Cost-Effective Sensing During Plan Execution

Eric A. Hansen

Department of Computer Science
University of Massachusetts
Amherst, MA 01003
hansen@cs.umass.edu

Abstract

Between sensing the world after every action (as in a reactive plan) and not sensing at all (as in an open-loop plan), lies a continuum of strategies for sensing during plan execution. If sensing incurs a cost (in time or resources), the most cost-effective strategy is likely to fall somewhere between these two extremes. Yet most work on plan execution assumes one or the other. In this paper, an efficient, anytime planner is described that controls the rate of sensing during plan execution. The sensing interval is determined by the state during plan execution, as well as by the cost of sensing, so that an agent can sense more often when necessary. The planner is based on a generalization of stochastic dynamic programming.

Introduction

The characteristic assumptions of classical planning — a complete and certain action model and deterministic plan execution — make sensing during plan execution unnecessary. A planner that knows the initial state of the world and can predict the effects of its actions with certainty has no reason for sensing. Hence, classical planners constructed open-loop plans.

The reactive approach pursued in recent planning research was developed for environments in which the assumptions made by classical planners are unjustified, in other words, for most realistic environments. In the real world, actions may not have their intended effects and the environment can change in unexpected ways. Because a planner cannot project the course of plan execution with certainty, the reactive approach is to construct a plan that specifies what action to take in each of many different states, and to sense the world after each action so that the agent can choose the next action based on the current state.

The question that is the starting point for this paper is whether it is cost-effective to sense the world after every action in a plan. There is often a cost, in time or resources, for acquiring and processing sensory information. If sensing is expensive, it may be better to sense less frequently, especially if there is not that much uncertainty about the state of the world during plan execution. Between sensing the world after every action (as in a reactive plan) and not sensing at all (as in an open-loop plan), lies a continuum of sensing strategies. The most cost-effective one is likely to fall somewhere between these two extremes.

Although the problem of sensing costs has been relatively neglected in planning research, two different approaches to it have been tentatively explored. In one, cost-effectiveness is achieved by sensing only a subset of the features of the environment (Chrisman & Simmons 1991; Tan 1991). This raises interesting issues, among them the problem of identifying a state from incomplete perception (Whitehead & Ballard 1991), but it still assumes an agent senses its environment after each action. The second approach allows an agent to sense at wider intervals than after every action, where the sensing interval is set based on factors such as the cost of sensing and the cost and likelihood of error. But in this approach, the simplifying assumption is usually made that the rate of sensing should be constant throughout plan execution. For example, Abramson (1993) gives a decision-theoretic analysis to show that a fixed sensing interval during plan execution is optimal, and a formula for calculating what the interval should be. Hendler and Kinny (1992) apply a similar analysis to TileWorld, and Langley, Iba, and Shrager (1994) also assume periodic sensing. All of these analyses are based on a model of plan execution in which errors occur, in Abramson's description, "spontaneously"; that is, they are no more likely in one state of the world than another, nor any more likely after one action than another. From the assumption that errors are spontaneous, it follows that a fixed rate of sensing is optimal. But in many realistic environments, some states are riskier than others and some actions more error-prone. As a result, the success of some parts of a plan will be less predictable than other parts, and this suggests that the frequency of sensing

This work was supported by ARPA/Rome Laboratory under contract #F30602-91-C-0076 and under an Augmentation Award for Science and Engineering Research Training.

should change depending on which part of a plan is being executed.

When the likelihood of plan error depends on the current state or on the action taken in that state, sensing becomes a planning problem; that is, decisions about when to sense the environment must take into account the actions in the plan and the projected state during plan execution. In order to treat the problem of when to sense during plan execution as a planning problem, this paper adopts the framework of Markov decision theory. Besides providing mathematical rigor, this framework has proven useful for formalizing planning problems in stochastic domains (Dean et al 1993; Koenig 1992), as well as for relating planning to reinforcement learning (Sutton 1990; Barto, Bradtke, & Singh 1993). In a conventional Markov decision problem, a policy (or plan) is executed by automatically sensing the world after each action, without considering the cost this might incur. Because this is exactly the assumption being questioned in this paper, we begin by showing how to incorporate sensing costs and formulate sensing strategies in the framework of Markov decision theory. Then an efficient, anytime planner is described that can adjust the rate of sensing during plan execution, depending on the state and actions of a plan. The approach developed here is directly applicable to work on planning for stochastic domains using techniques based on dynamic programming, and is suggestive for work on integrating sensing with plan execution in general.

Including Sensing Costs in a Markov Decision Problem

A discrete-time, finite state and action Markov decision problem is described by the following elements. Let S be a finite set of states, and let A be a finite set of actions. A *state transition function* $P: S \times A \times S \rightarrow [0,1]$ specifies the outcomes of actions as discrete probability distributions over the state-space. In particular, $P_{xy}(a)$ gives the probability that action a taken in state x produces state y. A *payoff function* $R: S \times A \rightarrow \Re$ specifies rewards (or costs) to be maximized (or minimized) by a plan. In particular, $R(x,a)$ gives the expected single-step payoff for taking action a in state x. A *policy* $\pi: S \rightarrow A$ specifies an action to take in each state. The set of all possible policies is called the *policy space*. To weigh the merit of different policies, a *value function* $V_\pi: S \rightarrow \Re$ gives the expected cumulative value received for executing a policy, π, starting from each state. The value function for a policy satisfies the following system of simultaneous linear equations:

$$V_\pi(x) = R(x, \pi(x)) + \lambda \sum_{y \in S} P_{xy}(\pi(x)) V_\pi(y), \qquad (1)$$

where $0 \le \lambda < 1$ is a *discount factor* that gives higher importance to payoffs in the near future. A policy π^* is optimal if $V_{\pi^*}(x) \ge V_\pi(x)$ for all states x and policies π. *Dynamic programming* is an efficient way of searching the space of possible policies for an optimal policy, using the value function to guide the search. There are several versions of dynamic programming and the ideas in this paper can be adapted to any of them, but Howard's policy iteration algorithm is used as an example (Howard 1960). It consists of the following steps:

1. *Initialization*: Start with an arbitrary policy $\pi: S \rightarrow A$.
2. *Policy evaluation*: Compute the value function for π by solving the system of simultaneous linear equations given by equation (1).
3. *Policy improvement:* Compute a new policy π' from the value function computed in step 2 by finding, for each state x, the action a that maximizes expected value, or formally:

$$\forall x: \quad \pi'(x) = \arg\max_{a \in A} \left[R(x,a) + \lambda \sum_{y \in S} P_{xy}(a) V_\pi(y) \right].$$

4. *Convergence test*: If the new policy is the same as the old one, it is optimal and the algorithm stops. Otherwise, set $\pi := \pi'$ and go to step 2.

The policy evaluation step requires solving a system of simultaneous linear equations and so has complexity $O(n^3)$ using a conventional algorithm such as Gaussian elimination, and at best $O(n^{2.8})$, where n is the size of the state set. The complexity of the policy improvement step is $O(mn^2)$, where m is the size of the action set. The policy iteration algorithm is guaranteed to improve the policy each iteration and to converge to an optimal policy after a finite number of iterations. It is also an anytime algorithm that can be stopped before convergence to return a policy that is monotonically better than the initial policy as a function of computation time.

In the conventional theory of Markov decision problems described so far, each action takes a single time-step and the world is automatically sensed at each step to see what action to take next. We want to find some way to represent problems in which an arbitrary sequence of actions can be taken before sensing. To do so, we define "multi-step" versions of each of the elements of a Markov decision problem with the exception of the state space.

A *multi-step action set* $\overline{A}$ includes all possible sequences of actions up to some arbitrary limit z on their length, where the arbitrary limit keeps the action set finite. The bar over the A is used to indicate "multi-step." Note that the multi-step action set includes both primitive actions, which take a single time step, and sequences of primitive actions. A tuple $\langle a_1 .. a_k \rangle$ represents a k-length sequence of actions, where $1 \le k \le z$. A *multi-step transition function* $\overline{P}: S \times \overline{A} \times S \rightarrow [0,1]$ gives the state transition probabilities for the multi-step action set. The

multi-step transition probabilities for an action sequence $\langle a_1..a_k \rangle$ can be computed by multiplying the matrices that contain the single-step transition probabilities for the actions in the sequence, or formally, $\overline{P}(\langle a_1..a_k \rangle) = P(a_1)...P(a_k)$, where the matrix $P(a)$ contains the single-step transition probabilities for action a. In particular, $\overline{P}_{xy}(\langle a_1..a_k \rangle)$ gives the probability that taking the sequence of actions $\langle a_1..a_k \rangle$ starting from state x results in state y. A *multi-step payoff function* gives the expected "immediate" payoff received in the course of a sequence of actions. It is the sum of the expected payoffs received each time step during execution of the action sequence, with a cost for sensing, C, incurred at the end (when the world is sensed again),[1] and is defined formally as follows:

$$\overline{R}(x, \langle a_1..a_k \rangle) = R(x, a_1) - \lambda^k C + \sum_{j=1}^{k} \lambda^j \sum_{y \in S} \overline{P}_{xy}(\langle a_1..a_j \rangle) R(y, a_j).$$

Note that the time step becomes the exponent of the discount factor to ensure discounting is done in a time-dependent way. The equation also shows that the cost of sensing can be controlled by varying the length of action sequences. A *multi-step policy* $\overline{\pi}: S \rightarrow \overline{A}$ maps each state to a sequence of actions to be taken before sensing again. The *multi-step value function* for $\overline{\pi}$ satisfies the following system of simultaneous linear equations,

$$\overline{V}_{\overline{\pi}}(x) = \overline{R}(x, \overline{\pi}(x)) + \lambda^{length(\overline{\pi}(x))} \sum_{y \in S} \overline{P}_{xy}(\overline{\pi}(x)) \overline{V}_{\overline{\pi}}(y), \quad (2)$$

where the length of an action sequence is again the exponent of the discount factor to ensure time-dependent discounting.

With multi-step versions of the action set, state transition function, payoff function, policy, and value function, we have a well-defined Markov decision problem for which policy iteration can find an optimal policy for interleaving acting and sensing. The multi-step version of the policy iteration algorithm is as follows:

1. *Initialization*: Start with an arbitrary policy $\overline{\pi}: S \rightarrow \overline{A}$.
2. *Policy evaluation:* Compute the value function for $\overline{\pi}$ by solving the system of simultaneous linear equations given by equation (2).
3. *Policy improvement:* Compute a new policy, $\overline{\pi}'$, from the value function computed in step 2 by finding, for each state x, the sequence of actions that maximizes expected value, or formally:

$$\forall x: \quad \overline{\pi}'(x) = \arg\max_{\langle a_1..a_k \rangle \in \overline{A}} \Big[\overline{R}(x, \langle a_1..a_k \rangle) + \lambda^k \sum_{y \in S} \overline{P}_{xy}(\langle a_1..a_k \rangle) \overline{V}_{\overline{\pi}}(y) \Big].$$

4. *Convergence test*: If the new policy is the same as the old one, it is optimal and the algorithm stops. Otherwise, set $\overline{\pi} := \overline{\pi}'$ and go to step 2.

The policy evaluation step still requires solving a system of n simultaneous linear equations and so has the same complexity as for a single-step Markov decision problem. However, the complexity of the policy improvement step has been dramatically increased. Naively performing the policy improvement step now requires evaluating, for each state, every possible sequence of actions up to the fixed limit z on the length of a sequence. Even assuming that all multi-step transition probabilities and payoffs are pre-computed,[2] the complexity of the policy improvement step would be $O(m^z n^2)$ where the factor m^z comes from the fact that there are approximately m^z different action sequences of length at least 1 and no greater than z, where m is the number of primitive actions. In other words, allowing a sequence of actions to be taken before sensing causes an exponential blowup in the size of the policy space and an apparently prohibitive increase in the complexity of the dynamic programming algorithm.

An Efficient Planning Algorithm

Although searching the space of all possible sequences of actions exhaustively for each state is prohibitive, a practical approach to the problem of cost-effective sensing is still possible within this formal framework. Many action sequences are implausible or counterproductive, either because actions lead away from the goal or because they do not make sense in a given state. By using the value function to estimate the relative merit of different sequences, the space of possible action sequences can be searched intelligently in order to find a good, if not optimal, multi-step policy.

The multi-step action set is first organized as a search tree in which each node corresponds to an action sequence, and the successors of a node are created by adding a single action to the end of the sequence. The root of the search tree corresponds to the null action (which we do not consider part of the action set), level 1 of the tree contains

[1]For simplicity, in this paper we assume that sensing has a fixed cost. It is a straightforward generalization to make the cost of sensing a function of state and/or action.

[2]The combinatorial explosion of the number of possible action sequences usually makes it prohibitive to precompute and store all multi-step transition probabilities and payoffs. Fortunately for the efficient algorithm described in the next section, the vast majority are never needed and those that are can be cached to avoid repeated recomputation.

all sequences of length 1, level 2 contains all sequences of length 2, and so on. An action sequence with a good expected value can be found efficiently by starting from the root and choosing a greedy path through the tree. At each node, the expected values of the successor nodes are computed and the path extended to the node with the highest expected value, as long as it is higher than the expected value of the current node. If no successor node is an improvement, the search is stopped. This corresponds to constructing an action sequence by starting with the null sequence and adding actions to the end of the sequence one at a time, using the following two heuristics:

greedy heuristic: Always extend a sequence of actions with the action that gives the resulting sequence the highest expected value.

stopping heuristic: Stop as soon as extending a sequence of actions with the best next action reduces the expected value of the resulting action sequence.

The stopping heuristic simply means to assume that if it does not pay to take one more action before sensing, it will not pay to take two or more actions before sensing either — a very reasonable heuristic. Note that the stopping heuristic makes it possible to stop the search before the arbitrary, fixed limit z on the length of an action sequence is reached.

In addition to these two search heuristics, the greedy search algorithm needs one more feature. A crucial property of policy iteration is that it guarantees monotonic improvement of the policy each iteration; in particular, it guarantees that the value of each state is equal to or better than the value for the same state on the previous iteration. This property, together with a finite policy space, guarantees that policy iteration converges in a finite number of iterations. However, because the efficient search algorithm searches the space of possible action sequences greedily, local maxima may lead it to an action sequence that has a lower value than the action sequence found on the previous iteration. To prevent the value of any state from decreasing from one iteration to the next, the following rule is invoked at the end of the greedy search:

monotonic improvement rule: If the value of the action sequence found by greedy search is not better than the value of the action sequence specified by the current policy, do not change the current action sequence.

This rule ensures that a state's value is always equal to or greater than its value on a previous iteration because if an improved action sequence is not found, the previous action sequence has a value at least as good.

The greedy search algorithm for the policy improvement step is summarized as follows:

For each state x:

1. Initialize $\langle sequence \rangle$ to the single action with the highest expected value.
2. Compute the expected value of each possible extension of $\langle sequence \rangle$ by a single action.
3. *Stopping heuristic*: If no extension has a higher expected value than the current sequence, stop the search and go to step 5.
4. *Greedy heuristic*: If at least one extension has an improved value, set $\langle sequence \rangle$ to the extension with the highest value and go to step 2.
5. *Monotonic improvement rule*: If the sequence found by greedy search has a higher expected value than the sequence specified by the current policy, change the policy for this state to $\langle sequence \rangle$, otherwise, leave the policy for this state unchanged.

From the monotonic improvement rule and the finiteness of the policy space, the multi-step version of policy iteration using the greedy search algorithm is guaranteed to converge after a finite number of iterations. However, it is impossible to say whether it will converge to a globally optimal policy or to one that is a local optimum. Nevertheless, the opportunity to vary the sensing interval during plan execution, based both on the state and the cost of sensing, makes it possible for the multi-step algorithm to converge to a policy that is better – often considerably better – than a policy that automatically senses each time step. Moveover, the greedy search algorithm can find a good multi-step policy efficiently, despite the exponential explosion of the policy space.

When the greedy search algorithm is used in the policy improvement step, the number of action sequences that must be evaluated for each state is reduced from m^z to km, where again, m is the number of different actions, z is the maximum length of an action sequence, and $k \leq z$ is the average length of an action sequence. In other words, the complexity of searching for an improved action sequence for a state is reduced from exponential to linear. If the multi-step transition probabilities and payoffs were already available, the average time complexity of the policy improvement step would be $O(kmn^2)$, only a constant factor greater than for the single-step algorithm. In most cases, however, the multi-step transition probabilities and payoffs are not given at the outset and must be computed from the single-step transition probabilities and payoffs. If they are computed on the fly each time they are needed, the time complexity of the policy improvement step is $O(kmn^3)$. This can be alleviated somewhat by saving or caching the multi-step transition probabilities and payoffs the first time they are computed. Because most regions of the search space are never explored, most multi-step transition probabilities and payoffs do not need to be

computed. Those that do tend to be re-used from iteration to iteration, making caching very beneficial. With caching, the time complexity of the policy improvement step becomes $O(pkmn^2)+O((1-p)kmn^3)$, where p is the proportion of cache hits on a particular iteration. Because the proportion of cache hits tends to increase from one iteration to the next, the policy improvement step runs faster as the algorithm gets closer to convergence.

Generating the multi-step transition probabilities and payoffs is by far the most computationally burdensome aspect of the multi-step algorithm. Nevertheless, because the time complexity of the policy evaluation step is already $O(n^3)$, or at best $O(n^{2.8})$, the overall asymptotic time complexity of a single iteration of the multi-step algorithm, consisting of both policy evaluation and improvement, is still about the same or only a little worse in order notation, although the constant is greater for the multi-step algorithm. This is important because it means that scaling up dynamic programming to problems with large state spaces, the characteristic weakness of dynamic programming, is not made appreciably more difficult by the multi-step algorithm.

Of course, it takes longer for the multi-step version of policy iteration to converge than for the single-step version. However, both are iterative (anytime) algorithms that can have a good policy ready before convergence. Moreover, it is easy to ensure that multi-step policy iteration always has a policy as good or better than the single-step algorithm in the same amount of computation time. Simply run the single-step algorithm until convergence, then continue from that point with the multi-step algorithm. This approach is used in the example described in the next section. In most cases, computing the optimal single-step policy before beginning the multi-step algorithm leads to faster initial improvement of plan quality because it postpones the expense of computing multi-step probabilities and payoffs. For much the same reason, gradually adjusting upward the maximum allowable length of an action sequence from one iteration of the multi-step algorithm to the next can accelerate convergence.

A Path-Planning Example

The multi-step policy iteration algorithm can be used for any planning problem that is formalizable as a Markov decision problem. The following path-planning example provides a very simple illustration. Imagine a robot in the simple grid world shown in figure 1. Each cell in the grid represents a state. The actions the robot can take to move about the grid are {North, South, East, West}, but their effects are unpredictable. Whenever the robot attempts to move in a particular direction, it has 0.8 probability of success. However, with 0.05 probability it moves in a direction that is 90 degrees off to one side of its intended direction, with 0.05 probability it moves in a direction that is 90 degrees off to the other side, and with 0.1 probability it does not move at all. For example, if it attempts to move north it is successful with 0.8 probability, but it moves east with 0.05 probability, moves west with 0.05 probability, and does not move at all with 0.1 probability. If the robot hits a wall, it stays in the same cell. In this problem, the robot cannot be sure where it has moved without sensing, and becomes less sure about where it is the more it moves without sensing. Sensing its current state has a cost of 1. Hitting a wall has a cost of 5. Therefore, this is a cost-minimization problem in which the robot must balance the cost of sensing against the value of knowing where it is as it tries to find its way to the goal. Costs stop accumulating and the problem ends when the robot senses that its current state is the goal state. The discount factor used is 0.99999.

Figure 2 shows the anytime improvement of plan quality with multi-step policy iteration, beginning from the point at which the conventional, single-step algorithm converges. The multi-step algorithm converges to a policy that performs better than the single-step policy by almost a factor of two. The numbers in the grid cells of figure 3 give the sensing interval for each state for the multi-step policy at convergence. Due to lack of space, only one of the action sequences is shown. The others do what one would expect: move the robot towards the goal. In the action sequence shown, the robot moves away from the wall first, to avoid accidentally hitting it, before moving through the middle of the room towards the goal.

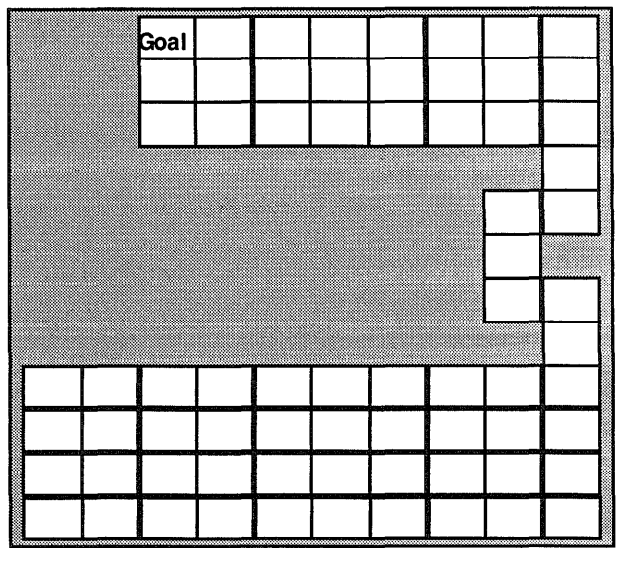

Figure 1: In this simple grid world, the robot tries to minimize the combined cost of sensing and hitting walls on its way to the goal.

Note that in the west side of the large room on the bottom of the grid world, the robot senses less frequently because there is less danger of hitting a wall, and then it senses more frequently as it approaches the east wall where it must turn into the corridor. The sharp corners of the narrow corridor cause the robot to sense nearly every time step. When it enters the room with the goal, it senses less frequently at first, given the wider space, then more frequently as it approaches the goal, to make sure it enters the goal state successfully. This sensing strategy is an intuitive one. A person would do much the same if he or she tried to get to the goal blindfolded, and was charged for stopping and removing the blindfold to look about.

Conclusion

This paper describes a generalization of Markov decision theory and dynamic programming in which sensing costs can be included in order to plan cost-effective strategies for sensing during plan execution. Within this framework, an efficient search algorithm has been developed to deal with the combinatorial explosion of the policy space that results from allowing arbitrary sequences of actions before sensing. The result is an anytime planner that can adjust the rate of sensing during plan execution in a cost-effective way.

There are some limitations to this approach. First, it applies specifically to planners that are based on dynamic programming techniques, although it is suggestive for the problem of interleaving sensing and acting in general. Second, it assumes a simplistic sensing model in which sensing is a single operation that acquires a perfect snapshot of the world. Many real-world problems involve sensor uncertainty and multiple sensors that each return a fraction of the information available. Nevertheless, the simple sensing model assumed here is often useful and is assumed by a number of planners and reactive controllers. In contexts in which this sensing model is appropriate, the anytime planner described here provides an efficient method for dealing with sensing costs.

Finally, the formal framework within which this approach is developed makes it possible to confirm a very natural intuition. The rate of sensing during plan execution should depend on the cost of sensing and on the projected state during plan execution. This makes it possible for an agent to sense less frequently in "safe" parts of a plan, and more frequently when necessary, as the path-planning example illustrates.

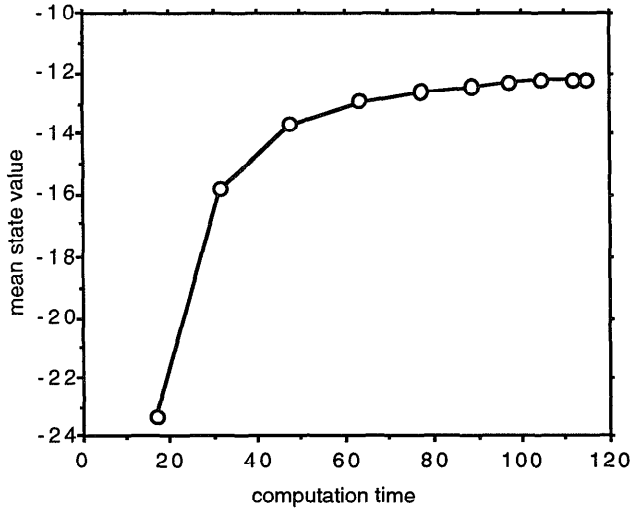

Figure 2: Anytime improvement in plan quality with multi-step policy iteration for the grid world example of figure 1. The single-step algorithm is started at time zero and the first circle marks the point at which it converges and the multi-step algorithm begins. Each circle after that represents a new iteration of the multi-step algorithm. The iterations are closer together near convergence because of previous caching of multi-step transition probabilities and payoffs. Computation time is in cpu seconds.

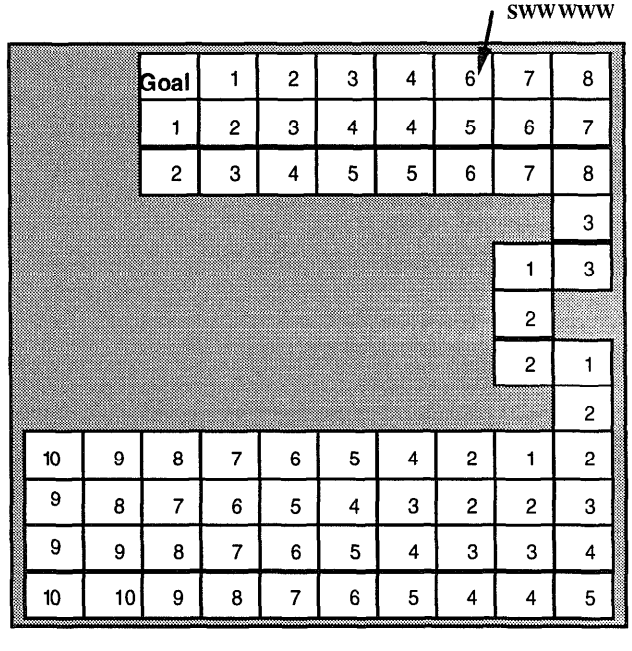

Figure 3: Multi-step policy at convergence. Each cell (or state) contains the length of the action sequence to take from that state before sensing again. One of the action sequences is shown as an example.

Acknowledgements

This research is supported by ARPA/Rome Laboratory under contract #F30602-91-C-0076 and under an Augmentation Award for Science and Engineering Research Training. The US Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

The author is grateful to Scott Anderson for valuable help and contributions to this work, to Paul Cohen for supporting this research, and to Andy Barto and the anonymous reviewers for useful comments.

References

Abramson, B. 1993. A Decision-Theoretic Framework for Integrating Sensors into AI Plans. *IEEE Transactions on Systems, Man, and Cybernetics* 23(2):366-373. (An earlier version appears under the title "An Analysis of Error Recovery and Sensory Integration for Dynamic Planners" in Proceedings of the Ninth National Conference on Artificial Intelligence, 744-749.)

Barto, A.G.; Bradtke, S.J.; and Singh, S.P. 1993. Learning to Act Using Real-Time Dynamic Programming. University of Massachusetts CMPSCI Technical Report 93-02. To appear in the *AI Journal.*

Chrisman, L., and Simmons, R. 1991. Sensible Planning: Focusing Perceptual Attention. In Proceedings of the Ninth National Conference on Artificial Intelligence, 756-761.

Dean, T.; Kaelbling, L.P.; Kirman, J.; and Nicholson, A. 1993. Planning with Deadlines in Stochastic Domains. In Proceedings of the Eleventh National Conference on Artificial Intelligence, 574-579.

Hendler, J., and Kinny, D. 1992. Empirical Experiments in Selective Sensing with Non-Zero-Cost-Sensors. In *Working Notes for the AAAI Spring Symposium on Control of Selective Perception*, 70-74.

Howard, R.A. 1960. *Dynamic Programming and Markov Processes.* MIT Press, Cambridge, MA.

Koenig, S. 1992. Optimal Probabilistic and Decision-Theoretic Planning using Markovian Decision Theory. UC Berkeley Computer Science technical report no. 685.

Langley, P., Iba, W., and Shrager, J. 1994. Reactive and Automatic Behavior in Plan Execution. To appear in *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems.*

Sutton, R.S. 1990. Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming. In *Proceedings of the Seventh International Conference on Machine Learning*, 216-224.

Tan, M. 1991. Learning a Cost-Sensitive Internal Representation for Reinforcment Learning. In *Proceedings of the Eighth International Workshop on Machine Learning*, 358-362.

Whitehead, S.D., and Ballard, D.H. 1991. Learning to Perceive and Act by Trial and Error. *Machine Learning* 7:45-83.