Learning Planning Operators by Observation and Practice

Xuemei Wang*
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213
wxm+@cs.cmu.edu

Abstract

The work described in this paper addresses learning planning operators by observing expert agents and subsequent knowledge refinement in a learning-by-doing paradigm. The observations of the expert agent consist of: 1) the sequence of actions being executed, 2) the state in which each action is executed, and 3) the state resulting from the execution of each action. Planning operators are learned from these observation sequences in an incremental fashion utilizing a conservative specific-to-general inductive generalization process. In order to refine the new operators to make them correct and complete, the system uses the new operators to solve practice problems, analyzing and learning from the execution traces of the resulting solutions or execution failures. We describe techniques for planning and plan repair with incorrect and incomplete domain knowledge, and for operator refinement through a process which integrates planning, execution, and plan repair. Our learning method is implemented on top of the PRODIGY architecture(Carbonell, Knoblock, & Minton 1990; Carbonell *et al.* 1992) and is demonstrated in the extended-strips domain(Minton 1988) and a subset of the process planning domain(Gil 1991).

The Challenge

There is considerable interest in learning for planning systems. Most of this work concentrates on learning search control knowledge (Fikes, Hart, & Nilsson 1972; Mitchell, Utgoff, & Banerji 1983; Minton 1988; Veloso 1992). Previous work on acquiring planning operators refines incomplete operators by active experimentation(Gil 1992). This paper describes a framework for learning planning operators by observing expert agents and subsequent knowledge refinement in a learning-by-doing paradigm(Anzai & Simon 1979).

*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.

The learning system is given at the outset the description language for the domain, which includes the types of objects and the predicates that describe states and operators. The observations of an expert agent consist of: 1) the sequence of actions being executed, 2) the state in which each action is executed, and 3) the state resulting from the execution of each action. Our planning system uses STRIPS-like operators(Fikes & Nilsson 1971) and the goal of this work is to learn the preconditions and the effects of the operators. Operators are learned from these observation sequences in an incremental fashion utilizing a conservative specific-to-general inductive generalization process. In order to refine the new operators to make them correct and complete, the system uses them to solve practice problems, analyzing and learning from the execution traces of the resulting solutions.

Learning by observation and by practice provides a new point in the space of approaches for problem solving and learning. There are many difficulties for such integration of learning and planning, to name a few: there are many irrelevant features in the state when the operators are applied; the planner needs to plan with the newly acquired, possibly incomplete and incorrect operators; plan repair is necessary when the initial plan cannot be completed because of some unmet necessary preconditions in the operators in the plan; planning and execution must be interleaved and integrated. This paper describe our approach to address these problems. Techniques for planning and plan repair with partially incorrect and incomplete domain knowledge are developed, and operators are refined during an integration of planning, execution, and plan repair. The PRODIGY architecture(Carbonell, Knoblock, & Minton 1990; Carbonell *et al.* 1992) is the test-bed for our learning method.

Our learning method applies when the domains can be modelled with discrete actions, the states are observable, the description language for representing the states and the operators (i.e. the set of predicates) is known. In many cases, such as in the process planning domain (Gil 1991), learning the operator representations is a difficult and important task. Our method can significantly reduce the effort of knowledge engineering in such cases [1].

[1]Note that the term "observation" refers to a form of input to the learning system. It can come from visual or audio input, or it can be input to the system typed in by a person, or it can be input

Learning Planning Operators – the Architecture

The architecture for learning by observation and practice in planning include *the observation module, the learning module, the planning module,* and *the plan execution module.* Figure 1 illustrates the functionality of each module and the information transferred among these modules.

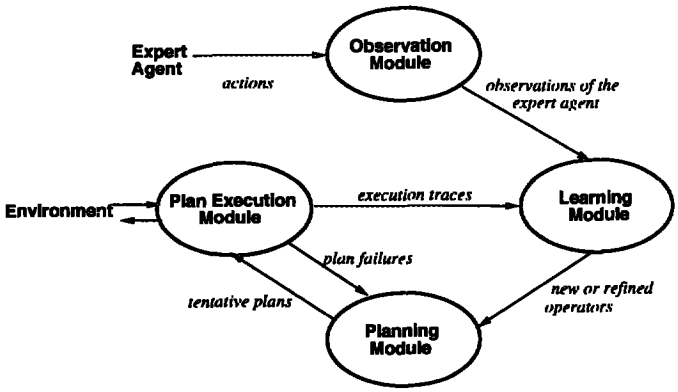

Figure 1: Overview of the data-flow among different modules in the learning system. The observation module provides the learning module with observations of the expert agent. The learning module formulates and refines operators in the domain, and these operators are used by the planning module. The planning module is essentially the PRODIGY planner modified to use some heuristics for planning with incomplete and incorrect operators as well as for plan repair, it generates tentative plans to solve practice problems. The plan execution module executes the plans, providing the learning module with the execution traces, and passing the plan failures to the planning module for plan repair.

This work makes the following assumptions:

1. The operators and the states are deterministic. If the same operator with the same bindings is applied in the same states, the resulting states are the same.
2. Noise-free sensors. There are no errors in the states.
3. The preconditions of operators are conjunctive predicates.
4. Everything in the state is observable.
5. There are no conditional effects.

The Observation Module

The observation module monitors the execution traces of the expert agents. An *observation* of the other agent consists of the name of the operator being executed, the state before the operator is executed, called *pre-state*, and the state after the operator is executed, called *post-state*. Computing the difference between the *post-state* and *pre-state*, we obtain a *delta-state*. An *episode* is a sequence of *observations*.

read in from a case library.

The Learning Module

The learning module formulates and refines the representations of the operators in the domain from the observations of the expert agent and the execution traces of the system's own actions. Operators are learned in an incremental fashion utilizing a conservative specific-to-general inductive generalization process and a set of domain-independent heuristics. At any point of time during learning, any learned operator always has all the correct preconditions, but it may also have some unnecessary preconditions. These unnecessary preconditions can be removed with more observations and practice. The learning module also learns heuristics for subgoal ordering from the observed episodes of the expert agent.

Learning Operators from Observations Given an observation, if the action is observed for the first time, we create the corresponding operator such that its precondition is the parameterized *pre-state*, and its effect is the parameterized *delta-state*. The type for each variable in the operator is the most specific type in the type hierarchy that the corresponding parameterized object has[2]. This is the most conservative constant-to-variable generalization. Generalizing the observation in Figure 2 yields the operator shown in Figure 3.

Pre-state:

(connects dr13 rm1 rm3) (connects dr13 rm3 rm1)
(dr-to-rm dr13 rm1) (dr-to-rm dr13 rm3)
(inroom robot rm1) (inroom box1 rm3) (pushable box1)
(dr-closed dr13) (unlocked dr13) (arm-empty)

Delta-state: add: (next-to robot dr13)

Figure 2: An observation of the state before and after the application of the operator GOTO-DR

```
(operator goto-dr
  (preconds ((<v1> door) (<v2> object) (<v3> room) (<v4> room))
   (and (inroom robot <v4>)
        (connects <v1> <v3> <v4>)
        (connects <v1> <v4> <v3>)
        (dr-to-rm <v1> <v3>)
        (dr-to-rm <v1> <v4>)
        (unlocked <v1>)
        (dr-closed <v1>)
        (arm-empty)
        (inroom <v2> <v3>)
        (pushable <v2>)))
  (effects nil ((add (next-to robot <v1>)))))
```

Figure 3: Learned operator GOTO-DR when the observation in Figure 2 is given to the learning module

Given a new observation, when the system already has an operator representation, the preconditions are generalized by removing facts that are not present in the *pre-state* of the

[2]Objects that do not change from problem to problem are treated as constants and are not parameterized, such as the object ROBOT in the extended-strips domain.

new observation[3]; the effects are augmented by adding facts that are in the *delta-state* of the observation. For example, given the observation in Figure 4, the operator GOTO-DR is refined: (dr-closed <v1>) is removed from the preconditions, and the effects become:

```
(effects ((<v5> Object))
  (add (next-to robot <v1>)) (del (next-to robot <v5>)))
```

Pre-state:

(connects dr12 rm2 rm1) (connects dr12 rm1 rm2)
(dr-to-rm dr12 rm1) (dr-to-rm dr12 rm2)
(next-to robot a) (dr-open dr12) (unlocked dr12)
(inroom robot rm1) (inroom c rm2) (inroom b rm1)
(inroom a rm1) (pushable c) (pushable b) (arm-empty)

Delta-state: add: (next-to robot dr12) del: (next-to robot a)

Figure 4: The second observation of the operator GOTO-DR

Refining Operators with Practice Because of the incorrectness and incompleteness of the learned operators, the plans produced by the planning module may be incorrect. This will be explained in section . While executing these plans in the environment, one of the following two outcomes is possible for each operator:

(1) The operator is *executed* if the state changes predicted by the effects of the operator happen after the operator execution. In this case, the execution is treated as an observation and the preconditins and the effects of the operator are modified as described in the previous sectione.

(2) The operator is *un-executed* if the state does not change as predicted because of unmet necessary preconditions of the operator in the environment. When an operator is un-executed, the learning module computes the unmet preconditions of the operator. If there is only one such precondition, we mark this precondition as a necessary precondition of the operator. Otherwise, none of the preconditions is marked as necessary because of insufficient information. *When all the preconditins of an operator are marked as necessary, the system has learned the correct representaion of the operator.*

Learning Negated Preconditions The learned operators are used to solve practice problems. During practice, if an operator is *un-executed* even when all the preconditions of the operators are satisfied, the learning module discovers that some negated preconditions are missing. The negations of all the predicates that are true in the current state when the operator is *un-executed*, but are not true in the observations of this operator are conjected as potential negated preconditions, and are added to the preconditions of the operator. Some of the added negated preconditions may be unnecessary, and they will be removed in the same way as all other preconditions with more observations and practices.

Inferring Subgoal Structure of the Observed Agent When the system observes an episode, it assumes that the observed agent is achieving its own top-level goals by carrying out the observed actions based on the "rational agent" assumption (Newell 1982). Inferring what the top-level goals are and how the preconditions of each operator in the observations is achieved provides the system heuristics for planning. For example, the system can prefer to subgoal on the preconditions of an operator that were subgoaled on by the observed agent.

The system infers how the preconditions of each operator are achieved by creating an Inferred Subgoal Graph (ISG). An ISG is a directed acyclic graph (DAG) whose nodes are either operators or goals. An edge in an ISG connects an operator to the goal it achieves, or connects a goal to the operator that requires it as a precondition. The root of an ISG is the virtual operator **finish** that is connected to the partially inferred goals.

The system initializes the ISG by setting its root to the virtual operator **finish**. The last operator in the episode must be achieving some top-level goals. Since everything in the *delta-state* of the observation can be a potential top-level goal, and since it is difficult to distinguish the true top-level goals from subgoals or side effects, the complete *delta-state* is presumed to be part of the top-level goals. A node is created for these goals and is connected to the root of the ISG, i.e. to operator **finish**. A node is also created for the operator in the observation, and is connected to the top-level goals it achieves. The system then infers how each precondition of the operator is achieved. A precondition is inferred to be achieved by an observation if it is added in the *delta-state* of the observation. Then the system recursively processes the episode and goes backwards the episode, determines how the preconditions of the operator are achieved. In an ISG, the preconditions of an operator that are connected to some operator that achieves them are considered to have been subgoaled on by the observed agent. These preconditions are called *subgoaled-preconds* of the operator.

The Planning Module

The operators acquired by the learning module are themselves incorrect or incomplete. To plan with such operators, PRODIGY's search algorithm is augmented with some domain-independent heuristics to generate plans that are possibly incomplete or incorrect.

Given a practice problem, the system first generates an tentative plan. Then the plan is executed in a simulated environment[4], and repaired if some operators in the plan cannot be executed because of unmet necessary preconditions in the environment. The following sections describe our approach to plan with incorrect and incomplete operators and to repair a failed plan.

Planning with Incomplete or Incorrect Domain Knowledge PRODIGY's standard planner assumes a correct action model. But before our learner acquires a complete model of the domain, the operators can be incomplete or incorrect in the following ways: (a) *over-specific precondi-*

[3] A precondition is removed iff the predicate of the precondition is not present in the new *pre-state*.

[4] The simulated environment is implemented using a complete and correct model of the domain.

tions: when a learned operator has unnecessary preconditions. In this case, the system has to achieve its unnecessary preconditions during planning. Not only does this force the system to do unnecessary search, but also it can make many solvable problems unsolvable because it may not be possible to achieve some unnecessary preconditions. (b) *incomplete effects:* when some effects of an operator are not learned because they are present in the *pre-state* of the operator. While searching for a plan, PRODIGY applies in its internal model an operator that is missing some effects, the state becomes incorrect.

These problems make it very difficult to generate a correct plan for a problem. Therefore, we relax the assumption that all the plans generated by the planner are correct. *over-specific preconditions* prevents the system from generating any plan at all, and are dealt with by using *applicable threshold heuristic* and *subgoal ordering heuristic* to generate plans that includes operators with unmet preconditions. *incomplete effects* is not dealt with explicitly in our planner. When an operator cannot be executed because of unmet necessary preconditions, the plan repair mechanism is invoked to generate a modified plan. PRODIGY's basic search algorithm is modified by using the following two heuristics to generate plans that are possibly incomplete or incorrect:

- *Applicable threshold heuristic:* While computing the applicable operators, a threshold variable **applicable-threshold** is introduced such that an operator is considered applicable in the planner's internal model during planning when all the *subgoaled-preconds* and the necessary preconds are satisfied, and when the fraction of the satisfied preconditions exceeds **applicable-threshold**. Of course, this introduces the possibility of incomplete or incorrect plans in that a necessary precondition may be unsatisfied, necessitating plan repair (see the next section).
- *Subgoal ordering heuristic:* While choosing a goal to work on next, goals that are *subgoaled-preconds* are preferred over those that are not.

For example, Figure 5 illustrates an example problem and the solution generated using the above heuristics with a set of learned operators that still are incorrect and incomplete. and **applicable-threshold** set to 0.7[5]. This plan is compared to a correct plan.

Plan Repair Plan repair is necessary when a planned operator fails to execute in the environment because of unmet necessary preconditions. Since there can be multiple unmet preconditions, and since some of them may be unnecessary preconditions, it is not necessary to generate a plan to achieve all of them. Therefore, the plan repair mechanism generates a plan fragment to achieve one of them at a time. The plan repair algorithm is illustrated in Figure 6. In step 1, heuristics from variable bindings are used to determine the order in which the unmet preconditions are achieved. An unmet precondition of the highest priority is chosen in step 2 as the goal to achieve. If a goal is chosen, the system attempts to achieve it using planning heuristics described previously. When a plan is thus generated, it is added to the initial plan to form the repaired plan. When the goal cannot be achieved, it is remembered as an *unachievable precondition*, and the system loops back to step 2 to achieve the next unmet precondition. When the failed operator still cannot be executed in the environment after the system has attempted to achieve each unmet precondition in the order determined in step 1, the system propose another operator to achieve the goal by using the *one step dependency directed plan repair* heuristics: while generating bindings for the failed operator, bindings leading to *unachievable preconditions* are rejected.

[5]The **applicable-threshold** is currently set manually, how to set it automatically is a part of future work.

(a) Initial State:

(connects dr12 rm1 rm2) (dr-to-rm dr12 rm2)
(connects dr12 rm2 rm1) (dr-to-rm dr12 rm1)
(unlocked dr12) (dr-open dr12) (inroom robot rm2) (arm-empty)
(inroom a rm2) (inroom b rm2) (pushable c) (inroom c rm1)
(inroom key12 rm1) (carriable key12) (is-key dr12 key12)

(b) Goal Statement:

(and (inroom key12 rm1) (inroom c rm2))

Incorrect and incomplete plan generated by the planner with the above heuristics:

<goto-dr dr12 rm2 rm1>
<go-thru-dr rm2 rm1 c dr12>
<carry-thru-dr rm1 c rm2 b c dr12>

A Correct plan that can be executed in the environment:

<goto-dr dr12 rm2 rm1>
<go-thru-dr rm2 rm1 c dr12>
<push-to-dr dr12 c key12 c b rm1 rm2>
<push-thru-dr dr12 c rm1 rm2 key12 c b>

Figure 5: A Sample Problem. The incorrect and incomplete plan generated by the planner using the above heuristics is compared with a correct plan that can be executed in the environment.

Input: *inst-op* that failed to execute, *top-goals*, *state*
Output: *repaired-plan*

1. Decide the order in which the unmet preconditions of *inst-op* are to be achieved
2. Choose the highest priority unmet precondition as the goal *g* to achieve
3. If *g* is chosen, do:
   3.1. Generate a plan to achieve *g* from *state*
   3.2. If the plan is not empty, it is added to the initial plan to form *repaired-plan*. Return *repaired-plan*.
   3.3. Otherwise, goto 2
4. If *g* is empty, generate a plan to achieve *top goals*, but avoid the operators with *unachievable preconditions*.

Figure 6: The Plan Repair Process

The Plan Execution Module

The plan execution process is integrated with plan repair. For each operator in the initial plan, the plan execution

module tests it in the environment. If it is executed, then the learning module refines the operator according to the states before and after the execution. If the operator is not executed, plan repair mechanism is invoked to generated a repaired plan and the execution continues. This process is illustrated in Figure 7.

Input: *plan, goals, state*
Output: Whether *goals* are achieved or not

For each instantiated operator *inst-op* in *plan*, do:

1. Test if *inst-op* is applicable in the environment or not
2. If it inst-op is applicable, then do:
   - 2.1. *state* ← apply *inst-op* in *state*
   - 2.2. Modify the operator with execution
   - 2.3. Check if *goals* are achieved. If so, terminate
3. If *inst-op* is not executed because of unmet preconditions, then do:
   - 3.1. Modify the operator with the un-executed
   - 3.2. *new-plan* ← Repair-plan (*inst-op, goals, state*)
   - 3.3. If *new-plan* is empty, terminate execute-plan with failure
   - 3.4. Otherwise, abandon the initial *plan* and restart execution: execute-plan (*new-plan goals state*).

Figure 7: Integration of Plan Execution and Plan Repair.

Examples

Let's trace how the problem in Figure 5 is solved when the system has learned a set of operators that are incorrect or incomplete. The initial plan is shown in Figure 5. While executing the plan, the first two steps, i.e. <goto-dr dr12 rm2 rm1> and <go-thru-dr rm2 rm1 c dr12> are executed in the environment. <carry-thru-dr rm1 c rm2 b c dr12> is not because of unmet preconditions. Therefore, the plan repair procedure is called. The unmet preconditions of this operator are: (holding c) (carriable c) (pushable b), and the planner decides to first achieve (holding c) using the variable bindings heuristics. The following two steps are added to the initial plan through plan repair:

```
<goto-obj dr12 c rm1 rm2 rm2 rm1 dr12>
<pickup-obj c key12 c rm1 rm1 dr12>
```

<goto-obj dr12 c rm1 rm2 rm2 rm1 dr12> is executed, but <pickup-obj c key12 c rm1 rm1 dr12> fails. The only unmet precondition of this operators is (carriable c), therefore, it is marked as a necessary precondition of the operator by the *learning module*:

```
Marking (carriable <v140>)
as a necessary precondition
of the operator PICKUP-OBJ.
```

Since (carriable c) cannot be achieved by the planner. <carry-thru-dr rm1 c rm2 b c dr12> thus fails, and its *unachievable precondition* is (holding c). The planner generates a new plan to achieve the top-level goals (inroom key12 rm1) and (inroom c rm2) using *one step dependency directed plan repair* heuristics while avoiding the operators with the *unachievable precondition* (holding c). The new plan is:

```
<push-thru-dr dr12 c rm1 rm2 key12 c b>
```

<push-thru-dr dr12 c rm1 rm2 key12 c b> fails to execute, and the unmet preconditions are ((next-to robot dr12) (next-to c dr12) (inroom key12 rm2)). A new plan step <push-to-dr dr12 c key12 c b rm1 rm2> is generated to achieve the precondition (next-to robot dr12), and it is executed. Finally, <push-thru-dr dr12 c rm1 rm2 key12 c b> is also executed. The *learning module* notices that the precondition (inroom <v184> <v178> is not satisfied in the state when <push-thru-dr dr12 c rm1 rm2 key12 c b> is executed, therefore, is an unnecessary precondition of the operator PUSH-THRU-DR:

```
Removing unnecessary precondition
(inroom <v184> <v178>)
from the operator PUSH-THRU-DR.
```

At this point, the system has achieved all the top-level goals.

Empirical Results

The learning algorithm described in this paper has been tested in the extended-STRIPS(Minton 1988) domain and a subset of the process planning domain(Gil 1991). It also easily learns operators in more simple domains such as the blocksworld domain and the logistics domain. As already discussed, the learned operators always have all the correct preconditions (except for negated preconditions), but they may also have unnecessary preconditions that can be removed with more observations and practice. Therefore, we use the percentage of unnecessary preconditions in the learned operators to measure the effectiveness of the learning system. We also measure the number of problems solved with the learned operators because the goal of learning is to solve problems in the domains.

The following table shows the system's performance in the extended-strips and process planning domain. Both the training and the practice problems in these domains are randomly generated. In the extended-strips domain, these problems have up to 3 goals and solution lengths of 5 to 10 operators. In the process planning domain, these problems have up to 3 goals, and an average solution length of 20. We are currenlty performing more extensive tests in the complete process planning domain that has total 76 operators. In both domains, we see that the learned operators can be used to solve the practice problems effectively, and the unnecessary preconditions of the learned operators are removed with practice so that the operators will eventually converge to the correct ones.

	extended-strips	process planning
# of training probs	7	20
# of learned ops	21	12
% of unnecessary preconds	45%	43%
# of practice probs	32	10
# of practice probs solved	26	10
% of practice probs solved	81%	100%
% of unnecessary preconds (after practice)	25%	38%

Related Work

There is considerable interest in learning for planning systems. Most of this work concentrates on learning

search control knowledge (Fikes, Hart, & Nilsson 1972; Mitchell, Utgoff, & Banerji 1983; Minton 1988; Veloso 1992). EXPO(Gil 1992) is a learning-by-experimentation module for refining incomplete planning operators. Learning is triggered when plan execution monitoring detects a divergence between internal expectations and external observations. Our system differs in that 1) The initial knowledge given to the two systems is different. EXPO starts with a set of operators that are missing some preconditions and effects. Our system starts with no knowledge about the preconditions or the effects of the operators. 2) EXPO only copes with incomplete domain knowledge, i.e. over-general preconditions and incomplete effects of the operators, while our system also copes with incorrect domain knowledge, i.e. operators with over-specific preconditions. 3) Operators are learned from general-to-specific in EXPO, while they are learned from specific-to-general in our system.

LIVE (Shen 1989) is a system that learns and discovers from environment. It integrates action, exploration, experimentation, learning, problem solving. It also constructs new term, that our system does not deal with. It does not learn from observing others, therefore it cannot learn search heuristics for planning. The preconditions of the operators in LIVE are learned mostly from general to specific, although over-specific preconditions can be eliminated using complementary discrimination learning algorithm. Our system deals explicitly with incomplete and incorrect operators by using a set of domain-independent heuristics for planning and plan repair, whereas LIVE depends on a set of domain-dependent search heuristics.

On the planning side, some planning systems do plan repair (Simmons 1988; Wilkins 1988; Hammond 1989; Kambhampati 1990). However, all these plan repair systems use a correct domain model. In our system, on the other hand, a correct domain model is not available, therefore it is necessary to plan and repair plan using incomplete and incorrect domain knowledge. Our plan repair method gives feedback to learning module, this is not the case for other plan repairs systems.

Future Work and Conclusions

Future work includes relaxing some of the assumptions described in this paper. For example, if there is noise in the sensor, we can have a threshold for each precondition so that a predicate is removed only when it is absent from the state more often than the threshold.

In conclusion, we have described a framework for learning planning operators by observation and practice and have demonstrated its effectiveness in several domains. In our framework, operators are learned from specific-to-general through an integration of plan execution as well as planning and plan repair with possibly incorrect operators.

References

Anzai, Y., and Simon, H. A. 1979. The Theory of Learning by Doing. *Psychological Review* 86:124–140.

Carbonell, J.; J.Blythe; O.Etzioni; Gil, Y.; Joseph, R.; Kahn, D.; Knoblock, C.; Minton, S.; Pérez, M. A.; Reily, S.; Veloso, M.; and Wang, X. 1992. PRODIGY 4.0: The Manual and Tutorial. Technical report, School of Computer Science, Carnegie Mellon University.

Carbonell, J. G.; Knoblock, C. A.; and Minton, S. 1990. PRODIGY: An Integrated Architecture for Planning and Learning. In VanLehn, K., ed., *Architectures for Intelligence*. Hillsdale, NJ: Erlbaum.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* 2(3,4):189–208.

Fikes, R. E.; Hart, P. E.; and Nilsson, N. J. 1972. Learning and Executing Generalized Robot Plans. *Artificial Intelligence* 3:251–288.

Gil, Y. 1991. A Specificication of Manufacturing Processes for Planning. Technical Report CMU-CS-91-179, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

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

Hammond, K. 1989. *Cased-Based Planning: Viewing Planning as a Memory Task.* Academic Press.

Kambhampati, S. 1990. A Theory of Plan Modification. In *Proceedings of the Eighth National Conference on Artificial Intelligence.*

Minton, S. 1988. *Learning Search Control Knowledge: An Explanation-Based Approach.* Kluwer Academic Publishers.

Mitchell, T.; Utgoff, P.; and Banerji, R. 1983. Learning by Experimentation: Acquiring and Refining Problem-solving Heuristic. In *Machine Learning, An Artificial Intelligence Approach, Volume I.* Palo Alto, CA: Tioga Press.

Newell, A. 1982. The Knowledge Level. *Artificial Intelligence* 18(1):87–127.

Shen, W. 1989. *Learning from Environment Based on Percepts and Actions.* Ph.D. Dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

Simmons, R. 1988. A Theory of Debugging Plans and Interpretations. In *Proceedings of the Sixth National Conference on Artificial Intelligence.*

Veloso, M. 1992. *Learning by Analogical Reasoning in General Problem Solving.* Ph.D. Dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA.

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

Acknowledgments

Thanks to Jaime Carbonell for many discussions and insights in this work. This manuscript is submitted for publication with the understanding that the U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes, notwithstanding any copyright notation thereon.