Searching for Planning Operators with Context-Dependent and Probabilistic Effects

Tim Oates and Paul R. Cohen
Computer Science Department, LGRC
University of Massachusetts
Box 34610
Amherst, MA 01003-4610
oates@cs.umass.edu, cohen@cs.umass.edu

Abstract

Providing a complete and accurate domain model for an agent situated in a complex environment can be an extremely difficult task. Actions may have different effects depending on the context in which they are taken, and actions may or may not induce their intended effects, with the probability of success again depending on context. We present an algorithm for automatically learning planning operators with context-dependent and probabilistic effects in environments where exogenous events change the state of the world. Empirical results show that the algorithm successfully finds operators that capture the true structure of an agent's interactions with its environment, and avoids spurious associations between actions and exogenous events.

Introduction

Research in classical planning has assumed that the effects of actions are deterministic and the state of the world is never altered by exogenous events, simplifying the task of encoding domain knowledge in the form of planning operators (Wilkins 1988). These assumptions, which are unrealistic for many real-world domains, are being relaxed by current research in AI planning systems (Kushmerick, Hanks, & Weld 1994) (Mansell 1993). However, as planning domains become more complex, so does the task of generating domain models. In this paper, we present an algorithm for automatically learning planning operators with context-dependent and probabilistic effects in environments where exogenous events change the state of the world.

We approach the problem of learning planning operators by first defining the space of all possible operators, and then developing efficient and effective methods for exploring that space. Operators should tell us when and how the state of an agent's world changes in response to specific actions. The degree to which an operator chosen from operator space captures such structure can be evaluated by looking at the agent's past experiences. Has the state of the world changed in the manner described by the operator significantly often in the past? Exploration of operator space is performed by an algorithm called Multi-Stream Dependency Detection (MSDD) that was designed to find dependencies among categorical values in multiple streams of data over time (Oates *et al.* 1995) (Oates & Cohen 1996). MSDD provides a general search framework, and relies on domain knowledge both to guide the search and reason about when to prune. Consequently, MSDD finds planning operators efficiently in an exponentially sized space.

Our approach differs from other work on learning planning operators in that it requires minimal domain knowledge; there is no need for access to advice or examples from domain experts (Wang 1995), nor for initial approximate planning operators (Gil 1994). We assume that the learning agent's initial domain model is weak, consisting only of a list of the different types of actions that it can take. The agent initially knows nothing of the contexts in which actions produce changes in the environment, nor what those changes are likely to be. To gather data for the learning algorithm, the agent explores its domain by taking random actions and recording state descriptions.[1] From the agent's history of state descriptions, the learning algorithm produces planning operators that characterize how the agent's world changes when it takes actions in particular contexts.

Domain Model

Our approach to learning planning operators requires minimal domain knowledge: we assume that the learning agent has knowledge of the types of actions that it can take, the sensors by which it can obtain the state of the world, and the values that can be returned by those sensors. With this information, we define a space of possible planning operators.

The Agent and its Environment

The agent is assumed to have a set of m sensors, $\mathcal{S} = \{s_1, \ldots, s_m\}$, and a set of n possible actions, $\mathcal{A} = \{a_1, \ldots, a_n\}$. At each time step, each sensor produces a single categorical value, called a *token*, from a

[1] Clearly, random exploration may be inefficient; nothing in our approach precludes non-random exploration.

finite set of possible values. Let $\mathcal{T}_i = \{t_{i_1}, \ldots, t_{i_k}\}$ be the token values associated with the i^{th} sensor, and let s_i^t denote the value obtained from sensor s_i at time t. Each sensor describes some aspect of the state of the agent's world; for example, s_2 may indicate the state of a robot hand, taking values from $\mathcal{T}_2 = \{\texttt{open, closed}\}$. The state of the world as perceived by the agent at time t, denoted $x(t)$, is simply the set of values returned by all of the sensors at that time. That is, $x(t) = \{s_i^t | 1 \le i \le m\}$ is a *state vector*.

Agent actions are encoded in a special sensor, s_a, so that s_a^t indicates which of the possible actions was attempted at time t. In general, $s_a \in \mathcal{T}_{action} = \mathcal{A} \cup \{\texttt{none}\}$. For any time step t on which the agent does not take an action, $s_a^t = \texttt{none}$. Actions require one time step, only one action is allowed on any time step, and resulting changes in the environment appear a constant number of time steps later. (These restrictions are not required by the MSDD algorithm, but are instituted for the particular domain of learning planning operators.) Without loss of generality, we will assume that the effects of actions appear one time step later. We assume that the state of the world can change due to an agent action, an exogenous event, or both simultaneously. The latter case makes the learning problem more difficult.

Consider a robot whose task it is to pick up and paint blocks. (This domain is adapted from (Kushmerick, Hanks, & Weld 1994), where it is used to explicate the Buridan probabilistic planner.) The robot has four sensors and can determine whether it is holding a block (HB), has a dry gripper (GD), has a clean gripper (GC), and whether the block is painted (BP). In addition, the robot can take one of four actions. It can dry its gripper (DRY), pick up the block (PICKUP), paint the block (PAINT), or obtain a new block (NEW). In terms of the notation developed above, the robot's initial domain model can be summarized as follows:

$$\begin{aligned}
\mathcal{S} &= \{\texttt{ACTION, BP, GC, GD, HB}\} \\
\mathcal{A} &= \{\texttt{DRY, NEW, PAINT, PICKUP}\} \\
\mathcal{T}_{ACTION} &= \{\texttt{DRY, NEW, PAINT, PICKUP, NONE}\} \\
\mathcal{T}_{BP} &= \{\texttt{BP, NOT-BP}\}, \ \mathcal{T}_{GC} = \{\texttt{GC, NOT-GC}\} \\
\mathcal{T}_{GD} &= \{\texttt{GD, NOT-GD}\}, \ \mathcal{T}_{HB} = \{\texttt{HB, NOT-HB}\}
\end{aligned}$$

Planning Operators

Operator representations used by classical planners, such as STRIPS, often include a set of preconditions, an add list, and a delete list (Fikes & Nilsson 1971). The STRIPS planner assumed that actions taken in a world state matching an operator's preconditions would result in the state changes indicated by the operator's add and delete lists without fail. We take a less restrictive view, allowing actions to be attempted in any state; effects then depend on the state in which actions are taken. Specifically, an operator $\mathcal{O} =< a, c, e, p >$ specifies an action, a context in which that action is expected to induce some change in the world's state, the state that results from the change, and the probability of the change occurring. If the state of the world matches the context c and the agent takes action a, then on the next time step the state of the world will match the effects e with probability p.

Contexts and effects of operators are represented as *multitokens*. A multitoken is an m-tuple that specifies for each sensor either a specific value or an assertion that the value is irrelevant. To denote irrelevance, we use a wildcard token $*$, and we define the set $\mathcal{T}_i^* = \mathcal{T}_i \cup \{*\}$. A multitoken is any element of the cross product of all of the $\mathcal{T}_i^*$; that is, multitokens are drawn from the set $\mathcal{T}_1^* \times \ldots \times \mathcal{T}_m^*$. Consider a two-sensor example for which $\mathcal{T}_1 = \mathcal{T}_2 = \{\texttt{A, B}\}$. Adding wildcards, $\mathcal{T}_1^* = \mathcal{T}_2^* = \{\texttt{A, B, *}\}$. The space of multitokens for this example ($\{\texttt{A, B, *}\} \times \{\texttt{A, B, *}\}$) is the following set: $\{$(A A), (A B), (A *), (B A), (B B), (B *), (* A), (* B), (* *)$\}$.

An operator's context specifies a conjunct of sensor token values that serve as the operator's precondition. For any given action, the values of some sensors will be relevant to its effects and other sensor values will not. For example, it might be more difficult for a robot to pick up a block when its gripper is wet rather than dry, but the success of the pickup action does not depend on whether the block is painted. A multitoken represents this contextual information as (* * GD *), wildcarding irrelevant sensors (e.g. the sensor that detects whether a block is painted) and specifying values for relevant sensors (the sensor that detects whether the gripper is dry).

While contexts specify features of the world state that must be present for operators to apply, effects specify how features of the context change in response to an action. We allow effects to contain non-wildcard values for a sensor only if the context also specifies a non-wildcard for that sensor. We also require that each non-wildcard in the effects be different from the value given by the context for the corresponding sensor. That is, operators must describe what changes in response to an action, not what stays the same. This restriction is similar to Wang's use of *delta-state* (Wang 1995), the difference between the states of the world before and after the execution of an action, to drive learning of operator effects. Likewise, Benson (Benson 1995) uses differences between state descriptions to identify the effects of actions when learning from execution traces generated by domain experts.

Assume that our block-painting robot's interactions with the world are governed by the following rules: The robot can successfully pick up a block 95% of the time when its gripper is dry, but can do so only 50% of the time when its gripper is wet. If the gripper is wet, the robot can dry it with an 80% chance of success. If the robot paints a block while holding it, the block will become painted and the robot's gripper will become dirty without fail. If the robot is not holding the block, then painting it will result in a painted block and a dirty gripper 20% of the time, and a painted block the remaining 80% of the time. Finally, when the robot requests a new block, it will always find itself in a state in which it is not holding the block, the block is not painted, and its gripper is clean; however, the gripper will be dry 30% of the time and wet 70% of the time. This information is summarized in our representation of planning operators in Figure 1.

```
<pickup, (* * GD NOT-HB), (* * * HB), 0.95>
<pickup, (* * NOT-GD NOT-HB), (* * * HB), 0.5>
<dry, (* * NOT-GD *), (* * GD *), 0.8>
<paint, (NOT-BP * * *), (BP * * *), 1.0>
<paint, (* GC * HB), (* NOT-GC * *), 1.0>
<paint, (* GC * NOT-HB), (* NOT-GC * *), 0.2>
<new, (BP * * *), (NOT-BP * * *), 1.0>
<new, (* NOT-GC * *), (* GC * *), 1.0>
<new, (* * * HB), (* * * NOT-HB), 1.0>
<new, (* * GD *), (* * NOT-GD *), 0.7>
<new, (* * NOT-GD *), (* * GD *), 0.3>
```

Figure 1: Planning operators in the block-painting robot domain.

The MSDD Algorithm

The MSDD algorithm finds dependencies—unexpectedly frequent or infrequent co-occurrences of values—in multiple streams of categorical data (Oates *et al.* 1995) (Oates & Cohen 1996). MSDD is general in that it performs a simple best-first search over the space of possible dependencies, terminating when a user-specified number of search nodes have been explored. It is adapted for specific domains by supplying domain-specific evaluation functions.

MSDD assumes a set of streams, $\mathcal{S}$, such that the i^{th} stream, s_i, takes values from the set $\mathcal{T}_i$. We denote a *history* of multitokens obtained from the streams at fixed intervals from time t_1 to time t_2 as $\mathcal{H} = \{x(t) | t1 \leq t \leq t2\}$. For example, the three streams shown below constitute a short history of twelve multitokens, the first of which is (A C B). MSDD explores the space of dependencies between pairs of multitokens. Dependencies are denoted $prec \overset{k}{\Rightarrow} succ$, and are evaluated with respect to $\mathcal{H}$ by counting how frequently an occurrence of the precursor multitoken *prec* is followed k time steps later by an occurrence of the successor multitoken *succ*. k is called the lag of the dependency, and can be any constant positive value. In the history shown below, the dependency (A C *) $\overset{1}{\Rightarrow}$ (* * A) is strong. Of the five times that we see the precursor (A in stream 1 and C in stream 2) we see the successor (A in stream 3) four times at a lag of one. Also, we never see the successor unless we see the precursor one time step earlier.

```
Stream 1: A D A C A B A B D B A B
Stream 2: C B C D C B C A B D C B
Stream 3: B A D A B D C A C B D A
```

MSDD performs a general-to-specific best-first search over the space of possible dependencies. Each node in the search tree contains a precursor and a successor multitoken. The root of the tree is a precursor/successor pair composed solely of wildcards; for the three streams shown earlier, the root of the tree would be (* * *) $\Rightarrow$ (* * *). The children of a node are its specializations, generated by instantiating wildcards with tokens. Each node inherits all the non-wildcard tokens of its parent, and it has exactly one fewer wildcard than its parent. Thus, each node at depth d has exactly d non-wildcard tokens distributed over the node's precursor and successor.

The space of two-item dependencies is clearly exponential. MSDD performs a *systematic* search, thereby avoiding redundant generation without requiring lists of open and closed nodes. Specifically, the children of a node are generated by instantiating only those streams to the right of the right-most non-wildcarded stream in that node. This method ensures that each dependency is explored at most once, and it facilitates reasoning about when to prune. For example, all descendants of the node (* A *) $\Rightarrow$ (B * *) will have wildcards in streams one and three in the precursor, an A in stream two in the precursor, and a B in stream one in the successor. The reason is that these features are not to the right of the rightmost non-wildcard, and as such cannot be instantiated with new values. If some aspect of the domain makes one or more of these features undesirable, then the tree can be safely pruned at this node.

Refer to (Oates & Cohen 1996) for a more complete and formal statement of the MSDD algorithm.

Learning Planning Operators with MSDD

To learn planning operators, MSDD first searches the space of operators for those that capture structure in the agent's interactions with its environment; then, the operators found by MSDD's search are filtered to remove those that are tainted by noise from exogenous events, leaving operators that capture true structure. This section describes both processes.

First, we map from our operator representation to MSDD's dependency representation. Consider the planning operator described earlier:

```
<pickup, (* * NOT-GD NOT-HB), (* * * HB), 0.5>
```

The context and effects of this operator are already represented as multitokens. To incorporate the idea that the pickup action taken in the given context is responsible for the changes described by the effects, we include the action in the multitoken representation:

(pickup * * NOT-GD NOT-HB) $\Rightarrow$ (* * * * HB)

We have added an action stream to the context and specified pickup as its value. Because MSDD requires that precursors and successors refer to the same set of streams, we also include the action stream in the effects, but force its value to be *. The only item missing from this representation of the operator is p, the probability that an occurrence of the precursor (the context and the action on the same time step) will be followed at a lag of one by the successor (the effects). This probability is obtained empirically by counting co-occurrences of the precursor and the successor in the history of the agent's actions ($\mathcal{H}$) and dividing by the total number of occurrences of the precursor. For the robot domain described previously, we want MSDD to find dependencies corresponding to the planning operators listed in Figure 1.

Guiding the Search

Recall that all descendants of a node n will be identical to n to the left of and including the rightmost non-wildcard in n. Because we encode actions in the first (leftmost) position of the precursor, we can prune nodes that have no action instantiated but have a non-wildcard in any other position. For example, the following node can be pruned because none of its descendants will have a non-wildcard in the action stream:

(* * * GD *) $\Rightarrow$ (* * * * *)

Also, our domain model requires that operator effects can only specify how non-wildcarded components of the context change in response to an action. That is, the effects cannot specify a value for a stream that is wildcarded in the context, and the context and effects cannot specify the same value for a non-wildcarded stream. Thus, the following node can be pruned because all of its descendants will have the value BP in the effects, but that stream is wildcarded in the context:

(pickup * * GD *) $\Rightarrow$ (* BP * * *)

Likewise, the following node can be pruned because all of its descendants will have the value GD instantiated in both the context and the effects:

(pickup * * GD *) $\Rightarrow$ (* * * GD *)

The search is guided by a heuristic evaluation function, $f(\mathcal{H}, n)$, which simply counts the number of times in $\mathcal{H}$ that the precursor of n is followed at a lag of one by the successor of n. This builds two biases into the search, one toward frequently occurring precursors and another toward frequently co-occurring precursor/successor pairs. In terms of our domain of application, these biases mean that, all other things being equal, the search prefers commonly occurring state/action pairs and state/action pairs that lead to changes in the environment with high probability. The result is that operators that apply frequently and/or succeed often are found by MSDD before operators that apply less frequently and/or fail often.

Filtering Returned Dependencies

We augmented MSDD's search with a post-processing filtering algorithm, FILTER, that removes operators that describe effects that the agent cannot reliably bring about and that contain irrelevant tokens. FILTER begins by removing all dependencies that have low frequency of co-occurrence or contain nothing but wildcards in the successor. Co-occurrence is deemed low when cell one of a dependency's contingency table is less than the user-specified parameter *low-cell1*. Those that remain are sorted in non-increasing order of generality, where generality is measured by summing the number of wildcards in the precursor and the successor. The algorithm then iterates, repeatedly retaining the most general operator and removing from further consideration any other operators that it subsumes and that do not have significantly different conditional probabilities (measured by the G statistic). All of the operators retained in the previous step are then tested to ensure that the change from the context to the effects is strongly dependent on the action (again measured by the G statistic). When G is used to measure the difference between conditional probabilities, the conditionals are deemed to be "different" when the G value exceeds that of the user-specified parameter *sensitivity*. We have omitted pseudocode for FILTER due to lack of space.

Empirical Results

To test the efficiency and completeness of MSDD's search and the effectiveness of the FILTER algorithm, we created a simulator of the block-painting robot and its domain as described earlier. The simulator contained fives streams: ACTION, BP, GC, GD and HB. Each simulation began in a randomly-selected initial state, and on each time step the robot had a 0.1 probability of attempting a randomly selected action. In addition, we added varying numbers of noise streams that contained values from the set $\mathcal{T}_n = \{$A, B, C$\}$. There was a 0.1 probability of an exogenous event occurring on each time step. When an exogenous event occurred, each noise stream took a new value, with probability 0.5, from $\mathcal{T}_n$.

The goal of our first experiment was to determine how the number of nodes that MSDD expands to find all of the interesting planning operators increases as the size of the search space grows exponentially. We ran the simulator for 5000 time steps, recording all stream values on each iteration. (Note that although the simulator ran for 5000 time steps, the agent took approximately 500 actions due to its low probability of acting on any given time step.) These values served as input to MSDD, which we ran until it found dependencies corresponding to all of the planning operators listed in Figure 1. As the number of noise streams, $\mathcal{N}$, was increased from 0 to 20 in increments of two, we repeated the above procedure five times, for a total of 55 runs of MSDD. A scatter plot of the number of nodes expanded vs. $\mathcal{N}$ is shown in Figure 2. If we ignore the outliers where $\mathcal{N} = 12$ and $\mathcal{N} = 20$, the number of nodes required by MSDD to find all of the interesting planning operators appears to be linear in $\mathcal{N}$, with a rather small slope. This is a very encouraging result. The outliers correspond to cases in which the robot's random exploration did not successfully exercise one or more of the target operators very frequently. Therefore, the search was forced to explore more of the vast space of operators (containing 10^{24} elements when $\mathcal{N} = 20$) to find them.

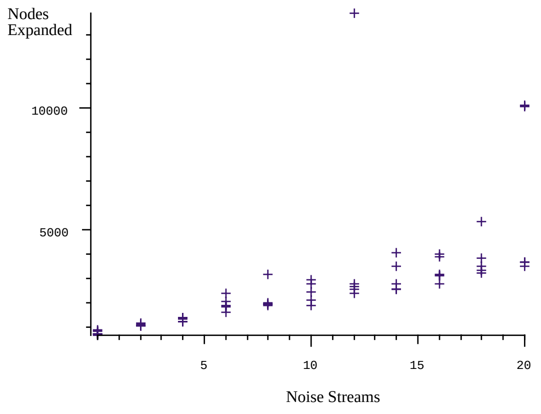

Figure 2: The number of search nodes required to find all of the target planning operators in the block-painting robot domain as a function of the number of noise streams.

In a second experiment, we evaluated the ability of the FILTER algorithm to return exactly the set of interesting planning operators when given a large number of potential operators. We gathered data from 20,000 time steps of our simulation, with 0, 5, 10, and 15 noise streams. (Again, the agent took far fewer than 20,000 actions due to its low probability of acting on any given time step.) For each of the three data sets, we let MSDD generate 20,000 operators; that is, expand 20,000 nodes. Figure 2 tells us that a search with far fewer nodes will find the desired operators. Our goal was to make the task more difficult for FILTER by including many uninteresting dependencies in its input. We used *low-cell1* = 6 and *sensitivity* = 30, and in all three cases FILTER returned the same set of dependencies. The dependencies returned with $\mathcal{N} = 0$ are shown in Figure 3. Note that all of the operators listed in Figure 1 are found, and that the empirically-derived probability associated with each operator is very close to its expected value. For $\mathcal{N} > 0$, the noise streams never contained instantiated values.

Interestingly, the last two operators in Figure 3 do not appear in Figure 1, but they do capture implicit structure in the robot's domain. The penultimate operator in Figure 3 says that if you paint the block with a clean gripper, there is roughly a 40% chance that the gripper will become dirty. Since that operator does not specify a value for the HB stream in its context, it includes cases in which the robot was holding the block while painting and cases in which it was not. The resulting probability is a combination of the probabilities of having a dirty gripper after painting in each of those contexts, 1.0 and 0.2 respectively. Similarly, the last operator in Figure 3 includes cases in which the robot attempted to pick up the block with a wet gripper (50% chance of success) and a dry gripper (95% chance of success).

```
<pickup, (* * GD NOT-HB), (* * * HB), 0.98>
<pickup, (* * NOT-GD NOT-HB), (* * * HB), 0.49>
<dry, (* * NOT-GD *), (* * GD *), 0.77>
<paint, (NOT-BP * * *), (BP * * *), 1.0>
<paint, (* GC * HB), (* NOT-GC * *), 1.0>
<paint, (* GC * NOT-HB), (* NOT-GC * *), 0.18>
<new, (BP * * *), (NOT-BP * * *), 1.0>
<new, (* NOT-GC * *), (* GC * *), 1.0>
<new, (* * * HB), (* * * NOT-HB), 1.0>
<new, (* * GD *), (* * NOT-GD *), 0.71>
<new, (* * NOT-GD *), (* * GD *), 0.31>
<paint, (* GC * *), (* NOT-GC * *), 0.38>
<pickup, (* * * NOT-HB), (* * * HB) 0.70>
```

Figure 3: Operators returned after filtering 20,000 search nodes generated for a training set with $\mathcal{N} = 0$ noise streams.

Related Work

Existing symbolic approaches to learning planning operators via interaction with the environment have typically assumed a deterministic world in which actions always have their intended effects, and the state of the world never changes in the absence of an action (Gil 1994) (Shen 1993) (Wang 1995). One notable exception is (Benson 1995), in which the primary effect of a durative action is assumed to be deterministic, but side effects may occur with some probability. In contrast, the work described in this paper applies to domains that contain uncertainties associated with the outcomes of actions, and noise from exogenous events. Subsymbolic approaches to learning environmental dynamics, such as reinforcement learning (Mahadevan & Connell 1992), are capable of handling a variety of forms of noise. Reinforcement learning requires a reward function that allows the agent to learn a mapping from states to actions that maximizes reward. Our approach is not concerned with learning sequences of actions that lead to "good" states, but rather attempts to acquire domain knowledge in the form of explicit planning operators.

Much of the work on learning planning operators assumes the availability of fairly sophisticated forms of domain knowledge, such as advice or problem solving traces generated by domain experts (Benson 1995) (Wang 1995), or initial approximate planning operators (Gil 1994). Our approach assumes that the learning agent initially knows nothing of the dynamics of its environment. A model of those dynamics is constructed based only on the agent's own past interactions with its environment.

MSDD's approach to expanding the search tree to avoid redundant generation of search nodes is similar to that of other algorithms (Rymon 1992) (Schlimmer 1993) (Riddle, Segal, & Etzioni 1994). MSDD's search differs from those mentioned above in that it explores the space of rules containing both conjunctive left-hand-sides and conjunctive *right-hand-sides*. Doing so allows MSDD to find structure in the agent's interactions with its environment that could not be found by the aforementioned algorithms (or any inductive learning algorithm that considers rules with a fixed number of literals on the right-hand-side).

Conclusions and Future Work

In this paper we presented and evaluated an algorithm that allows situated agents to learn planning operators for complex environments. The algorithm requires a weak domain model, consisting of knowledge of the types of actions that the agent can take, the sensors it possesses, and the values that can appear in those sensors. With this model, we developed methods and heuristics for searching through the space of planning operators to find those that capture structure in the agent's interactions with its environment. For a domain in which a robot can pick up and paint blocks, we demonstrated that the computational requirements of the algorithm scale approximately linearly with the size of the robot's state vector, in spite of the fact that the size of the operator space increases exponentially.

We will extend this work in several directions. Our primary interest is in the relationship between exploration and learning. How would the efficiency and completeness of learning be affected by giving the agent a probabilistic planner and allowing it to interleave goal-directed exploration and learning? However, our first task will be to apply our approach to larger, more complex domains.

Acknowledgements

This research was supported by ARPA/Rome Laboratory under contract numbers F30602-91-C-0076 and F30602-93-0100, and by a National Defense Science and Engineering Graduate Fellowship. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes not withstanding any copyright notation hereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements either expressed or implied, of the Advanced Research Projects Agency, Rome Laboratory or the U.S. Government.

References

Benson, S. 1995. Inductive learning of reactive action models. In *Proceedings of the Twelfth International Conference on Machine Learning*, 47–54.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(2):189–208.

Gil, Y. 1994. Learning by experimentation: Incremental refinement of incomplete planning domains. In *Proceedings of the Eleventh International Conference on Machine Learning*, 87–95.

Kushmerick, N.; Hanks, S.; and Weld, D. 1994. An algorithm for probabilistic least-commitment planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1074–1078.

Mahadevan, S., and Connell, J. 1992. Automatic programming of behavior-based robots using reinforcement learning. *Artificial Intelligence* 55(2–3):189–208.

Mansell, T. M. 1993. A method for planning given uncertain and incomplete information. In *Proceedings of the Ninth Conference on Uncertainty in Artificial Intelligence*, 350–358.

Oates, T., and Cohen, P. R. 1996. Searching for structure in multiple streams of data. To appear in *Proceedings of the Thirteenth International Conference on Machine Learning.*

Oates, T.; Schmill, M. D.; Gregory, D. E.; and Cohen, P. R. 1995. Detecting complex dependencies in categorical data. In Fisher, D., and Lenz, H., eds., *Finding Structure in Data: Artificial Intelligence and Statistics V.* Springer Verlag.

Riddle, P.; Segal, R.; and Etzioni, O. 1994. Representation design and brute-force induction in a boeing manufacturing domain. *Applied Artificial Intelligence* 8:125–147.

Rymon, R. 1992. Search through systematic set enumeration. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning.*

Schlimmer, J. C. 1993. Efficiently inducing determinations: A complete and systematic search algorithm that uses optimal pruning. In *Proceedings of the Tenth International Conference on Machine Learning*, 284–290.

Shen, W.-M. 1993. Discovery as autonomous learning from the environment. *Machine Learning* 12(1–3):143–165.

Wang, X. 1995. Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of the Twelfth International Conference on Machine Learning.*

Wilkins, D. E. 1988. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann.