In this section, we explore the effect on the expected utility of the system's behavior of using a reduced decision model, in which these expectations are ignored. We report on the results of experiments in three domains: time-dependent planning problems as defined in Section 5, the Robot Courier, and the Shooting Gallery.[15]

[15] More detailed descriptions of these domains and the experiments described in this section may be found in [4].

7.2.1 Time-Dependent Planning

The deliberation schedules generated by the procedure **DS** in Section 5 are optimal for a decision model that does not include expectations on the future occurrence of events. Suppose that we are given a dynamic environment in which *observations* occur as time passes. Each observation specifies a new condition (event), including a time of occurrence. We define a deliberation scheduler for this domain that uses `DS` to generate new schedules for all known events whenever observations occur and otherwise follows the allocations given by the last deliberation schedule constructed. We call this a "static" scheduler and the decision model that does not include any expectations on events the "static case."

The situation actually faced by the system is one in which new observations and events may occur at any time. We demonstrate empirically that applying this static deliberation scheduler to a dynamic environment results in deliberation allocations that on average are very close to optimal for a decision model that includes expectations on the occurrence of events. We do this by comparing the performance of the static scheduler to a deliberation scheduler that has perfect information (i.e., knows every query and its deadline). The perfect information scheduler is running `DS` on the entire schedule, including conditions for which the corresponding observations have not occurred yet, with the restriction that the scheduler is not allowed to make allocations for a condition before the time the corresponding observation occurs. This perfect-information scheduler has expected performance better than that of an optimal deliberation-scheduling policy that makes use of expectations on the future occurrence of events.

The comparison involves repeatedly generating histories of observations and events, using a fixed probability that an observation would occur for any time unit (0.01), out to a fixed endpoint (30000 time units). The deliberation schedulers were then simulated on the resulting schedules and the expected utilities of the resulting allocations were computed and averaged across trials.

The results of these experiments are graphed in Figure 15. The independent variable is the ratio $\overline{d}$ of the time between observation and condition to the expected time between observations (100 time units). Varying this ratio affects the average number of condition the static procedure knows about when it constructs a deliberation schedule. As $\overline{d}$ decreases, both algorithms do worse. This is not surprising: the only possible time for allocation is between observation and condition.

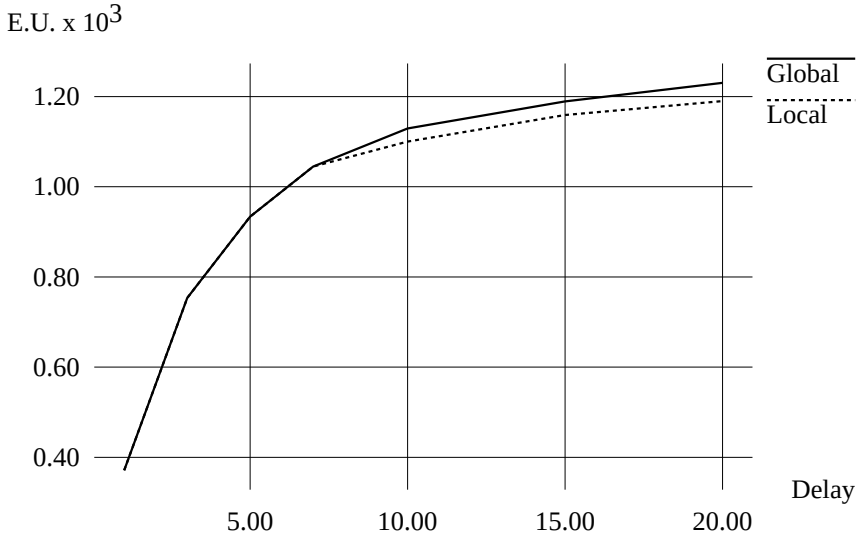

Figure 15: Relative performance of Static and Perfect-Information Schedulers

For small values of $\overline{d}$, there is usually at most a single predicted event, and so no tradeoffs can be made to increase expected utility. There is also simply less time: in any part of a history where conditions occur at less than the expected rate, $\overline{d} = 1$ means that there is time that cannot be used, time that would be available (and would be allocated) for a larger value of $\overline{d}$.

The most interesting feature of these experiments is the relative performance of the local (static) algorithm and global algorithms. Not only does the local algorithm do very well, it does better (relative to the global one) for smaller values of $\overline{d}$, which is somewhat counterintuitive. One might expect that increasing the prediction delay, and thus the piece of the schedule that the system knows about when it generates a deliberation schedule, would improve the relative performance of the local algorithm. What happens in practice is that this effect is dominated by the constriction in the possible tradeoffs as $\overline{d}$ diminishes. As the prediction delay gets smaller, the number of conditions for which deliberation time can be traded off gets smaller as well, decreasing the opportunities for the global algorithm to exploit its additional information. The fact that the global algorithm has no advantage until $\overline{d}$ is about 8 and a minimal advantage all the way out to $\overline{d} = 20$ indicates that the efficient local scheduling algorithm produces schedules that are very close to

optimal for a wide range of time-dependent planning problems.

The results of these experiments indicate that for at least some time-dependent planning problemsthis local version of **DS** has expected performance close to the theoretical maximum, given the available information. This is true even (in fact especially) when the horizon over which the algorithm can "see" is a small multiple of the expected time between observations. The basic intuition explaining the near-optimal performance of the local algorithm is that new information rarely changes the optimal allocation for the part of the deliberation schedule currently being executed.

7.2.2 The Robot Courier

Suppose we are in charge of designing the control program for a robot courier for a Manhatten delivery service. The function of these couriers is to pick up small parcels and deliver them to specified locations. Assuming that a courier completes all of its assigned deliveries, the main factor in evaluating its performance will be the speed with which it accomplishes this task. The faster deliveries are made, the more deliveries a courier can make, and the more income it will generate. The robot must fix upon an order in which to visit the locations on its current delivery list. We call the process of choosing this ordering *tour-improvement* planning. Once the robot has an ordering for the locations, it may spend time determining how best to get from one of them to another (*path* planning).

Boddy [4] presents three different deliberation-scheduling algorithms for different variants of this problem:

- Pr-I, in which only path planning may be scheduled.
- Pr-II, in which all tour improvement must occur before the robot moves or does any path planning.
- Pr-III, in which the robot may plan paths for and move between some set of points while doing tour improvement and path planning for the rest of the points on its list.

We performed a series of experiments involving randomly-generated enviroments and sets of locations for the robot to visit. We compared the performance of these

three algorithms to one another in both static and dynamic environments (i.e., either the entire list of requests is known from the start, or it arrives over time). In both static and dynamic environments, Pr-II is a substantial improvement over Pr-I, but Pr-III has less of an advantage over Pr-II, particularly in the dynamic case.

We also investigated the degree to which ignoring the future occurrence of events affected the performance of the three schedulers in a dynamic environment. The answer to this question depends on the mean time between requests. The performance of all three schedulers falls off as the request probability drops (eventually the robot spends a significant amount of time waiting for a new destination). So we choose an intermediate value, on the same order as the mean time to travel between locations, and compare the schedulers' performance to their performance in the static case. There are other factors affecting the dynamic behavior of the system as well, for example the relative cost of tour improvement and path planning. For a more complete description of the effect of these factors, see [4].

For a set of experiments in which the "clock rate" for path planning was twice that for tour improvement, the gain in the average time required for Pr-I in the dynamic case is less than 10%. Pr-II and Pr-III do not fare as well, suffering about a 40% increase in average completion time. One possible explanation for the difference between Pr-I and Pr-II or Pr-III is the qualitative difference between path planning and tour improvement: new requests can render useless most or all of the path planning done so far, if they make it advisable to reorder the tour, while new requests simply added to the end of the current tour have less effect on what earlier path-planning allocations should be. Doubling the tour-improvement "clock rate" results in a situation where the increase in completion time in dynamic environments for Pr-II and Pr-III is less than 15%.

7.2.3 The Shooting Gallery

For time-dependent planning and the Robot Courier, simplifying the decision model in fairly obvious ways had relatively little effect on the system's performance. In this section, we present a third example, in which similar simplifications impose a significant cost in terms of the expected utility of the system's performance.

Suppose that a robot stands in front of an arcade shooting gallery. Targets appear at irregular intervals on one side of a screen some distance in front of the

Figure 16: The Arcade

robot and cross to the other side. The robot uses a gun to try to hit these targets. Hitting one of the targets with a projectile has a known value. The value of each target the robot hits is added to its score. The robot's task is to accumulate as many points as possible from a fixed set of targets. This domain, the *Arcade*, is shown in Figure 16.

The robot also has two sensors of different kinds. The first sensor provides information essentially for free, but is not very useful; it tells the robot when a new target appears, but only roughly where, and can also be used to determine when a target disappears off the other side of the screen. The second sensor can be used to find the approximate position of a specified target. This sensor is expensive to use in that there is a delay between asking for a sensor reading and obtaining the information. Since the sensor cannot be used again during this delay, there is a fixed minimum interval between sensor readings.

At each point in time, the only decisions available to the robot are: whether or not to fire the gun, in what direction to move the gun's aim, and whether or not to employ the active sensor and if so on which target. The robot has a simple control system that operates in the absence of any scheduled deliberation. The movement and firing of the gun are controlled by a procedure that determines the direction to move to intercept the oldest target as soon as possible. If it can determine that the target cannot be intercepted before disappearing from the target area, it moves to the next-oldest target. When the gun is pointing at the best estimate of where to

shoot to hit the target, it fires. All sensor readings are taken on the target the gun is currently moving to intercept, or failing that, the oldest target in the target area.

The no-deliberation control system goes after the oldest target in the target area that it can get to before the target disappears. If several targets are on the screen, as is frequently the case, this may not be a very good ordering, since it may be possible to reorder the targets so as to reduce the time spent in moving the gun around. In addition, consider a situation in which there are several targets that all appeared at about the same time, with about the same velocity. All of these targets may appear to be reachable before they disappear, though in fact it may be possible to reach only one or two of them before they all disappear at roughly the same time. Constructing an explicit schedule that takes into account the time required to get from one target to the next will allow the robot to avoid this error.

We split the scheduling problem facing the robot into two pieces: deciding what target to shoot at next and when, and deciding what sensor readings to take in order to maximize the probability of hitting the targets aimed at. We have implemented a target scheduler that determines a schedule of targets to try to hit and fixes the maximum number of shots to be taken at each target. For a given target schedule, the robot can allocate time to scheduling the sensor so as to maximize the expected value of the targets hit. Since the sensor is a scarce resource, we model it as a set of "slots" separated by the fixed sensor delay, each corresponding to a separate reading. Sensor scheduling involves deciding what target will get a reading in each of these slots.

Deliberation scheduling for the Shooting Gallery consists of determining when to do target scheduling and when to do sensor scheduling. The two forms of scheduling are interdependent: sensor readings provide additional information that makes target scheduling more reliable. Reliable target scheduling means that scheduled sensor readings are more likely to be useful, because the target involved will in fact be fired on at the expected time. As a first test of the effect of these two schedulers, we ran a series of trials. For each trial, we noted the utility of the robot's performance (the value of the targets it hit) for four different cases: doing no deliberation, doing only target scheduling, doing only sensor scheduling, and doing both target and sensor scheduling. We ignored the cost of deliberation in these experiments; the object was to establish upper bounds on the improvement in the robot's behavior resulting from time allocated for deliberation using these two schedulers. The results are summarized in Table 1.

Deliberation	Avg Utility
None	0.44
Sensor only	0.45
Target only	0.53
Both	0.54

Table 1: The benefits of deliberation

Sensor scheduling does not appear to be very effective: its improvement on the expected utility of the robot's performance is only about 2%, even when we ignore the cost of deliberation and run the scheduler to completion. This compares to about a 20% improvement in expected utility for the target scheduler, when the cost of deliberation is again ignored.

Given these results, we do not use the sensor scheduler at all. Deliberation scheduling is trivial; we run the target scheduler until one of three things happens:

- A new target appears.
- A target is hit.
- A target disappears without being hit.

These events modify the set of targets the robot knows about, and the target scheduler is restarted with the new information.

The benefit of target scheduling is shown in Figure 17. The leftmost value on the graph is the expected utility when the robot does no deliberation at all. For the rightmost, deliberation is free. In between, the cost of deliberation increases to the left. This graph has one very interesting feature: the cost of the approximations made in constructing expectations has become significant. There is a point at which the expected utility of the robot's performance is actually worse for having done some deliberation than it is for no deliberation at all. This suggests that the expectations the robot uses are not a good model for the actual behavior of the system.

The expectations the robot uses involve the same simplifications as for time-dependent planning and Robot Courier: future events are ignored and expected values, not distributions, are combined, but the cost of these approximations has become significant.

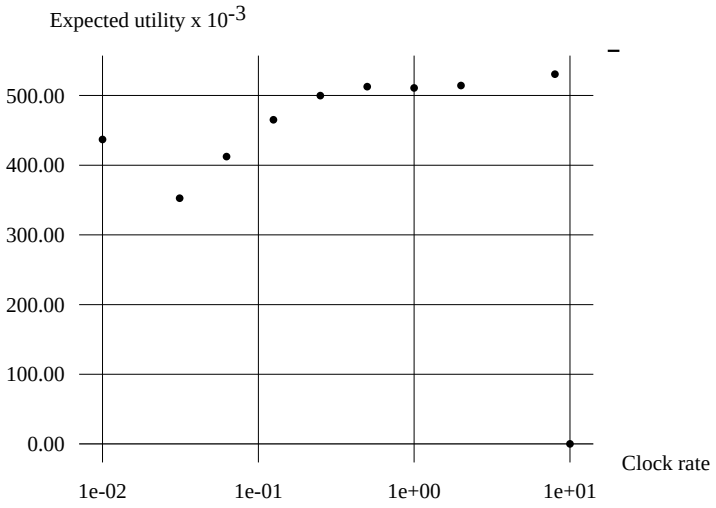

Figure 17: Performance profile for target scheduling

8 Conclusion

In this paper, we describe *deliberation scheduling*: an approach to making tradeoffs among competing demands for computation and action in dynamic environments. We show how the use of anytime algorithms simplifies the problem of providing effective deliberation schedulers and analyzing the behavior of the resulting systems. We demonstrate the application of deliberation scheduling to a simple class of abstract time dependent problems. We also discuss some of the complications encountered in applying this approach to more complex problems.

All of the approaches described in this paper make rather restrictive assumptions in order to avoid the combinatorics involved in dealing with unlimited decision-making horizons and complicated interactions between information sources. We need to extend the current approaches to handle computational models that reflect the complexity of existing problem-solving systems. We need experience with real applications so that the research will be driven by real issues and not artifacts of our mathematical models. An interesting area for future research involves identifying and dealing with restricted types of interactions and providing a disciplined approach to extending decision-making horizons.

The work in time-critical problem solving will have far reaching implications for the whole research community. Time is an issue in any problem solving task. Theoretical results concerning agents with limited computational resources should shed light on a number of basic representation issues. For instance, the notion of a "plan" as a persistent belief does not make sense until you take computational considerations into account. Plans enable a system to amortize the cost of deliberation over an interval of time. If time were not an issue, there would be no justification in committing to a plan.

References

[1] Barnett, V., *Comparative Statistical Inference*, (John Wiley and Sons, 1982).

[2] Bellman, R.E., *Dynamic Programming*, (Princeton University Press, Princeton, NJ, 1957).

[3] Boddy, Mark, Anytime Problem Solving Using Dynamic Programming, *Proceedings AAAI-91, Anaheim, CA*, AAAI, 1991.

[4] Boddy, Mark, *Solving Time-Dependent Problems: A Decision-Theoretic Approach to Planning in Dynamic Environments*, Technical Report CS-91-06, Brown University Department of Computer Science, 1991.

[5] Boddy, Mark and Dean, Thomas, Solving Time-Dependent Planning Problems, *IJCAI89, Detroit, Michigan*, 1989.

[6] Bodin, L. and Golden, B., Classification in Vehicle Routing and Scheduling, *Networks*, **11** (1981) 97–108.

[7] Brooks, Rodney A., *A Robust Layered Control System for a Mobile Robot*, A.I. Memo No. 864, MIT Artificial Intelligence Laboratory, 1985.

[8] Chernoff, Herman and Moses, Lincoln E., *Elementary Decision Theory*, (John Wiley and Sons, New York, 1959).

[9] Dean, Thomas, Decision-Theoretic Control of Inference for Time-Critical Applications, *International Journal of Intelligent Systems*, **to appear** (1991).

[10] Dean, Thomas and Boddy, Mark, An Analysis of Time-Dependent Planning, *Proceedings AAAI-88, St. Paul, Minnesota*, AAAI, 1988, 49–54.

[11] Drummond, Mark and Bresina, John, Anytime Synthetic Projection: Maximizing the Probability of Goal Satisfaction, *Proceedings of the Eighth National Conference on Artificial Intelligence*, 1990, 138–144.

[12] Einav, David, Computationally-Optimal Real-Resource Strategies for Independent, Uninterruptible Methods, *Proceedings of the Sixth Workshop on Uncertainty in Artificial Intelligence, Cambridge, MA*, 1990, 73–81.

[13] Elkan, Charles, Incremental, Approximate Planning, *Proceedings of the Eighth National Conference on Artificial Intelligence*, 1990, 145–150.

[14] Etzioni, Oren, Tractable Decision-Analytic Control, *First International Conference on Principles of Knowledge Representation and Reasoning*, 114–125, (Morgan-Kaufmann, 1989), 114–125.

[15] Garey, Michael R. and Johnson, David S., *Computing and Intractibility: A Guide to the Theory of NP-Completeness*, (W. H. Freeman and Company, 1979).

[16] Georgeff, Michael P. and Lansky, Amy L., Reactive Reasoning and Planning, *Proceedings AAAI-87, Seattle, Washington*, AAAI, 1987, 677–682.

[17] Good, I. J., *Good Thinking*, (University of Minnesota Press, 1976).

[18] Graham, R.L., Lawler, E.L., Lenstra, J.K., and Rinnooy Kan, A.H.G., Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey, *Proceedings Discrete Optimization, Vancouver*, 1977.

[19] Hansson, Othar and Mayer, Andrew, The Optimality of Satisficing Solutions, *Proceedings of the Fourth Workshop on Uncertainty in Artificial Intelligence, Minneapolis, MN*, 1988.

[20] Harel, David, *ALGORITHMICS: The Spirit of Computing*, (Addison-Wesley, 1987).

[21] Hayes-Roth, Barbara, Washington, Richard, Hewett, Rattikorn, Hewett, Michael, and Seiver, Adam, Intelligent Monitoring and Control, *Proceedings IJCAI 11, Detroit, Michigan*, IJCAI, 1989, 243–249.

[22] Henrion, M., *The Value of Knowing How Little You Know*, PhD thesis, Carnegie Mellon University, 1984.

[23] Henrion, M., Propagating Uncertainty by Logic Sampling in Bayes' Networks, *Proceedings of the Second Workshop on Uncertainty in Artificial Intelligence*, 1986.

[24] Horvitz, E. J., Suermondt, H. J., and Cooper, G. F., Bounded Conditioning: Flexible Inference for Decisions Under Scarce Resources, *Proceedings of the Fifth Workshop on Uncertainty in Artificial Intelligence, Windsor, Ontario*, 1989.

[25] Horvitz, Eric J., Reasoning About Beliefs and Actions Under Computational Resource Constraints, *Proceedings of the Third Workshop on Uncertainty in Artificial Intelligence, Seattle, Washington*, 1987.

[26] Horvitz, Eric J., Reasoning Under Varying and Uncertain Resource Constraints, *Proceedings AAAI-88, St. Paul, Minnesota*, AAAI, 1988, 111–116.

[27] King, John R. and Spachis, Alexander S., Scheduling: Bibliography and Review, *International Journal of Physical Distribution and Materials Management*, **10**(3) (1980) 105–132.

[28] Korf, Richard, Real-Time Heuristic Search, *Artificial Intelligence*, **42**(2) (1990) 189–212.

[29] Pearl, Judea, *Heuristics*, (Addison-Wesley, 1985).

[30] Pearl, Judea, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*, (Morgan-Kaufman, Los Altos, California, 1988).

[31] Raiffa, Howard, *Decision Analysis: Introductory Lectures on Choices Under Uncertainty*, (Addison-Wesley, Reading, Massachusetts, 1968).

[32] Robert, F., *Discrete Iterations: A Metric Study*, (Springer-Verlag, 1986).

[33] Rosenschein, Stan and Kaelbling, Leslie Pack, The Synthesis of Digital Machines with Provable Epistemic Properties, Halpern, Joseph Y., (Ed.), *Theoretical Aspects of Reasoning About Knowledge, Proceedings of the 1986 Conference*, Morgan-Kaufman, 1987, 83–98.

[34] Ross, Sheldon, *Introduction to Stochastic Dynamic Programming*, (Academic Press, New York, NY, 1983).

[35] Russell, Stuart J. and Wefald, Eric H., On Optimal Game-Tree Search using Rational Meta-Reasoning, *Proceedings IJCAI 11, Detroit, Michigan*, IJCAI, 1989, 334–340.

[36] Russell, Stuart J. and Wefald, Eric H., Principles of Metareasoning, *First International Conference on Principles of Knowledge Representation and Reasoning*, (Morgan-Kaufmann, 1989).

[37] Shekhar, S. and Dutta, S., Minimizing Response Times in Real Time Planning, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, Detroit, Michigan*, IJCAI, 1989, 972–978.

[38] Simon, H. A. and Kadane, J. B., Optimal Problem-Solving Search: All-or-None Solutions, *Artificial Intelligence*, **6** (1975) 235–247.

[39] Sproull, Robert F., *Strategy Construction Using a Synthesis of Heuristic and Decision-Theoretic Methods*, Technical Report CSL-77-2, Xerox PARC, 1977.

[40] Varga, R. S., *Matrix Iterative Methods*, (Prentice-Hall, Englewood Cliffs, NJ, 1962).