Dynamic Sensor Policies

Kurt Krebsbach
Department of Math and Computer Science
Shippensburg University
Shippensburg, PA 17257
(717) 532-1424
kdkreb@ark.ship.edu

Maria Gini
Computer Science Department
University of Minnesota
4-192 EE/CSci Building
200 Union Street SE
Minneapolis, MN 55455
gini@cs.umn.edu

Introduction

When an agent's task environment is largely benign and partially predictable (although uncertain), and the goals involve accomplishing tasks, we can make the agent more adaptive by planning to acquire unknown or uncertain information during execution of the task. Of course we pay a price for this flexibility. In this paper we dicuss a strategy for measuring this price in a realistic way and reducing it by making rational decisions about how to acquire unknown environmental information with imperfect sensors. Ultimately, we are interested in a general framework for making optimal sensor decisions which will minimize a cost (or set of costs) we expect to incur by employing sensors.

In particular, we propose to generate a tree of possible sensing policies *offline* (using dynamic programming), cache the optimal sensor decisions at each level, and subsequently use actual world states as indexes into this structure to make a rational sensor choice *online*. Because no states are discarded in the dynamic programming process (only non-optimal paths), we are always guaranteed of having earlier cached an optimal decision for each state we anticipate possibly encountering at execution time. This combination of offline and online computation allows the sensor selection process to be sensitive to actual events, while making use of assumptions and regularities inherent in the structure of the domain.

The work presented here builds on some of our earlier work in which we have discussed strategies for static (i.e., offline) sensor scheduling [Krebsbach *et al.*, 1992, Olawsky *et al.*, 1993], and in which we have used techniques similar to these to combat the inherent computational complexity of the problems involved [Krebsbach, 1993]. Many of these ideas have been suggested by others employing decision-theoretic methods for a wide variety of optimization and control problems, including those related to planning and sensing [Bellman, 1957, Boddy, 1991b]. Boddy proposes the use of dynamic programming for constructing anytime algorithms [Boddy, 1991a]. Hager and Mintz [1991] have proposed methods for sensor planning based on probabilistic models of uncertainty. Abramson [1991] casts sensory integration as a decision problem and presents a formula for deciding how often to sense, depending on the rate of change of the environment. Goodwin and Simmons [1992] use a decision-theoretic approach to incorporate the achievement of a new goal into a partially executed plan, and Chrisman and Simmons employ Markov Decision Processes in order to handle a significant amount of uncertainty in the outcomes of actions [1991].

General Methodology

We intend in this work to demonstrate in a general way how to endow a planner with a mechanism for making cost-effective choices about gathering uncertain information necessary to continue planning. Conceptually, the general method we advocate can be described in the following steps:

1. Choose a planning problem and **decide what information will be unknown** (or uncertain) to the planner at the start of planning. If this is most of the information in the domain it is possible that planning does not make sense, and a more reactive approach may be warranted. However, a more important consideration than the amount of unknown information is its type.

   There are two considerations here. The first is whether or not the **unknown information relates in a straightforward way to the goals**. If so, since we can choose a goal ordering in advance, we can determine the order in which the unknown information will be needed. Even if the exact order is not completely determined, it may be possible to at least determine the type of information needed at each decision point. If this is reasonable, we can then treat the sensor selection problem as a sequential decision problem and apply the methods described shortly.

   A second consideration is the **degree to which goals interact**. Some of our earlier work involves domains deliberately designed so that actions for solving one goal may adversely affect the achievement of another. In this paper we describe how to compute these costs and tradeoffs in another domain with somewhat different characteristics.

2. Adding sensors to the plan will increase effort in some way. Since it is likely to increase effort in a number of ways, it is necessary to **determine one criterion or a combinable set** of criteria (e.g., diagnostic cost and printer availability gain) to optimize. In the example shown later we use a complex variation on minimizing execution cost. Other examples might include reducing planning cost or increasing the probability of generating plans which contain no unnecessarily redundant actions.
3. A connection must be made between our sensor choices and their effect on the parameter(s) we are trying to optimize. This can be done empirically or analytically. We employ analytic means for illustration here.
4. Among the most difficult aspects of making this approach useful is to develop a **computationally feasible** way to compare alternative sensor schedules in order to make an optimal decision at each decision point. We take the approach of casting the problem of dynamic sensor selection as a *sequential decision problem*. We can then use the method of stochastic dynamic programming to make the computation involved polynomial rather than exponential. This involves a number of steps. The most important of these steps involve defining **cost functions** from one state to another via a sensor decision, **state abstractions**, and **state transitions**. These steps will now be discussed.

Sensor Policies

A *sensor policy* tells an agent what decision to make at each sensor decision point. If the decision is not made until the decision point is reached during execution, we call the policy *dynamic*. Constructing a sensor policy for a planner can be viewed as a *sequential decision problem* (SDP)[1] [Ross, 1983, Boddy, 1991b]. There are a number of characteristics of sequential decision problems which serve to categorize them.

Discrete vs. Continuous: A particular decision is said to be discrete if the effect of a decision is to choose one of a finite, or at least countable set of possibilities. A decision is continuous otherwise. Since we are concerned with choosing among a small set of sensor strategies, we focus on the discrete case.

Deterministic vs. Stochastic: When the outcome of a given decision is certain (i.e., uniquely determined by the choice of this decision), the process is said to be deterministic. The aim of solving deterministic SDPs is to find the decision policy that maximizes the *value* of a sequence of decisions. In our case, maximizing value is equivalent to *minimizing cost*. The problem becomes more complex when the outcome of a given decision is uncertain. For these problems, known as *stochastic* SDPs, each decision is treated as a random variable with an associated probability distribution. This distribution is a function describing the probabilities of various outcomes of a given decision. For stochastic SDPs, the aim becomes maximizing the *expected value* of a sequence of decisions. We focus on the stochastic case since we must evaluate sensor policies based on expected cost.

Formal Statement of the Problem

We have described how sensor policy construction can be viewed as a discrete, stochastic sequential decision problem. Stochastic dynamic programming [Ross, 1983] is a solution methodology for problems of this sort. We now formally describe the components of our problem as a sequential decision problem.

- Let $\mathcal{S}$ be the set of **states** the system can assume. While it may appear this would include all possible world states, much of the information included in a particular state in that space has no direct relevance to an impending sensor decision. Thus, we restrict our attention to a subset of that information, which varies from domain to domain. We also make use of precomputed cost profiles of various types of failures. We will call this reduced world model, a *simplified world model*, (SWM), and a state in it a *simplified world state* (SWS).
- Let $\mathcal{O}$ be the set of **sensor options** to choose from at a given decision point d_i.
- Let $\mathcal{D}$ be the set of **decisions** to be made. For our purposes, this set is more specifically an ordered sequence of sensor decisions $\mathcal{D} = \{d_1, d_2, \ldots, d_n\}$. A solution to the problem will consist of an n-element vector of decision choices.
- Let $\phi : \mathcal{S} \times \mathcal{D} \rightarrow \mathcal{S}$ be a **function mapping from the current state and a decision to the next state.**
- Let $C : \mathcal{S} \times \mathcal{D} \rightarrow \mathcal{R}$ be a **cost function** mapping from the current state and a decision to the real numbers[2]. Conceptually the cost function represents the expected execution cost that a given decision in a given state implies. Of course, the resulting solution is highly dependent on our choice of cost function, which must take into account the probability of success of its decision, and the expected costs resulting from its decision. The cost function can be thought of

[1]Sequential decision problems are a special case of so-called *multi-stage decision problems*[Bellman, 1957].

[2]This is traditionally called the *reward function* in operations research.

as a numerical representation of the relative advantages and disadvantages of one sensing policy over another.

Dynamic Programming

Dynamic programming is a mathematical technique often useful for making a sequence of interrelated decisions [Hillier and Lieberman, 1980]. In contrast to linear programming, there does not exist a standard mathematical formulation of the dynamic programming problem. It is a more general type of approach to problem solving in which the particular equations used must be carefully crafted to fit the problem domain.

All dynamic programming problems, however, do share a number of basic characteristics, which are now briefly summarized and related to the problem of sensor policy construction.

Fundamental Characteristics

1. The problem can be divided into **stages**, with a **policy decision** required at each stage. For sensor policy construction, each stage corresponds to a point at which a single-point sensor policy decision must be made.
2. Each stage has a number of **states** associated with it. These of course are the Simplified World States discussed earlier. They contain information sufficient for making sensor policy decisions.
3. The effect of the policy decision at each stage is to transform the current state into a state associated with the next stage (in this case, according to a probability distribution). The function ϕ provides this transformation.
4. Given the current state, an **optimal policy** for the remaining stages is **independent** of the policy adopted in previous stages. This crucial property means that knowledge of the current state of the system conveys all the information about its previous behavior necessary to make optimal policy decisions from that stage on. It is sometimes referred to as the *principle of optimality* [Bellman, 1957], and is a special case of the *Markovian property*, which states that the conditional probability of any future event, given any past event and the present state is independent of the past event and depends only upon the present state of the process.
5. The solution procedure begins by finding an optimal policy for each state of the **final** stage. This is often trivial. For instance, in the domain we discuss, this is simply the time slice before the scheduling period ends.
6. A **recursive function** is available which identifies an optimal policy for each state at stage i, given an optimal policy for each state at stage $i+1$. For our purposes, finding an optimal (one step) decision from each state in stage i involves computing the least costly path from each state in stage i to the relevant states in stage $i+1$, and then finding which decision at each state minimizes overall cost to a goal state. The overall cost can be computed by adding each one-step cost to the accumulated (backed up) cost from the relevant successor state to the goal. All other $(state, decision)$ pairs can be discarded if the principle of optimality holds. This is where a great deal of computation and storage is saved by using dynamic programming. The function that identifies an optimal policy for each state at stage i is traditionally expressed as:

$$f_i^*(s) = min\{c_{sx_i} + f_{i+1}^*(x_i)\} \qquad (1)$$

where

x_i is the choice of action made at decision point i.

c_{sx_i} is the cost of the transition from state s at stage i to a state in stage $i+1$ by choosing action x_i.

An optimal policy then, consists of finding the minimizing value of x_i.

7. Using this recursive relationship, the solution procedure moves backward stage by stage, each time determining an optimal decision for each state at that stage which optimizes the sequence of decisions from that state forward. This continues until it finds an optimal decision at the initial stage. Note that intuitively this entails caching optimal policies corresponding to longer and longer sequences of decisions, in this case, sensor policies.

   While it is possible to proceed from the first to final stage (forward) for some dynamic programming problems, it is generally not possible when stages correspond to time periods. This is because possibly optimal sequences of decisions might be discarded if they look more costly early on when some commitments must be made about optimal subsequences of decisions. However, this is never a problem when working backward from the final to initial stage as long as the principle of optimality holds, since an optimal policy to each state in the current stage, while not yet determined, can be assumed optimal.

Stochastic Sensor Selection

Figure 1 contains the algorithm for constructing stochastic sensor policies. The algorithm is fairly straightforward. The outer loop moves moves the process backward from the final stage to the initial stage. All possible states at each stage are then generated and stored in the list S_i. From each of these stages, we must compute the expected one-step cost of making each possible decision, add these one-step costs to the accumulated cost for each resulting state, and cache the decision which yields the minimum value.

In most domains, the run-time behavior of the sensors (i.e., whether they actually succeed or fail) will greatly influence the overall cost of executing a plan containing

```
Procedure $Stochastic(States, Stages, SensorOptions)$

For $i \leftarrow Stages$ downto 1
    begin
    $S_i \leftarrow PossibleStates(i)$
    For $s$ in $S_i$
        begin
        $C_{min} \leftarrow 0$
        For $d$ in $SensorOptions$
            begin
                $StateProbPairs \leftarrow ProjectStoch(s, d)$
                $C\prime \leftarrow 0$
                For $Pair$ in $StateProbPairs$
                    begin
                    $P\prime \leftarrow prob(Pair)$
                    $S\prime \leftarrow state(Pair)$
                    $C\prime \leftarrow C\prime + P\prime \cdot CostSoFar(S\prime)$
                    end
                If $C\prime < C_{min}$
                  begin
                        $C_{min} \leftarrow C\prime$ ; Best so far?
                        $d_{min} \leftarrow d$   ; Optimal decision
                  end
            end
        $CostSoFar(s) \leftarrow C_{min}$
        Cache $d_{min}$ as optimal decision at $(i, s)$
        end
    end

;;; Make Optimal Sensor Decisions as Execution Proceeds

$Path_{min} \leftarrow NIL$
$CurState \leftarrow s_{init}$

For $i \leftarrow 1$ to $Stages$
    begin
    $d \leftarrow GetCachedDecision(i, CurState)$
    $Path_{min} \leftarrow Append(Path_{min}, d)$
    $CurState \leftarrow Execute(CurState, d)$
    end

Return($Path_{min}$)
```

Figure 1: Stochastic Sensor Scheduling

sensor operations. In this case, the best we can do is minimize expected execution cost, and model this cost probabilistically. For this reason, the projection function must return not a single state, but a probability distribution over the set of possible states that could result. (We have named this function *ProjectStoch* to make the distinction clear.) Each member of this set of (*Probability, State*) pairs is then evaluated to determine the contribution each possible result will have on making an optimal decision.

Finally, the optimal decisions at each level are cached and used at execution time to make the proper context-dependent, online sensor decisions.

The Printer Diagnostics Domain

We now demonstrate our method with a fairly simple example. Imagine the following domain, which we will call the *Printer Diagnostics Domain* (PDD): A large software house has a row of n laser printers for use by its programmers and staff. These printers periodically break down, but when these printers will require service is completely random and unpredictable. A computer program polls each of the n printers every 15 minutes and reports whether or not a printer has changed status from operational to down. To make the simulations more understandable, we will assume only one printer failure can be detected per 15-minute time slice (although a second failure could be noticed in the next time slice). The total number of down printers is limited only by the total number of printers (n).

The first step in determining the proper course of action to repair a broken printer involves diagnosing the problem. This can be viewed as a context-dependent sensor selection process, where choosing whether and which diagnostic tests need to be performed is equivalent to choosing whether and which sensors need to be fired.

The task of the dynamic programming algorithm is to determine an optimal diagnostic to run (if any) to minimize the amount of wages to pay service personnel while maximizing the amount of printer-hours available for use by the programmers and staff. We will discuss the relevant cost functions involved shortly.

First however, we wish to point out that the PDD differs from other pure (although uncertain) planning domains we have previously studied in several important respects.

Sources of Uncertainty

In the PDD, there are two types of uncertainty:

1. **Environmental Uncertainty**, which consists of the agent not knowing for a given time slice whether a printer will go down or not. In the PDD the agent has no expectations to go on (i.e., no default values) because printers are assumed to go down in a purely random fashion. The best the agent can do is decide on the best diagnostic to use (if any), given the context when a printer does fail.

2. **Sensor Uncertainty**, which consists of not knowing if a certain diagnostic test (sensor) will be sufficient to isolate the printer's problem and facilitate a repair. This uncertainty is a probabilistic function of the diagnostic test chosen.

Diagnostics as Sensors

In the PDD, at each decision point a diagnostic test is chosen, which takes some amount of time. Let $\mathcal{O}$ be the set of **sensor options** to choose from at a given decision point d_i. For the PDD, this is the set $\mathcal{O} = \{NA, S3, S6, S9\}$, which stand for no action, or one of the three actual diagnostic tests. In each of the problems, diagnostic S3 has a duration of 3 time units, S6 lasts 6 time units, and S9 lasts 9 time units. Of course NA has a duration of 0 time units. It is also the case that the reliabilities of the diagnostics are better for longer diagnostics, as one would expect.

When a diagnostic action is administered, the correct repair action is either determined (a positive test) or it is still undetermined (a negative test). A positive test corresponds to a correct sensor reading while a negative test corresponds to a failed sensor reading after which we may want to acquire the requisite information in some other way. This is accomplished by determining the context in the current time slice, and looking up the optimal sensor option for this context in the precomputed lookup structure.

Leaving Some Environmental Information Unknown

In some domains, the agent must eventually acquire all unknown environmental information in order to achieve its goals. This is not the case in the PDD. If it is more cost-effective to let one or more of the n printers stay down for the rest of the monitored time period rather than diagnose it, then the relevant unknown information might never be acquired.

Variable-Duration Diagnostics

Since some diagnostics can take a long time, much can happen in the environment while a single test is being run. Other printers can break down, workers can leave work for the day, and so on. For this reason, following a prescribed sequence of sensor strategies is not very sensitive to the context in which they may eventually be employed. That feature is what makes this a good domain for *dynamic sensor policies*, in which we can make local context-dependent choices about sensors which optimize some global parameter.

Factors Influencing Sensor Choice

A number of factors influence optimal sensor choices. These factors can be roughly divided between state variables and domain features.

State Variables

We have identified four state variables, which when considered together, make up the Simplified World State representation for the PDD.

S:Stage The current stage number. Each stage represents the end of a 15-minute time slice.

J:Jobs The number of diagnostic jobs currently in progress. This number has an upper bound which reflects the service personnel resources available. For our simulations we have set this upper bound (max jobs) to be 5.

L:Load The total number of 15-minute time slices currently allotted for the J jobs. Since diagnostics have differing durations, the load falls in the range

$$J \leq L \leq max - duration \cdot J$$

for a given value of J. $Max - duration$ is the duration of the most expensive diagnostic. In our simulations, the most expensive diagnostic, S9, has a duration of 9.

D:Down The number of printers down which have not been scheduled for diagnostics. This is limited only to the number of printers that exist.

Domain Features

Duration: The expected time needed to run each diagnostic test. This number is rounded off to the nearest quarter hour.

Probability of Success ($P(D)$): The probability that running this diagnostic test will fully determine the problem. This is represented as a real number on the inclusive range [0.0 1.0] and is rounded to the nearest tenth.

Gain to Cost Ratio (γ): In order to compare costs due to running diagnostics (measured in time slices of work scheduled) and the gain derived from having a down printer back up for some amount of time (measured in time slices of operationality), we express the relative worth of these two quantities in the *gain-to-cost ratio*. This parameter is used in the cost function C, discussed shortly.

Cost Multiplier ($\mu(stage)$): We assume diagnostics are cheaper to run during some times of the day than during others. The idea here is that service personnel are paid their normal wage during normal business hours, time and a half from the end of the working day until midnight, and double time in the early morning hours. Thus, the *cost-multiplier* function multiplies the base cost by the appropriate constant for the periods to be spanned by the scheduled diagnostic test. Our algorithm automatically splits the stages into three roughly equal periods for which each of 2.0, 1.0, and 1.5 are appropriate multiplier constants.

Domain Predicate
(TIME 8:45) (PR-1 UP)
(PR-2 DOWN)
(PR-3 UP)
(PR-4 DOWN)
(PR-5 DOWN)
(UNDER-REPAIR PR-3)
(UNDER-REPAIR PR-5)
(TIME-REMAINING PR-3 4)
(TIME-REMAINING PR-5 1)
(UNSCHEDULED PR-2)

Table 1: Snapshot of 5-Printer Problem World State.

Domain Factor	Value
Stage (s)	35
Jobs in progress (j)	2
Load = total resources scheduled (l)	5
Down (printers down & unscheduled) (l)	1

Table 2: Snapshot of 5-Printer Problem Simpified World State.

Importance Coefficient($\iota(d)$): The degree to which diagnosing remaining printers that go down becomes important increases with the number of printers already down. We define the *importance coefficient* (ι) as follows:

$$\iota(d) = \begin{cases} 1 & \text{if } d \leq 1 \\ \frac{1}{1-\frac{d-1}{d_{max}}} & \text{otherwise} \end{cases}$$

Number of Stages (n): Clearly, the number of stages has a bearing on which diagnostics make sense. We enforce the constraint that all scheduled diagnostics be completed by the end of the last time slice, so a diagnostic with a duration of 9 time slices is really never appropriate for a problem of only 10 stages, for instance.

The Simplified World Model: Defining ϕ

Since the function $\phi : \mathcal{S} \times \mathcal{D} \rightarrow \mathcal{S}$ provides a mapping from a current SWS to the next SWS, we must define precisely what should be represented in a SWS, and how ϕ transforms one SWS and a decision into another. Figures 1 and 2 provide an example of the difference between a predicate-based planner state and an abstracted SWS for the PDD.

A given SWS should contain any information relevant to making the next sequential decision d_i. Ideally, it should reflect the agent's expectations of important factors of future events. Each of the four state variables which together constitute an SWS for the PDD is described below.

S:Stage The stage number is incremented by one.

J:Jobs Jobs which have just been completed in the current time slice must be subtracted. At most one job can be added in this time slice. No jobs may be added if all resources are being used or if no unattended jobs exist.

L:Load The load is decreased by the number of current jobs in each time slice. The load is increased by the duration of a newly scheduled job.

D:Down The number of down machines is increased by one if a random printer fails and the diagnostic NA (no action) is chosen. It is decreased when a diagnostic is scheduled for a pending (down) job.

The Cost Function: Defining C

The general form of the incremental cost function C_{total} describing the expected cost for making decision D in state S_i is simply:

$$C_{total}(S_i, D) = Cost(S_i, D) - Gain(S_i, D) \qquad (2)$$

$$Cost(S_i, D) = \sum_{k=i}^{i+\delta(D)} \mu(k) \qquad (3)$$

$$Gain(S_i, D) = \gamma \cdot \iota(d) \cdot (P(D)(n - i - \delta(D))) \qquad (4)$$

where

i	=	the current stage.
n	=	the number of stages.
S_i	=	the current state.
D	=	the current decision.
$\delta(D)$	=	the duration of the decision D.
$\mu(k)$	=	the cost multiplier for stage k.
$\iota(d)$	=	the importance coefficient for d down machines.
γ	=	the gain-to-cost ratio constant.
$P(D)$	=	the probability that decision D will work.

Results

The method we have described has been fully implemented and applied to several domains, including the PDD. Although space considerations preclude inclusion of experimental results and complexity analyses here, representative examples of simulations of the diagnostic scheduler operating in the PDD can be found in [Krebsbach, 1993]. Reliabilities of the four diagnostic tests are varied, as is the probability of printer failure at each point. Finally, a demonstration and discussion of the time complexity advantage gained with this method is presented.

Conclusion

We have shown that a methodology developed originally to select sensors offline for a planning and executing agent is also appropriate for agents which must choose among sensing alternatives in an adaptive, context-dependent manner at execution time. The chief idea behind this dynamic method is that optimal decisions are cached offline with abstractions of actual world states encountered acting as indexes into this structure during execution. Since the optimal decisions are computed and cached using stochastic dynamic programming, this method can be shown to be polynomial in both space and time.

References

[Abramson, 1991] Bruce Abramson. An analysis of error recovery and sensory integration for dynamic planners. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 744–749, 1991.

[Bellman, 1957] Richard Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, 1957.

[Boddy, 1991a] Mark Boddy. Anytime problem solving using dynamic programming. In *Proceedings of AAAI-91*, Boston, MA, 1991.

[Boddy, 1991b] Mark Boddy. Solving time-dependent problems: A decision-theoretic approach to planning in dynamic environments. Tech Report CS-91-06, Brown University, Department of Computer Science, 1991.

[Chrisman and Simmons, 1991] Lonnie Chrisman and Reid Simmons. Sensible planning: Focusing perceptual attention. In *Proceedings of AAAI-91*, Los Angeles, CA, 1991.

[Goodwin and Simmons, 1992] Richard Goodwin and Reid Simmons. Rational handling of multiple goals for mobile robots. In *Proceedings of the First International Conference on AI Planning Systems*, pages 70–77, College Park, Maryland, June 1992.

[Hager and Mintz, 1991] G. Hager and M. Mintz. Computational methods for task-directed sensor data fusion and sensor planning. *International Journal of Robotics Research*, 10:285–313, 1991.

[Hillier and Lieberman, 1980] Frederick S. Hillier and Gerald J. Lieberman. *Introduction to Operations Research.* Holden-Day, Inc., San Francisco, CA, 3rd edition, 1980.

[Krebsbach *et al.*, 1992] Kurt Krebsbach, Duane Olawsky, and Maria Gini. Sensing and deferral in planning: Empirical results. In *Proceedings of the First International Conference on AI Planning Systems*, College Park, Maryland, June 1992.

[Krebsbach, 1993] Kurt Krebsbach. Rational sensing for an AI planner: A cost-based approach. Ph.D. dissertation, University of Minnesota, 1993.

[Olawsky *et al.*, 1993] Duane Olawsky, Kurt Krebsbach, and Maria Gini. An analysis of sensor-based task planning. Technical Report TR 93-43, University of Minnesota Department of Computer Science, Minneapolis, MN, 1993.

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