Learning Planning Operators with Conditional 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. In addition, the contexts and probabilities that govern the effects and success of actions may change over time. 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. Our approach assumes that a situated agent has knowledge of the types of actions that it can take, but initially knows nothing of the contexts in which an action produces change in the environment, nor what that change is likely to be. The algorithm accepts as input a history of state descriptions observed by an agent while taking actions in its domain, and produces as output descriptions of planning operators that capture structure in the agent's interactions with its environment. We present results for a sample domain showing that the computational requirements of our algorithm scale approximately linearly with the size of the agent's state vector, and that the algorithm successfully locates operators that capture true structure and avoids those that incorporate noise.

Keywords: learning, knowledge acquisition, uncertainty

1 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 [13]. These assumptions, which are unrealistic for many real-world domains, are being relaxed by current research in AI planning systems [5] [7]. 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.

Our approach assumes a situated agent and a weak initial domain model, consisting only of a list of the different types of actions that the agent 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 *predictive* planning operators that characterize how the agent's world changes when it takes actions in particular contexts.

In section 2 we define a space of planning operators, then we show how to search this space efficiently for predictive operators. The search is performed by an algorithm called Multi-Stream Dependency Detection (MSDD, see section 3) which finds statistical dependencies among categorical values in multiple data streams over time [9]. MSDD is a general search algorithm; it relies on domain knowledge to guide its search and decide when to prune in an otherwise exponential space of planning operators (see section 4). Not all points in operator space represent true dependencies between agent actions and effects—some may describe changes that are actually due to exogenous events—so we augment MSDD with a filtering algorithm to find operators that describe effects that the agent itself can reliably bring about.

The efficiency and effectiveness of our approach are demonstrated in a simulated robot domain. Eleven planning operators with conditional and probabilistic effects characterize the robot's interactions with its environment. The number of search nodes required by MSDD to find these target operators scales approximately linearly with the size of the agent's state description, even though the size of the operator space increases exponentially (see section 5). Moreover, the algorithm consistently returns small sets of operators that contain the target operators, as well as operators that capture structure that is implicit in the definition of the sample domain but that was not explicitly codified in the target operators. We contrast our approach with related work in section 6 and describe future research directions in section 7.

2 Domain Model

The agent's domain model is weak in the sense that it includes only the agent's repertoire of actions and sensors, and values that can be returned by sensors. With this information, we define a space of possible planning operators.

[1] Clearly, random exploration is inefficient, especially when actions need to be combined into long sequences to achieve certain states. The utility of goal-directed exploration and learning is well documented [10]. However, nothing in our approach precludes non-random exploration (see Section 7).

2.1 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 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, $x(t) = \{s_i^t | 1 \leq i \leq m\}$ is a *state vector*.

Agent actions are encoded in a special sensor, s_a; effects show up as changes in state vectors observed on subsequent time steps. In general, $s_a \in \mathcal{T}_{action} = \mathcal{A} \cup \{\texttt{none}\}$, and indicates which one of the possible actions was attempted. 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 agent action is allowed on any time step, and resulting changes in the environment appear a constant number of time steps later. (Section 7 discusses ways to eliminate those restrictions.) 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 complicates the learning problem.

Consider a simple robot whose task it is to pick up and paint blocks. (This domain is adapted from [5], 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:

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

2.2 Planning Operators

The STRIPS operator representation includes a set of preconditions, an add list, and a delete list [2]. 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 change itself, and the probability of the change occurring. If the agent is in a state matching the context c and it takes action a, then it will enter a state matching 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. To understand this restriction, consider the following operator:

<dry, (* * * *), (* * GD *), 0.5>

Because the context of this operator does not specify whether the gripper is wet or dry, it is impossible to interpret the probability associated with the operator. Perhaps the base probability of a dry gripper in the robot's environment is 0.45 and the dry action succeeds 10% of the time; alternatively these probabilities could be 0.17 and 0.4, respectively. Both pairs of probabilities would produce $p = 0.5$ in the operator above. Thus, an agent with a wet gripper cannot use this operator to calculate its chances of obtaining a dry gripper via the **dry** action. The situation is very different in the following operator, which specifies non-wildcards in the effects only for sensors containing non-wildcards in the context:

<dry, (* * NOT-GD *), (* * GD *), 0.5>

In this case it is clear that attempting a **dry** action with a wet gripper (**NOT-GD**) results in a dry gripper 50% of the time.

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.

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. Note the wildcards. In the first operator, for example, the success of the **pickup** action depends on whether the gripper is dry (**GD** in the context) but it doesn't depend on whether the block is painted.

```
<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.

3 The MSDD Algorithm

The MSDD algorithm finds dependencies—unexpected co-occurrences of values—in multiple streams of categorical data [9]. MSDD is general in that it performs a simple best-first search over the space of possible dependencies. 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 $prec \stackrel{k}{\Rightarrow} succ$, 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 *)` $\stackrel{1}{\Rightarrow}$ `(* * 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
```

When counting occurrences of a multitoken, the wildcard matches all other tokens; for example, both `(D B A)` and `(C D A)` are occurrences of `(* * A)`, but `(C D D)` is not.

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. For example, both `(A * *)` $\Rightarrow$ `(* * *)` and `(* * *)` $\Rightarrow$ `(* D *)` are children

of the root node; and both `(* * B)` $\Rightarrow$ `(* C *)` and `(* * *)` $\Rightarrow$ `(* A A)` are children of nodes at depth one.

The space of two-item dependencies is clearly exponential. The number of possible multitokens is given by $|\mathcal{T}_1^* \times \ldots \times \mathcal{T}_m^*| = \prod_{i=1}^{m} |T_i^*|$. If each stream contains k distinct tokens (including *) and there are m streams, then the number of possible multitokens is k^m, and the number of possible dependencies is k^{2m}. Given the size of the search space, MSDD requires domain knowledge to guide the search and to allow efficient pruning. Section 4 describes how these requirements are met; here, we confine ourselves to the general framework of MSDD search. 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 doesn't say *which* children should be generated before others (see section 4.1), but it does ensure 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.

Formal statements of both the MSDD algorithm and its node expansion routine are given in Algorithms 3.1 and 3.2. The majority of the work performed by MSDD lies in evaluating f for each expanded node. Typically, f will count co-occurrences of the node's precursor and successor, requiring a complete pass over $\mathcal{H}$. Assuming that $\mathcal{H}$ contains l vectors of size m, the computational complexity of MSDD is $O(m * l * maxnodes)$.

Algorithm 3.1 MSDD

MSDD($\mathcal{H}, f, maxnodes$)

1. $expanded = 0$
2. $nodes$ = ROOT-NODE()
3. while NOT-EMPTY($nodes$) and $expanded < maxnodes$ do
   a. remove from $nodes$ the node n that maximizes $f(\mathcal{H}, n)$
   b. EXPAND(n), adding its children to $nodes$
   c. increment $expanded$ by the number of children generated in (b)

Algorithm 3.2 EXPAND

EXPAND(n)

1. for i from m downto 1 do
   a. if $n.precursor[i] \neq$ '*' then
      return children
   b. for $t \in \mathcal{T}_i$ do
      i. $child$ = COPY-NODE(n)
      ii. $child.precursor[i] = t$
      iii. push $child$ onto $children$
2. repeat (1) for the successor of n
3. return $children$

4 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 the operator representation of section 2.2 to MSDD's dependency representation. Consider the planning operator described earlier:

$\mathcal{O} =< a, c, e, p >$ = `<pickup, (* * NOT-GD NOT-HB), (* * * HB), 0.5>`

The context and effects of that 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 sample domain described previously, we want MSDD to find dependencies corresponding to the planning operators listed in Figure 1. These dependencies are given in Figure 2.

```
(pickup * * GD NOT-HB) ⇒ (* * * * HB)
(pickup * * NOT-GD NOT-HB) ⇒ (* * * * HB)
(dry * * NOT-GD *) ⇒ (* * * GD *)
(paint NOT-BP * * *) ⇒ (* BP * * *)
(paint * GC * HB) ⇒ (* * NOT-GC * *)
(paint * GC * NOT-HB) ⇒ (* * NOT-GC * *)
(new BP * * *) ⇒ (* NOT-BP * * *)
(new * NOT-GC * *) ⇒ (* * GC * *)
(new * * * HB) ⇒ (* * * * NOT-HB)
(new * * GD *) ⇒ (* * * NOT-GD *)
(new * * NOT-GD *) ⇒ (* * * GD *)
```

Figure 2: The planning operators in the block-painting robot domain represented as MSDD dependencies.

4.1 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, the domain model of Section 2 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.[2] 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.

4.2 Filtering Returned Dependencies

Not all of the dependencies produced by MSDD's search are equally interesting. Interesting dependencies tell us about structure in the environment; that is, they tell when changes in the environment are associated with particular actions more or less often than we would expect by random chance. Suppose our robot spins a roulette wheel on each time step, and the color of the outcome is available via one of its sensors. Further, there is a button underneath the wheel that the robot can push. Assuming a fair wheel, we expect `(* * RED)` to be be followed by `(* * BLACK)` with a probability of 0.5. If we also find that `(push * RED)` is followed by `(* * BLACK)` with a probability of 0.5, we are not interested. The `push` action does not affect the relationship between `RED` and `BLACK`. However, if we find that `(push * RED)` is followed by `(* * BLACK)` with a probability of 0.6, our interest is piqued; and if that probability is 0.8 or 0.2, we are even more interested. Therefore, a dependency, when viewed as a planning operator, is interesting to the extent that the probability of the effects given the context depends on the action. Said differently, we want to know for each operator $\mathcal{O} =< a, c, e, p >$ how different $p(e|c, a)$ and $p(e|c)$ are.

We must be careful to avoid non-wildcards that "freeload" on interesting operators. In our sample block-painting robot domain, it is the case that `(paint * GC * HB)` is always followed by `(* * NOT-GC * *)`. If, by random chance, the robot's gripper is dry when it paints a block that it is holding with a clean gripper, then `(paint * GC GD HB)` will also be followed by `(* * NOT-GC * *)` without fail. In this case, `GD` is a freeloader. It has no impact on the effects of `(paint * GC * HB)` or on the probability of the effects given the context and the action. All other things being equal, we prefer operators that are more general; that is, we prefer `(paint * GC * HB)` $\Rightarrow$ `(* * NOT-GC * *)` to `(paint * GC GD HB)` $\Rightarrow$ `(* * NOT-GC * *)`. The question that needs to be answered is whether the probability of the effects in one context

[2] Rules with successors containing only wildcards are problematic because they predict nothing. We assign such nodes a value equal to the average of $f(\mathcal{H}, n)$ for their children, and delete them during a later step.

is significantly different from the probability of the effects in a more specific context. If not, we keep the general operator and discard the more specific one.

The G statistic computed for 2x2 contingency tables provides a simple metric for the degree to which two empirically-derived conditional probabilities differ. Returning to our wheel-spinning robot, suppose the wheel comes up red 100 times, followed by black 48 times. Further, on 20 of the time steps for which the wheel came up red, the robot also performed a **push** action, followed on the next time step by black 12 times. We can summarize this information with two conditional probabilities: $p(black|red \wedge \overline{push}) = 36/80 = 0.45$ and $p(black|red \wedge push) = 12/20 = 0.6$. We can also use a contingency table representation as shown in Figure 3. We want to know whether the distribution of red and black following an occurrence of red depends on whether a **push** action was also attempted. The G statistic measures nonindependence, with larger values of G representing more dependence between the precursor and successor. The value of G in this case is 1.45, which is small; however, if black followed red and **push** 16 times rather than 12, G would be 10.77, and we would be inclined to conclude that **push** does affect the relationship between red and black.[3]

	$black$	$\overline{black}$
$red \wedge \overline{push}$	36	44
$red \wedge push$	12	8

Figure 3: A contingency table that summarizes the distribution of black and red ($\overline{black}$) following either red and no **push** action or red and a **push** action.

Pseudocode for the FILTER routine is shown in Algorithm 4.1. FILTER accepts as input a list of dependencies returned by MSDD, D, the history used during MSDD's search, $\mathcal{H}$, and two parameters that control the sensitivity of the algorithm, n and g. It returns a list of dependencies that represent the interesting operators found by MSDD. To evaluate dependencies as planning operators, we need to map from the former to the latter. Let $e(d)$ denote dependency d's effects, and $n(d)$ denote the the number of times that d's precursor was followed by d's successor in $\mathcal{H}$. We define the function SUBSUMES(d_1, d_2) to return true if dependency d_1 is a generalization of dependency d_2. We define the function $G(d_1, d_2, \mathcal{H})$ to return the G statistic computed for the contingency table used to determine whether the conditional probability of d_1's successor given its precursor is different from the conditional probability of d_2's successor given its precursor. For example, the function G would compute a G statistic for a table such as the one shown in Figure 3 for the two dependencies **(* * RED)** $\Rightarrow$ **(* * BLACK)** and **(push * RED)** $\Rightarrow$ **(* * BLACK)**. The parameter g is used in steps $(4.c.i)$ and $(5.b)$ as a threshold which $G(d_1, d_2, \mathcal{H})$ must exceed before d_1 and d_2 are considered to represent "different" conditional probabilities. The parameter n is used in step (1) of the algorithm to remove dependencies with low frequency of co-occurrence, because such dependencies are likely to be spurious.

FILTER begins in step 1 by removing all dependencies that have low frequency of co-occurrence or contain nothing but wildcards in the successor. Step 4 processes operators in order of generality, due to the sort in step 2, repeatedly retaining the most general operator and removing from further consideration any other operators that it subsumes and that do

[3]We can, of course, test whether a value of G is *statistically significant*, but this raises problems of multiple testing [1] and it is not necessary if we use G only to rank operators.

Algorithm 4.1 FILTER

FILTER$(D, \mathcal{H}, n, g)$

1. remove from D all dependencies d such that $n(d) < n$ or $e(d)$ contains only wildcards
2. sort D in non-increasing order of generality
3. $S = \emptyset$
4. while NOT-EMPTY(D) do
   a. $s =$ POP(D)
   b. PUSH(s, S)
   c. for $d \in D$ do
      i. if SUBSUMES(s, d) and $G(s, d, \mathcal{H}) < g$ then remove d from D
5. for $s \in S$ do
   a. let s' be a copy of s with the action in the precursor wildcarded
   b. if $G(s, s', \mathcal{H}) < g$ then remove s from S
6. return S

not have significantly different conditional probabilities. All of the operators retained in step 4 are then tested in step 5 to ensure that the change from the context to the effects is strongly dependent on the action, where the degree of dependence is measured by G.

5 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 in Section 2. (We have successfully applied MSDD to much larger problems in other domains. A brief summary of that work is given in Section 6.) 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 $\mathcal{T}_n$.

We ran the simulator for 5000 time steps, recording all stream values on each iteration. 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. The goal 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. A scatter plot of the number of nodes expanded vs. $\mathcal{N}$ is shown in Figure 4. 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.

To explain the outliers, consider what happens to the size of the search space as $\mathcal{N}$ increases. When $\mathcal{N}$ is 0, the space of possible two item dependencies contains more than 164,000 elements. Compare this with a space of more than 10^{24} elements with 20 noise streams. Recall that MSDD

finds operators that apply frequently and/or succeed often before operators that apply less frequently and/or fail often. 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 to find them.

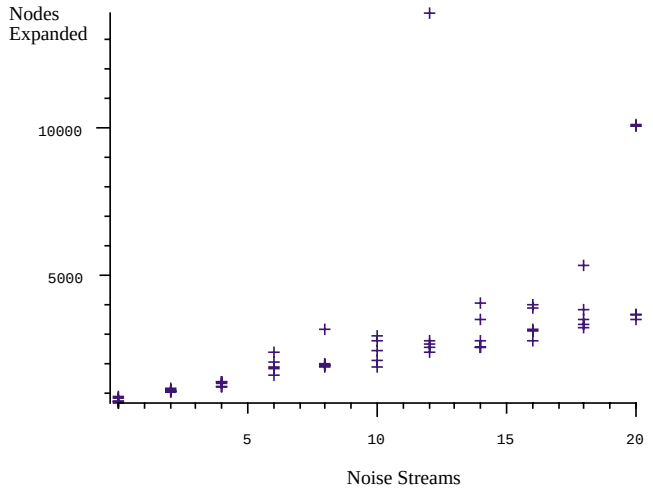

Figure 4: The number of search nodes required to find all of the target planning operators in the block-painting robot domain as a function of the number of noise streams.

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 5, 10, and 15 noise streams. For each of the three data sets, we let MSDD generate 20,000 operators; that is, expand 20,000 nodes. Figure 4 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 $n = 6$ and $g = 30$, and in all three cases FILTER returned the same set of dependencies. The dependencies returned with $\mathcal{N} = 5$ are shown in Figure 5. 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. The five noise streams occupy the first five slots of the context and effects multitokens, and none contain instantiated values.

Interestingly, the last two operators in Figure 5 do not appear in Figure 1, but they do capture structure in the robot's domain that was implicit in the description of Section 2.2. The penultimate operator in Figure 5 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 5 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).

The effect of g on the sensitivity of FILTER can be demonstrated by looking at the operators returned when g is lowered from 30 to 20. Recall that g is used as a threshold when determining whether two conditional probabilities are different. When we lower g, we find all of the operators shown in Figure 5, as well as the following:

```
<pickup, (* * * * * * * GD NOT-HB), (* * * * * * * * HB), 0.984>
<pickup, (* * * * * * * NOT-GD NOT-HB), (* * * * * * * * HB), 0.494>
<dry, (* * * * * * * NOT-GD *), (* * * * * * * GD *), 0.773>
<paint, (* * * * * NOT-BP * * *), (* * * * * BP * * *), 1.0>
<paint, (* * * * * * GC * HB), (* * * * * * NOT-GC * *), 1.0>
<paint, (* * * * * * GC * NOT-HB), (* * * * * * NOT-GC * *), 0.175>
<new, (* * * * * BP * * *), (* * * * * NOT-BP * * *), 1.0>
<new, (* * * * * * NOT-GC * *), (* * * * * * GC * *), 1.0>
<new, (* * * * * * * * HB), (* * * * * * * * NOT-HB), 1.0>
<new, (* * * * * * * GD *), (* * * * * * * NOT-GD *), 0.714>
<new, (* * * * * * * NOT-GD *), (* * * * * * * GD *), 0.313>
<paint, (* * * * * * GC * *), (* * * * * * NOT-GC * *), 0.383>
<pickup, (* * * * * * * * NOT-HB), (* * * * * * * * HB) 0.701>
```

Figure 5: Operators returned after filtering 20,000 search nodes generated for a training set with 5 noise streams.

```
<paint, (* * * * * GC * * NOT-GD), (* * * * * NOT-GC * * *), 0.263>
<paint, (* * * * * GC * * GD), (* * * * * NOT-GC * * *) , 0.505>
```

Both of these operators are subsumed by the operator in which the **paint** action is attempted with a clean gripper, resulting in a dirty gripper 38% of the time (the penultimate operator in Figure 5). It is interesting to note that, according to these operators, the probability of having a dirty gripper after painting is higher if the gripper is dry than if the gripper is wet. If the robot had attempted to pick up the block before painting during its random exploration, these probabilities would make perfect sense. With a wet gripper, the robot's attempts to pick up the block often fail, resulting in a dirty gripper only 10% of the time after painting. If the gripper is dry, the robot almost always succeeds in picking up the block, resulting in a dirty gripper 100% of the time. Looking at the execution trace of the robot confirms that this is exactly what is going on. By lowering g from 30 to 20 we have found more structure in the environment, but this structure includes subtle effects that we may or may not wish to enshrine in a planning operator.

6 Related Work

MSDD's approach to expanding the search tree is similar to that of Rymon's Set Enumeration trees (SE-trees) [11], which in turn are related to Knuth's trie data structure [4]. Rymon uses SE-trees to systematically enumerate the elements of the power set of a set given a total ordering on the set's elements. He demonstrates the flexibility of that approach by implementing efficient SE-tree based versions of several algorithms (such as Reiter's hitting-set algorithm). Our contribution with MSDD is a novel approach to the problem of finding structure in multiple streams of data, and showing how that approach is both general and efficient. We have successfully applied MSDD to classification problems [8] and to learning rules in a shipping network that relate current states to future pathologies [9]. That work involved data sets with more than 50 streams, indicating that MSDD scales well with problem size.

Symbolic approaches to learning planning knowledge 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 [3] [12]. The work described

in this paper applies in 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 [6], 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 learn explicit, conditional and probabilistic planning operators.

7 Conclusions and Future Work

As research in planning attacks increasingly complex domains, the task of obtaining models of those domains becomes more difficult. 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 sample 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. The algorithm consistently returns a small set of planning operators that contains all of the target operators for our sample domain, and never includes noise introduced by exogenous events. In addition, the algorithm finds structure in the domain that is implicit in our construction of the domain, but that was not explicitly formulated as a target planning operator.

We will extend this work in several directions. We can easily relax the requirement that the effects of an action appear a constant number of time steps after the action. One approach would be to run MSDD with different lags on the same history, collecting a profile over time of the effects of an action. Another approach would be to encode multiple time steps in the successor of a dependency. That is, rather than using $x(t+1)$ following an action at time t as the successor, we could concatenate $x(t+1)$, $x(t+2)$, ..., $x(t+k)$, and use the resulting multitoken of size km as the successor. Similarly, we can concatenate multiple state vectors in the precursor, capturing contexts with temporal extent. Another interesting area is 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?

Acknowledgements

This research was supported by ARPA/Rome Laboratory under contract numbers F30602-91-C-0076 and F30602-93-0100. 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.

The authors would like to thank Marc Atkin and Rob St. Amant for their comments on earlier versions of this paper.

References

[1] Paul R. Cohen. *Empirical Methods for Artificial Intelligence.* The MIT Press, 1995.

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

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

[4] Donald E. Knuth. *The Art of Computer Programming.* Addison-Wesley, 1973.

[5] Nicholas Kushmerick, Steve Hanks, and Daniel Weld. An algorithm for probabilistic least-commitment planning. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, pages 1074–1078, 1994.

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

[7] Todd Michael Mansell. A method for planning given uncertain and incomplete information. In *Proceedings of the Ninth Conference on Uncertainty in Artificial Intelligence*, pages 350–358, 1993.

[8] Tim Oates. MSDD as a tool for classification. EKSL Memorandum 94-29. Department of Computer Science, University of Massachusetts at Amherst, 1994.

[9] Tim Oates, Matthew D. Schmill, Dawn E. Gregory, and Paul R. Cohen. Detecting complex dependencies in categorical data. In Doug Fisher and Hans Lenz, editors, *Finding Structure in Data: Artificial Intelligence and Statistics V.* Springer Verlag, 1995.

[10] Ashwin Ram and David B. Leake. *Goal-Driven Learning.* The MIT Press, 1995.

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

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

[13] David E. Wilkins. *Practical Planning: Extending the Classical AI Planning Paradigm.* Morgan Kaufmann, 1988.