A Multistrategy Learning System for Planning Operator Acquisition

Xuemei Wang*
Computer Science Department
Carnegie Mellon University
5000 Forbes Avenue
Pittsburgh, PA 15213, U.S.A
wxm@cs.cmu.edu

Abstract

This paper describes a multistrategy learning approach for automatic acquisition of planning operators. The two strategies are: (*i*) learning operators by observing expert solution traces, and (*ii*) refining operators through practice in a learning-by-doing paradigm.

During observation, OBSERVER uses the knowledge that is *naturally observable* when experts solve problems, without the need of explicit instruction or interrogation. During practice, OBSERVER generates its own learning opportunities by solving practice problems. The *inputs* to our learning system are: the description language for the domain, experts' problem solving traces, and practice problems to allow learning-by-doing operator refinement. Given these inputs, our system *automatically* acquires the preconditions and effects (including conditional effects and preconditions) of the operators.

Our approach has been fully implemented in a system called OBSERVER on top of a non-linear planner PRODIGY. We present empirical results to demonstrate the validity of our approach in a process planning domain. These results show that the system learns operators in this domain well enough to solve problems as effectively as human-expert coded operators. The results also show the learned operators are more effective if both strategies are used, than if only one strategy is used. Therefore, both learning strategies contribute significantly to the learning process.

Introduction

Acquiring and maintaining domain knowledge is a key bottleneck in fielding planning systems for realistic domains (Chien *et al.* 1995). For example, significant effort has been devoted to writing and debugging planning operators for successful applications of planning systems (desJardins 1994; Chien 1996; Gil 1991).

While considerable research has focused on *knowledge acquisition tools* for rule-based systems (see (Boose and Gaines 1989) for a summary), and some on such tools for specialized planning systems (Chien 1994), these systems all require a considerable amount of direct interactions with domain experts. It is possible to *automate* the process of knowledge acquisition when a simulator is available, although there has been little work in this area.

Simulators are commonly available in areas such as CAD, robotics assembly planning, manufacturing, process planning. Simulating is different from planning. Simulators simply act as simplified surrogates for the real world, bypassing problems of coping with sensors and physical effectors. In contrast, planning is the process of generating a sequence of actions to transform an initial situation into a goal situation. Planning requires a specific form of knowledge that simulators do not have.

We have developed a multistrategy learning system OBSERVER, that automatically transforms simulator knowledge into planning operators that can be used for planning. Our approach is to learn planning operators by observing expert solution traces and to further refine the operators through practice by solving problems in the simulator in a learning-by-doing (Anzai and Simon 1979) paradigm. During observation, OBSERVER uses the knowledge *naturally observable* when experts solve problems, without need of explicit instruction or interrogation. Learning from observation allows OBSERVER to bootstrap itself, otherwise since OBSERVER initially does not have any knowledge about the preconditions and the effects of the operators, it has to uses random exploration, which is much less efficient. During practice, OBSERVER generates its own learning opportunities by solving practice problems.

The *inputs* to OBSERVER are:

- The description language for the domain. This includes the types of objects and the predicates that describe states and operators.
- Observations of an expert agent consisting of: 1) the sequence of actions being executed, 2) the state in which each action is executed (*pre-state*), and 3) the state resulting from the execution of each action (*post-state*).

*This research is sponsored by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant number F33615-93-1-1330. Views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing official policies or endorsements, either expressed or implied, of Wright Laboratory or the United States Government. Thanks to Jaime Carbonell, Jill Fain, Douglas Fisher, Herb Simon, and Manuela Veloso for their suggestions and support.

- Practice problems used by the learning system for practice in order to further refine incomplete operators.

Given these inputs, our system *automatically* acquires the preconditions and the effects of the operators. This representation is exactly what is required by most operator-based planners such as STRIPS (Fikes and Nilsson 1971), TWEAK (Chapman 1987), PRODIGY (Carbonell *et al.* 1992; Veloso *et al.* 1995), SNLP (McAllester and Rosenblitt 1991), and UCPOP (Penberthy and Weld 1992). Our learning method has been fully implemented on top of the PRODIGY architecture, a complete, state-space, non-linear planner (Veloso and Stone 1995).

The learning method described in this paper is domain-independent, although the examples are from the process planning domain. The process planning task is to generate plans to produce parts given part specifications, such as the shape, the size along each dimension, and the surface quality, by using processes such as drilling and milling. Different machines, such as drills, milling-machines, and different tools such as spot drills, twist-drills are available.

The structure of this paper is as follows. We first present a high level view of the learning system, followed by discussions on the issues that arise from learning by observation and practice. We then describe the detailed algorithm for learning operators from observation. These initially learned operators maybe be incomplete and incorrect, they are further refined during practice. We describe our algorithms for planning with incomplete and incorrect operators during practice, and our algorithm for refining operators during practice. Finally we present empirical results and analysis to demonstrate the effectiveness of the learning system, and the significance of both learning strategies. We end with discussions on related work, future work, and conclusions.

Learning architecture overview

OBSERVER's learning algorithms make several assumptions. First, since OBSERVER is operating within the framework of classical planners, it assumes that the operators and the states are deterministic. Second, OBSERVER assumes noise-free sensors, i.e., there are no errors in the states. And finally, we notice that in most application domains, the majority of the operators have conjunctive preconditions only. Therefore, OBSERVER assumes that the operators have conjunctive preconditions. This assumption greatly reduces the search space for operator preconditions without sacrificing much of the generality of learning approach.

Figure 1 shows the architecture of our learning system OBSERVER. There are three main components:

Learning operators from observation: OBSERVER inductively learns an initial set of operators from the observations of expert solutions given very little initial knowledge. Details of the learning algorithm are described in this paper.

Planning, plan repair, and execution: The initial set of operators learned from observation can be incomplete and incorrect in many ways. They are further refined when OBSERVER uses them to solve practice problems. Given a practice problem, OBSERVER first generates an initial plan to solve the problem. The initial plan is executed in the environment, resulting in both successful and unsuccessful executions of operators. OBSERVER uses these executions as positive or negative training examples for further operator refinement. The planner also repairs the failed plans upon unsuccessful executions. The repaired plans are then executed in the environment. This process repeats until the problem is solved, or until a resource bound is exceeded. Details of the integration of planning, execution, and learning can be found in (Wang 1996b).

Refining operators during practice: The successful and unsuccessful executions generated during practice are effective training examples that OBSERVER uses to further refine the initial imperfect operators. Details for operator refinement during practice are described in this paper.

Issues of learning planning operators

OBSERVER learns STRIPS-like operators that include preconditions and effects (including conditional effects and preconditions). Figure 2 is an example of such an operator from the process planning domain. Note that in Figure 2, `(not (has-burrs <part>))` is a *negated precondition*, meaning that HOLD-WITH-VISE can only be applied when `(has-burrs <part>)` is absent in the state. Also note that `(add (holding-weakly <machine> <holding-device> <part> <side>))` and `(add (holding <machine> <holding-device> <part> <side>))` are *conditional effects*, with their corresponding *conditional preconditions* being `(shape-of <part> CYLINDRICAL)` and `(shape-of <part> RECTANGULAR)`. Each conditional effect occurs *only* when its corresponding conditional preconditions are satisfied.

An important point is that negated preconditions are rare in most application domains. For example, in our process planning domain, there are only 25 negated preconditions among 350 preconditions in the human-expert coded operators. In many other domains implemented in PRODIGY, such as the extended-strips domain, there are no negated preconditions. This is because people tend to prefer thinking in terms of "something should be true in the state," which corresponds to non-negated preconditions, to thinking "something can not be true", which corresponds to negated preconditions. Also, one can easily convert a negated precondition to a non-negated precondition by introducing a new predicate,

OBSERVER learns operator preconditions by generalizing from the observed *pre-state*. In our application domains, the number of facts in the *pre-state* and *post-state* are typically much larger than the number of preconditions and effects of the corresponding operators. This is because many facts in the state are not relevant for the operator. For example, in the process planning domain, the *pre-state*, or *post-state* typically includes 50 to 70 assertions, while an operator usually has 2 to 6 preconditions or effects. In the absence of background knowledge for identifying the portion of the world state relevant to the planning operator, it

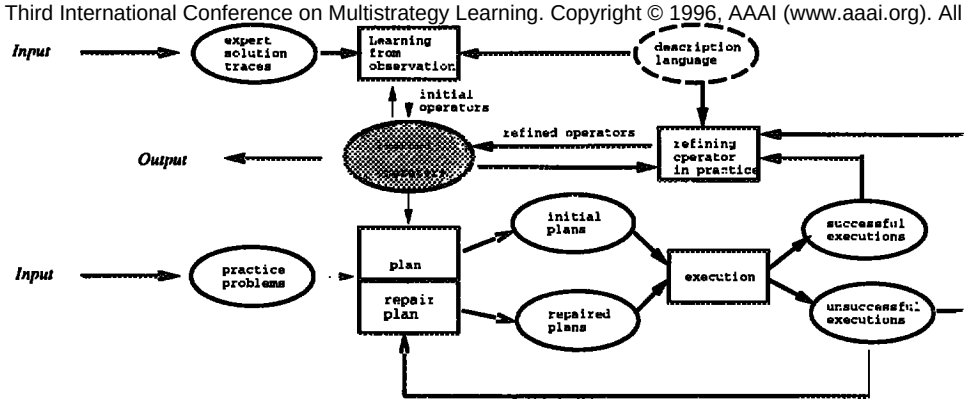

Figure 1: Overview of OBSERVER's learning architecture.

```
(Operator HOLD-WITH-VISE
 (preconds ((<hd> VISE) (<side> Side)
            (<machine> Machine) (<part> Part))
  (and (has-device <machine> <hd>)
       (not (has-burrs <part>))
       (is-clean <part>)
       (on-table <machine> <part>)
       (is-empty-holding-device <hd> <machine>)
       (is-available-part <part>)))
  (effects
   (del (on-table <machine> <part>))
   (del (is-available-part <part>))
   (del (is-empty-holding-device <hd> <machine>))
   (if (shape-of <part> CYLINDRICAL)
       (add (holding-weakly
              <machine> <hd> <part> <side>))))
   (if (shape-of <part> RECTANGULAR)
       (add (holding
              <machine> <hd> <part> <side>)))))
```

Figure 2: Operator HOLD-WITH-VISE from the process planning domain. This operator specifies the preconditions and effects of holding a part with a vise. This is operator OBSERVER learns after observation and practice.

is computationally expensive to find the maximally specific common generalization MSCG from the observed *pre-state*. In fact, Haussler (Haussler 1989) shows that finding an existential conjunctive concept consistent with a sequence of m examples over an instance space defined by n attributes is NP-complete. Furthermore, he shows that the size of MSCG can grow exponentially with the number of examples m. Although Haussler notes that heuristic methods for learning conjunctive concepts can be effective, the existing inductive algorithms (Vere 1980; Hayes-Roth and McDermott 1978; Watanabe and Rendell 1990) do not apply well to our operator learning problem. For example, Vere's counterfactual algorithm (Vere 1980) requires both positive and negative examples and is not incremental. Hayes-Roth's interference matching algorithm (Hayes-Roth and McDermott 1978) uses some complex heuristics to prune the space of all the matches between two examples, and thus prevent some operator preconditions from being learned. It also uses the one-to-one parameter binding assumption that does not hold for our operator preconditions (in general, two or more different variables can be bound to the same object). Watanabe and Rendell's X-search algorithm (Watanabe and Rendell 1990) prunes the search space by using connectivity constraint. However operator preconditions usually form only one connected component and thus preventing the heuristic from being effective. FOIL (Quinlan 1990), a greedy algorithm, does not apply well to our problem either, because FOIL requires both positive and negative training instances and it is not incremental.

OBSERVER uses an incremental algorithm to learn an operator's preconditions by building a general representation (**G-rep**) and a specific representation (**S-rep**) in a manner similar to the version spaces method (Mitchell 1978). The **S-rep** is a collection of literals that represents a specific boundary of the precondition expression being learned. The **S-rep** is updated both from from observations and during practice. The **G-rep** is a collection of literals that represents a general boundary of the precondition expression being learned. The **G-rep** is initialized to the empty set and is updated using negative examples OBSERVER generates during practice.

Learning operators from observation

During learning from observation, OBSERVER first initializes each operator using the first observation for the operator, then it refines each operator by learning the **S-rep** of operators preconditions and the operator effects.

Initializing the operators

Given the first observation of an operator, OBSERVER initializes the specific representation S-rep to the parameterized pre-state of the observation, and the effects of the operator to the parameterized delta-states. During parameterization, objects (except for *domain constants*) are generalized to typed variables. Domain constants are given to OBSERVER as part of the input, they are only generalized to a variable if OBSERVER notices a different constant in a later observation. The type for each variable in the operator is the most specific type in the type hierarchy for the corresponding object. See operator in Figure 6 learned from one observation in Figure 3 as an example. Note that

constants `width, rectangular, iron` etc are not generalized to variables. Also note that the operator learned from this one observation has some extraneous preconditions such as `(size-of <v1> height 2.75)`. They will later be generalize or removed with more observations and/or practice.

```
(operator hold-with-vise
  (preconds ((<v3> Drill) (<v2> Vise)
             (<v1> Part) (<v5> Spot-drill))
    (and (has-device <v3> <v2>)
         (is-available-table <v3> <v2>)
         (is-empty-holding-device <v2> <v3>)
         (holding-tool <v3> <v5>)
         (size-of <v1> width 2.75)
         (size-of <v1> height 4.25)
         (size-of <v1> length 5.5)
         (shape-of <v1> rectangular)
         (on-table <v3> <v1>)
         (hardness-of <v1> soft)
         (is-clean <v1>)
         (material-of <v1> copper)
         (is-available-part <v1>)
  (effects
   (add (holding <v3> <v2> <v1> side5))
   (del (on-table <v3> <v1>))
   (del (is-available-part <v1>))
   (del (is-empty-holding-device <v2> <v3>))))))
```

Figure 6: Learned operator HOLD-WITH-VISE when the observation in Figure 3 is given to OBSERVER. The preconditions shown in this figure is the S-rep of the preconditions of the operator HOLD-WITH-VISE.

Learning operators incrementally with new observations

After initializing the operators from the first observation, these initial operators are further generalized incrementally with more observations. OBSERVER refines the corresponding operator with each new observation as follows: first OBSERVER matches the effects of the operator against the delta-state of the new observation. The matching produces partial bindings for variables in the operator. Then OBSERVER uses the matching results to update the specific representation S-rep of the operator preconditions. And finally the effects of the operators are updated.

Learning variable bindings of the operator from observation

Learning the variable bindings can help reducing the ambiguity in finding the mapping between an operator and a new observation of the operator. OBSERVER learns bindings by matching the effects of the operator against the delta-state of the observation. In the presence of ambiguity during matching, this procedure chooses the mapping between the operator effects and the delta-states in the observation that leads to fewer conditional effects being learned. This implies that any effect that is unifiable with a unique element in the delta-state should be unified with it.

In addition to learning the bindings of the variables in the effects, OBSERVER generalizes a constant to a variable if this constant is unified with a different constant in the bindings. It also generalizes the type of a variable to the least common ancestor type of the type of the variable in the operator and the type of the corresponding object in the bindings.

For example, given the second observation of the operator HOLD-WITH-VISE shown in Figure 7, and the initial operator shown in Figure 6 that was learned from the first observation, OBSERVER matches the effects of the operator against the delta-state of the observation. `(holding <v3> <v2> <v1> side5)` is unified with `(holding milling-machine1 vise1 part1 side4)` to produce *bindings* = {part1/<v1>, vise1/<v2>, milling-machine1/<v3>, side4/side5}. The type of the variable `<v3>` is generalized to the least common type of milling-machine and drill, i.e. Machine, and the constant `side5` is generalized to a new variable `<v4>` whose type is `Side`.

Updating the S-rep from observations

Figure 10 shows the procedure **update_S_rep_from_obs** for updating the S-rep of the operator preconditions given an observation. OBSERVER updates the **S-rep** by removing from it those literals that are not present in the pre-state of the observation, and by generalizing *domain constants* to variables if different constants are used in the pre-state. Determining which preconditions in the S-rep are not present in the pre-state of the observation is non trivial when bindings for the variables are unknown. In fact, there are multiple generalizations under different mappings of variables to objects. OBSERVER's approach is to remove preconditions that are definitely not met in the pre-state (i.e. is not unifiable with any literal in the pre-state) under all the possible bindings for the variables in the operator, as long as they are consistent with the bindings returned by **learn_variable_bindings**.

How does OBSERVER know if a precondition is not unifiable with any literal in the pre-state? For each precondition *prec* in the **S-rep** of the operator, OBSERVER unifies *prec*, substituted with the bindings returned by **learn_variable_bindings**, with every literal in the pre-state of the observation. For every literal *lit* in the pre-state, let $\theta(\{l_1/p_1, l_2/p_2, ..., l_n/p_n\})$ = **unify**(*lit*, (**substitute**(*p*, *bindings*)). We call *lit* a **potential-match** of *prec* if and only if *lit* and *prec* are unifiable with respect to bindings *bindings*. **find_potential_matches**(*prec, obs, bindings*) finds all the potential matches for a precondition *prec*. If *prec* does not have any potential matches, that is, *prec* can not be present in the pre-state no matter what variable bindings are given, then it is removed from the S-rep. This usually happens when no literal in the pre-state has the same predicate as the precondition *prec*. If *prec* has exactly one potential match, then *prec* is generalized through unification with its potential-match in the pre-state. Otherwise, *prec* has several potential matches, and OBSERVER handles this ambiguity by keeping *prec* as it is in the S-rep.

As an example, given the second observation of the operator HOLD-WITH-VISE shown in Figure 7, and the initial operator shown in Figure 6 that was learned from the first observation, and *bindings* = {`<v1>/part1, <v2>/vise1, <v3>/milling-machine1, side5/side4`}, learned by matching

op-name: hold-with-vise

Pre-state:

```
(has-device drill0 vise0)
(on-table drill0 part0)
(is-clean part0)
(is-empty-holding-device vise0 drill0)
(is-available-table drill0 vise0)
(holding-tool drill0 spot-drill0)
(is-available-part part0)
(hardness-of part0 hard)
(material-of part0 iron)
(size-of part0 width 2.75)
(size-of part0 height 4.25)
(size-of part0 length 5.5)
(shape-of part0 rectangular)
```

Delta-state:

adds:

```
(holding drill0 vise0 part0 side5)
```

dels:

```
(is-empty-holding-device vise0 drill0)
(on-table drill0 part0)
(is-available-part part0)
```

Figure 3: An observation of the state before and after the application of the operator HOLD-WITH-VISE. Many literals in the pre-states are irrelevant information for applying the operator HOLD-WITH-VISE.

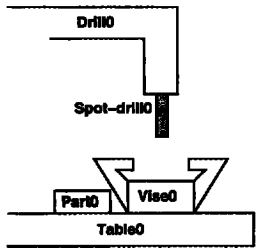

Figure 4: Pre-state of observation in Figure 3. Note that the part is on the table.

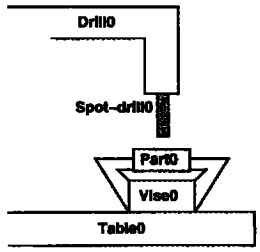

Figure 5: Post-state of observation in Figure 3. Note that the part is being held.

op-name: hold-with-vise

Pre-state:

```
(has-device milling-machine1 vise1)
(on-table milling-machine1 part1)
(is-available-table milling-machine1 vise1)
(is-empty-holding-device vise1 milling-machine1)
(is-available-tool-holder milling-machine1)
(is-available-part part1)
(is-clean part1)
(size-of part1 length 4)
(size-of part1 width 3)
(size-of part1 height 2.25)
(shape-of part1 rectangular)
(hardness-of part1 soft)
(material-of part1 bronze)
```

Delta-state:

adds:

```
(holding milling-machine1 vise1 part1 side4)
```

dels:

```
(is-empty-holding-device vise1 milling-machine1)
(on-table milling-machine1 part1)
(is-available-part part1)
```

Figure 7: The 2nd observation of the operator HOLD-WITH-VISE. The main differences with the first observation shown in Figure 3 are (1) the machine in use is a milling-machine instead of a drill, (2) there is no drill-tool being held by the machine, (3) the part is made of different material, and has different sizes.

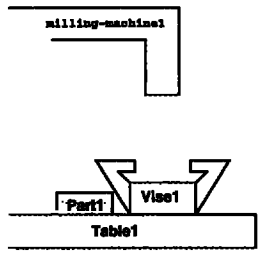

Figure 8: Pre-state of observation in Figure 7. The part is on the table.

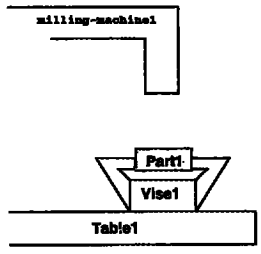

Figure 9: Post-state of observation in Figure 7. The part is being held by vise1.

procedure: update_S_rep_from_obs

Input: *op, obs, bindings*
Output: S-rep, the updated specific precondition representation

1. for each *prec* ∈ S-rep)
2. *potential-matches(prec)* ← **find_potential_matches**(*prec,pre-state(obs),bindings*)
3. if *potential-matches(prec)*=∅
4. then S-rep ← S-rep \{*prec*}
5. if *potential-matches(prec)* has 1 element
6. then S-rep ← (S-rep \{*prec*}) ∪**generalize**(*prec, bindings*)
7. if *potential-matches(prec)* has more than 1 element
8. then do not modify the S-rep

Figure 10: Updating the **S-rep**, a specific boundary of the operator preconditions, given an observation. This algorithm runs in polynomial time in terms of the number of literals in the pre-state or the preconditions.

```
(operator hold-with-vise
  (preconds ((<v3> Machine) (<v2> Vise)
             (<v1> Part) (<v4> Side))
   (and (has-device <v3> <v2>)
        (is-available-table <v3> <v2>)
        (is-empty-holding-device <v2> <v3>)
        (shape-of <v1> rectangular)
        (on-table <v3> <v1>)
        (is-clean <v1>)
        (hardness-of <v1> soft)
        (is-available-part <v1>)
  (effects
   (add (holding <v3> <v2> <v1> <v4>))
   (del (on-table <v3> <v1>))
   (del (is-available-part <v1>))
   (del (is-empty-holding-device <v2> <v3>))))
```

extraneous preconditions:

```
(holding-tool <v3> <v5>)
(size-of <v1> width <v6>)
(size-of <v1> height <v7>)
(size-of <v1> length <v8>)
(material-of <v1> <v10>)
```

Figure 11: Learned operator HOLD-WITH-VISE when the observations in Figures 3 and 7 are both given to OBSERVER. The preconditions listed here are the literals in the S-rep of the operator HOLD-WITH-VISE. Note that the type of variable <v3> has been generalized to Machine.

the effects with the delta-state, OBSERVER computes the *potential-matches* for each precondition in the specific representation of the operators giving *bindings*. For example, there are no *potential-matches* for `(holding-tool <v3> <v5>)` because none of the element in the pre-state has the predicate `holding-tool`, i.e. for every `i` in the pre-state **unify**(`i`, `(holding-tool <v3> <v5>)`) = NULL. Here are some examples of what **find_potential_matches** returns:

potential-match(`(holding-tool <v3> <v5>)`) = ∅;
potential-match(`(size-of <v1> width 2.75)`)
= {`(size-of part1 width 3)`};

Therefore, `(holding-tool <v3> <v5>)` is removed from S-rep. `(size-of <v1> width 2.75)` is generalized to `(size-of <v1> width <anything>)` by unifying it with `(size-of part1 width 3)`. Since `<anything>` is not constrained by any other preconditions in S-rep, OBSERVER learns that the width of a part can be any value, and therefore is irrelevant to the operator and is removed from the S-rep. Similarly, preconditions `(size-of <v1> height 2.75)`, `(size-of <v1> length 5.5)`, `(material-of <v1> copper)` are removed from the preconditions. The modified operator is shown in Figure 11, along with the extraneous preconditions that are removed from the initial operator.

Updating the effects from observations

Figure 12 describes the procedure **update_effects_from_obs** for updating the effects of the operator from an observation. OBSERVER first generalizes those effects that have unambiguous matches in the delta-state (steps 4-8). Steps 9–17 describes how OBSERVER learns conditional effects. For every effect of the operator, if it does not unify with any element in the delta-state of the observation, then it is learned as a conditional effect. The specific representation of the preconditions of this conditional effect is initialized to the **S-rep** of the preconditions of the operator (steps 9–12). Every element in the delta-state that does not unify with any effect is also learned as a conditional effect (steps 13–17), and its specific representation for the preconditions of this conditional effect is the parameterized pre-state of the current observation. OBSERVER does not update the effects that have ambiguous mappings with elements in the delta-state.

To illustrate how OBSERVER generalizes an effect, let's look at the learned operator HOLD-WITH-VISE as shown in Figure 6 learned from the first observation, and the second observation shown in Figure 7. Matching the effects of the operator against the delta-state of the observation generates the following bindings: {part1/<v1>, vise1/<v2>, milling-machine1/<v3>, side4/side5}. The effect `(add (holding <v3> <v2> <v1> side5))` is generalized to `(add (holding <v3> <v2> <v1> <v4>))` because the constant `side5` in the effect is substituted with a different constant `side4` in the delta-state. Other effects are kept as they are. Also since the mapping between the effects and the *delta-state* is unambiguous, no conditional effects are learned. The updated operator is shown in Figure 11.

The next example illustrates how conditional effects are learned. Given the third observation shown in Figure 13, and the operator that are learned from the first two observations as shown in Figure 11, OBSERVER matches the effects of the operator against the delta-states of the third observation and produces bindings {drill2/<v3>, part2/<v1>, vise2/<v2>}. Effects `(del (on-table <v3> <v1>))`, `(del (is-available-part <v1>))`, and `(del (is-empty-holding-device <v2> <v3>))` have unambiguous matches in the delta-states, i.e. `(del (on-table drill2 part2))`, `(del (is-available-part part2))`, and `(del (is-empty-holding-device vise2 drill2))`, respectively. Effect `(add (holding <v3> <v2> <v1> <v4>))` is not unifiable

procedure: update_effects_from_obs

Input: *bindings*, learned operator *op*, new observation *obs*.
Output: modified effects(op).

1. *effects* ← *Effects*(*op*).
2. *deltas* ← *delta-state*(*obs*).
3. for every $e \in$ **Effects**(*op*)
4. if there $\exists\ d \in$ *deltas* s.t. **is_unambiguous_pair**(*e,d*) then
5. $e' \leftarrow$ **generalize**(*e, bindings*).
6. replace *e* with e'.
6. if *e* is a conditional effect, update its conditional preconditions
7. *effects* ← *effects* \ {*e*}
8. *deltas* ← *deltas* \ {*d*}
9. for every $e \in$ *effects*
10. if $\forall\ d \in$ *deltas*, **unify**(*e, d1*) = NULL then ;;; learning a conditional effects
11. **conditional_effects**(*op*) ← **conditional_effects**(*op*) ∪ *e*
12. S-rep(e) ← S-rep(op)
13. for every $d \in$ *deltas*,
14. if $\forall\ e1 \in$ *effects*, **unify**(*e1, d*) = NULL then ;;; learning a conditional effects
15. *new-effect* = **parameterize**(*d*)
16. **conditional_effects**(*op*) ← **conditional_effects**(*op*) ∪ {*new-effect*}
17. *S* − *rep*(*new-effect*) ← **parameterize**(*pre-state*(*obs*))

Figure 12: Updating the effects of an operator based on the new observation. Every effect that has a unique match with the delta-state is generalized according to the match. Every effect that is not unifiable with any element in the delta-state is learned as a conditional effect. Every element in the delta-state that is not unifiable with any effect is also learned as a conditional effect.

with any literal in the delta-state, and therefore is learned as a conditional effect of the operator. Delta-state `(add (holding-weakly drill2 vise2 part2 side0))` is not unifiable with any effect of the operator, and therefore is parameterized and learned as a conditional effect. The resulting operator with conditional effect is shown in Figure 16.

```
(operator hold-with-vise
  (preconds ((<v3> Machine) (<v2> Vise)
             (<v1> Part) (<v4> Side))
    (and (has-device <v3> <v2>)
         (is-available-table <v3> <v2>)
         (is-empty-holding-device <v2> <v3>)
         (on-table <v3> <v1>)
         (is-clean <v1>)
         (hardness-of <v1> soft)
         (is-available-part <v1>)
  (effects (<v6> spot-drill))
    (if (and (size-of <v1> diameter 4.75)
             (size-of <v1> length 5.5)
             (shape-of <v1> cylindrical)
             (hardness-of <v1> soft)
             (material-of <v1> bronze)
             (holding-tool <v3> <v6>))
        (add (holding-weakly <v3> <v2> <v1> side0)))
    (if (and (shape-of <v1> rectangular)
             (has-device <v3> <v2>)
             (is-available-table <v3> <v2>)
             (is-empty-holding-device <v2> <v3>)
             (on-table <v3> <v1>)
             (is-clean <v1>)
             (is-available-part <v1>)
        (add (holding <v3> <v2> <v1> <v4>)))
    (del (on-table <v3> <v1>))
    (del (is-available-part <v1>))
    (del (is-empty-holding-device <v2> <v3>)))))
```

extraneous preconditions:

```
(holding-tool <v3> <v5>)
(size-of <v1> width <v6>)
(size-of <v1> height <v7>)
(size-of <v1> length <v8>)
(material-of <v1> <v10>)
(shape-of <v1> <v11>)
```

Figure 16: Operator HOLD-WITH-VISE after learning from observations shown in Figures 3, 7, 13. Note the conditional effects and conditional preconditions.

Planning while learning

The operators learned from observation can be incomplete and incorrect in several ways. The planning system must be able to plan using these operators during practice. This challenges planning in the following ways:

1. Classical planners presume a correct domain model. In our learning system however, the newly acquired operators are possibly incomplete and incorrect. How can the planner generate plans to solve practice problems?
2. Because of incomplete and incorrect operators used during practice, the plans generated by the planner may be incorrect, which in turn may lead to execution failures. Thus plan repair upon execution failure is necessary. How can the planner effectively repair the incorrect plan using incomplete and incorrect operators?
3. How should planning and execution be interleaved so that the system can solve practice problems effectively

Pre-state:

```
(has-device drill2 vise2)
(on-table drill2 part2)
(is-available-table drill2 vise2)
(is-clean part2)
(is-empty-holding-device vise2 drill2)
(holding-tool drill2 spot-drill2)
(is-available-part part2)
(size-of part2 diameter 4.75)
(size-of part2 length 5.5)
(material-of part2 bronze)
(hardness-of part2 soft)
(shape-of part2 cylindrical)
```

Delta-state:

adds:

```
(holding-weakly drill2 vise2 part2 side0)
```

dels:

```
(is-empty-holding-device vise2 drill2)
(on-table drill2 part2)
(is-available-part part2)
```

Figure 13: The 3rd observation of the operator HOLD-WITH-VISE. Note that the effect has "holding-weakly" instead of "holding" as in previous observations. OBSERVER thus learns conditional_effects.

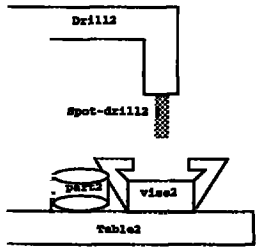

Figure 14: Pre-state of observation in Figure 13

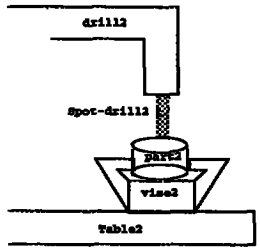

Figure 15: Post-state of observation in Figure 13

and concurrently generate learning opportunities for refining operators using incomplete and incorrect domain knowledge?

Our approach to address the above issues is to use the **G-rep** of the operator preconditions for planning. The individual plans generated for achieving some top-level goals achieve the preconditions in the **G-rep** of each operator, but do not require achieving preconditions in the **S-rep** of the operators. This has the advantage of generating an initial plan quickly and of generating opportunities for operator refinement. While executing an operator in the environment, there are the following two possible outcomes, generating negative or positive training instances, respectively, for refining operators.

- The first outcome is that the state does not change after executing *op*. In this case, we say that *op* is executed *unsuccessfully*.

  An operator may execute unsuccessfully because OBSERVER achieves the preconditions in the **G-rep** of each operator during planning without necessarily achieving all the preconditions in the **S-rep**. This introduces the possibility of incomplete or incorrect plans in the sense that a real precondition may be unsatisfied. Unsuccessful executions form the negative examples that OBSERVER uses for refining operators as discussed in (Wang 1995). Upon each unsuccessful execution, OBSERVER updates the **G-rep** by learning *critical preconditions* using *near misses* of negative examples. OBSERVER also attempts to generate a *repaired-plan*. If such a plan is found, OBSERVER continues execution using the repaired plan; otherwise, OBSERVER removes the failed operator and continues execution with the remaining plan.
- The other possible outcome is that the state changes after executing *op*. In this case, we say that *op* is executed *successfully*.

  Successful executions form the positive examples that OBSERVER uses for refining operator as described in (Wang 1995). Note that a successful execution may still have incorrect predictions of how the state changes due to the possibility of incomplete operator effects. Upon each successful execution, OBSERVER updates the **S-rep** by removing from it the preconditions that are not satisfied in *current-state*, and updates the effects of *op* if missing effects are observed in the *delta-state* of the execution. OBSERVER then updates the current state and continues execution with the remaining plan.

Detailed descriptions of the algorithms for planning with incomplete and incorrect operators and plan repair are described in (Wang 1996b).

Refining operators during practice

This section describes how OBSERVER refines operators using the successful and unsuccessful executions generated during practice. This involves updating the **S-rep** of operator preconditions, learning the **G-rep** of the operator preconditions, and updating the effects, including conditional and conditional preconditions.

Updating the S-rep of operator preconditions

During observations, OBSERVER cannot observe the bindings for the variables in the operators. However, during practice, the bindings are determined during planning. Therefore, when refining operators during practice, OBSERVER no longer needs to learn the bindings through matching the effects with the delta-state. This eliminates the ambiguity in the matching process that is necessary for learning from observations. Therefore, removing extrane-

ous preconditions from the S-rep of the operator preconditions during practice is relatively straightforward

OBSERVER may also learn negated preconditions upon unsuccessful executions: if all the preconditions in the **S-rep** are satisfied in the pre-state, and the operator fails to apply, then the failure is due to the existence of negated preconditions. The potentially negated preconditions are those literals that are true in the pre-state of the execution but are not specified in the **S-rep** of the operator preconditions, nor in the extraneous preconditions previously removed from the operator. OBSERVER conjectures the negations of these literals as negated preconditions, and adds them to the **S-rep**. Some of the negated preconditions thus added are extraneous preconditions, and are removed from the **S-rep** accordingly using the algorithm for updating the **S-rep**.

Learning the G-rep of operator preconditions

The **G-rep** of an operator is learned incrementally as OBSERVER confirms elements of the **S-rep** to be true preconditions of the operator. The **G-rep** is only updated when the negative example is a *near miss*. A near miss is a negative example where only one literal in the **S-rep** is not satisfied in the state. *Critical-preconditions* are learned from near miss negative example. A *critical-precondition p* of an operator *op* is a precondition such that there exists a pair of states s_1 and s_2 that OBSERVER encountered during practice such that: (1) p is satisfied in s_1 but not in s_2; (2) everything else in s_1 and s_2 are identical; (3) *op* is applicable in s_1 but not in s_2. The **G-rep** is specialized by adding to it the *critical-preconditions*. More details for learning the **G-rep** can be found in (Wang 1996a).

Refining operator effects

The procedure for refining operator effects during practice is very similar to the procedure **update_effects_from_obs** described in Figure 12 for updating the effects based on observations. The main difference is that when learning from executions, there is no ambiguity in variable bindings, and therefore learning is more efficient.

Empirical results

The learning algorithm described in this paper has been fully implemented on top of the PRODIGY4.0 architecture. In this section, we present empirical results to demonstrate that OBSERVER learns operators in a process planning well enough to solve problems as effectively as using human-expert coded operators in terms of the total number of solvable test problems. The human expert coded operators are encoded in a period of 6 months by an AI expert through interactions with domain experts. We also present empirical results to show that both learning from observation and learning from practice contribute significantly to the learning process.

Experiment set up

Our experiments include the following three phases:

1. *Learning operators from observation phase,* where OBSERVER learns operators from expert solutions traces.
2. *Refining operators during practice phase,* where OBSERVER refines operators during practice.
3. *Testing phase,* where the learned operators are compared against human expert-coded operators in terms of the number of solvable problems in a set of randomly generated test problems.

OBSERVER is implemented in the context of PRODIGY4.0(Carbonell *et al.* 1992). PRODIGY4.0 is a non-linear operator-based planner. There are many choice points for plannings: choice points for choosing a goal, for choosing an operator to achieve the goal, etc. In the absence of control knowledge, PRODIGY4.0's default strategy is to always choose the first alternative. This pre-determined order is arbitrary but introduces a systematic bias in search time that we want to factor out. In order for the planner not to be biased by a pre-determined ordering of operators, we introduced a random selection mechanism at each choice point in our experiments, and ran the planner multiple times to measure average performance.

The effectiveness of the overall learning system

OBSERVER first learns 33 new operators from observed solutions of 100 problems. Then it is given a set of 180 problems to refine operators from practice. OBSERVER's performance on a test set of 34 problems that are *different* from the learning problems is measured by the total number of solvable test problems using the learned operators. We measure the performance 5 times and computes the averages at the end of observationk, and after every 15 practice problems.

Figure 17 compares the total number of problems solved using learned operators and expert human-codes operators. The spikes of the curve are due to the fact that the planner makes random choices at each decision point. The curve indicates that after observation and practice, the total number of solvable problems are the same using OBSERVER-learned operators or expert human-coded operators.

Previous results given in (Wang 1995) showed that the learned operators are as effective as the expert human-coded operators in problem solving according to two other criteria: (*i*) the average amount of planning time to solve each problem, and (*ii*) the average length of the plans generated to solve each problem.

The role of observation

To evaluate the role of observation, OBSERVER is run multiple times. During each run, OBSERVER is given the same problems during practice, but *different number* of initial observation problems that are randomly drawn from a fixed set of observation problems. We measure the total number of solvable test problems after observation and practice, given *different number* of initial observation problems.

Figure 18 shows how the total number of problems that solvable after the same set of practice problems is given, but

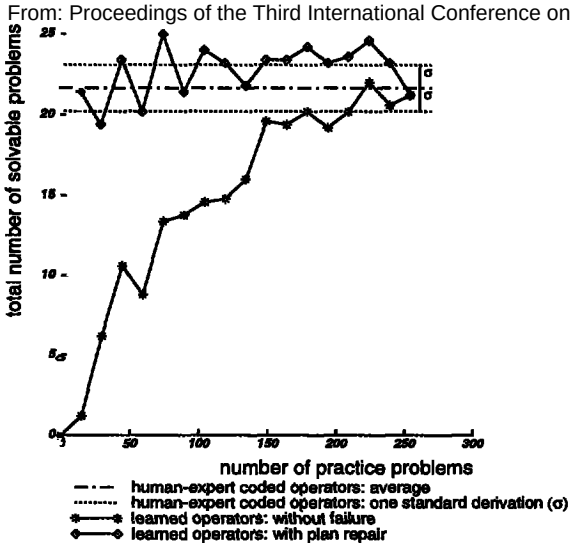

Figure 17: Total number of solvable test problems as a function of the number of practice problems in training.

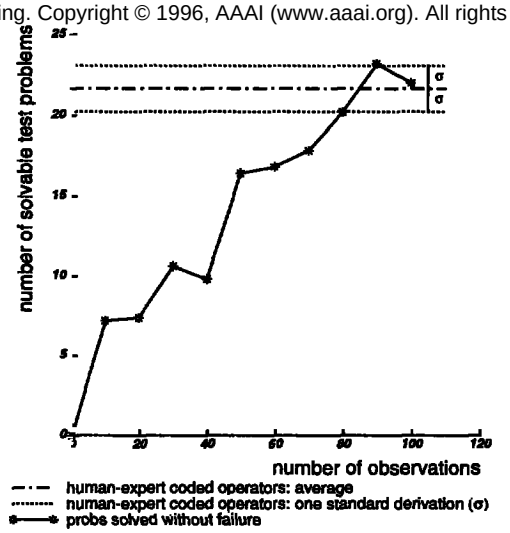

Figure 18: Total number of problems solvable with the same set of practice problems, but with different amount of observation before practice, as a function of the number of initial observation problems in training. We see that after 80 observation training problems in the process planning domain, more observation problems do not significantly increase the solvability. This indicates the right point to stop learning from observation, and to start practicing.

expert solutions traces of different number of observation problems are given to OBSERVER initially.

The role of practice

In order to demonstrate the role of practice, we show that OBSERVER is able to learn operators better and faster if it practices using the planning and plan repair algorithms described in this paper, after initial operator acquisition by observation, than if it is given only the observations of expert solutions of the total sequence of problems (both observation sequence and practice problems presented as additional observations).

During evaluation, OBSERVER is first given expert solutions traces for a set of observation problems. OBSERVER thus learns a set of operators from observation. These operators may be incomplete and incorrect in a number of ways. We then compare how OBSERVER refines these learned operators in the following two scenarios in terms of total number of solvable test problems.

scenario 1: OBSERVER is given a set of problems to practice. During practice, OBSERVER uses the planning and plan repair algorithms described in this paper to generate plans to solve the practice problems, and refines the initial incorrect and incomplete operators using both successful and unsuccessful executions.

scenario 2: OBSERVER is given the same set of problems. However, OBSERVER is only allowed to refine the operators based on the expert solutions for these problems, and is not allowed to practice searching for its own solutions.

The evaluation of solvable test problems includes the following two cases:

1. The total number of solvable test problems, including problems solved where executions failures occur but are subsequently repaired using our plan repair algorithm.
2. The total number of solvable test problems without execution failure, and therefore without requiring plan repair.

Figure 19 illustrates the comparison of the learned operators that are refined through practice (scenario 1), with the operators that are refined based on observation only (scenario 2), in terms of the total number of solvable problems in the test set *without execution failure*. This figure shows that:

- With practice where OBSERVER uses the planning and plan repair algorithms in this paper, the total number of solvable test problems without execution failure increases steadily as the number of training problems increases.
- If OBSERVER is only given the expert solution traces of the same problems, OBSERVER can not solve any test problem without failure.

Figure 20 illustrates the comparison of the operators that are further refined through practice (scenario 1) with the operators that are learned and refined based on observations (scenario 2) only, in terms of the total number of solvable test problems *without execution failure, and therefore without requiring plan repair.* This figure shows that:

- In both scenarios, the total number of solvable test problems increases as the number of training problems increases, whether OBSERVER uses the training problems for practice or as observations;
- The total number of solvable problems increases much faster in scenario 1, where OBSERVER practices using our planning and plan repair algorithms, than in scenario 2, where OBSERVER is only given the expert solution traces of the same problems.

Figure 19: *Total number of solvable test problems without execution failure during testing, as a function of the number of practice problems in training.* This figure compares the operators learned from scenario 1 (OBSERVER practices) with operators learned in scenario 2 (OBSERVER does not practice, and learns from observation only). In the comparison, OBSERVER first learns a set of operators from observation in both scenarios. Then OBSERVER refines operators from practice in scenario 1, and OBSERVER learns from observations only in scenario 2.

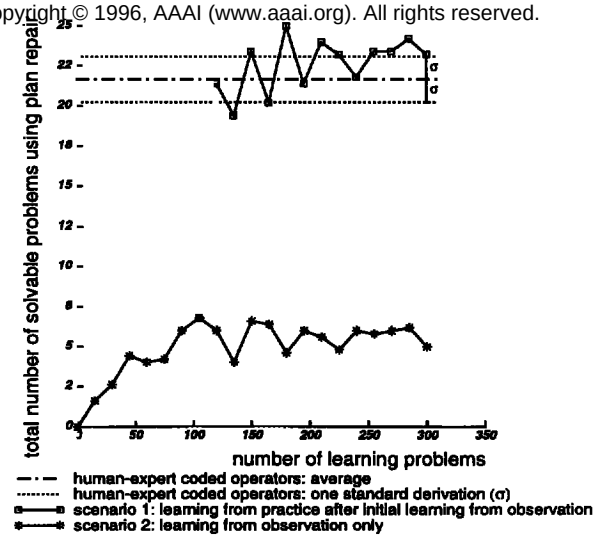

Figure 20: *Total number of solvable test problems allowing execution failure and plan repair during testing, as a function of the number of practice problems in training.* This figure compares the operators learned from scenario 1 (OBSERVER practices) with operators learned in scenario 2 (OBSERVER does not practice, and learns from observation only). In the comparison, OBSERVER first learns a set of operators from observation in both scenarios. Then OBSERVER refines operators from practice in scenario 1, and OBSERVER learns from observations only in scenario 2.

The above results can be explained by the fact that OBSERVER uses the **G-rep** for planning, and that the correctness of the plans generated depends on how well OBSERVER has learned the **G-rep**. After practice, OBSERVER is able to update the **G-rep** of the operator preconditions based on its own executions (scenario 1). However, without practice, OBSERVER can never update the **G-rep** (scenario 2) — observations of expert solutions do not have negative examples, and the **G-rep** can only be made more specific using negative training examples.

Comparing Figure 20 and Figure 19, we see that in scenario 1 where OBSERVER practice, although the number of solvable test problems given plan repair during testing increases faster than without plan repair, the total number of solvable test problems in both cases for the total number of solvable test problems are comparable at the end of learning. However, in scenario 2 where OBSERVER only learns from observation, more test problems can be solved where plan repair is used when solving test problems than if plan repair is not used when solving test problems even at the end of learning. This indicates that plan repair plays an important role when solving problems with incomplete and incorrect operators.

Related work

LEX(Mitchell *et al.* 1983) is a system that learns heuristic problem-solving strategies through experience in the domain of symbolic integration. It starts with exact preconditions for the operators, but lacks preconditions (control rules) for when should the operators be applied. The control knowledge is learned from experience as additional preconditions. OBSERVER is learning the preconditions of the basic domain operators, not search heuristics, and OBSERVER also learns from observations of expert solution traces.

LIVE (Shen 1994) is a system that learns and discovers from the environment. It integrates action, exploration, experimentation, learning, and problem solving. However it does not learn from observing others. LIVE has only been demonstrated in simplistic domains (e.g. the Little Prince World with only about 15 possible states), and does not scale up to large complex domains OBSERVER has been applied to. LIVE also avoids the complexity of planning with incomplete and incorrect operators by using a set of *domain-dependent search heuristics* during planning. These heuristics are part of the input to LIVE. OBSERVER differs in that it deals explicitly with imperfect operators without relying on any human-coded, domain-dependent heuristics.

EXPO (Gil 1992) is a learning-by-experimentation module for refining incomplete domain knowledge. Learning is triggered when plan execution monitoring detects a divergence between internal expectations and external observations. The initial knowledge given to EXPO and OBSERVER is different in that EXPO is given operators with over-general preconditions and incomplete effects, whereas OBSERVER does not have any knowledge about the preconditions or the effects of the operators. OBSERVER learns

from observations of expert solution traces that EXPO does not need to address, since EXPO already starts with workable, if incomplete operators. EXPO designs experiments to determine which preconditions and effects are missing from the operators. During execution failure, EXPO resets the problem to its original initial state. This differs from OBSERVER in that planning, execution, and plan repair form a closed loop in OBSERVER.

Benson's system (Benson 1995) learns actions models from its own experience and from its observation of a domain expert. Benson's work uses an action model formalism that is suited for reactive agents, which is different from STRIPS operators. It does not have a complicated planning and plan repair mechanism, and relies on an external teacher when an impasse is reached.

In DesJardins' work (desJardins 1994), users enter partially specified operators that reflect a rough description of how a subgoal may be solved. These operators usually have predicates with underspecified arguments. The learning system inductively fills these in by generalizing from feedback from evaluation models and from the user's planning. OBSERVER differs in that it starts with minimal initial knowledge, and must learn the preconditions themselves as well as the arguments in each precondition.

Future work and conclusions

We have presented a novel multistrategy learning method to automatically learn planning operators by observation and practice. Unlike previous approaches, our approach does not require a considerable amount of direct interactions with domain experts, or initial approximate planning operators, or strong background knowledge. We have shown that our system learns operators in a large process planning domain well enough to solve problems as effectively as human-expert coded operators, and that both strategies, i.e., learning operators from observation, and refining operators during practice, contribute significantly to the learning process.

Future work involves extending the learning algorithms to handle uncertainty in the domains. There are several sources of uncertainties that are possible in a domain. The first is perception noise in which observation of states may be incorrect — some properties may be missing in the observation, while some maybe erroneously observed, and others incorrectly observed. The second form of uncertainties is in the operators — operators may have effects that occur only with certain probability in a manner not fully predictable by the planner. The third type of uncertainty comes from external exogenous events that change the state.

In order to develop a learning system to handle uncertainty in the domain, one must first study what types of noise is present. Assuming no noise in the domain enables OBSERVER to converge fast in complex domains (e.g. two to three hundreds problems suffices for the learning to converge in the complex process planning domain). However, our framework for learning operators from observation and practice is still valid for acquiring operators in the presence of noise. The learning algorithm can be extended to handle noise by, for example, maintaining an occurrence count of each literal in the pre-state of the observations or practice, and only removing preconditions from the specific representation of the corresponding operator if the occurrence count is lower than a pre-specified frequency (the frequency is a function of the maximal level of noise the system will tolerate), and only adding a precondition to the general representation if the occurrence count is higher than a pre-specified frequency. But the learning rate will be much slower if there is noise in the domain. Some recent work has started research along this direction (Oates and Cohen 1996; Tae and Cook 1996). However, these approaches have only been applied to simple domains.

Our work makes two major contributions. First, it demonstrates that inductive learning techniques can be applied effectively to the problem of learning complex action descriptions - an open question for the machine learning community. Second, our work on automatic learning of planning operators provides a good starting point for fielding planning systems as it addresses a key bottleneck for this endeavor, namely, the knowledge acquisition problem.

References

Y. Anzai and H. A. Simon. The theory of learning by doing. *Psychological Review*, 86:124–140, 1979.

S. Benson. Inductive learning of reactive action models. In *Proceedings of 12th International Conference on Machine Learning*, Tahoe City, CA, July 1995.

J. H. Boose and B. R. Gaines. Knowledge acquisition for knowledge-based systems: Notes on the state-of-the-art. *Machine Learning*, 4:377–394, 1989.

Jaime G. Carbonell, and the PRODIGY Research Group: Jim Blythe, Oren Etzioni, Yolanda Gil, Robert Joseph, Dan Kahn, Craig Knoblock, Steven Minton, Alicia Pérez, (editor), Scott Reilly, Manuela Veloso, and Xuemei Wang. PRODIGY4.0: The manual and tutorial. Technical Report CMU-CS-92-150, School of Computer Science, Carnegie Mellon University, June 1992.

D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32:333–378, 1987.

S. Chien, R. Jr Hill., X. Wang, T. Estlin, K. Fayyad, and H. Mortensen. Why real-world planning is difficult: A tale of two applications. In M. Ghallab, editor, *Advances in Planning*. IOS Press, 1995.

S. Chien. Towards an intelligent planning knowledge base development environment. In *AAAI-94 Fall Symposium Series: Planning and Learning: On to Real Applications*, New Orleans, LA, 1994.

S. Chien. Static and completion analysis for planning knowledge base development and verification. In *Proceedings of the Third International Conference on AI Planning Systems*, Edinborgh, Scotland, 1996.

M. desJardins. Knowledge development methods for planning systems. In *AAAI-94 Fall Symposium Series: Planning and Learning: On to Real Applications*, New Orleans, LA, 1994.

R. E. Fikes and N. J. Nilsson. STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3,4):189–208, 1971.

Yolanda Gil. A specification of process planning for PRODIGY. Technical Report CMU-CS-91-179, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, August 1991.

Y. Gil. *Acquiring Domain Knowledge for Planning by Experimentation*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1992.

D. Haussler. Learning conjunctive concepts in structural domains. *Machine Learning*, 4:7–40, 1989.

F. Hayes-Roth and J. McDermott. An interference matching technique for inducing abstractions. In *CACM*, volume 26, pages 401–410, 1978.

D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, 1991.

T. Mitchell, P. Utgoff, and R. Banerji. Learning by experimentation: Acquiring and refining problem-solving heuristic. In *Machine Learning, An Artificial Intelligence Approach, Volume I*. Tioga Press, Palo Alto, CA, 1983.

T. Mitchell. *Version Spaces: An Approach to Concept Learning*. PhD thesis, Stanford University,, 1978.

T. Oates and P. Cohen. Searching for planning operators with context-dependent and probabilistic effects. In *Proceedings of the 13th National Conference on Artificial Intelligence*, Portland, Oregon, August 1996. AAAI Press/The MIT Press.

J.S. Penberthy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proceedings of KR-92*, 1992.

R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

W. Shen. *Autonomous Learning from the Environment*. Computer Science Press, W.H. Freeman and Company, 1994.

K. Tae and D. Cook. Experimental knowledge-based acquisition for planning. In *Proceedings of 13th International Conference on Machine Learning*, Bari, Italy, July 1996.

Manuela Veloso and Peter Stone. Flecs: Planning with a flexible commitment strategy. *Journal of Artificial Intelligence Research*, 3, 1995.

M. Veloso, J.G. Carbonell, M. A. Pérez, D. Borrajo, E. Fink, and J. Blythe. Integrating planning and learning: The PRODIGY architecture. *Journal of Experimental and Theoretical Artificial Intelligence*, 7(1), January 1995.

S. A. Vere. Multilevel counterfactuals for generalizations of relational concepts and productions. *Artificial Intelligence*, 14:139–164, 1980.

X. Wang. Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of 12th International Conference on Machine Learning*, Tahoe City, CA, July 1995.

X. Wang. *Learning Planning Operators by Observation and Practice*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 1996.

X. Wang. Planning while learning operators. In *Proceedings of the Third International Conference on AI Planning Systems*, Edinborgh, Scotland, 1996.

L. Watanabe and L. Rendell. Effective generalization of relational descriptions. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, Boston, MA, July 1990. AAAI Press/The MIT Press.