LEARNING PROCEDURAL PLANNING KNOWLEDGE IN COMPLEX ENVIRONMENTS

by

Douglas John Pearson

A dissertation submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
(Computer Science and Engineering)
in The University of Michigan
1996

Doctoral Committee:
Associate Professor John E. Laird, Chair
Professor John H. Holland
Professor Keki B. Irani
Associate Professor Colleen M. Seifert

This research was supported under contract N66001-95-C-6013 from the Advanced Systems Technology Office of the Advanced Research Projects Agency and the Naval Command and Ocean Surveillance Center, RDT&E division; and by a University of Michigan Rackham Predoctoral Fellowship.

ABSTRACT

LEARNING PROCEDURAL PLANNING KNOWLEDGE IN COMPLEX ENVIRONMENTS

by
Douglas John Pearson

Chair: John E. Laird

In complex, dynamic environments, an agent's knowledge of the environment (its domain knowledge) will rarely be complete and correct. Existing approaches to learning and correcting domain knowledge have focused on either learning procedural knowledge to directly guide execution (e.g. reinforcement learners) or learning declarative planning knowledge (e.g. theory revision systems). Systems that only learn execution knowledge are generally only applicable to small domains. In these domains it is possible to learn an execution policy that covers the entire state space, making planning unnecessary. Conversely, existing approaches to learning declarative planning knowledge are applicable to large domains, but they are limited to simple agents, where actions produce immediate, deterministic effects in fully sensed, noise-free environments, and where there are no exogenous events.

This research investigates the use of procedural knowledge to support the learning of planning knowledge in large and complex environments. We describe a series of environmental properties that constrain learning and are violated by existing approaches to learning planning knowledge. We then present an operator-based representation for planning knowledge that is sufficiently expressive to model complex, conditional actions that produce sequential effects over time. We then present IMPROV, a system for learning and correcting errors in this planning knowledge that only requires procedural access to the knowledge. This procedural restriction ensures that learning remains tractable, even over these large, expressive representations. We first explain how IMPROV incrementally learns operator precondition knowledge. We then demonstrate how a hierarchical, operator-based representation can be used to reduce the problem of learning operator effects to the problem of learning operator preconditions. This result allows IMPROV to use a single learning method to learn both operator preconditions and effects. This also allows IMPROV to learn complex models of actions that produce conditional or sequential effects. Finally, we test the system in two sample domains and empirically demonstrate that it satisfies many of the constraints faced by learning agents in complex and challenging environments.

To Helen
For each and every, perfectly ordinary and perfectly wonderful day.

ACKNOWLEDGEMENTS

My thanks go firstly to my advisor, John, for having the grand dreams and boundless enthusiasm to work on what's really the most exciting goal for AI, a general intelligent agent. It's great to work with smart people and, of course, John's a very smart guy, but there are a lot of smart people in AI or any academic field. John's different because he combines a fun-loving personality, with an attitude that while his research is an important part of this life, it's only part of his life; which is perhaps one of the most important and difficult lessons for any graduate student to learn. It's often said that the only important decision a graduate student makes is who their advisor will be, and I've never doubted that was one decision I got right. I'd also like to thank the rest of the Soar Mafia; the old crowd: Scott, Craig, Clare, Arie, Mike, Paul, Mark, Eric and the new: Randy, Karen, Feets, Mike, Seth, Bob, Ron, Frank, Joe, Peter, Sayan, Chris, for many helpful and interesting conversations along with the odd round of golf, hand of cards or duel in the skies. Also I'd like to thank the rest of my friends around the lab and Ann Arbor, for making the whole place more livable and the experience more enjoyable. My thanks also go to rest of the Soar community, spread across many countries, for putting up with all of those frantic 15 minute talks.

I'd also like to thank the rest of my committee, John, Keki and Colleen for many helpful suggestions and probing questions. Those, together with many ideas and discussions from our classes together, have greatly shaped this thesis.

Outside of my work life, I must thank all of my friends from the local badminton club: Anson, Jeff, John, Sue, Ukim, John, Nigel, Dan, Gil, Steve, Greg, Mel, Phil, Jason, and so many more, for being my friends without knowing or caring about the quality of my research. Without being able to release the tension that builds up from fighting the research beast, as a series of blistering smashes; I'm sure I'd have long since tossed a workstation through a window.

Finally, my thanks to my family. To my mum and dad, for prophetically saying that getting a Ph.D. is never a bad idea. And most of all to Helen, for keeping me sane and bringing sunlight into even the darkest of days. Now, let's go enjoy the new dawn...

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF APPENDICES

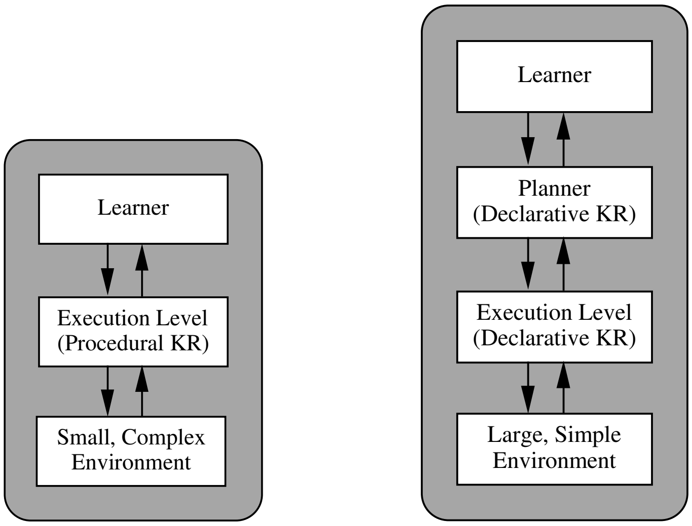
APPENDIX

CHAPTER 1

Introduction

Of all the capabilities that are integral to the success of human intelligence, perhaps two of the most striking are our abilities to think and to learn. Thinking, as opposed to reacting, requires the ability to construct and manipulate internal models, in other words, the ability to plan. Learning allows us to adapt to changing environments and to incrementally improve our ability to perform tasks in a complex world. This research project investigates an approach to these central issues, how to combine planning and learning into a single, generally intelligent agent that can function in complex and dynamic environments.

In complex, dynamic environments an agent's knowledge about the environment (its *domain knowledge* or *domain theory*) will rarely be complete and correct. The agent cannot expect to have exhaustive knowledge to guide its behavior in all possible situations except in the simplest domains. Additionally, changes in the environment over the life of the agent can make any preprogrammed knowledge outdated and incorrect. Thus, to succeed, an autonomous agent must have the ability to learn new domain knowledge and correct errors in its existing knowledge. Learning allows the agent to adapt to unexpected situations and update its knowledge in response to a changing environment.



(a) Reinforcement (b) Theory Revision

Figure 1.1: Traditional classes of agents that learn domain knowledge

Existing research on learning domain knowledge for planning and execution falls into two broad classes. Figure 1.1(a) shows the main components of agents in the first class, reinforcement learners (e.g. Q-Learning [Watkins and Dayan, 1992], Classifiers [Holland, 1986], Backpropogation [Rumelhart *et al.*, 1986]). These systems use weak inductive learning methods to directly modify an agent's execution knowledge. The execution knowledge is generally represented procedurally (e.g. in a neural net). These systems are robust in dynamic and complex environments but generally do not support planning or the pursuit of multiple goals. As a result they are usually only applied to domains with small state and goal spaces. Also, they learn slowly as a result of their weak methods. In contrast, the second category (shown in Figure 1.1(b)) consists of symbolic theory revision systems (e.g. EITHER [Ourston and Mooney, 1990], EXPO [Gil, 1992], OCCAM [Pazzani, 1988]). These systems learn declarative planning knowledge through stronger methods that explicitly reason to identify and correct errors in the agent's domain knowledge. However, these more powerful systems are generally only applicable to simpler agents where actions are assumed to produce immediate, deterministic effects in fully sensed environments where there are no exogenous events.

This research explores learning *procedural* planning knowledge through *deliberate* reasoning about the correctness of an agent's knowledge, as shown in Figure 1.2. The system, IMPROV [Pearson and Laird, 1996], uses an expressive knowledge representation so that it can learn complex actions that produce conditional or sequential effects over time. By developing learning methods that only require limited procedural access to the agent's knowledge, IMPROV's learning remains tractable as the agent's knowledge is scaled to large problems. IMPROV learns to correct operator precondition and effect knowledge in complex environments that include such properties as noise, multiple agents, irreversible actions and time-critical tasks. Additionally, the deliberate reasoning about correctness leads to stronger, more directed learning and allows other knowledge sources (e.g. causal theories) to be smoothly integrated into the learning. In this way, IMPROV, draws on the strengths of the existing classes of systems that learn domain knowledge, combining the powerful learning of theory revision systems with the robust performance in complex environments of reinforcement learners.

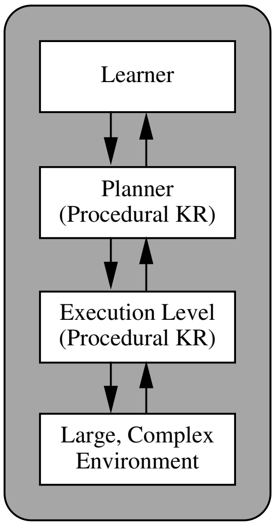


Figure 1.2: Deliberate learning of procedural planning knowledge

In addition to exploring the issues involved in building a system that learns procedural planning knowledge, this research is also exploring two related questions. First, what are the constraints and interactions between execution, planning and learning in an agent-based system? Many existing systems that learn planning knowledge are not directly connected to an execution environment. Therefore they do not address the question of when learning should occur or how training instances are generated. Often the approach that is taken is to consider each phase of execution, planning and learning as being a distinct module. There has been little work done on how these phases constrain each other and on integrating them into a complete autonomous agent that learns on-line, while still functioning in the environment. For example, time spent learning in a time-critical domain reduces the time available for planning and execution; but without learning, a task may be impossible if the agent's knowledge is incomplete or incorrect. One goal of this research is to better explore this interaction, outlining the constraints on learning, planning and execution and presenting one approach to satisfying those constraints.

The second, related goal for this research is to develop a weak method for learning planning knowledge. The goal here is to transfer the learner's bias from the structure of the system to the agent's knowledge. Instead of encoding a strong learning bias within the system itself, the intention is to develop a method that can be easily guided by additional agent knowledge. This allows the agent to flexibly use a range of different kinds of knowledge, rather than being limited to knowledge in a single form. For example, IMPROV defaults to using a weak method for credit assignment, based on differences between training instances. Additional knowledge can be added (for instance by adding a causal theory or through guidance from an instructor) to make the learning stronger and more directed.

IMPROV exists as both a theoretical system for the deliberate learning of procedural planning knowledge and as a specific implementation of this theory within a particular cognitive architecture, Soar[Laird *et al.*, 1987]. In presenting a theoretical or functional description, as well as a specific implementation, the intention is to help identify the contributions to other learning systems. For example, a Soar agent's knowledge is encoded as production rules. In general, an IMPROV agent's knowledge representation must support efficient associative retrieval and while production rules are one choice, other alternatives (e.g. as neural networks) would also be sufficient. Naturally, different implementations would embody different strengths and weaknesses. We recognize that while we feel the Soar architecture provides a good basis for building intelligent agents this is clearly an open area of research. Throughout the dissertation we will return to the question of what constraints the IMPROV theory places on an implementation and how Soar provides one particular approach to meeting those constraints.

1.1 Thesis Summary

This section summarizes the main elements of the thesis and IMPROV's learning method. This summary can be read as an introduction and overview of the remainder of the thesis. It can also be used by readers only interested in specific topics (e.g. knowledge representation or the learning of operator effects) to gain a general understanding of the system before jumping to the relevant chapter for further details.

In the body of this dissertation (Figure 1.3), we will first explore the constraints that complex environments place on planning and execution. We will then present an operator-based representation that is sufficiently expressive to represent complex actions and to support planning in challenging environments. We then derive a class of possible errors that can exist in a planning agent's knowledge and show how these knowledge level errors can lead to a range of performance failures. To keep learning tractable in large, complex environments we limit the access IMPROV has to the agent's knowledge. This restriction presents a challenge for monitoring plan execution

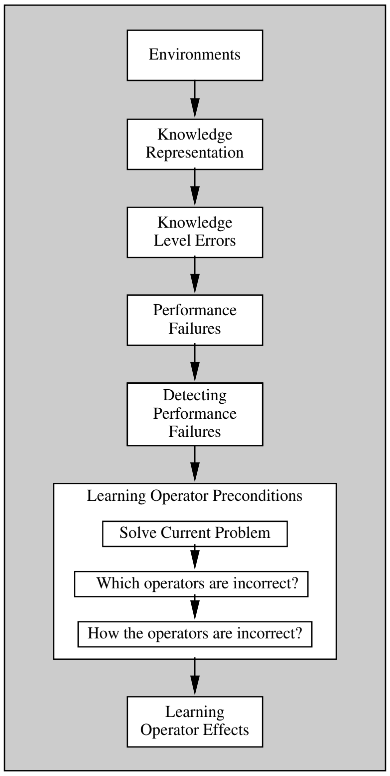


Figure 1.3: Stages of dissertation and error correction method

and detecting errors so we will present a procedural method for detecting performance failures. We then present a method for correcting operator precondition knowledge that is based on generating and executing plans until a successful plan is discovered; then analyzing and training an inductive learner on the sequences of operators and states collected during this search. We call this *k-incremental* learning as the number of training instances considered during learning, k, only increases until a successful plan is found, ensuring learning is still incremental. IMPROV represents potentially complex operator effects, such as conditional, sequential effects, as a sequence of more primitive *single-effect* operators. This hierarchical representation also allows IMPROV to solve the problem of learning operator effects by learning the correct *preconditions* of these single-effect operators. We evaluate IMPROV's performance on two test domains and demonstrate that IMPROV can recover from errors in its knowledge of operator preconditions or effects in environments which exhibit a wide range of complexities (such as noise, multiple agents, time-critical performance, evolving target domain etc.). Finally, we will demonstrate how IMPROV's learning can be seen as a weak method and can be guided by additional task knowledge to improve its performance.

1.1.1 The Environment and Procedural Access to Knowledge

IMPROV is designed as a method for learning planning knowledge for autonomous agents. The learning is constrained by the environments that we expect the agents to face. Each property of the environment shown in Figure 1.4 constrains the design of the agent and IMPROV's learning method (see Chapter 2). The agent plans (rather than just relying on an execution policy that covers all states) because the state and goal spaces may be large and because planning knowledge is usually more general and can be used for many different tasks. The agent is assumed to start with some initial, approximate domain knowledge that IMPROV learns to extend and correct as

Environmental Property	Constraint on Agent
E1. Large, Dynamic State Space	Need to plan.
E2. Large Goal Space	Need to plan, explicit goal representation
E3. Actions with Complex Structure	Expressive Knowledge Representation
E4. Actions that Apply in a Large Range of States	Expressive, Efficient and Scalable KR.
E5. Irreversible Environmental Changes	Learning Spans Multiple Episodes
E6. Time-critical Tasks	Efficient Learning, Planning & Execution.
E7. Long-term Tasks and Continual Existence	Incremental Learning.
E8. Limited Sensing	Tolerance for Noise/Delays etc.
E9. Environmental Changes Independent of the Agent	Difficult to Predict Environment
E10. Evolving Environmental Processes	Tolerance to Changing Target Knowledge.
E11. Many Forms of Feedback.	Use of Multiple Knowledge Sources.

Figure 1.4: Environmental constraints on planning and learning

the agent completes tasks in the external world.

As actions may produce complex effects, including sequential effects that occur over time, conditional effects or iterative behavior, the agent requires an expressive knowledge representation for the effects of actions. In making the representation expressive, there is a danger that the agent will become inefficient. In many systems, the time to correct existing knowledge grows in proportion to the size of the agent's planning knowledge (Chapter 3). This is undesirable as the agent's performance slows as it learns more. One approach to ensuring that performance does not degrade as the agent learns, is to limit the agent's access to its knowledge. IMPROV's methods only require procedural access to the agent's knowledge; that is, the planning knowledge can be executed but cannot be directly searched, examined or modified. As the agent is unable to search or otherwise directly examining its knowledge, learning time is guaranteed to be independent of the size of the knowledge base. This allows IMPROV to use a highly expressive representation for its knowledge, because learning will not examine or otherwise analyze this representation. The agent's knowledge is divided into operators, with preconditions and actions expressed as sets of production rules. IMPROV uses a compact, intentional representation based on production rules, to keep the space requirements of the agent's knowledge as low as possible. More extensional representations, such as listing states (popular in some reinforcement learning) or using sets of attribute-value pairs to define regions of state space (popular in symbolic learning) would both require large amounts of space to represent complex functions, such as the preconditions for the operator shown in Figure 1.5.

Action : Move-Arm (<x>, <y>, <r>)

Preconditions: if $\sin(x^7)*\cos(y^{0.3}) + x^2\log(y^5) + xy^2 - 12 > 0$ then move-arm

Effects : $x = r^2\cos(r)$
$y = -r^2\sin(r) + \log(r)$

Figure 1.5: Complex planning knowledge

To summarize, IMPROV learns planning knowledge that is represented intentionally and is only accessed procedurally. This approach provides sufficient expressiveness to represent complex planning knowledge, while ensuring efficient learning and task performance.

IMPROV builds plans through a state-space search technique called *Uncertainty Bounded Iterative Deepening* (UBID) (see Chapter 4). This is similar to iterative deepening, but the depth of the search is limited by an uncertainty measure, with each operator being assigned an uncertainty that reflects how closely its preconditions match the current state. This leads to a deeper search in areas of the search space that have earlier proved useful to the agent. Plans are represented procedurally, as a collection of rules to reactively guide the agent at each state during plan execution.

1.1.2 Framework for Error Correction

The main stages in correcting an agent's knowledge are:

1. Classification of errors

   Determining the errors that can occur in an agent's knowledge and the range of performance failures the knowledge errors can cause.

2. Detecting performance failures

   Recognizing that a performance failure has occurred, either during planning or during plan execution.

3. Solving the current problem

   Deciding what is the correct course of action in the current situation. This stage is often folded into the learning phase, where the approach is to learn and then replan.

4. Learning a general correction for the future

   Generalizing from the current situation to correct the agent's knowledge and avoid the error in the future. This learning can be further divided into 3 separate problems:

   (a) Credit assignment–Which operators are incorrect?

   Determining which operator, or operators, have the incorrect knowledge that lead to the performance failure.

   (b) Credit assignment–How the operators are incorrect?

   Having identified the incorrect operator, or operators, the agent must decide how the knowledge about that operator is incorrect. For example, which additional tests to add, to specialize the operator preconditions.

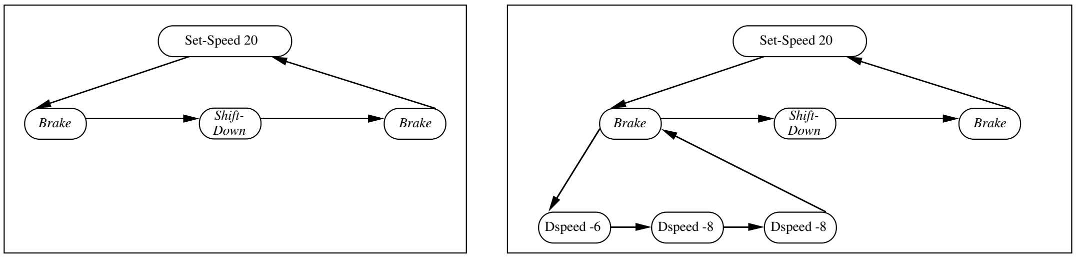
   (c) Changing the domain knowledge

   Finally, the agent must modify its knowledge to avoid the error in future. IMPROV's restriction to only executing its knowledge means that it must solve this problem without directly modifying the existing, incorrect knowledge. This is achieved by learning new knowledge that works together with the existing knowledge to generate the correct behavior.

1.1.3 Correcting Operator Preconditions

Knowledge Representation

IMPROV represents the agent's domain knowledge as a hierarchy of operators (Chapter 3). This hierarchy represents the goal–subgoal structure that is common to many symbolic reasoning systems. IMPROV's hierarchy is similar to standard top-down structured programming, with each layer of operators being analogous to a layer of procedures. The distinction is that in IMPROV, the hierarchy is built dynamically using operator precondition rules to determine which operators are included in a particular hierarchy. Control knowledge is folded into the preconditions for an operator, so that for a given goal and state, only a single operator should have its preconditions matched. High-level, abstract operators, are used to represent plans. These high-level operators are implemented by a series of operators that can themselves be sub-plans or traditional, motor-level operators that generate external behavior. An example is shown in Figure 1.6(a). The



(a) During Execution (b) During Planning

Figure 1.6: Domain knowledge as an operator hierarchy

`Set-Speed 20` operator cannot be achieved directly, so it becomes a goal for the agent with an implementation (or plan) to achieve it consisting of braking and changing gear. During plan execution, the `Brake` and `Shift-Down` operators generate external behavior in the environment, leading the agent to press pedals and change gear (Figure 1.6(a)). During planning, the same operators are used and motor-level operators, such as the `Brake` operator, are further expanded into a model of their expected effects, represented as a series of primitive, *single-effect* operators (Figure 1.6(b)). In this example, the `DSpeed` operators indicate that the rate of rate of deceleration will change as the agent brakes. These single-effect operators are not required during plan execution, just during planning, although they can be useful in detecting plan execution failures. Thus, operator actions consist of both execution knowledge that generates external behavior and planning knowledge for internal simulations.

Each operator has its own preconditions, represented as a set of production rules for when that operator should be chosen. The hierarchies each show a single expansion for a particular problem, in this case when the car is initially traveling faster than 20mph. If the car was initially traveling at only 10mph, then the hierarchy implementing `Set-Speed 20` would include operators for pressing the accelerator, as the preconditions for the `Accelerate` operator would match instead.

Classification of errors

IMPROV's operator based model for representing domain knowledge defines the scope of possible knowledge errors (Chapter 5). These are limited to:

- Incorrect operator preconditions

  These can be overgeneral, overspecific or a combination.

- Incorrect operator effects

  The agent's model for the effects of actions can include extra effects, missing effects or a combination.

- Completely missing operators

These errors are in the agent's knowledge about the processes of the domain. IMPROV does not correct errors in the agent's sensed data about the domain, that is, its state knowledge or state representation.

IMPROV learns new domain knowledge and corrects errors in operator preconditions (overgeneral, overspecific or a combination) or operator effects (extra effects, missing effects or a combination) but it cannot learn completely new operators. In certain constrained environments and with a sufficiently expressive state representation, IMPROV is guaranteed to converge to the correct operator knowledge.

Incorrect knowledge can lead to a range of performance failures, either during planning or during execution. By defining the classes of knowledge level errors we can derive the performance failures.

Detecting performance failures

To detect planning failures the agent requires extra knowledge about the planning process, beyond the task knowledge required to solve a problem. For example, the agent must know that a problem is soluble to realize that the inability to build a plan is because of an error in its knowledge.

Failures during execution can be detected by comparing the agent's planned behavior to the agent's actual behavior in the environment during execution. Traditionally, this comparison is made by explicitly monitoring each step of plan execution. The agent verifies that all of the preconditions for the next operator in the plan are satisfied before execution and then verifies that all of the expected effects have occurred. This form of explicit monitoring detects failures on the basis of the agent's ability to predict changes in the environment. In stochastic environments, or environments with multiple interacting agents or other external processes, accurate predictions may be impossible. IMPROV takes an alternative, weaker approach to detecting execution failures, by determining when the agent is unable to make progress towards its current goal (Chapter 6). The agent may incorrectly predict environmental changes but as long as those incorrect predictions not prevent the agent from achieving its goals, no failure is detected.

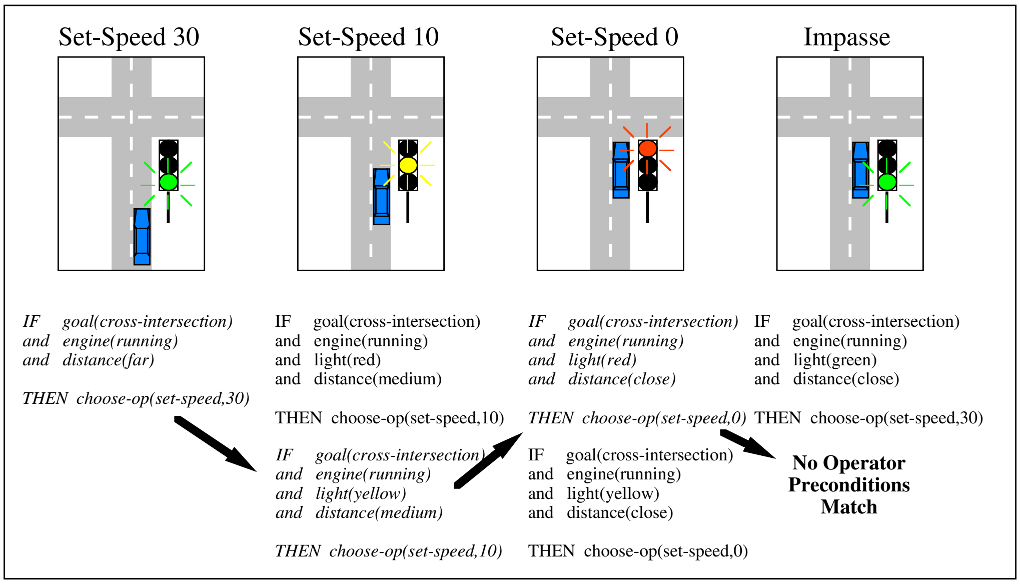


Figure 1.7: Inability to select an operator during execution signals an error

IMPROV detects a failure during plan execution if the agent reaches a state where it has conflicting suggestions (or no suggestions) on which operator to select next. This approach meshes well with IMPROV's distributed plan representation, where the plan is represented as a series of production rules that reactively guide the choice of operators during execution, rather than as a single, monolithic plan structure. Figure 1.7 shows an example of a plan to cross an intersection (represented as a set of rules). During execution (rules that fire are shown in *italics*) the agent is unaware that it must change to a lower gear, so the car stalls after `Set-Speed 0`. As a precondition for `Set-Speed 30` is that the engine is running, no operator's preconditions are matched and this is detected as an error. This approach can be extended to detecting errors in the expected effects of operator actions, by representing the effects of an operator as a sequence of more primitive operators and detecting an inability to select an operator at the lower level (see Figure 1.6(b)). IMPROV's method for determining when an agent is still on the path to the goal is augmented by a loop detection method, to ensure that it is making progress to that goal.

IMPROV's error detection method, as with the rest of its learning, is designed as a weak method. The methods for detecting errors are general purpose methods that make few assumptions about the environment or the agent's knowledge. It is important to realize that these weak methods can easily be made stronger by the addition of domain specific knowledge. For example, an explicit theory of failure states can be added to IMPROV to enhance its base error detection method. Knowledge can be added easily as all of the agent's reasoning is represented as production rules within a general purpose architecture for intelligent reasoning.

Solving the current problem

IMPROV casts the task of correcting an agent's knowledge as two search problems (Chapter 9). The first is a search through the space of plans and the second is a search through the space of operator preconditions. During the first search, plans are generated and executed in turn until the agent finds a plan that succeeds for the current goal. The sequences of operators executed during the first search are used to train an inductive learner, leading to the second search for correct operator preconditions.

IMPROV searches for alternative plans in decreasing order of similarity to the original, incorrect plan. As each plan is generated, IMPROV temporarily assumes that the agent's planning knowledge about the effects of the operators in the plan is correct. It simulates the sequence of operators to determine the outcome of the plan, allowing IMPROV to reject plans that it believes lead to failure. Each plan that reaches the goal in the internal simulation is executed in the world, until the agent finds a plan that succeeds. If a correct plan cannot be found, IMPROV proceeds to try to find a correction within the actions of the operators.

The agent's actions may be irreversible (for example, moving a chess piece during a game). In this case, the search for correct behavior will be spread across multiple problem solving episodes (multiple games of chess). IMPROV recalls previous failures and previous attempted corrections when it returns to a context similar to where the original failure occurred (Chapter 7). This allows it to search across multiple, temporally disjoint, problems.

As each plan is executed, IMPROV records the sequence of states, operators and the outcome of the plan (success or failure). IMPROV uses these records as positive and negative training instances for the inductive learner, once a successful plan has been found, (see Figure 1.8).

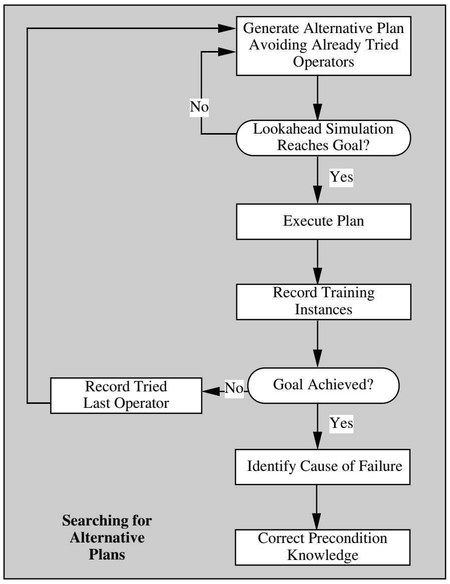


Figure 1.8: Searching for a correct plan

Learning a general correction for the future

IMPROV delays learning until a successful plan has been discovered, which allows it to use the comparisons between the successful plan and the unsuccessful attempts to improve credit assignment during learning (Chapter 9). IMPROV's delayed learning also helps it avoid incorrect early learning, which can be particularly harmful to an active learner because early learning influences the later instances the agent will see. For example, a driver that learns to stop for every bus, rather than just school buses, will avoid passing buses and so may take a long time to discover that its original learning was overgeneral.

1. Credit assignment–Which operators are incorrect?

   IMPROV's approach to determining the operators that caused a failure is to compare the successful plan to the original, incorrect plan, and use the differences to determine the operators that are in error. For each state in the successful plan, IMPROV determines which operator would have been chosen using the original, possibly incorrect, planning knowledge applied to that state. This original operator is compared to the operator used in the successful plan.

   Figure 1.9 shows an example where the agent is unaware that it must change gear before stopping the car, or it will stall. By comparing the successful plan to the incorrect plan, IMPROV concludes that the preconditions for `Change-Gear` should be generalized (so that operator will be chosen in the future), while the preconditions of `Set-Speed 0` should be specialized (so it is only chosen once the agent has changed gear).

   This approach can more accurately locate the incorrect operator than existing incremental approaches because IMPROV has access to more information in the form of the successful plan. Traditional approaches that only consider a single incorrect plan during learning are forced to rely on a fixed bias. For example, temporal difference methods assign most blame to the final step of a plan (`Set-Speed 0`). It is very difficult for such a system to discover that the true correction is earlier in the plan, while still maintaining the final `Set-Speed 0` operator, which is required in any successful plan.

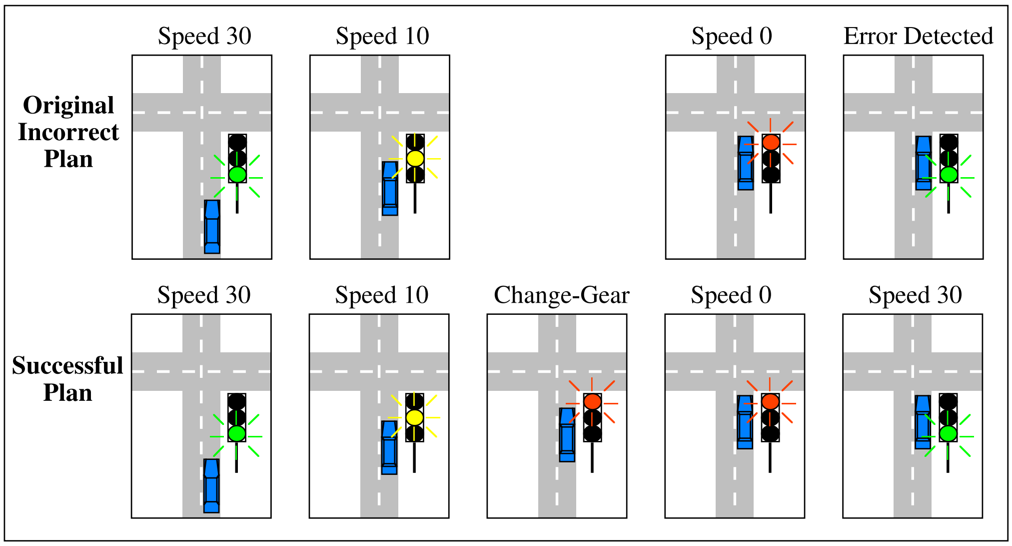


Figure 1.9: Differences between plans used for credit assignment

2. Credit assignment–How the operators are incorrect?

   Having determined which operators are incorrect, IMPROV must decide how to specialize or generalize the agent's precondition knowledge. IMPROV achieves this by inductively learning the correct operator precondition knowledge. The inductive, symbolic category learner, SCA2 (Chapter 8), is trained on each instance of an operator succeeding or failing, for a particular state and goal. IMPROV analyzes the set of training examples to determine which state features are most likely to have caused the success or failure of the operator. IMPROV biases the induction towards features that are present in positive instances, but missing or different in negative instances.

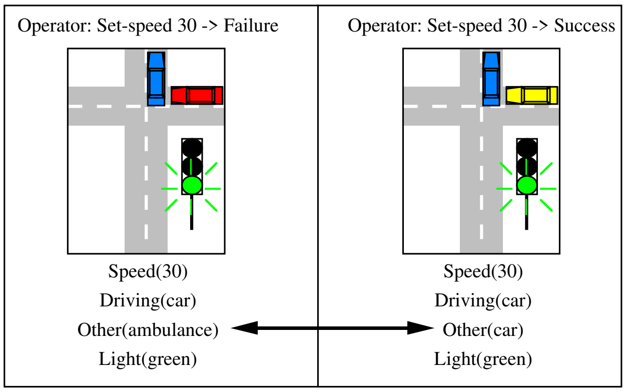


Figure 1.10: Differences between states used for credit assignment

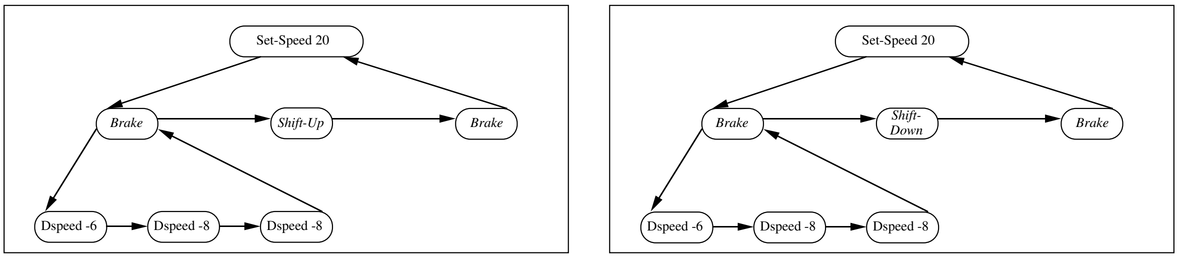
Figure 1.10 shows an example of how comparing a positive instance to a negative instance can improve learning. No matter how good the inductive learner is, there is insufficient information from just the initial negative instance to determine the cause of the failure. IMPROV benefits by delaying its learning until it has found a successful plan and therefore avoids making an incorrect early induction.

As the induction is based on a set of instances, we call this *k-incremental* learning. The k refers to the size of the set of instances passed to the learner during training and the learning is still *incremental* as the set of instances only increases until a successful plan is discovered. K will vary from problem to problem, as k is the number of trials of an operator before a success is discovered. However, as the number of instances considered during learning does not grow over the life of the agent, the learning is still incremental. This weak inductive learning can also be made stronger by the addition of domain specific heuristics.

3. Changing the domain knowledge

   IMPROV's procedural access to the agent's domain knowledge means the agent cannot directly examine and modify the incorrect knowledge. Instead of searching its rule-base for the incorrect knowledge (a potentially expensive process), IMPROV learns additional rules that correct the decision about which operator to select. Operator preconditions are specialized by learning rules that indicate the operator should not be chosen. Preconditions are generalized by learning additional rules for when the operator should be selected.

   The preconditions of an operator determine whether it is included in a particular operator hierarchy. An operator can be added to the hierarchy by generalizing its preconditions, or an operator can be removed by specializing its preconditions.



(a) Initial Theory (b) Final Theory

Figure 1.11: Correcting preconditions for Shift-Up and Shift-Down

For example, the agent's initial knowledge in Figure 1.11(a) is incorrect as the `Shift-Up` operator is included in the implementation (or plan) to achieve `Set-Speed 20`. The correct operator, `Shift-Down`, is included in the final hierarchy (Figure 1.11(b)), by generalizing its preconditions so that it is chosen when decelerating to 20mph. At the same time, the incorrect `Shift-Up` operator is removed by specializing its preconditions so that it is not chosen when decelerating.

1.1.4 Correcting Operator Effects

IMPROV corrects the planning knowledge that models the effects of external actions. The corrected planning knowledge is then used for subsequent planning and to learn and correct execution knowledge, as shown in Figure 1.12. Thus, the task for IMPROV is to learn the correct operator effects.

IMPROV corrects planning knowledge for the effects of operators by correcting the preconditions of a sequence of more primitive operators at the next lower level of the operator hierarchy (Chapter 11). To see how operator preconditions can be used to correct operator effects, consider the example shown in Figure 1.13. In this example, the agent's initial knowledge models the effects of pressing the brake pedal as producing a faster initial rate of deceleration than actually occurs (-4

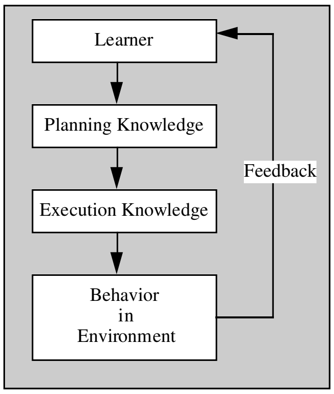
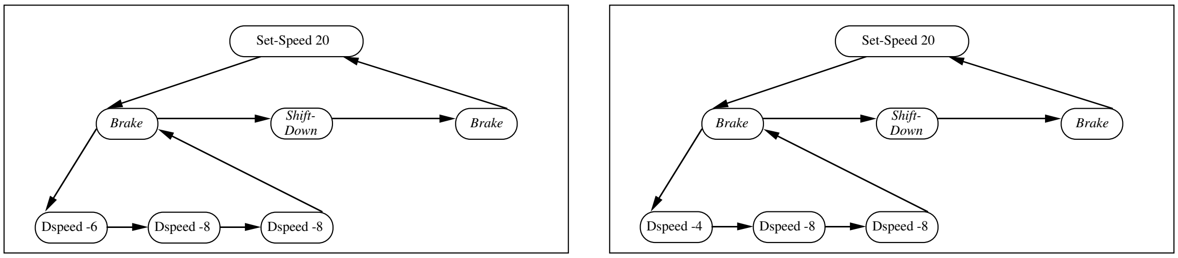


Figure 1.12: Correcting external behavior by correcting planning knowledge



(a) Initial Theory

(b) Final Theory

Figure 1.13: Correcting preconditions for DSpeed -4 and DSpeed -6

rather than -6). To learn the correct effects, IMPROV specializes the preconditions of `DSpeed -6` and generalizes the preconditions of `DSpeed -4`. This is completely analogous to the earlier example (Figure 1.11), with the correction being applied to a different level in the operator hierarchy. It's important to realize that the lowest level of operators only represent planning knowledge; the agent's external actions are the same in both cases, which is to press the brake pedal when the `Brake` operator is chosen. By changing the sequence of single-effect operators that are chosen in implementing `Brake` during planning, the agent's model for the effects of braking are changed, so that the agent expects braking to occur more slowly. Then, when the agent replans, it will select the `Brake` operator earlier, as it expects slower braking.

This approach allows IMPROV to represent and learn complex models of sequential or conditional effects that occur over time. For example, IMPROV can learn the multiple, dynamic effects of braking shown in Figure 1.14. This figure shows the rate of deceleration (`DSpeed`) and pressure

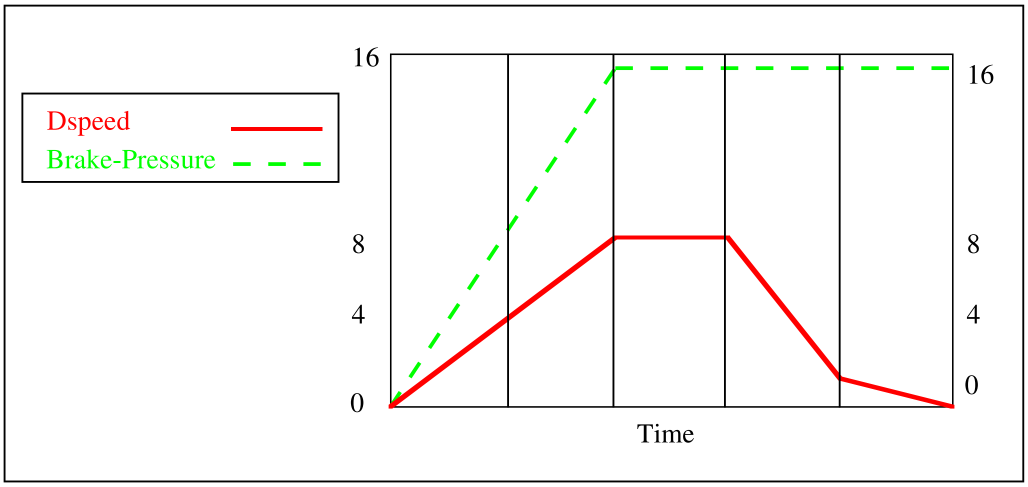


Figure 1.14: Correct model for effects of braking

on the brake pedal (`Brake-Pressure`) over time, in response to a *single* external action (to press the brake pedal). IMPROV models this sequence of effects as a series of single-effect operators and learns the correct preconditions for those operators. Figure 1.15 shows the operator hierarchy

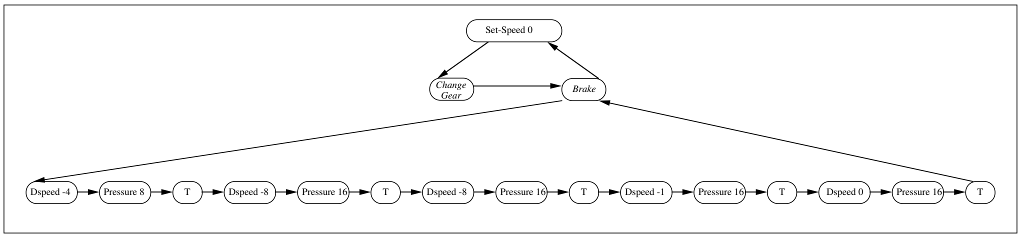


Figure 1.15: IMPROV's final representation for the effects of braking

after learning (`T` operators indicate when time advances). Existing systems that learn the effects of operator actions are unable to represent or learn this type of sequential effect. IMPROV's approach to learning operator effects, by correcting precondition knowledge at a lower level, is guaranteed to terminate at the single-effect level. This is because single-effect operators only manipulate a single symbol (e.g. `DSpeed`) and therefore the planning knowledge for these operators is guaranteed to be correct. The question is whether or not to include one of these operators in the effects of a motor-level operator, rather than changing the effects modeled by the single-effect operators; a decision which is based on the precondition knowledge of the `DSpeed` operators.

1.1.5 Evaluation

We have evaluated IMPROV on two test domains: a simulated robotic manipulation task and a simulated car driving domain. We tested IMPROV's performance on each of the environmental properties described in Figure 1.4 and demonstrated that IMPROV can efficiently learn planning knowledge in complex environments that include large goal and state spaces, noise, evolving target domains and time-critical tasks. We also demonstrated that IMPROV's *k-incremental* learning leads to better quality learning through improved credit assignment by comparing the *k-incremental* approach to a version of IMPROV that learns from each training instance immediately. Further, we demonstrated how the addition of just a few rules or some simple instructions from a human instructor, could significantly improve the performance of each phase of IMPROV's learning. By developing a weak learning method, IMPROV's learning can be guided by any knowledge that is readily available, without relying on specific knowledge being present for every task or in every domain.

1.1.6 Contributions

The main contributions of this thesis are:

1. Presenting a framework for learning planning knowledge
2. Presenting a method for learning planning knowledge in complex environments
3. Casting learning as a weak method within a general problem solving architecture
4. Presenting K-Incremental learning
5. Demonstrating that operator effects can be learned by correcting operator preconditions
6. Demonstrating that procedural access is sufficient for learning operator knowledge

CHAPTER 2

Environmental Constraints on Planning and Learning

2.1 Introduction

The focus of this research project is on developing autonomous agents that learn, plan and act in an external environment. This chapter describes how the environment constrains the design of the agent. For example, time-critical tasks and a continual existence constrain the agent to learn incrementally. Our intention is to develop a system that is as flexible and general as possible, rather than being specific to one target set of environments. As we will see, this leads to IMPROV being a weak, general purpose learner that can be strengthened by the addition of domain-specific knowledge. In the next section we describe the environmental properties and briefly mention how they constrain the agent's design. In the remainder of the dissertation, we discuss these constraints in more details and present methods that satisfy them.

2.2 Environmental Properties

Figure 2.1 summarizes the list of environmental properties and how they constrain the agent's design. If the agent is to function in a given type of environment, it must meet the constraints that environment imposes on the agent. In this discussion, the "environment" is taken to include the tasks the agent will be asked to perform and also the actions that the agent can take, as both constrain the agent's design.

E1. Large, dynamic state space

There may be a large number of goal-related objects in the environment. Each object may have many properties and relations to other objects. The set of objects in the environment may increase or change over time. For example, an agent that navigates in a changing world where the destinations and possible routes change over time. This environmental property motivates the use of a planner rather than just relying on a complete execution policy that maps states and goals to actions. As the number of possible states increases, perhaps even being infinite, it can be infeasible to maintain a complete execution policy (see Chapter 3).

E2. Large, dynamic goal space

The number and range of possible tasks may be large and the set of tasks may change dynamically. Again, the agent navigating to arbitrary destinations in a changing world is an example of this. The size of the agent's goal space, also motivates the use of a planning system, rather than storing a complete policy to cover all states and goals (Chapter 3).

Environmental Property	Constraint on Agent
E1. Large, Dynamic State Space	Need to plan.
E2. Large Goal Space	Need to plan, explicit goal representation
E3. Actions with Complex Structure	Expressive Knowledge Representation
E4. Actions that Apply in a Large Range of States	Expressive, Efficient and Scalable KR.
E5. Irreversible Environmental Changes	Learning Spans Multiple Episodes
E6. Time-critical Tasks	Efficient Learning, Planning & Execution.
E7. Long-term Tasks and Continual Existence	Incremental Learning.
E8. Limited Sensing	Tolerance for Noise/Delays etc.
E9. Environmental Changes Independent of the Agent	Difficult to Predict Environment
E10. Evolving Environmental Processes	Tolerance to Changing Target Knowledge.
E11. Many Forms of Feedback.	Use of Multiple Knowledge Sources.

Figure 2.1: Environmental constraints on planning and learning

Additionally, this constraint may be a problem for learning systems. Many reinforcement learners attempt to maximize "reinforcement" without explicit structures to represent goals, leading to problems of task interference, where the trained system loses skill on one task as it learns another [McCloskey and Cohen, 1989; Ratcliff, 1990].

E3. Actions may have a complex structure

Actions may have *duration*, taking time to produce changes in the world and possibly producing a series of *sequential* or *transitory effects*. The effects of an action may also be *conditional* on the current state of the world. For example, pressing the brake pedal in a car produces different effects depending on the current road, weather and tire conditions. Effects may occur in *parallel* or require *iterative* behavior. This property constrains the agent's representation of the effects of its actions. The representation should be sufficiently expressive to model these effects if the agent is to plan effectively (Chapter 11).

E4. Actions may apply to a large range of situations

Actions may be appropriate in a wide range of different situations. For example, you may brake to avoid another vehicle, to stay under the speed limit or to signal a tailgater. This property constrains the agent to use a representation that is at least expressive enough for large sets of disjunctive and conjunctive conditions. When the range of conditions is scaled to hundreds per operator for large numbers of operators, it also constrains the system to have efficient access to its domain knowledge. Ideally, the access cost should remain constant as the amount of knowledge increases.

This condition also constrains learning. The presence of large sets of preconditions for an action can lead to slow and inefficient learning if the learning method relies on analyzing the full set of preconditions during any correction (Chapter 3).

E5. Environmental changes may be irreversible

The agent's actions or other external processes produce changes that cannot be reversed through the agent's actions. For example, an agent that makes the wrong move while playing chess cannot take it back and make another move. This situation often arises when the agent is unable to recreate the problem state. For example, an agent that incorrectly drives through a red light at an intersection cannot back up and try again. The driver cannot immediately recreate the state that precedes running the red light. Similarly, a pilot who has a bumpy landing usually cannot take-off and try again.

This condition constrains a learner to consider instances that do not occur during the same problem solving episode (Chapter 7). Some analytic approaches assume that the environment is reversible or repeatable and therefore, that the agent will see enough instances clustered together to identify the cause of a failure during a single episode. If the instances are spread across multiple episodes, the learner must contend with correcting multiple errors in its knowledge simultaneously (the pilot who has trouble driving a car and landing a plane). Naturally, one approach to this is to treat each episode as completely independent, but then the agent potentially loses useful information that can be extracted when related training instances are considered together (Chapter 9).

E6. The task may be time-critical

There is limited time available for the agent to reason and learn. For example, when driving, the car continues to move as the agent plans, acts and learns. This property constrains the agent to not be arbitrarily[1] slow in selecting actions in the environment. This is a weak constraint–the agent must not become progressively slower as more knowledge is learned. However, the agent is not required to reason about the time available for actions, in order to achieve its goals. Our agents have sufficient time to reason, as long as their learning does not progressively slow the agent down. Many traditional deliberate learning systems become progressively slower as the amount of domain knowledge increases and as the number of training instances expands. Both of these mechanisms lead to systems that grow slower as they learn, eventually leading to arbitrarily slow task performance (Chapter 3).

E7. Tasks may be long-term and existence is continual

The agent may be continually active in the environment without time to consolidate or update its knowledge during a period of "down time". This constraint requires that the agent incorporate new knowledge efficiently into its existing knowledge base. More specifically, this constraint, together with E6 leads to a requirement that the learning be *incremental*, where the time to learn from a new instance is not dependent on the total number of instances already seen (Chapter 9).

E8. Sensing may be limited

The agent's knowledge about the current state of its environment may be incomplete or incorrect. Sensing may be *noisy*, where an incorrect value is reported by a sensor. It also may be *delayed* where it takes time for the current value in the environment to be sensed by the agent. Certain properties may be *hidden* where the agent is simply unable to sense them at all. Because of these limitations on sensing it is important that a learning system does not rely on complete and immediate sensing. Many deliberate learning systems make this assumption to simplify credit assignment and therefore limit the range of environments where they are applicable (Chapter 9).

Sensing is potentially a very complex issue. In this research we have focused on passive sensing, where the agent receives (possibly incorrect) input without cost. More generally, sensing can require effort by the agent and may be adjustable. For example, a telescope has

[1] This term is used in its mathematical sense. In other words, for any time bound, by choosing the inputs the system can be made to perform more slowly than the bound.

to be pointed at a particular location and can be zoomed in or out to change the scale of the observed objects. This is *active* sensing and is not directly considered in this project.

E9. The environment may change independently of the agent

There may be other agents or other processes acting on the environment. This is an issue when executing plans because the agent may have to react to unexpected changes in the environment. For example, when approaching a red traffic light, the agent may have built a plan to stop and cross when the light turns green. If the light changes before the agent reaches the intersection it should not stop, it should just cross the intersection. Equally, the agent should be able to build a plan that lets it stop if a pedestrian happens to step out into the road.

This also presents a challenge for learning. Traditionally, deliberate systems have credited all changes in the environment to the last action of the agent. This is clearly a useful heuristic, but it cannot be assumed to be always correct. Learning should account for the fact that an action's effects may occur at the same time as other independent changes in the environment (Chapter 11).

E10. Environmental processes may change over time

The processes that govern the environment may evolve over time. For example, as tires wear out, braking may take more time. This form of dynamism in the environment is more complex than occurs when a simple feature changes value (for example, a traffic light changing color). When the processes change, the agent's knowledge of the environment will also have to evolve or it will become incorrect (Chapter 10).

In some cases the distinction between an evolving environment and a changing feature is blurred. The wearing out of a car's tires may be seen either as a change in the way the environment works or as simply moving to a new part of a more "global" model. In this global model, there are processes for tires that are new and processes for tires that are worn. This distinction is largely unimportant. What matters is the effect of these types of change on the agent's knowledge and whether it becomes invalid or incomplete. In general, the agent may be unable to sense the triggering property, in which case the agent is usually forced to model the domain as truly evolving.

E11. Feedback about the environment may take many forms

Feedback may take many different forms and may be external (from the environment) or internal (extra, possibly intractable, internal knowledge bases). For example, the obvious source of feedback from driving a car is whether the agent achieves (or fails to achieve) its goals. However, the agent's knowledge about a car may also consist of an observed model of causes and effects–pushing the brake causes the car to slow down. It might also include deeper theories, such as a model of hydraulics and friction explaining why pressing the brake pedal slows the car, or it may include instructions from a driving instructor–brake whenever you're closer than 2 seconds from the car in front. These are all in addition to feedback about the success or failure of actions to reach the agent's goals. The general property is that the agent should be able to use a range of different sources of knowledge to guide its decisions about how to act in the world. Each of these types of knowledge may be presented and represented differently and an agent that can integrate them smoothly will be more effective that one that is limited to one form or has to reason about each source independently (Chapter 9, 10).

CHAPTER 3

Representation of Planning Knowledge

Before presenting an approach to learning, planning and execution that attempts to meet all of the environmental constraints from the last chapter, we will first explore the role that knowledge representation plays. In this chapter we will discuss the existing use of *declarative* access to an agent's domain knowledge and identify some of the limitations of this approach to knowledge representation. We will argue that assuming the agent has only *procedural* access to a compact *intentional* representation is more appropriate in a system that is to learn and act in the complex environments of Figure 2.1. As further evidence of this correlation, we will demonstrate that existing reinforcement learners typically rely on just procedural access to their knowledge and are applicable to these challenging environments. Meanwhile theory revision systems assume declarative access but are generally not applicable to these environments. We will further argue that *deliberate* learners, that explicitly reason about the correctness of an agent's knowledge, can learn from fewer examples than *implicit* learners, that just rely on general purpose induction algorithms.

3.1 Procedural vs Declarative Representations

As the terms *procedural*, *declarative*, *extensional* and *intentional* have many different interpretations in this field, we will use a computational model to define our use of them. We will describe three dimensions, use them to classify existing learners and demonstrate that there is a strong correspondence between these dimensions and the ability of the learner to function in the environments outlined in the last chapter. The three dimensions are:

1. Time to access data encoded in the representation during learning

   This is the cost to modify the agent's domain knowledge during learning. It typically varies from:

   ```
   >= O(size of representation) to O(1)
   ```

   Methods that are at the `O(size of representation)` end of this spectrum, we will refer to as having *declarative* access. While methods at the `O(1)` end we will refer to as having *procedural* access. We will shortly discuss some specific examples, but the intuitive distinction is whether the agent can search or otherwise manipulate the entire representation during learning. This is generally only possible when the agent has full, or declarative, access to its knowledge. In contrast, agents that have only limited access to their knowledge, such as procedural access where the agent can only execute the knowledge, will not have learning costs that are proportional to the size of the representation. It is important to recognize that while learning times proportional to the size of the representation require the agent to have declarative access to that knowledge, the reverse is not necessarily true. An agent with declarative access to its knowledge may not use that full access or may use indexing methods to reduce access times. However, an agent that is capable of full, declarative access that only makes limited accesses, is essentially working with a more limited level of access and is closer to a procedural system. Given that learning costs may be proportional to the size of the representation, size becomes our next dimension, especially as different systems require different space to store the same knowledge.

2. Size of the representation

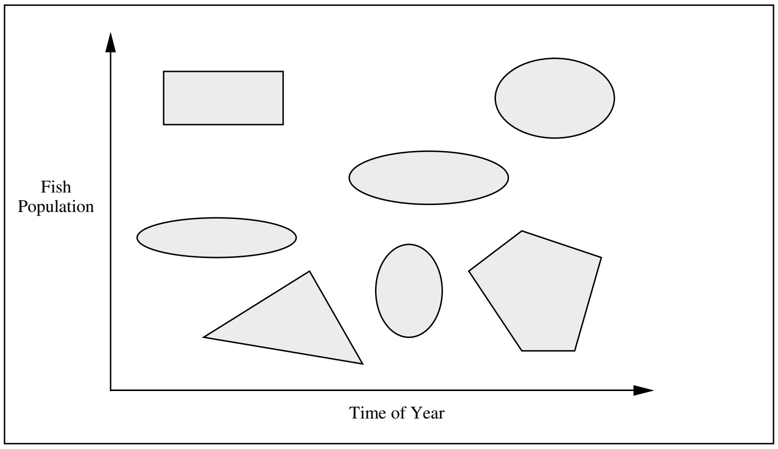


Figure 3.1: Regions of state space indicating when an action should be selected

This is the amount of space required to describe the agent's knowledge about the domain. We will focus on how to represent when to select an action, although there are parallels in representing the effects of an action. Let's assume the state space can be divided into n regions for when a particular action should be chosen. Figure 3.1 shows an example of deciding whether a fisherman should leave port, based on the current fish stocks and the time of year. The point is that the state space could be divided into a number of irregularly shaped regions depending on fishing regulations, holiday dates, tides etc. Naturally, a real example would be based on many factors and the regions would be in a multi-dimensional space.

In representing n regions of the state space, the size of the representation in learning systems typically varies from:

```
O(exponential(n)) to O(n) to O(log(n))
```

We will use the term *extensional* to refer to representations that list each state when an action should be taken. These representations are proportional to the size of the state space and therefore grow exponentially as the number of regions and the size of the space grows. They fall at the large end of this spectrum. In the middle of the spectrum are *semi-extensional* representations, typically based on a disjunctive normal form, to represent when an action should be selected. These representations use a series of attribute-value conditions to enumerate each region and therefore represent a more intentional representation. However, the representations are still proportional in size to the number of regions. At the smallest end of spectrum are fully *intentional* representations that use an unrestricted function to represent when an action should be selected. A single function can define a large number of regions, even an infinite number in extreme cases, using a very compact representation. This representation can therefore be sublinear in its space requirements.

To help clarify this, consider the example shown in Figure 3.2. This example helps to demonstrate that it is not always natural, or easy, to represent an agent's knowledge as either an extensional list of states or a semi-extensional disjunctive normal form. Computing the regions where this action will be chosen would require significant numerical analysis and could, in extreme cases, lead to an unbounded number of regions and therefore very large extensional or semi-extensional representations approximating the function.

Action : Move-Arm (<x>, <y>, <r>)

Preconditions: if $\sin(x^7)*\cos(y^{0.3}) + x^2\log(y^5) + xy^2 - 12 > 0$ then move-arm

Effects : $x = r^2\cos(r)$
$y = -r^2\sin(r) + \log(r)$

Figure 3.2: Action with complex preconditions

3. Explicit reasoning about the agent's knowledge

   This dimension defines the degree to which an agent reasons about the correctness of its knowledge, rather than just its task performance. This is a form of meta-level reasoning, as the agent reasons about the underlying knowledge that leads to task performance, rather than just reasoning based on the task performance itself. We will use the term *deliberate* learner to refer to an agent that reasons about errors in its knowledge and *implicit* learner to refer to agents that limit their reasoning to task performance.

We will now discuss existing methods for learning domain knowledge in light of these three dimensions and the environmental constraints we discussed in the previous chapter. In this discussion, we will particularly focus on the learning of precondition knowledge, knowledge about when to select an action. In Chapter 11 we will discuss approaches to learning the effects of actions.

The existing approaches generally fall into the categories summarized in Figure 3.3. This diagram is inevitably somewhat complex as it shows a 3-dimensional space. Systems at the "front" of the figure (lower left) learn deliberately, while those at the back are implicit learners. Each plane is then divided into procedural and declarative access against intentional and extensional or semi-extensional representations. Learning costs increase up and to the right, as the agent's knowledge is scaled to larger problems. Thus, we feel IMPROV represents an interesting part of the space to explore, using procedural access to a compact, intentional representation while still supporting deliberate learning.

3.1.1 Deliberate Learning of Declarative Representations

This section describes some representative examples of systems that learn by deliberately reasoning about errors in the agent's knowledge. They are often labeled theory revision systems. These systems typically rely on full, declarative access to the agent's domain knowledge and typically represent that knowledge in a semi-extensional form as a list of symbolic conditions for when to take an action. They are not generally applicable to the more challenging environments of Chapter 2.

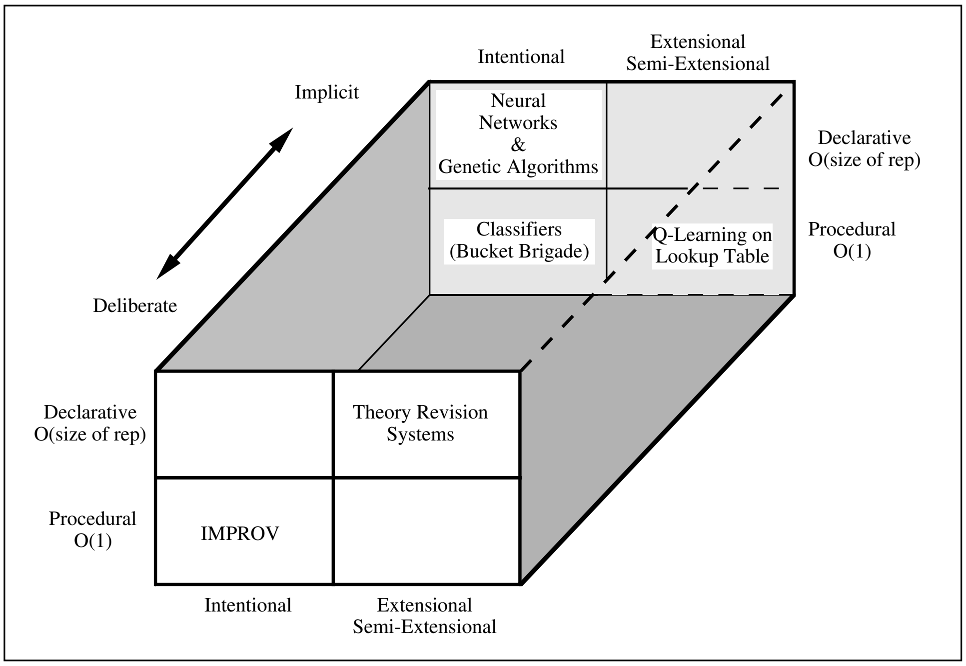


Figure 3.3: Classification of learning systems

EXPO, LIVE and OBSERVER

EXPO [Gil, 1991; Gil, 1993], LIVE [Shen and Simon, 1989] and OBSERVER [Wang, 1995; Wang, 1996] share a similar STRIPS-like [Fikes and Nilsson, 1971] representation. Operators are declaratively represented as structures with lists of preconditions and effects. The preconditions are represented in disjunctive normal form. The effects are limited to a single, state-to-state transition, although they can include conditional effects.

EXPO [Gil, 1991; Gil, 1993] learns corrections to its domain knowledge by designing experiments to refine initially overgeneral operator preconditions. Errors are detected by explicit monitoring of operator preconditions and actions, before and after an operator is executed. This detection process relies on the declarative access to the operator preconditions and effects. EXPO assumes complete and immediate sensing, allowing it to restrict the error to the last operator applied. EXPO designs and executes a series of experimental plans until a cause of the failure has been identified. The design of the experiments relies on declarative access to preconditions represented in disjunctive normal form. The search for the cause of a failure is guided by a series of heuristics, such as locality of action, similarity to other operators and generalization of previous experiences combined with a module to efficiently design experiments. Once the cause of the failure has been determined, EXPO changes the operator preconditions accordingly.

LIVE [Shen, 1989; Shen, 1994] and the more recent OBSERVER [Wang, 1995; Wang, 1996], learn domain knowledge by executing actions in the environment and observing the results. LIVE and OBSERVER rely on the assumption that changes in the environment are due to deterministic actions of the agent. Differences between the state before and then after an action are used as the basis of learning. In LIVE, features which disappear after an operator has been applied become preconditions, while new features become actions. When an inaccurate predicted effect is detected, LIVE recalls the last instance of applying this operator. The differences between the state now and the earlier state are used to specialize the operator preconditions. The unexpected features (which triggered the error) become new actions. OBSERVER initially learns operator preconditions from analyzing a series of expert execution traces, and then refines these through exploration. The set of operator preconditions is represented as two sets, one a most specific version and the other a most general version, consist with the observed training instances. These sets are refined in a manner similar to that of version spaces [Mitchell, 1982] and relies on operator preconditions being a single, conjunctive set.

The STRIPS-like planning representation allows these systems to reason in large state and goal spaces (E1and E2) and (for EXPO and OBSERVER) to use a full planner from the PRODIGY architecture [Minton *et al.*, 1989; Carbonell *et al.*, 1991]. Experiments or explorations are deliberately selected for their ability to guide learning, thereby re-using the internal planning as a source of guidance during learning (E11). However, the representation is not expressive enough to model complex actions such as ones that produce sequential effects over time (E3). Also, although the representation language is sufficiently expressive to include disjunctive operator preconditions, the learning methods assume only conjunctive preconditions (violating E4). The standard STRIPS solution to this problem is to split disjunctive operators into multiple conjunctive operators. However, this prevents any transfer of learning between these split operators (e.g. when the effects of the operator change). Learning time for these STRIPS-like representations is also proportional to the size of the operator structures (violating E6) although it can usually be incremental (satisfying E7). Sensing is assumed to be complete, immediate and noise-free (violating E8) and all changes in the environment are assumed to be due to the agent's actions, preventing these systems from being used in domains with external processes or other agents (E9). Finally, the learning is assumed to be correct and is generally irreversible. Therefore a domain with processes that evolve would be unlearnable (E10).

EITHER/NEITHER, FOIL/FOCL and TRAIL

EITHER [Ourston and Mooney, 1990] and the more recent NEITHER [Baffes and Mooney, 1993], correct errors in Horn-clause propositional logic domain theories. EITHER corrects errors in the theory as a whole. The theory's preconditions are antecedents of the Horn clauses that lead to an example being classified as belonging to the theory. The antecedents are represented in disjunctive normal form in EITHER and N-of-M form in NEITHER, both semi-extensional, although NIETHER's representation will be more compact on some problems. The actions are limited to category labeling. For example, the preconditions for an object are that it has a handle, a flat bottom and is an open vessel, and the action is labeling it as a cup. In EITHER, training examples include the label, reducing the credit assignment problem of determining which theories (or operators) need to be corrected. Overspecific and overgeneral theories are repaired by constructing all possible repairs to the theory and then ranking the theories according to a series of metrics. This process relies on having declarative access to the set of theories. The ranking is based on the number of examples covered, the size of the repair and the number of new, incorrectly classified examples it creates. A non-incremental, greedy algorithm iteratively adds the best repair to the new theory until all examples are covered.

FOIL/FOCL

FOIL [Quinlan, 1990] learns horn clause classification rules, or first-order logic programs, and is therefore an example of inductive logic programming. FOIL has been extended to FOCL [Pazzani *et al.*, 1991; Pazzani and Kibler, 1992], which also includes an explanation-based learning component. FOIL uses a greedy algorithm, based on adding literals with maximum information gain, to cover a set of positive training instances, while avoiding covering negative instances. The classification rules FOIL learns can be seen as operator preconditions, represented in a semi-extensional disjunctive normal form. As the algorithm is non-incremental and the representation is built up one literal at a time, learning time is proportional to the size of the representation.

TRAIL

TRAIL [Benson, 1995; Benson and Nilsson, 1996] learns planning knowledge in the form of teleo-operators (or TOPS). Operators consist of a precondition (or *preimage*) which can be disjunctive, an action, a single intended effect (or *postcondition literal*) and a set of probabilistic side effects. The action is continuously executed until the postcondition literal is achieved. The sequence of states the agent passes through, until it either achieves the postcondition or reaches a time limit and fails, are recorded and used as training instances for an inductive logic programming module. TRAIL requires that the preconditions of the operator remain true for the duration of the operator. This means that each state, along the course of a successful operator application, is a positive instance of the operator precondition. The ILP module [Muggleton and Feng, 1992] is trained on each of these instances. This learning is similar to IMPROV's training on a set of instances, but in IMPROV's case the instances are collected across multiple training episodes. TRAIL takes an interesting approach to learning operator the effects of actions with duration, which is discussed in Chapter 11 and TRAIL makes fewer assumptions about the environment than is typical of deliberate, declarative learners.

The domain knowledge learned in these deliberate systems can be used for planning in large state and goal spaces (E1, E2). Learning time is proportional to the size of theory and is usually not incremental (violating E6 and E7) and presenting problems for learning in an evolving domain as the training instances will contradict each other (violating E10). These systems generally assume complete and perfect sensing as the theory must cover all training instances (although this restriction can be relaxed to improve noise tolerance, as happens in TRAIL) (E8).

CHEF

CHEF [Hammond, 1989; Hammond, 1986] uses case-based reasoning to repair plans and recover from failures. CHEF records a library of plans and a library of patches that can be used to convert an existing plan into one applicable to a new goal. CHEF recovers from plan execution failures by analyzing an explanation of the failure. CHEF selects a plan repair method on the basis of classifying the explanation of why the failure occurred. It uses the repair method to correct the plan and stores the new plan for future use. The new plan is indexed on the basis of features derived explanation as being the cause of the failure. A crucial assumption of CHEF is the availability of the causal explanation, making CHEF more of an explanation based learner (although using the explanations in a different way from standard EBL/EBG), than a knowledge level (or inductive) learner.

These systems are only intended to be representative examples of deliberate, declarative learners. There are a great number of other theory revision systems, each embodying different strengths and properties. For example, FORTE [Richards and Mooney, 1991] extends EITHER to first order logic as does GENTRE [Asker, 1994]. DUCTOR [Cain, 1991] is similar to EITHER, but including additional abduction components, and is in turn similar to OCCAM [Pazzani, 1988; Pazzani, 1991]. CLIPS-R [Murphy and Pazzani, 1994] even applies theory revision to production rules. However, the character of these systems is similar to those discussed above, relying on declarative access to the domain theory to support deliberate learning.

3.1.2 Implicit Learning of Intentional Representations

This section describes some examples of systems that learn implicitly. These systems are often labeled as reinforcement learning systems. Learning is implicit because it identifies incorrect task performance, rather than incorrect knowledge. This prevents the learner from localizing the correction. Consider the example of an agent that drives through a corner too quickly, leading to a skid. The incorrect performance (the skid) will lead to negative feedback and credit will be assigned to each step in driving through the corner, including braking, turning etc. Eventually, the agent may drive more slowly, but this will never have been explicitly located as the cause of the failure. These learners typically learn a compact, intentional representation.

Classifier Systems

Classifiers [Holland, 1986; Booker *et al.*, 1989], represent domain knowledge as rules. The rules are typically fixed length binary strings which restricts their expressiveness and the agent relies on reactive execution rather than planning to select actions. As rules can be combined into sequences, or chains, that lead to actions in the world, a set of rules can represent complex functions and is therefore intentional. Error detection implicitly occurs when reinforcement is not received after taking an action. Rules have a strength which indicates the usefulness, or *fitness*, of the rule. Credit for successes is assigned through backward-chaining in the bucket brigade algorithm. The strength of a rule is reduced when it fires, thereby penalizing rules that do not lead to success (and positive reinforcement). As learning only requires access to the sequence of rules that contributed to a result, the bucket brigade does not require declarative access to the rule base. Errors in a specific rule's knowledge are not identified and corrected by classifier systems. Instead, rules that are generally successful are used to create new offspring through the use of a genetic algorithm, with the offspring replacing other low strength (and hopefully incorrect) rules. It can be shown [Booker *et al.*, 1989] that by doing this the more useful sections of a rule, or *building blocks*, will be preserved in future rule populations, while less useful parts are lost. The genetic algorithm requires declarative access to the rule base to produce the offspring generation. In classifier systems the size of the rule base is fixed. This ensures that learning time does not increase as the agent learns more. However, it does mean that classifiers suffer from over-training and task interference as previously valuable rules are discarded to make room for currently useful rules.

Backpropogation in Neural Networks

Neural networks trained using backpropogation [Rumelhart *et al.*, 1986], or related temporal difference methods [Samuel, 1959; Sutton, 1988; Tesauro, 1992], implicitly correct domain knowledge by adjusting the network to more accurately model the training information or environment. Domain knowledge is represented as a network, intentionally encoding a complex mathematical function. The network can be seen as representing the preconditions of an operator, with the network's output indicating the choice of action. Errors take the form of differences between the network's output and the desired output. As the network essentially represents a single, complex operator the credit assignment problem is to identify the parts of the network that contributed to an incorrect output. Credit is assigned in proportion to the activity of a node and the strength of its connection to the output. As credit is assigned to all of the features present during a failure, the incorrect knowledge is only implicitly detected and removed over many instances. Changes are made by adjusting the connection strengths between nodes to reduce the size of the error. Nodes that are active when the agent acts incorrectly will have their connection strengths reduced over time, removing them as conditions for the action. Nodes active during successes will have their connections strengthened, effectively adding them to the preconditions for an action. Although neural networks only have procedural access to the network during reasoning (connection strengths cannot be reasoned about), full declarative access to all of the network's connections is required during learning. Thus, learning time is proportional to the size of the network. As in classifiers, the size of the representation is fixed, ensuring that learning does not slow down over time, but leading to problems with over-training and task interference [McCloskey and Cohen, 1989; Ratcliff, 1990].

3.1.3 Implicit Learning of Extensional Representations

Q-Learning

Q-learning [Watkins, 1989; Watkins and Dayan, 1992] also implicitly corrects the agent's domain knowledge by improving the accuracy of the predicted reward for taking a particular action. Knowledge is represented by a set of Q-values, one for each state and action pair. As a result, Q-learning requires a full, extensional representation and as a result is generally only applicable to small state and goal spaces. These Q-values predict the expected reward for taking an action in a given state. Errors are based on the difference between the expected reward for the states before and after an action, along with the actual reward received. Q-learning assumes the environment provides immediate reward after each action, resolving the problem of determining which action should receive credit for the reward. Changes to the domain knowledge are made by adjusting the Q-values associated with a particular state and action. As these entries can be efficiently indexed, the agent does not require full declarative access. If less reward is received than was expected, the Q-value will be reduced, making that action less likely to be chosen in future. Together the Q-values for a particular state effectively represent the preconditions for the different actions that the agent could take. An example of a system that uses Q-learning for agent-based learning is Dyna-Q [Sutton, 1990].

Neural networks, Classifiers and Q-learning make few assumptions about their environment and do not model the effects of the agent's actions in the world. This makes them applicable to domains with limited sensing, exogenous events and processes that change over time (E8, E9, E10). The fixed size of the representations and the incremental learning algorithms mean learning time remains constant during the life of the agent satisfying E6 and E7. As the effects of actions are not modeled by the systems they can be complex in structure and widely applicable (E3,E4) although this lack of modeling also prevents the systems from planning. On the downside, reinforcement learners are usually applied to only a single task (and suffer interference effects if trained on multiple tasks) and are limited to a static state-space due to the static nature of the representations (thus violating E1 and E2). Also, they are generally only able to learn from a single source of knowledge–reinforcement (E11). Recently there has been some work on using other knowledge sources, such as instruction [Maclin and Shavlik, 1996], in neural network learning, but incorporating other knowledge sources remains a challenging problem.

3.1.4 Summary of Existing Approaches

To summarize, we can classify existing research in terms of its knowledge representation, the access to that knowledge and whether the knowledge is directly reasoned about. We can then draw correspondences between the knowledge representation and the environments where the system can be applied (see Figure 3.4).

The systems that are applicable to a wide range of environments (reinforcement learners) tend to have expressive intentional representations and either use limited procedural access or have

Environmental Property	Reinforcement Learners	Theory Revision	IMPROV
E1. Large, Dynamic State Space	NO	YES	YES
E2. Large Goal Space	NO	YES	YES
E3. Actions with Complex Structure	YES	NO	YES
E4. Actions that Apply in a Large Range of States	SOME	NO	YES
E5. Irreversible Environmental Changes	SOME	NO	YES
E6. Time-critical Tasks	YES	NO	YES
E7. Long-term Tasks and Continual Existence	YES	SOME	YES
E8. Limited Sensing	YES	NO	YES
E9. Environmental Changes Independent of the Agent	YES	NO	YES
E10. Evolving Environmental Processes	YES	NO	YES
E11. Many Forms of Feedback.	NO	SOME	YES

Figure 3.4: Coverage of environments by system type

declarative access to a representation of a fixed, small size. These systems are limited in their ability to plan, transfer their learning or work on problems that require dynamic changes to the state and goal representations. Also, they limit themselves to implicit learning, without the meta-reasoning to locate knowledge errors that can lead to more accurate learning.

The other class of systems (theory revision systems) require full declarative access to larger semi-extensional representations. These systems can plan and work on problems that involve large and dynamic state and goal spaces. However, they are not applicable to many of the other classes of environments that contain other agents, noisy sensing, fast, incremental learning etc. Also, they cannot efficiently represent complex preconditions that divide the state space into many regions, possibly leading to problems as the systems are scaled up or applied to control tasks.

In essence, declarative access to possibly large representations trades expressiveness for tractability. Having declarative access to the complete model of the agent's actions can be very useful in guiding learning, leading to deliberate learning and planning. However, it should not be assumed that declarative access comes without cost. Either the representation is fixed in size and does not include a model of the agent's actions (preventing planning and leading to task interference problems) or learning becomes slow and inapplicable to many complex environments.

This is not an argument that declarative access to knowledge is always inappropriate, rather that there are potential pitfalls from assuming declarative access is readily available. This result means that one interesting avenue to explore is the case of assuming the agent only has procedural access to a compact, intentional knowledge representation. By developing methods that assume only procedural access, learning times can be guaranteed to be independent of the size of the agent's representation. Also, by using an intentional representation, that representation can be kept small and efficient to execute. From this knowledge representation, we can build up methods that embody the benefits of deliberate learning and have a method that is applicable to challenging environments.

3.2 IMPROV's Procedural Representation of Planning Knowledge

As we would like IMPROV to function in the challenging environments of Chapter 2 and as full declarative access may inhibit this, we are exploring the hypothesis that declarative access to just the names of operators is sufficient for learning and correcting planning knowledge. The remainder of the agent's knowledge is only accessed procedurally, ensuring that IMPROV does not suffer from the potential pitfalls associated with declarative representations. In this section we present IMPROV's representation for domain knowledge.

Most domain theory learning systems that plan, represent their domain knowledge as sets of independent operators, where each operator corresponds to a different action the agent can perform in the world. These operators are used for planning as well as for execution of actions in the environment. During planning, the operator simulates the effects of an action on an internal model, while during execution, the operator initiates motor actions in the world. The operator based representation makes planning more efficient by supporting reasoning at a larger grain size for backtracking and problem decomposition. Without an operator (or other similar larger structure) to organize the agent's knowledge, reasoning is limited to individual inferences rather than the groups of inferences represented by an operator.

Collecting knowledge into operators brings an additional locality for that knowledge that can improve learning as well as planning. First, conflicts or gaps in the agent's knowledge can be detected when there is insufficient knowledge to decide which operator to select or apply next. Second, selection knowledge (when to do something) can be learned and corrected independently of implementation knowledge (what to do). We will return to both of these issues later in discussing IMPROV's error detection and correction methods.

IMPROV adopts the operator based model for its domain knowledge–although it limits the agent to just procedural access to this knowledge.

3.2.1 Planning Knowledge – Functional Requirements

IMPROV is an investigation of detecting errors and learning corrections of *procedural* planning knowledge. This basic approach, together with the set of environmental properties from Figure 2.1 and the discussion of Section 3.1 lead to this set of requirements for an IMPROV agent's domain knowledge:

1. Organized into Operators

   IMPROV's learning methods take extensive advantage of the assumption that the agent's knowledge is organized into operators. This organization facilitates credit assignment (as we will show) by allowing the agent to focus its learning on specific operators. It also allows selection knowledge (operator preconditions) to be learned and corrected independently of implementation knowledge (operator effects), as well as improving planning efficiency.

2. Expressiveness for Complex Effects

   The agent's planning knowledge must be sufficiently expressive to represent actions with complex structure (E3). This should include modeling actions with duration, sequential effects and conditional effects as well as parallel effects and looping behavior.

3. Expressiveness for Large Precondition Sets

   The agent's knowledge must be expressive enough to allow for multiple conjunctive and disjunctive sets of conditions for when an operator (E4) should be chosen.

4. Efficient Testing of Operator Preconditions

   An IMPROV agent requires a method to determine if operator preconditions are met by a particular state. This method should be efficient (to meet the temporal constrains of E6 and

E7). For instance, a declarative list of the regions of state space where the preconditions are satisfied could be expensive to access and store if the state-space is divided into an large number of regions. This is a further argument for only assuming and requiring procedural access.

5. Efficient Calculation of Operator Effects

   Determining the effects of an operator must be efficient, in the same way that computing whether the preconditions are met must be efficient. This is usually no problem when the operator effects are discrete, instantaneous changes to the internal planning representation but becomes more of an issue as the representation expands to include more complex effects, such as conditional or sequential ones.

6. Efficient Methods for Changing Operator Knowledge

   The agent should be able to modify it's operator knowledge as it learns, with the speed being independent of the amount of knowledge the agent has about the operator. Systems based on declarative access generally violate this requirement as the cost to locate the incorrect knowledge usually grows with the size of the theory. However, in a system based on procedural access to knowledge we must avoid slipping in a requirement for declarative access to the knowledge base in order to change it.

A key element of this list is that it does not contain any requirement that the agent be able to enumerate the preconditions or actions of its operators. If the agent has this level of access, it does not mean it will be unable to function in complex environments, but the system will require careful design or it will fail.

3.2.2 Planning Knowledge – Implementation

We have mentioned that the current implementation of IMPROV is built within the Soar architecture [Laird *et al.*, 1987][1]. In Soar, all activity is cast in terms of applying operators to move through a state space represented by attribute-value pairs. During problem solving, Soar repeatedly selects and applies operators. If Soar is unable to uniquely select an operator or cannot fully apply the operator (i.e. the agent has reached an impasse), Soar creates a new subgoal. In the subgoal, operators are again selected and applied in an effort to decide how to resolve the original impasse. Results from the processing are cached by Soar's only learning mechanism: chunking [Laird *et al.*, 1986], a form of Explanation-Based Learning [Mitchell *et al.*, 1986; DeJong and Mooney, 1986]. It is important to realize that chunking only analyzes the trace of working memory elements created and tested during problem solving, so learning costs are proportional to execution costs. There is no requirement to search or otherwise access the rule base during learning.

In Soar, all knowledge is represented as production rules that consist of a set of conditions and a set of actions. The rules are matched in parallel against the current state[2]. All of the rules that match, fire and can produce additions or deletions in the current state or generate external actions.

Production rules can only match against the agent's state not against other production rules. This is an important restriction as it means the agent is limited to purely procedural access of its knowledge encoded as production rules. Operator preconditions and effects are computed for the

[1] IMPROV, as of this writing, is implemented in version 7.0.1 of Soar. Complete manuals and the latest version of Soar can be obtained from http://www.isi.edu/soar/soar.html

[2] Of course, this match usually occurs on a single CPU machine and therefore is just a simulation of a truly parallel process. However, the important point is that first all rules are tested to see if they are matched, then all changes are applied to the state at once.

current state and the agent is unable to access the rules to determine other states where the operator would apply (or what effects it would produce). The production rules are only accessed through the match algorithm. No other part of Soar relies on access to the rules. This architectural restriction ensures that IMPROV's learning and correction methods cannot inadvertently depend on some form of declarative access to the agent's knowledge. It's important to realize that production rules, in themselves, are not inherently a declarative or procedural representation. It depends on the access the agent has to the knowledge in those rules that determines if they are procedural or declarative.

An example of a production rule that encodes the knowledge that the set-speed operator should be chosen when a car is going too fast might be represented as shown in Figure 3.5(a).

```
sp {controls*propose*operator*set-speed*too-fast
    (state <s>  ^superstate <ss>
                ^io-state <io>)
    (<ss> ^operator <so>)
    (<so> ^name drive-to ^destination <x>)
    (<io> ^car <car> ^sign <sign>)
    (<car> ^isa my-car)
    (<sign> ^isa speed-limit ^value <limit>)
    (<car> ^speed <speed>)
    (<car> ^speed > <limit>)
    -->
    (<s> ^operator <o> +)
    (<o> ^name set-speed ^car <car> ^speed <limit>)}
```

(a) Soar Rule

```
IF      goal(drive-to,<x>)
and     sign(speed-limit,<limit>)
and     isa(<c>,my-car)
and     speed(<c>,<speed>)
and     greater-than(<speed>,<limit>)

THEN  choose-operator(set-speed,<c>,<limit>)
```

(b) English Rule

Figure 3.5: An operator precondition as a Soar production rule

To avoid confusion over interpreting the complex syntax in these rules (e.g. ^x indicates an attribute, <x> is a variable, - signals a value should be deleted and + signals that a value should be added), in this report we will generally adopt an English pseudo-code for these production rules, just preserving the notation of <x> as a variable. The precondition rule from Figure 3.5(a) is shown in this English pseudo-code in Figure 3.5(b).

Operators in IMPROV are represented by three sets of production rules. An example of this representation for a simple operator is shown in Figure 3.6. The sets represent:

- The Operator Preconditions

  The conditions for when to select the operator are represented by a set of production rules and therefore can represent disjunctive preconditions. In IMPROV, search control is folded into these operator preconditions. An operator's preconditions only match when it *should* be chosen for the current task, not just when it *could* be chosen.

  The decision to incorporate search control into operator preconditions helps to simplify the correction process by reducing the range of types of knowledge in the system. However, it also means that general search control rules (such as not moving the same piece twice during a chess opening) may not be represented compactly and can be harder to learn.

- The Operator Actions

  The actions of an operator are also represented by a set of rules and may contain planning knowledge (e.g. that the accelerate operator should lead to a higher speed) and execution knowledge (e.g. that executing the accelerate operator in the external environment requires

```
Operator Preconditions

IF     goal(drive-to,<x>)                       IF     goal(drive-to, <x>)
and    sign(speed-limit,30)                     and    road-quality(unpaved)
and    isa(<c>,my-car)                          and    isa(<c>,my-car)
and    speed(<c>,<speed>)
and    greater-than(<speed>,30)                 THEN  choose-operator(set-speed,<c>,10)

THEN  choose-operator(set-speed,<c>,30)

Operator Actions

IF     current-operator(set-speed,<c>,<new>)    IF     current-operator(set-speed,<c>,<new>)
and    speed(<c>,<old-speed>)                   and    speed(<c>,<old-speed>)
and    not-equal(<old-speed>,<new>)             and    not-equal(<old-speed>,<new>)

THEN add speed(<c>,<new>)                       THEN remove speed(<c>,<old-speed>)

Operator Termination

IF     current-operator(set-speed,<c>,<new-speed>)
and    speed(<c>,<new-speed>)

THEN finished-operator(set-speed,<c>,<new-speed>)
```

Figure 3.6: Representing an operator as production rules

pressing the accelerator pedal). One of the tasks for IMPROV is to learn to associate the correct planning knowledge with the execution knowledge.

- The Operator Termination Condition

  The final set of production rules indicate when this operator is complete–what the goal is for this operator. For example, in a `set-speed 50` operator, this would be when the car had reached 50mph. In IMPROV, the meaning of the operator's name (and parameters) derives from its termination condition. That is, `set-speed 50` is the operator with a goal of getting the car's speed to 50mph[3]. This representation is consistent with standard operator models, as motor level operators will have the termination condition of generating external behavior. However, this explicit representation of termination conditions also allows operators to be used as plans.

  A limitation of IMPROV's goal language is that it only supports binary goal conditions. A goal is either achieved or not. Later, we will show that IMPROV only learns when its goals are not achieved. This means there is no implicit preference for optimality, for instance that shorter solutions should be preferred over longer solutions. IMPROV could be extended to learn when the agent failed to achieve a sufficiently optimal solution (using some metric). In this case, rather than learning why one sequence of actions leads to failure, while another leads to success; IMPROV would learn why one sequence of actions leads to a better success than an alternative sequence. Extending the goal language to include a wider range of evaluations would be a natural future extension to IMPROV.

  One important assumption is that operators that generate external behavior receive feedback from the environment indicating when they are still taking affect and when they are complete. For example, in a `brake` operator, that "braking" is occurring. This assumption limits the temporal extent of the operator's effects, simplifying the problem for the learner of determining what changes in the environment are a result of the agent's actions. Without this

[3] Note that this condition is separate from the knowledge of how to implement the operator, i.e. how to get the car to a speed of 50mph.

feedback assumption, the learner would have to decide when the external action had stopped producing changes in the environment.

This representation is sufficient to meet the requirements for an IMPROV agent as follows:

1. Organized into Operators

   The agent maintains declarative access to just the name of the operator (as it is explicitly added and removed from the state). This allows the agent to reason about individual operators and correct the knowledge associated with a particular operator.

2. Expressiveness for Complex Effects

   Production rules provide a very general model for the effects of an operator. Conditional effects can easily be represented by application rules that include more than just the operator in their conditions (see Figure 3.7(a)). Sequential effects and actions with duration are both encoded the same way, as a series of application rules. Each rule in the sequence tests for the effect of the previous rule, leading to a chain of effects over time (see Figure 3.7(b)). Looping and iterative behavior can be produced by rules that modify a counter or test for a condition that only eventually becomes true (see Figure 3.7(c)). In Chapter 11, we will show how these individual rules can be replaced by complete, primitive operators to enhance learning.

(a) Conditional Effects

```
IF    current-operator(brake)
and   weather(raining)
and   isa(<c>,my-car)
and   tires(<c>,worn)

THEN  add  skid(<c>)
```

(b) Sequential Effects

```
IF    current-operator(brake)              IF    current-operator(brake)
and   brake-pressure(<c>,0)                and   brake-pressure(<c>,10)
and   isa(<c>,my-car)                      and   isa(<c>,my-car)

THEN add     brake-pressure(<c>,10)        THEN add     brake-pressure(<c>,30)
     remove  brake-pressure(<c>,0)              remove  brake-pressure(<c>,10)
```

(c) Iterative Effects

```
IF    current-operator(brake)
and   isa(<c>,my-car)
and   speed(<c>,<speed>)
and   greater-than(<speed>,0)
and   time(<t>)

THEN add     speed(<c>,(<speed> - 5))
     add     time(<t> + 1)
     remove  speed(<c>,<speed>)
     remove  time(<t>)
```

Figure 3.7: Examples of complex operator actions

3. Expressiveness for Large Precondition Sets

   Individual production rules represent conjunctive preconditions. Together, a set of rules represent different disjunctive preconditions.

4. Efficient Testing of Operator Preconditions

   As we have already mentioned, all of IMPROV's domain knowledge is encoded as productions and those productions can only be accessed procedurally, through a fully symbolic match algorithm. All of IMPROV's planning, execution, error detection and error correction methods only rely on having declarative access to the names of operators and procedural access to the remainder of the agent's domain knowledge.

   The efficiency of this representation has been demonstrated for complex, real-time domains including the control of simulated aircraft [Pearson *et al.*, 1993a; Pearson *et al.*, 1993b] and tactical air combat [Laird *et al.*, 1995; Tambe *et al.*, 1995]. The match algorithm has also been tested on cases of matching over 100,000 rules [Doorenbos, 1993] and the counter-intuitive result has been shown that, for some systems, there is no increase in match cost as the number of rules increases.

5. Efficient Calculation of Operator Effects

   Operator effects are encoded as production rules. Therefore, the same efficient match algorithm is used to compute operator effects as is used to compute operator preconditions.

6. Efficient Methods for Changing Operator Knowledge

   As we have just shown, IMPROV's domain knowledge consists of a collection of production rules. One approach to changing this knowledge (as IMPROV learns) would be to identify and modify the rule(s) that lead to incorrect behavior. However, this approach can be expensive if it requires some form of matching against the full rule base, leading to slower corrections as the agent's knowledge grows in a similar manner to declarative systems. As we want to apply IMPROV to tasks that are time-critical (E6) we'd like to avoid this slow-down.

   Instead, our approach is to incrementally add rules that work together with the agent's existing knowledge to change the incorrect operator selection or implementation [Laird, 1988]. This method will be described in detail in Section 9.1.4. By adding just a few new rules, each correction can be made in constant time. As the old rules are not discarded this means the rule base will grow in size over the life of the agent. This could slow down the agent's execution performance. However, our hope (born out by the empirical evidence of matching 100,000 rules [Doorenbos, 1993] without increased cost) is that improved match algorithms will ensure no long term slow down as more rules are learned.

It is worth noting that the example of a naturally procedural operator from Figure 3.2, would have it's preconditions expressed in Soar rules as shown here, in Figure 3.8.

This operator would not present any problems for IMPROV's learning, planning or execution methods. However, the current implementation of IMPROV's inductive learner is not sufficiently expressive to learn this rule, in this form. The learner would be forced to approximate it symbolically. This operator could still be included, as shown here, as part of the agent's initial (possibly incorrect) domain knowledge and it would cause no problems. Errors in this knowledge would be corrected through symbolic approximations.

```
sp {propose*operator*move-arm*compute-sum
    (state <s>  ^x <x> ^y <y>)
    -->
    (<s> ^total (+ (* (sin (power <x> 7)) (cos (power <y> 0.3)))
                 + (* (power <x> 2) (log (power <y> 5)))
                 + (* (power <x> 1) (power <y> 2))
                 - 12))}

sp {propose*operator*move-arm*if-sum-positive
    (state <s>  ^total > 0 ^x <x> ^y <y> ^r <r>)
    -->
    (<s> ^operator <o> +)
    (<o> ^name move-arm ^x <x> ^y <y> ^r <r>)}
```

Figure 3.8: Encoding naturally procedural preconditions

CHAPTER 4

Planning

4.1 Planning vs Execution Policies

Autonomous agents functioning in an external environment must select actions to achieve their goals. The choice of actions can be made in at least two different ways. The agent can maintain an *exhaustive execution policy* that maps the current state and goal to an action. The mapping need not enumerate every goal and state combination, but the mapping must cover the entire space (see Figure 4.1(a)). In learning systems, this is the approach usually favored by reinforcement learning systems (see Section 3.1.2). However, relying on an exhaustive mapping presents problems when these methods are scaled to increasingly large environments. The usual alternative, Figure 4.1(b),

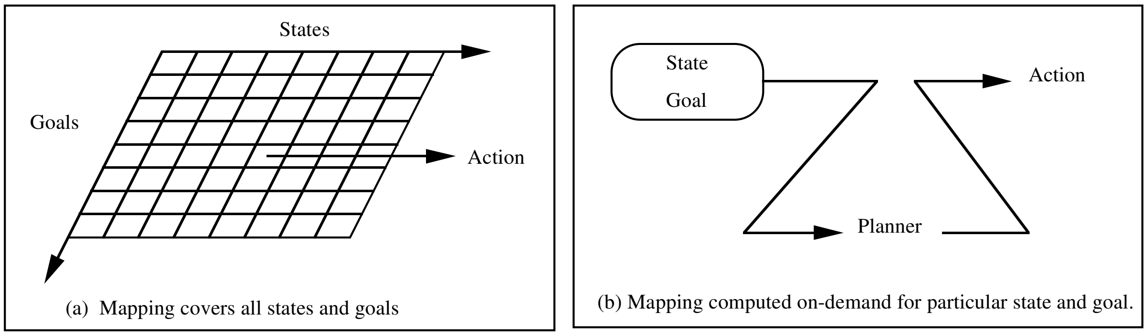


Figure 4.1: Alternative methods to select external actions

is to instead rely on an internal *planning* representation to determine the best action by simulating sequences of actions applied to the current state and goal. In this way, a planning agent determines the appropriate action on demand.

The main advantages of using an execution policy are that actions can be selected quickly (E6), without the cost of planning, and actions can produce arbitrarily complex effects (E3). This is because the agent does not maintain a model of the effects of an actions, it merely tries to decide when to take that action. The disadvantages of an execution policy are that it may be very large (for large state and goal spaces, E1and E2) and learning may not transfer well from one task to another. These limitations lead many reinforcement learners to limit the environment to a small, fixed state representation and a single goal.

The main benefits and disadvantages of planning are essentially the reverse. Actions are selected more slowly as the agent must plan, and actions are often limited to simple, discrete effects so that the planner can represent and reason about them. In return, planning can be applied to large state and goal spaces as the agent builds plans in response to a specific goal and state, rather than recording actions for each possible state and goal. Also, planning leads to better transfer of learning between tasks. For example, if the agent learns that braking happens more slowly in the rain, this new knowledge can be applied to all tasks that involve braking.

In an effort to combine the strengths of both methods, IMPROV uses a hybrid approach. Planning knowledge is used to determine the best action in response to a particular state and goal. The results of planning are cached as operator selection knowledge, leading the agent to incrementally build an execution policy over time. In solving future problems, the agent first checks the cached execution knowledge and only plans if no previously cached knowledge is appropriate. The underlying planning knowledge makes IMPROV applicable to tasks with large state and goal spaces (E1and E2) while the cached execution knowledge gives faster execution times (E6). The space required for the cache is not unbounded in the way that an exhaustive policy can be. Instead, the space required is bounded by the parts of the goal and state space that have been visited by the agent.

IMPROV's approach therefore defines its learning problem:

1. IMPROV learns and corrects planning knowledge

2. IMPROV learns and corrects execution knowledge based on corrected planning knowledge

IMPROV learns and corrects planning knowledge and then uses the corrected planning knowledge to learn new execution knowledge.

4.2 Plan Representation

An agent that can plan can solve a wider class of problems than an agent that is limited to a policy for selecting actions based on the current state and goal. The environment further constrains the representation of this plan. These constraints lead to a series of functional requirements for IMPROV's planning representation.

4.2.1 Plan Representation – Functional Requirements

1. Reactive Plan Execution

   The plan's representation should support efficient responses to unexpected (but planned for) changes in the environment. For example, in a plan to walk through a closed door, if someone happens to open the door before the agent reaches it, we would like to avoid replanning to determine that the agent can proceed through the door. This is particularly important in environments with unpredictable external processes (such as traffic lights) or multiple agents (E9).

2. Efficient Plan Execution

   The plans must be efficient to execute, to meet the requirements of time-critical environments (E6). This becomes a larger issue when plans are extended to include conditional branches or iterative steps.

3. Access to the Currently Executing Operator

   In executing the plan, the agent must have access to a symbol that represents the currently executing operator (i.e. its name). This access is necessary during learning as IMPROV will reason explicitly about the operators in a plan to improve credit assignment in identifying erroneous operators. This requirement prevents the plan being compiled into a fully procedural representation (such as a finite state machine) without maintaining access to the operators that were used in building the plan.

4.2.2 Plan Representation – Implementation

IMPROV represents plans procedurally, as a hierarchy of operators. The termination condition of a "plan operator" becomes the goal for the plan, the operator implementation consists of the steps of the plan and the operator preconditions indicate when this plan should be chosen. Each step in the plan can either be implemented directly as production rules or can itself be an operator with its own implementation, leading to a hierarchy of plans/operators. Soar makes no distinction between these operators. The hierarchy develops naturally as Soar attempts to apply operators. If the agent has insufficient knowledge to directly apply an operator, the architecture generates a new subgoal of completing that operator, leading to the hierarchy[1]. After learning, the results of the lower level operators can be compiled into the actions of the high level operators, producing macro-operators.

Figure 4.2(a) shows one possible hierarchy of operators for the task of driving across an intersection. This hierarchy represents a plan to achieve the highest level operator, `cross-intersection`, for crossing an intersection with a traffic light. In this example, this operator has no direct implementation knowledge. Instead, it is achieved by a series of increasingly primitive operators. First, it is decomposed into a series of `set-speed` operators, allowing the agent to plan to reduce speed, stop and then move off when the light changes. The opportunity for this decomposition arises automatically from the impasse and subgoal that arises when the agent is unable to complete the `cross-intersection` operator directly. The intermediate level helps the agent decompose the problem into subproblems. Then it can plan to achieve each of the sub-operators, for example how to slow down to 20mph. This is done in terms of the motor level operators (`brake`, `accelerate` and `change-gear`). To summarize, the hierarchy allows complex goals to be decomposed into simpler subgoals until finally those goals can be achieved by plans that generate behavior.

Each node in the hierarchy is a full operator. Some of the rules that make up the operators are also shown in Figure 4.2(b),(c). The upper group are preconditions for the `set-speed` operators, while the lower group are preconditions for the motor operators which together form the actions of the `set-speed 20` operator. The "hierarchical plan" is not explicitly represented anywhere in the system. It arises from the interaction of the rules that make up the operators in the hierarchy.

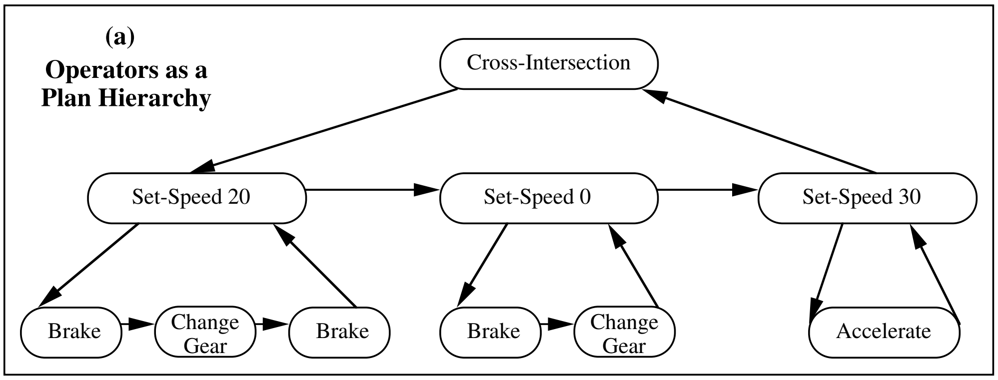
We will return to this very important identity between plans and operator actions later in discussing learning. Using the same representation for plans and operators, allows us to generalize correcting operator knowledge to correcting plans at any level in a plan hierarchy. It will also allow us to generalize a method for correcting operator preconditions to a method for correcting operator effects (see Chapter 11).

By representing plans as a hierarchy of operators, IMPROV can meet all of the functional requirements for plans:

1. Reactive Plan Execution

   The knowledge that guides the agent along a plan is encoded in the preconditions of the operators that implement that plan (see Figure 4.2(b) and (c)). There is no explicit "plan" data structure, instead the plan consists of many fine-grained "plan pieces" that activate

[1] The term "the agent's current goal" is interchangeable with the concept of achieving or implementing the current "super operator", i.e. the operator one level up from the lowest level of the hierarchy



(b)
Procedural Plan for Cross-Intersection

IF goal(cross-intersection)
and distance(far)
THEN choose-op(set-speed,20)

IF goal(cross-intersection)
and light(green)
and distance(close)
THEN choose-op(set-speed,30)

IF goal(cross-intersection)
and light(red)
and distance(close)
THEN choose-op(set-speed,0)

IF goal(cross-intersection)
and light(yellow)
and distance(close)
THEN choose-op(set-speed,0)

(c)
Procedural Plan for Set-Speed 20

IF goal(set-speed,20)
and isa(<c>,my-car)
and speed(<c>,<speed>)
and greater-than (<speed>,30)
THEN choose-op(brake)

IF goal(set-speed,20)
and isa(<c>,my-car)
and speed(<c>,<speed>)
and less-than (<speed>,30)
and gear(<c>,4)
THEN choose-op(change-gear,3)

IF goal(set-speed,20)
and isa(<c>,my-car)
and speed(<c>,<speed>)
and greater-than (<speed>,20)
and gear(<c>,3)
THEN choose-op(brake)

Figure 4.2: Example of a plan as an operator hierarchy

themselves as the environment changes. This representation ensures that the agent is reactive to the environment. To return to the example of walking through a closed door. If the door happens to open before the agent reaches it, the preconditions for `open-door` would no longer match while `go-through-door` would have its preconditions met, so it would be chosen. This means the agent would react to the change in the door's state without any need to re-plan.

An important additional property of this "distributed" plan representation is that behavior of the agent is not controlled by only the most recently generated plan. Instead, the agent's behavior is controlled by the runtime combination of all of its previous planning. As a result, the agent may begin using control rules learned from its most recent planning experience, but as a result of an unexpected change in the environment, rules from a much earlier planning experience might become relevant and guide the agent.

2. Efficient Plan Execution

   The knowledge for selecting the next step in a plan is encoded as operator preconditions, that are themselves production rules. This procedural knowledge is optimized for execution and can be matched very efficiently even as the size of the rule base grows.

3. Access to the Currently Executing Operator

   The name of the operator currently being executed is declaratively available to an IMPROV agent. The operator name (and parameters) are added to the agent's working memory, allowing the agent to reason explicitly about the operators that make up a plan. Because of IMPROV's restriction to procedural access this is the only part of the agent's plan knowledge that the agent has direct access to. The rest of the plan knowledge is encoded as rules that can only be accessed through execution.

4.3 Uncertainty Bounded Iterative Deepening (UBID)

IMPROV's commitment to a procedural knowledge representation constrains it's planning method. Most plan-space planners incrementally refine a plan by examining the preconditions and actions of STRIPS-style operators. IMPROV's procedural access to its operator knowledge along with the complex structure of its operators makes it difficult for IMPROV to search through a space of plans. Instead, IMPROV can use a variety of state-space planning methods, where the agent searches for a path between initial and goal states. As we expect this search can be usefully biased by prior experience we've adopted a planning method we call *Uncertainty Bounded Iterative Deepening* (UBID).

4.3.1 UBID – Functional Description

Uncertainty Bounded Iterative Deepening is a state-space planning method. UBID is an extension of the iterative deepening search method. In iterative deepening all plans are explored to a fixed depth. If the planner does not find a path to the goal, the depth bound is incremented and the search repeated. Iterative deepening is complete, is guaranteed to find the shortest plan and requires only a single state be maintained by the planner (keeping memory requirements low). UBID also explores all plans to a fixed limit. However, the limit is not based on the length of the plan as in iterative deepening. Instead, it is a measure of *uncertainty* associated with the plan. The uncertainty of a plan is the sum of the uncertainties associated with each operator in the plan. In this implementation, these uncertainties are derived from the learning component of the system and reflect how often the operator was useful in a similar situation. UBID includes all operators in its search (uncertainty measures guiding the search rather than relying on the correctness of operator preconditions) so eventually UBID will generate all possible operator sequences. This is important in searching for a sequence of operators to solve a problem when the agent's knowledge may be incorrect.

The key characteristic of the search is that the planner first explores paths that the agent believes are most likely to succeed based on its previous experience. This results in a deeper search in areas of the search space that have earlier proved useful to the agent, which can make planning much more efficient. An example of this is shown in Figure 4.3.

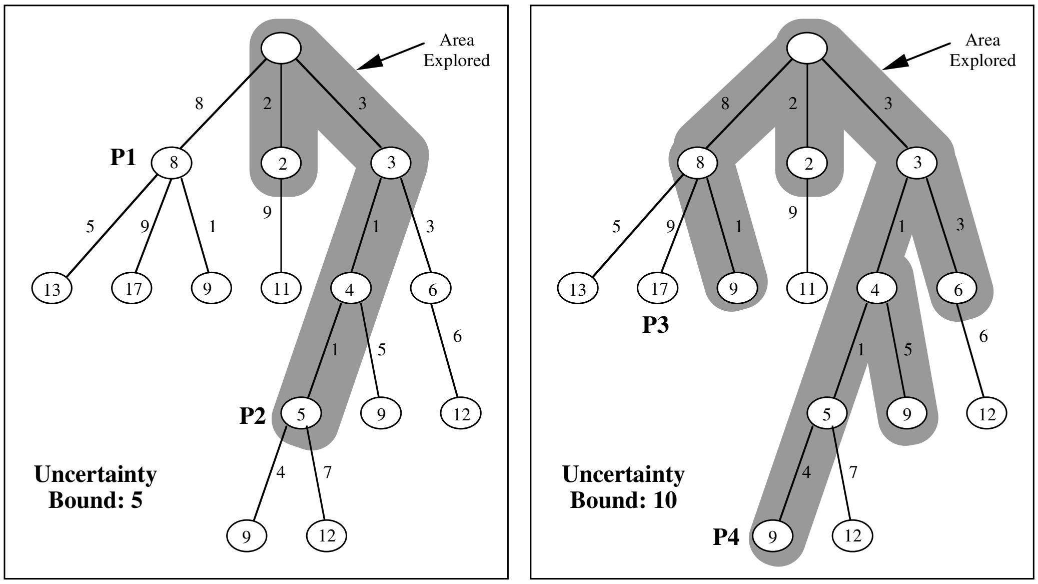


Figure 4.3: UBID explores previously useful parts of the space first

The nodes in this figure represent states, the branches operators, and the values on the branches are the uncertainties for each operator. A path from the top of the tree to a node represents a plan (or a partial plan) and the value in the state is the total uncertainty for that plan. The figure on the left shows the parts of the space explored when the uncertainty bound is 5. The figure on the right shows the nodes that have been explored when the bound is 10. In the left figure, notice that the path to `P2` is explored before `P1` and similarly in the right figure, `P4` is explored before `P3` even though `P2` and `P4` are deeper in the search tree.

4.3.2 UBID – Implementation

In this implementation of UBID, uncertainties are derived from the inductive learning module. The inductive module learns prediction rules that are increasingly specific (in a similar manner to the way decision tree learners build increasingly deep, and therefore more specific, trees during learning). In choosing an operator, the more specific the rule that matches to predict that operator, the higher its certainty (or the lower its uncertainty). For example, if an operator is chosen based on a very specific rule that tested `isa(<b>,block)`, `surface(<b>,rough)`, `on(<b>,table)`, `clear(<b>)` and `goal(stack,<b>,<a>)` then the operator would be assigned a lower uncertainty than an operator chosen on the basis of a very general rule (e.g. just `isa(<b>,block)`). When combined with the particular inductive learner used in IMPROV, the specificity of a rule is directly related to the number of times the operator was useful in similar situations, making it a useful heuristic to guide search.

In this implementation of UBID, uncertainties are in the range 1-20 and the bound is incremented by 20 for each iteration. The base uncertainty for an operator is (10-number of features matched, min. 1). The rules that suggest operators can record knowledge to avoid certain operators (e.g. don't choose pickup). To ensure a complete search, these operators are still considered, but 10 is added to their uncertainty values (leading to uncertainties in the range 11-20). The details of how these uncertainties are calculated and which operators are proposed will be discussed more fully in Section 8.2.2.

As we showed in Figure 4.2, IMPROV's plans are represented as a hierarchy of operators. The UBID search method can be applied to each level of this hierarchy as shown in Figure 4.4. In this example, the `cross-intersection` has no direct implementation knowledge, leading to an impasse and Soar creating the subgoal of completing `cross-intersection`. IMPROV then searches, using UBID, to find a sequence of operators that will achieve `cross-intersection`. Once this sequence has been found Soar's chunking mechanism, compiles the choices into new proposal rules for the `set-speed` operators. Examples of these rules are shown in Figure 4.2. When the plan is executed these rules will guide the choice of the operators and no search will be needed. The process can be extended recursively, as is shown in the lower part of Figure 4.4. In this case, the `set-speed 20` operator also has no direct implementation rules, leading to another impasse and a second UBID search. Again, once the search finds a plan that the agent believes will produce a speed of 20mph, new rules are created to guide the selection of the appropriate motor level operators (`brake` and `change-gear`). (A more detailed example is shown in Appendix A.1).

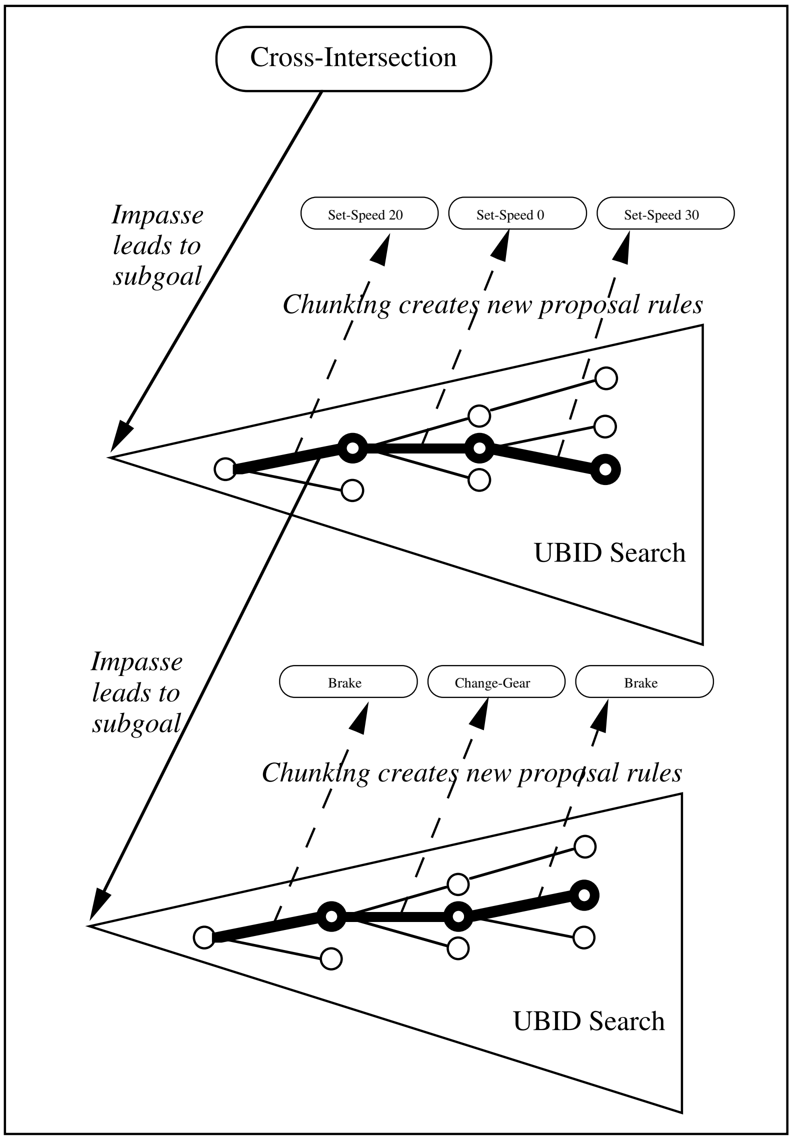


Figure 4.4: Repeated use of UBID to build a hierarchical plan

CHAPTER 5

Classifying Knowledge Errors and Performance Failures

5.1 Introduction

In designing an approach to correcting planning knowledge the main stages are:

1. Classification of errors

   Determining the errors that can occur in an agent's knowledge of the environment and the range of performance failures that can occur as a result of these knowledge errors. For example, not knowing that baking powder makes a cake rise is a knowledge error that leads to the performance failure of the cake falling.

2. Detecting performance failures

   Recognizing that a performance failure has occurred. The failure can occur and be detected during planning or during plan execution. For example, the ability to build a plan where the cake is in two places at once is a failure during planning; while burning a cake is a failure during execution. Both can result from an error in the agent's planning knowledge.

3. Solving the current problem

   Determining what is the correct behavior to achieve the agent's current goal. For example, finding a sequence of steps that leads to a properly cooked cake.

4. Learning a general correction for the future

   Deciding how to generalize from successes and failures to avoid performance failures in the future. In the baking example, this is the problem of deciding why one sequence of steps works and another does not. This generalization problem can be further decomposed into three parts:

   (a) Credit assignment–Which operators are incorrect?

   Determining which operator, or operators, has the incorrect knowledge that lead to the performance failure. If feedback from the environment is not complete and immediate, the agent will not necessarily know what operator caused an execution failure. For example, the failure of the cake to rise may be due to any of the steps that went into making the cake, the preparation of the ingredients, how they are combined or how they are cooked. The problem of identifying the cause of the failure is a problem of assigning credit (or blame) to the incorrect step.

(b) Credit assignment–How the operators are incorrect?

Having identified the operator, or operators, that lead to a performance failure, the agent must still determine what part of the operator is incorrect. The reasons for selecting the operator may be incorrect or the agent's model of the effects of the operator may be wrong. For example, the agent must decide that the reason the add-eggs step of the plan lead to a failure is that the egg shells had not been removed. In this case, the error is that the add-eggs operator lacks an extra precondition. The agent is assigning credit for the failure to a part of the operator's knowledge.

(c) Changing the domain knowledge

Having identified the cause of the failure, the agent must determine how to adjust its knowledge in order to avoid future errors. For example, how to modify the add eggs operator to reflect the new precondition.

We will explore each part of this error correction process over the course of the next chapters, starting with the question of how to classify errors and describing which of those errors IMPROV can correct.

Traditional approaches to classifying errors have been based on performance failures, such as failing to achieve a desired effect in the world [Mitchell *et al.*, 1986; Rajamoney and DeJong, 1987]. The scope of the learning system is defined in terms of the classes of performance failures that it can recover from. In this section we will propose an alternative method for classifying errors based on the agent's *knowledge*. By focusing on the domain knowledge, we can more clearly define the range of possible errors, and from these derive the range of possible performance failures. Then we can discuss the range of knowledge level errors that IMPROV can correct and which ones remain as topics for future work.

This analysis focuses on errors in the agent's domain knowledge, that is errors in the agent's understanding of the *processes* that govern the environment. To operate correctly, the domain theory depends on the agent having a correct knowledge of the *state* of the world and representing that state knowledge at the correct level of generality. Incomplete or noisy sensing can lead to an inaccurate state representation, in turn leading to performance failures. Even correct sensing can lead to errors if the agent's representation is inappropriate. For instance, if the agent represents car speeds as simply `moving` or `not-moving` it will be difficult for the agent to learn to control its own car or avoid others. This research does not address these forms of representational errors, focusing instead on the errors in the agent's process, or domain knowledge.

5.2 Knowledge Level Errors

In our formulation, the agent's domain knowledge consists of its knowledge about its operators. In particular, this means the preconditions, actions and termination conditions for those operators. As mentioned earlier, an operator's actions will consist of both planning knowledge and separate execution knowledge. For example, when driving a car, planning knowledge for pressing the brake would include the car slowing, while execution knowledge would be to send motor commands to press the brake pedal. In agents that plan, performance failures are the result of incorrect planning knowledge. For example, in trying to reach a speed of 20mph from 40mph, the agent incorrectly sends a command to press the accelerator pedal. In a planning agent, this is because the planning knowledge associated with the `accelerate` operator incorrectly includes the car slowing down. The error is corrected by learning the correct planning knowledge for `accelerate` and then using this corrected planning knowledge to correct the agent's execution knowledge (see Figure 5.1). This

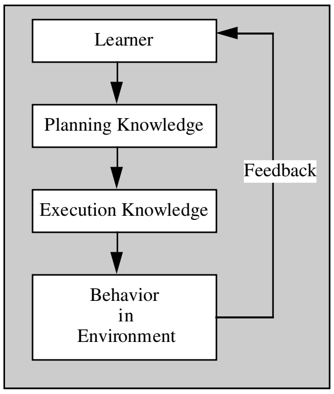


Figure 5.1: Correcting execution by correcting planning knowledge

leads the agent to avoid the `accelerate` operator when trying to reduce speed and so generate the correct external performance. In short, the task for agents that plan is to learn correct planning knowledge and this will lead to correct execution behavior.

This approach to the agent's domain knowledge ensures that all errors in the actions associated with an operator can be seen as errors in the planning knowledge for that operator. This observation, combined with the operator based model for agent knowledge, leads to a small set of possible domain knowledge errors:

- Overgeneral Preconditions

  A conjunctive operator precondition is missing or an extra disjunctive precondition is present. For example, the agent might think you must stop at all traffic lights, so the operator `set-speed 0` is missing the precondition that the light should be red. An example of an extra disjunction is an agent which believes you should stop for traffic lights or street lights. The disjunctive street light precondition makes the operator overgeneral. These overgeneralities could be programmed into the agent's initial domain knowledge, or could be the result of overgeneral learning.

- Overspecific Preconditions

  A disjunctive precondition is missing or an extra conjunctive precondition is present. For example, the agent may believe that you should only change gear when going up hill. The operator is missing the disjunctive precondition that you should also change gear when slowing down. An example of an extra conjunctive precondition is an agent that learned to drive at night, believing that a precondition of driving is that the highlights are on.

- Incomplete Effects

  The agent is not aware of all of the effects that executing the operator will produce. The planning knowledge incompletely describes the effects of executing the operator in the environment. For example, the agent is unaware that braking hard causes "fade" in the braking system, leading to longer stops in future.

- Extraneous Effects

  The agent believes that the operator produces more effects than it really does. For example, that coming to a stop at a red light causes the light to turn green.

- Missing Operator

  An entire operator is missing from the domain theory. The operator could be a motor level operator, for example, when the agent is unaware that a car has cruise control. It could also be when the agent lacks a more abstract operator that is useful for problem decomposition. These abstract operators can become subgoals during problem solving, allowing the agent to search for subplans to complete the operator. For example, a `change-lanes` operator is useful in building a plan to pass a slower car. In our formulation, errors in the termination conditions of operators are treated as cases where the operator with the correct termination condition is missing. For example, the `change-lanes` operator may have termination conditions that cause it to terminate before the agent has changed to the new lane. This `change-lanes` operator will not be particularly useful in decomposing problems. The solution, in our model, is to create a new `new-change-lanes` operator that has the correct termination conditions. The benefits and disadvantages of this approach have not been extensively explored as IMPROV is unable to create new operators or otherwise correct errors in termination knowledge.

The situation where a precondition or effect is incorrect can be seen as a combination of two other cases. For example, an incorrect effect is a case of incomplete effects (the correct effect is missing) and extraneous effects (the incorrect effect is present). In correcting these types of errors, we'll see later that IMPROV must generalize one operator while specializing another.

In summary, IMPROV can correct errors in operator preconditions and effects, but cannot create new operators or correct errors in termination conditions. IMPROV can recover from overgeneral or overspecific operator preconditions and correct for incomplete or extraneous effects. If the agent's initial set of operators is incomplete, IMPROV may be unable to solve certain complex tasks that require a problem to be decomposed to make it tractable. For example, searching for the correct route in a long cross-country journey, without an appropriate decomposition (such as defining intermediate stopping points along the way) may be intractable.

5.3 Mapping Knowledge Errors to Performance Failures

Errors in the agent's domain knowledge can lead to failures during either planning or plan execution. This mapping is summarized in Figure 5.2. For example, overspecific preconditions for an operator (believing that cars can only drive on paved roads) may mean the agent cannot construct a plan when faced with a dirt road. The important point from the figure is that each class of knowledge level error can give rise to a range of performance failures. The range of performance failures is described briefly here, summarizing our more detailed analysis in [Huffman *et al.*, 1993].

	Planning Failures		Execution Failures		
	Incomplete	Impossible State	Prec. Unmet	Effects Unmet	Explicit Detection
Overgen. Preconds.		•		•	•
Overspec. Preconds.	•				
Incomplete Effects	•	•	•	•	•
Extra Effects	•	•	•	•	•
Missing Operator	•				

Figure 5.2: Mapping from knowledge errors to performance failures

This section describes the range of possible performance failures that arise from domain knowledge errors and the approaches taken by existing learning systems to detect these failures.

1. Planning Failures

   (a) Incomplete Plan

   The agent is unable to construct a sequence of operators to reach the goal state. The agent must be able to distinguish between an unachievable goal and errors in its knowledge. For example, in trying to plan a path between two others cars, the agent might conclude that it's not possible. This is either because there is not enough space (so the goal is really unachievable) or because the agent's planning knowledge (about the width of the car being driven) is wrong and there is enough room.

   To detect this type of failure, the agent needs additional knowledge to signal that it should be possible to build a plan for the current goal. For example, a number of systems attempt to construct explanatory parses of training examples (e.g. [VanLehn, 1987; Hall, 1986; Hall, 1988]). The inability to build an explanation is equivalent to an incomplete plan failure.

   Rajamoney and DeJong [1988] also examined cases where the agent knows that only one plan should be formed, but multiple plans are found. In many domains, multiple paths to the goal are possible, so forming multiple plans does not necessarily constitute an error. Again, the agent must have additional knowledge indicating that only one plan should be possible. For example, in navigating a system of one way roads, if the agent finds two paths to the same point this may signal it is planning to go the wrong way up one street. This is a special case of not being able to build any plan, because in both cases the agent has knowledge about the space of plans that can be built.

   (b) Reaching an Impossible State

   The agent uses planning knowledge about its operators to search through states of the domain. If this knowledge is incorrect, the agent may reach a state that it knows should be impossible; for example, reaching a state where the agent is in two places at once. This is different from a failure state where the agent's goal is not achieved (for example, building a plan that takes the agent to the wrong place). Reaching a failure state during planning does not indicate an error in the agent's knowledge, while reaching an impossible state does.

   To detect an impossible state the agent must have explicit knowledge about the states that are invalid. For example, the knowledge that an agent can only be in one place at a time.

2. Execution Failures

   Execution failures occur when the agent generates a plan but is unable to successfully execute it. Failures occur when the agent's expectations (reflected in its planning knowledge) are not met in the external world. If the agent has no planning knowledge and instead relies on an exhaustive execution policy to guide the choice of operators during execution, it is limited to detecting errors through an explicit error signal from the environment (e.g. reward or reinforcement). This is because the agent is not explicitly modeling the effects of its actions and therefore cannot directly verify the accuracy of its predictions. Agents with planning knowledge can use it to recognize execution failures when the planned behavior differs from the observed behavior during execution. The classes of execution failures caused by incorrect planning knowledge are as follows:

(a) Incomplete Execution

This failure occurs when during plan execution, the agent is unable to reach the goal. For example, having built a plan to drive to a particular location, the agent arrives somewhere else.

To detect the incomplete execution of a plan the agent must compare the planned behavior to the actual results of execution. The standard approach to making this comparison is by explicitly comparing a declarative model of the operator to the states before and/or after the operator is applied. This is the approach used in EXPO [Gil, 1992; Gil, 1994], LIVE [Shen and Simon, 1989; Shen, 1994], OBSERVER [Wang, 1995; Wang, 1996] and TRAIL [Benson, 1995] (although TRAIL records a sequence of states to allow for actions with duration).

This monitoring usually assumes complete and immediate sensing to detect:

- Next operator's preconditions are not achieved
  This failure occurs when the next operator in the plan cannot be executed because one its preconditions does not hold in the external world. For example, the accelerate operator requires the engine to be running, but a missed gear change has stalled the car. In general, this type of failure indicates that an earlier operator failed to achieve the missing precondition, an earlier operator unexpectedly clobbered the condition or an external agent/process unexpectedly invalidated the precondition.
  To detect this type of failure the operator's preconditions are matched against the observed environment, prior to executing the operator. If the operator's preconditions are not met, an error may have occurred earlier in the execution of the plan. It is worth noting that the previous operator is not necessarily the one at fault. The agent can only be certain that the previous operator is at fault if the agent has complete, immediate sensing, there are no exogenous events and the expected effects of operators are explicitly verified. The agent may have correct domain knowledge but an unexpected (or unpredictable) event can lead to this type of execution failure. For example, the plan to cross an intersection includes the precondition that the light is green, but the light unexpectedly turns red, preventing the plan from executing successfully.
  These basic methods have been extended, in the planning community, to more complex plan monitoring schemes where state features, that are preconditions of later operators, are monitored during execution (e.g. [Reece and Tate, 1994; Sanborn and Hendler, 1988; Hard *et al.*, 1990]). These *active* monitoring approaches assume the same declarative access to operator descriptions.
- Previous operator's effects are not achieved
  This failure occurs when one or more of the intended effects of the last operator to be executed do not occur. For example, the agent checks that after executing a `close-door` operator, the door is really closed.
  To detect this failure each of the operator's actions are verified as having been achieved in the external world. If none of the actions have been achieved, this may suggest that the operator failed to apply at all, perhaps because there was an unknown precondition of the operator [Carbonell and Gil, 1987]. This monitoring usually assumes actions produce discrete, deterministic effects, limiting the scope of actions that can be modeled. TRAIL [Benson, 1995; Benson and Nilsson, 1996] is an exception to this, explicitly monitoring one primary effect of an operator but also probabilistically modeling a set of possible side-effects of the action.

These detection methods classify failures as the inability to accurately predict changes in the environment. In stochastic environments, or environments with multiple interacting agents or other exogenous events, accurate predictions may be impossible. IMPROV takes an alternative approach where failures are based on the agent's inability to achieve its goals, rather than accurately predicting the environment. IMPROV's method only requires procedural access to the agent's knowledge and is presented in Chapter 6.

(b) Failure State Reached

This failure occurs when the agent detects that it is impossible to reach the goal from the current state. The agent may have explicit knowledge in the form of a "theory of failure" to determine that the current state should be classed as a failure. Gupta [1987] used this approach with a theory that included knowledge of explicit failures, such as when a robot's gripper melts from grasping a hot part.

Reinforcement learning systems also use the approach of explicitly detecting when failure states have been reached. The agent has knowledge that a specific behavior should be achieved and any failure in achieving this behavior corresponds to an error. For example, the agent could be trained to drive at a certain speed. Any difference between the target speed and the agent's actual speed is taken to indicate an error. This method can be generalized to situations where feedback (or reward) is delayed. The predicted reward for a state should match the reward when that state is reached, allowing the feedback from the final state to be passed back to earlier states [Sutton, 1988; Watkins and Dayan, 1992].

To summarize, detecting planning failures requires the agent to have knowledge about the task, separate from knowledge about how to do the task. This knowledge is either that creating a plan should be possible or that specific states are impossible. In contrast, planning agents can detect execution failures by comparing the agent's planning knowledge about the task to the agent's behavior in the environment. In the next chapter we will look more closely at this capability and present a method for detecting execution failures that only requires procedural access to the agent's domain knowledge.

CHAPTER 6

Detecting Execution Failures

In the last chapter we developed a theory of errors based around the agent's domain knowledge and outlined how these knowledge errors lead to performance failures. We also described how most existing systems that build plans and then detect errors in their execution (e.g. CHEF [Hammond, 1989; Hammond *et al.*, 1990], EXPO [Gil, 1992] and Phoenix [Howe and Cohen, 1991]) typically rely on an explicit comparison of the sensed environment to the agent's operator knowledge at each step of a plan. This class of systems typically rely on three distinct components during plan execution:

1. A Declarative Plan

   The plan lists the sequence of operators that the agent believes should be executed to reach the goal.

2. Plan Monitoring

   Before each operator is executed its preconditions are checked to ensure they are satisfied. After execution, the agent checks that all of the expected effects have occurred. This is the step that requires declarative access to the operator's preconditions and effects (which we have argued in Section 3.1 may be expensive to compute).

3. Reactive Executive Module

   Critics or other executive modules monitor the plan's progress and decide if steps can be skipped or must be repeated. This allows the agent to react to unexpected events in the environment.

As IMPROV may have only procedural access to its domain knowledge, it cannot employ this form of explicit monitoring. This leads us to an alternative approach that combines these three, distinct modules into a single process.

6.1 Functional Description

The central observation that underlies IMPROV's error detection method is that the existence of a plan indicates that the agent should be able to reach its goal without further deliberation. The ability of the planner to construct a plan indicates that the agent now has sufficient *directly available* knowledge to reach its current goal. This was the purpose of planning in the first place. IMPROV detects errors by identifying when additional deliberation is required after the agent starts executing a plan. This approach only needs procedural access to the agent's operator preconditions and actions.

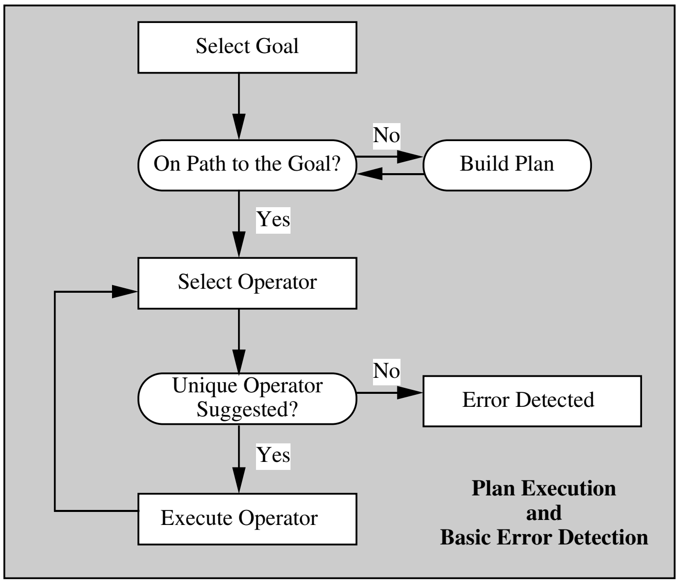


Figure 6.1: Plan execution and basic error detection

6.1.1 Detecting progress to the goal

The basic error detection method is shown in Figure 6.1. The first step is for the agent to recognize it has a plan that matches the current goal. If the agent doesn't have a plan, the default behavior is to build one. Once the agent starts executing the plan, IMPROV detects if the agent reaches a state where it no longer knows how to make progress towards the goal. This occurs when the agent reaches a state where no operators are suggested or when the agent has conflicting suggestions on how to proceed.

Consider the example of a car driving through an intersection using the plan shown in Figure 6.2. If the light turns red as the car approaches then the agent will execute `set-speed(30)`, `set-speed(10)` and `set-speed(0)`. At each step the agent's knowledge suggests exactly one operator to be executed next, so the agent is deemed to be making progress to the goal. If however, the agent is unaware of the need to change to a lower gear, the car will stall and the preconditions for `set-speed(30)` will not be met when the light turns green (namely `engine(running)`). As no other operator will be proposed, this is detected as a plan execution failure (see Figure 6.3, *italics* show the rules that fire during plan execution).

This basic method can be extended to detecting errors at multiple levels in an operator hierarchy. In particular, errors in the planned effects of operators that produce external behavior are detected through the implementation of the external operator as a series of more primitive, *single-effect* operators (see Figure 6.4 for an example). The `brake` operator will generate external behavior when executed, causing the agent to press the brake pedal. The lower level, `dspeed` operators model the expected series of changes to the agent's speed. This sequence of single-effect operators is sufficient to represent complex conditional or sequential effects that occur over time.

The primitive `dspeed` operators are not required to produce external behavior as they represent the planned effects of the agent's actions. After the `brake` operator has been chosen, the agent sends the brake command. At this point, the preconditions for the first `Dpseed -3` operator are

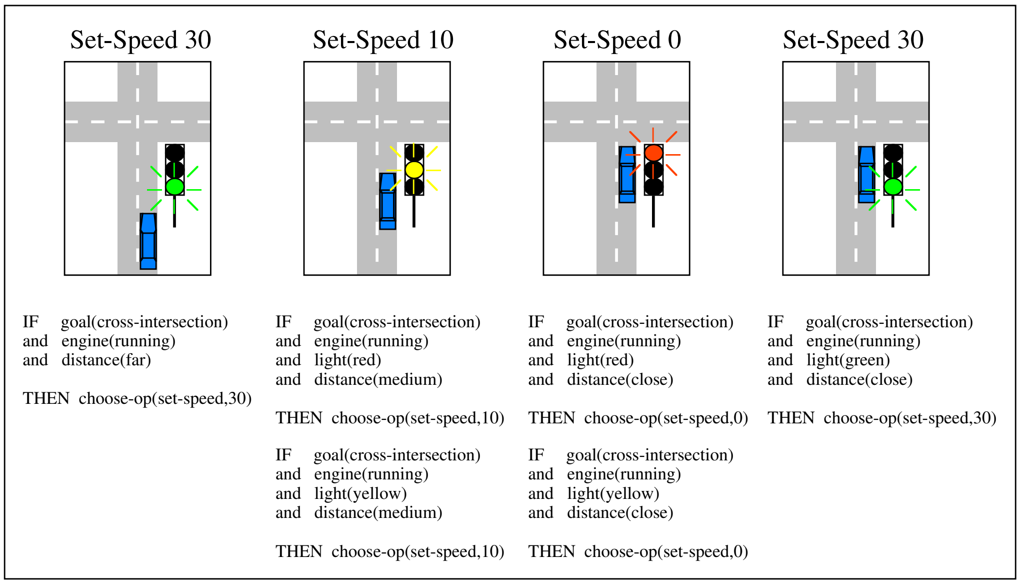


Figure 6.2: Example plan to drive across an intersection

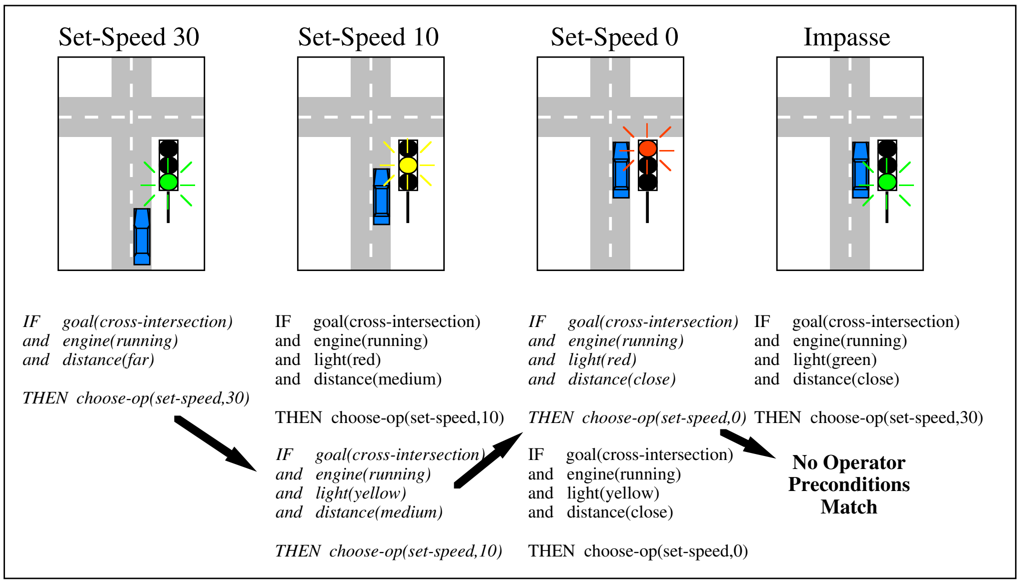


Figure 6.3: Impasse signals an execution error

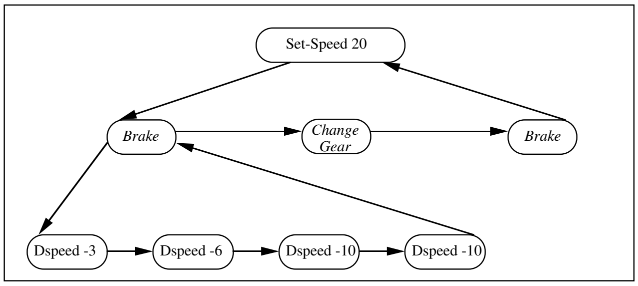


Figure 6.4: Planned effects of external brake command

met, so this operator is selected. IMPROV waits for the agent to sense the environment, moving the agent to a new state. If no primitive operators match the newly sensed state, an error is detected in the same manner as in Figure 6.2. There is no explicit comparison that the expected effects were observed. For example, if the agent decelerates by -3 (has a Dspeed of -3) then the next operator in the sequence (`Dspeed -6`) will match and will be selected. If instead the car unexpectedly accelerates, for instance, then no operator will match and IMPROV will detect an error. In this way, IMPROV extends its method for detecting errors to all levels of the operator hierarchy, relying on just procedural access to the preconditions and actions of the operators in that hierarchy. We return to this issue in Chapter 11 on learning operator effects.

6.1.2 Detecting loops

IMPROV's basic error detection method fails if the agent inadvertently cycles back to an earlier part of the plan, in which case it will believe that it is still making progress. In our example, if the agent has a `start-engine` operator then after stalling it may repeatedly turn the key and stall the car again, because it has still not changed gear. The basic method fails to detect this as an error as the agent repeatedly selects the same operator, believing that this is the next step in the plan. This problem is complicated by the fact that the states may never repeat exactly, as the agent moves to states in which the only differences are irrelevant to the current task (e.g. the pedestrians on the sidewalk have moved). The key is to recognize that the *task-relevant* parts of the state have been repeated.

IMPROV's approach is to divide the state space into equivalence classes based on operator preconditions. For each operator precondition (conjunctive clause), all states matched by that precondition are equivalent[1]. For example, if one precondition for the `set-speed(30)` operator is `light(green) and at(intersection3)` then all states where the light is green and the car is at the intersection are considered equivalent with respect to this precondition. IMPROV detects a loop if the agent returns to a state in a previously visited equivalence class. For example, having chosen `set-speed(30)` using the above precondition, if the agent later returns to a state where it's `light(green) and at(intersection3)` it will recognize this as an error. The classes are linked to the instance of the agent's current goal, so the agent does not detect a loop when it crosses the same intersection on a different day (as it will no longer be pursuing the same goal instance).

By basing the loop detection method on the conditions in operator preconditions IMPROV ignores differences that are not task-related. In our example, the agent would still detect a loop even when there are different cars present at the intersection on the second occasion, as these were not relevant to the selection of the operator.

This method allows the same operator to be selected multiple times during the course of a goal, as long as different preconditions are used in each selection. Iterative tasks are possible if there is either an observable external change or a change to the agent's internal state. For example, in turning a finely threaded screw the agent may be unable to sense the external change but could still maintain an internal counter, allowing the agent to distinguish between the different states.

It is possible to construct iterative plans where these conditions don't hold. For example, the agent's plan might include a step to turn a volume control until sound becomes audible. For IMPROV to correctly determine each small turn does not constitute a loop, the planner must make explicit the concept that turning the control will eventually lead to audible sound (a concept it needs to build the plan in the first place). Otherwise, the knowledge in the plan is insufficient to determine whether the agent is making progress or not. The accuracy of IMPROV's loop detection

[1]More precisely, the precondition must also be matched using the same values for all variables in the precondition.

method is bounded by the explicit knowledge encoded in the plan. The planner should therefore include any implicit knowledge used to build the plan as an explicit part of the plan.

IMPROV's combined method for detecting errors, by detecting lack of progress to the goal supplemented by loop detection is shown in Figure 6.5. Errors can be detected either from an

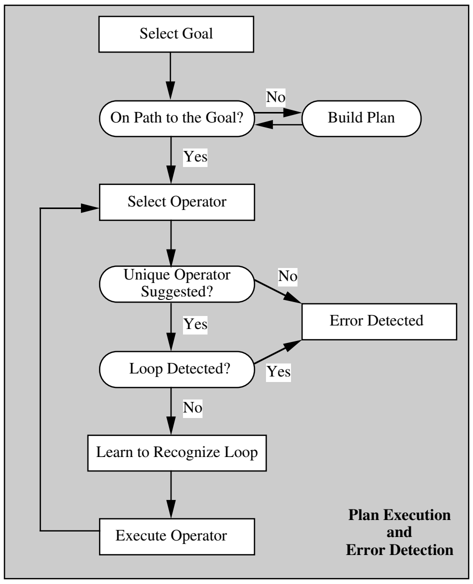


Figure 6.5: Plan execution and full error detection

unexpected impasse, indicating that the agent has deviated from the plan or through detecting a loop back to an earlier part of the plan. To summarize, the functional requirements for an agent to use IMPROV's error detection method are:

1. Detect when on a path to the goal

   The agent has determined a sequence of steps, a plan, that will take it to the goal. More explicitly, the agent now has sufficient directly available knowledge to reach the goal.

2. Detect when have left the path to the goal

   The agent must recognize when it has moved beyond the scope of the original plan. This is achieved by verifying that the current state matches the preconditions of precisely one operator. This is appropriate as the operator preconditions, in this framework, include control knowledge, ensuring only one operator at a time has its preconditions satisfied.

3. Detect loops to earlier states

   The agent needs a method to detect when it has returned to an earlier state in the plan. This problem is simplified to detecting the re-use of the same conjunctive precondition for an operator in achievement of the current goal (or super-operator).

This set of requirements is very different from the traditional requirements we presented earlier:

1. A Declarative Plan

2. Plan Monitoring

3. Reactive Executive Module

One important property of IMPROV's approach is that a single mechanism, matching operator preconditions, supports reactive plan execution and error detection. If the environment changes unexpectedly, moving the agent to a different part of the plan (e.g. someone opening a door before the agent reaches it) the agent will recognize that the preconditions of a different operator (planned for later use) are now satisfied and this operator will be executed next. Similarly, the agent can move between plans built at different times and compiled into operator precondition knowledge. For example, an agent with two plans, one for driving through an intersection when the light is green and another for crossing when the light is red would effectively switch between the plans if the light, unexpectedly, changed color. The light changing color causes the preconditions of operators in one plan to become unachieved, but as preconditions from another plan are now achieved the agent can continue execution and cross the intersection. The calculation to select the next operator to execute, efficiently combines reactivity and plan monitoring, unlike the separate mechanisms traditionally used in systems with declarative plan representations.

6.2 Implementation

There are three processes required by IMPROV's error detection method:

1. Recognize have a path to the goal
2. Recognize agent no longer on path to the goal
3. Detect loops to earlier states

In this section, we will discuss how each method is achieved by the current implementation of IMPROV within the Soar architecture. Readers primarily interested in IMPROV's learning mechanisms can skip this section without hindering their ability to understand the later chapters.

6.2.1 Detect when on a path to the goal

After the construction of a plan, IMPROV learns a rule which indicates that the agent can reach the goal from the initial state. Once this rule fires it indicates that the agent is now following a plan and should reach the goal without further deliberation. This approach is used in all subplans within the complete sequence of steps that make up a full plan. Therefore, rules are learned for each state in the sequence of states that make up the plan. These additional rules are useful in future problem solving, where the agent may start from a different initial state.

An example of the sequence of rules that is learned is shown in Figure 6.6. These rules have the same conditions as the rules representing the preconditions of the operators in the plan. If the agent only learned operator precondition rules as the result of planning, matching the preconditions of an operator would be sufficient to indicate the start of a plan. However, the agent may have additional operator precondition knowledge, perhaps pre-programmed or learned from other sources (such as instruction).

6.2.2 Detect when have left the path to the goal

IMPROV detects that the agent is no longer making progress towards the goal when the agent's knowledge is insufficient to decide which operator to select next. In Soar, this takes the form of an *impasse*. The impasse is either a *tie impasse* because multiple operators are suggested or a *state no-change impasse* because no operators have been suggested. An impasse reflects the absence of directly applicable knowledge on how to proceed in solving the current problem.

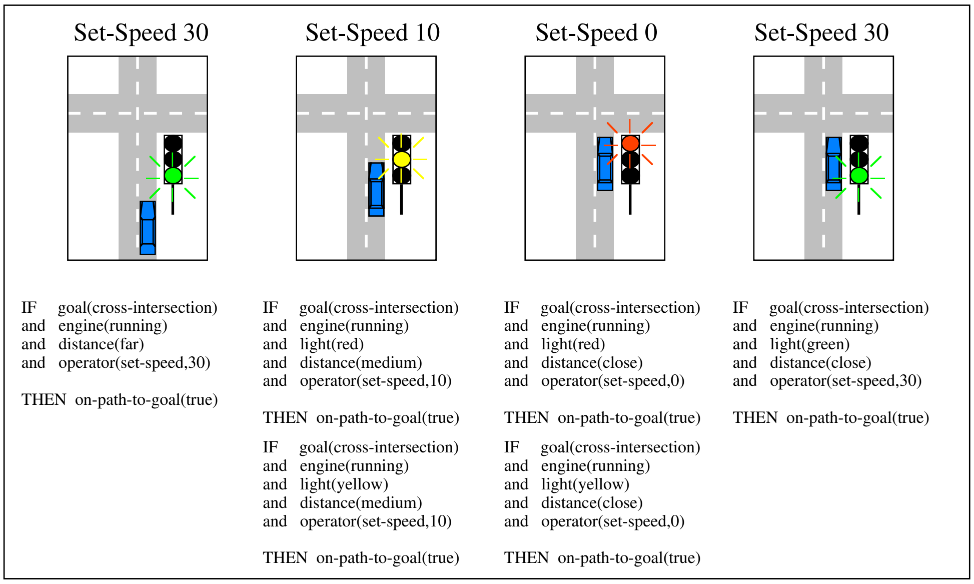


Figure 6.6: Rules to detect the presence of a plan to reach the goal

For example, in slowing to cross an intersection, the agent may be unaware that it should change gear to avoid stalling the car. IMPROV detects this as an execution failure when the agent reaches an impasse because no operator is suggested after the car stalls.

6.2.3 Detect loops to earlier states

To detect loops, IMPROV checks for the re-use of an operator precondition in the achievement of the current goal. IMPROV's restriction to purely procedural access to the agent's domain knowledge means it cannot directly identify the precondition rule that caused an operator to be selected. Instead, at each step during execution, IMPROV builds an explanation of why the next operator is being chosen. This explanation leads, through Soar's Explanation-Based Learning method, *chunking* [Laird *et al.*, 1986], to a new rule that matches the precondition of the operator being chosen, along with a test for the current goal instance. This process is shown in Figure 6.7.

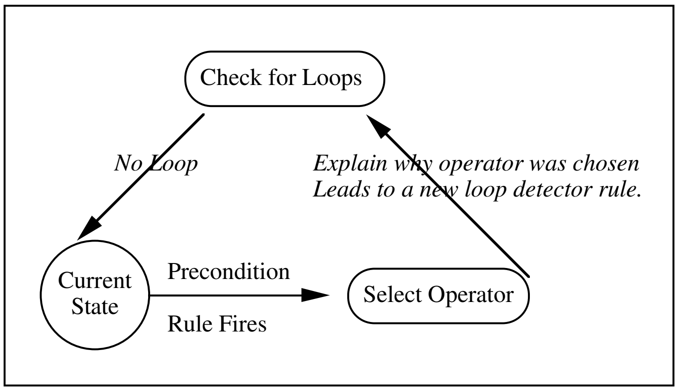


Figure 6.7: Learning rules to detect cycles

Whenever one of these loop detection rules fire, IMPROV detects a cycle in the agent's progress towards the current goal. Figure 6.8 shows an example of this process. As the plan is executed, IMPROV learns the series of loop detection rules shown in the figure. Notice that the goal is a specific instance of the general goal of crossing an intersection, ensuring that these rules will not match on a later attempt to cross an intersection having achieved this goal. When `Start-Engine` is chosen for a second time, the third rule fires, indicating that a loop has occurred.

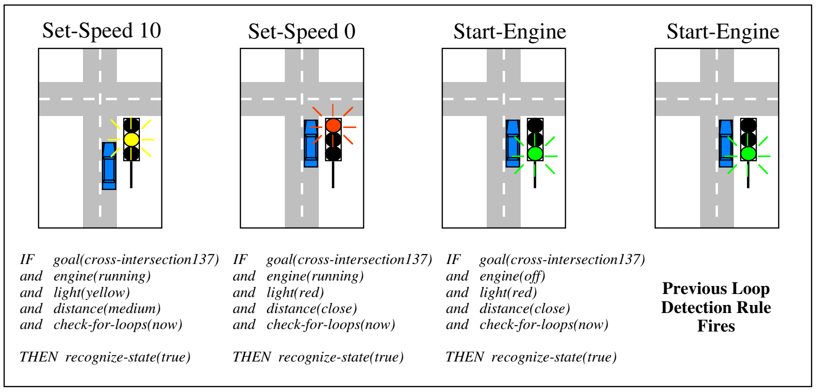


Figure 6.8: Loop signaled when an earlier loop detection rule fires

Soar's chunking mechanism maintains a trace of the rules that fire during problem solving and the working memory elements that are created as a result of the rules' firing. This trace allows it to construct an explanation of why a particular working memory element was created and compile the sequence of rules that lead to its creation into a single new rule. This access is still limited to the procedural cost of executing rules as the chunking mechanism does not search or otherwise analyze the rule base in building new rules. It simply examines the trace of rules that have already fired. This is important, as it ensures that the cost to build a new rule does not increase as the number of rules comprising the agent's domain knowledge grows. Chunking is Soar's only learning mechanism and is used in the creation of all new rules.

An alternative approach to IMPROV's current implementation of loop detection, would be to expand this trace to maintain a record of the rules that had fired and signal errors when an operator precondition rule fired twice in service of the same goal. This alternative solution results in a trade-off. The alternative approach would lead to a faster implementation, as generating explanations is more expensive than maintaining this form of expanded trace. The disadvantage of this approach is that it makes loop detection an architectural process. This is undesirable, firstly because it increases the number of basic processes within the architecture and secondly, because it prevents the agent from reasoning about and controlling its loop detection method. The agent may have additional domain specific knowledge that should override this general purpose detection method. For example, a taxi driver looking for fares may intentionally loop back to an earlier location. When the loop detection process is implemented as part of the agent's knowledge it is easier to modify or replace it than would be the case for an automatic, architectural mechanism.

6.3 Discussion

IMPROV's failure detection method is a weak, domain independent detection method. Errors are only detected when they interfere with the agent's ability to achieve its current goal. It does not detect errors in the agent's ability to make accurate predictions. IMPROV's error detection method is weaker than methods that rely on plan monitoring and explicit verification of each prediction made during planning. In deterministic environments with no exogenous events, explicit monitoring can lead to detecting errors closer to the part of the plan where they occurred, making credit assignment easier. In complex, nondeterministic environments with exogenous processes, detecting lack of progress may be the best that can be done. For example, if an action involves a random event (such as the roll of a die), IMPROV will not attempt to learn to predict the random event once it has learned plans for dealing with all of the possible outcomes of the event. IMPROV's method imposes a weaker constraint on the plan's correctness. As long as the agent's planning leads to sufficient knowledge to allow the agent to react to changes in the environment and eventually reach it's goal, the plan is deemed sufficiently correct. Explicit plan monitoring methods require that the plan precisely describe the changes that will occur in the environment, a harder constraint to meet in complex environments.

IMPROV combines plan monitoring and support for reactive execution into a single process. The matching of operator preconditions during execution provides reactivity and can be extended to detect failures by recognizing that the agent is following a previously learned plan. By combining these two processes into a single process, execution can be made more efficient. The overhead to support IMPROV's failure detection method beyond normal plan execution is limited to the loop detection process and the addition of rules indicating the start of a plan. As IMPROV only engages its learning and correction methods in response to a failure (rather than after every trial) there is no learning overhead during or after normal plan execution. This approach ensures that the speed of an IMPROV agent that can learn, is very close to that of a static, non-learning agent, during normal plan execution.

IMPROV's method for detecting failures means that the agent does not require immediate feedback. If feedback is not available, due to delayed or incomplete sensing, the agent will continue to execute the plan on the basis of internal state changes. Once feedback becomes available again, the agent will either be in a situation it recognizes (i.e. has knowledge of how to proceed) or in an unfamiliar state (leading to an impasse and an error).

The loop detection method may incorrectly detect loops on iterative tasks when state changes are imperceptible and are not represented explicitly in the agent's plan. As we discussed earlier, each step in an iterative task must produce a change to the agent's internal state, for the states to be distinguishable as not forming a loop. Additionally, the heuristic of detecting repeated operator preconditions will not detect loops when the state representations are equivalent but represented differently.

This weak, domain independent error detection method can easily be augmented with additional domain specific knowledge. In evaluating IMPROV we compare the weak approach to one augmented by instruction (an external agent signaling errors [Pearson and Huffman, 1995]) and by an additional theory of failure states (knowledge of states that will lead inevitably to failure). The results of these experiments are detailed in Chapter 10.

CHAPTER 7

Recalling Previously Detected Failures

Irreversible environmental processes (constraint E5) prevent an agent from always determining the cause of a failure during a single problem solving episode. For example, in playing a card game, the agent cannot return to the same situation and try a different card in an effort to avoid a failure. Instead, the learner must wait until another similar situation occurs at a later time and then it can try an alternative action. This requires the learner to consider episodes that are temporally disjoint and interspersed with other, completely unrelated problems.

The agent must take some approach to this class of problems, where the opportunity for learning is spread over a range of temporally disjoint training instances. Most existing systems either record all training instances (making them non-incremental) or try to learn from each instance independently (losing potentially valuable information).

Reinforcement learners treat each training instance as an independent opportunity for learning. The agent makes an inductive guess at the cause of a failure leading to permanent changes in the agent's knowledge. This induction is based on the existing state of the agent's knowledge and the current instance. This limits the ability of these learners to consider information that is available on the basis of comparing multiple instances, (as we'll show later in Section 9.1.2).

Deliberate theory revision systems either assume that environmental processes are reversible or immediately repeatable (e.g. EXPO) or store all training instances and update a single, global domain theory (e.g. EITHER, FOIL). Systems such as EXPO assume that the agent repeats the task until a cause for the failure is identified, before proceeding to a new task. Systems such as EITHER record all training instances. This gives the learner access to earlier, relevant problem solving episodes but makes learning non-incremental.

As IMPROV is designed for environments that may be time-critical (E6 and E7) we require learning to be incremental and not based on all previous instances. Also, we want IMPROV to reason deliberately, analyzing multiple instances at once during training to improve the quality of its learning. This leads IMPROV to recall previously detected failures and attempted solutions when it returns to a similar context. These previous failed attempts are only collected until IMPROV discovers a successful plan that avoids the failure, bounding the number of instances and ensuring that learning remains incremental. This approach allows IMPROV access to training instances recorded as a result of problem solving that occurred at different times. It also allows IMPROV to work on correcting multiple, unrelated errors in its knowledge at the same time, as information related to each error is only recalled when the agent returns to that context. For example, failures related to playing cards and failures related to driving are only recalled when the agent is working on the relevant type of problem.

7.1 Functional Description

IMPROV's approach is to recall previously detected failures when the agent returns to a similar situation. When an failure is detected, IMPROV records the error, indexed by the current goal (i.e. super operator). Whenever the agent is working on an instance of this goal, during later problem solving, the failure is recalled. The record of this error is also used to record previous attempts at solving the problem and avoiding the failure.

The error information is discarded after IMPROV has attempted to correct the domain knowledge associated with achieving this goal. To return to the earlier example of driving through an intersection. On the first try, the agent crosses when the light is red, leading to an error. IMPROV records that the super operator (`cross-intersection`) lead to an error. Later, possibly much later, the agent comes to another intersection. The goal (super operator) is the same so the error is recalled (leading to more cautious behavior, starting with replanning). This time, the agent successfully crosses when the light is green. IMPROV induces the cause of the failure as the color of the light, updates the planning knowledge of how to cross an intersection (to include the correct light color) and removes the error record. If the induction is incorrect, the agent will fail again in a future situation, but that will be treated as a new, independent error instance.

To summarize, the functional properties are:

1. Record errors, indexed by goal
2. Recall errors, when goal matches
3. Remove errors, after attempting a correction

This approach allows IMPROV to work on correcting multiple errors at the same time. The agent may have made several errors for different goals and have not yet determined their cause. The errors will be recalled in their correct context, based on the current choice of super operator. An agent that is having trouble getting the car started and is also having trouble crossing intersections, will recall each as distinct, independent errors.

The error is indexed by the super operator, rather than being indexed by the last operator to be executed. This is because the error may have occurred earlier in the plan to achieve the super operator. Recalling the error before starting to execute the plan, gives IMPROV a better chance to avoid repeating the failure. For example, in crossing an intersection, the point to take corrective action is at the start of the intersection, not when the brake is pressed in an emergency attempt to avoid a collision.

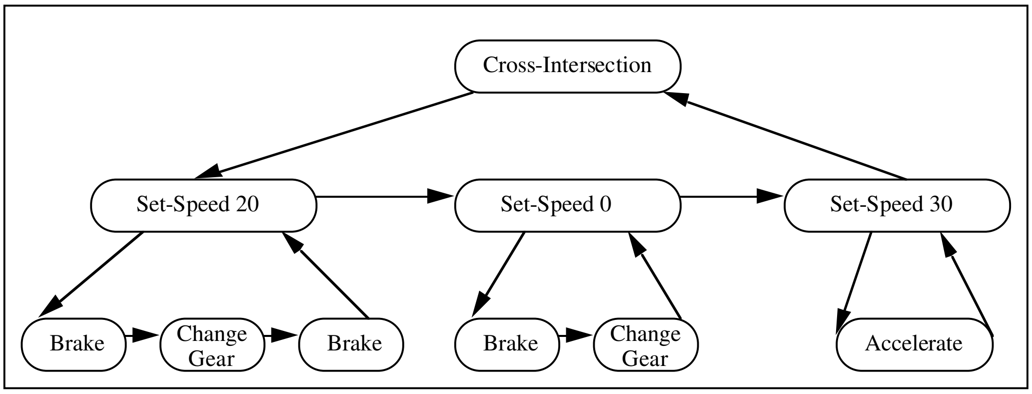


Figure 7.1: Operator hierarchy for crossing an intersection

7.2 Implementation

IMPROV records the information concerning an error as rules, rather than in the agent's working memory. This ensures that as IMPROV simultaneously works on multiple, unrelated errors, the size of the agent's working memory will not grow. In Soar, it is much more efficient to store information in long term memory, as rules, and only recall the relevant information to working memory when the agent returns to the context of the failure.

This approach means that the main issue in the implementation is how to discard the record of the error, and the previous attempts to correct it, once IMPROV has learned new (and hopefully correct) planning knowledge. This is an issue because IMPROV is only assumed to have procedural access to the agent's knowledge and therefore cannot search for the rule that records the error information and remove it.

Instead, IMPROV associates an extra symbol with each operator and tests for the presence of that symbol in the rules relating to an error. When IMPROV corrects the agent's knowledge, it changes the symbol associated with the operator, preventing the error information from being recalled in future. This process is discussed in detail in Section 9.3.4 as this same symbol, or *version number*, is needed to let IMPROV correct errors in the agent's knowledge. At this point, it is sufficient to understand that this symbol changes to a new value when IMPROV changes the agent's knowledge in an effort to correct an error in that knowledge.

To return to the functional requirements of IMPROV's error recall method:

1. Record errors, indexed by super operator

   Once a failure has been detected, IMPROV learns a new rule that tests for the super operator and the version number symbol. The rule creates an identifier for this particular error instance. In the driving example, this would be the rule shown in Figure 7.2. This rule will fire if the agent chooses the same super operator in a future situation, providing the necessary recall of the error.

   ```
   IF        operator(set-speed,0,version 1)

   THEN  add    error(error127)
   ```

   Figure 7.2: Indexing errors by operators

   Additional information about the error is also recorded, such as the state immediately prior to the failure being detected. This information is used during error correction and is described in Chapter 9.

2. Recall errors, when super operator matches

   When a similar situation arises and the agent selects the same super operator (i.e. is working on the same goal), the rule described above fires, signaling to the agent that an error has previously occurred in implementing this operator. This knowledge leads IMPROV to move into a more cautious, error correcting mode and consider alternative plans that may successfully achieve the current goal.

3. Remove errors, after attempting a correction

As IMPROV learns new knowledge, in an effort to correct existing errors, it will construct new versions of the failed operators. That is, it will build new precondition rules that suggest an operator with a higher version number (see Section 9.3.4). In future problem solving, if the operator's preconditions match it will be this operator with a higher version number that is suggested. For example, after learning, `set-speed(0) version 2` will be chosen in future driving situations.

As the rule for recalling the error (see Figure 7.2) tests the original version number (`version 1`), it will no longer match when a correction has been made and an operator with a higher version number is suggested. The process of correcting the agent's knowledge automatically prevents recall of the errors the correction is intended to fix.

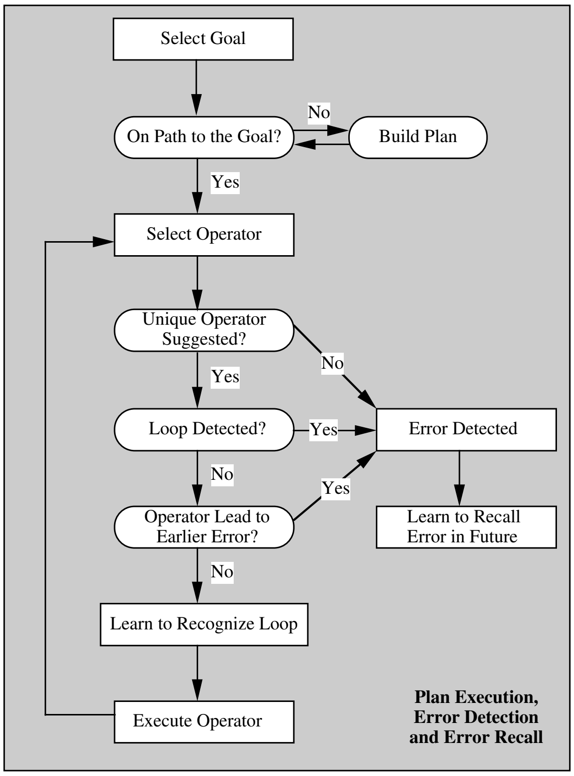


Figure 7.3: Plan execution, error detection and error recall

This complete error detection and recall process is shown in Figure 7.3, including the steps where the error record is learned and later recalled.

CHAPTER 8

Inductive Learning

The last three chapters described the first two stages in the error correction process that we outlined earlier, namely:

1. Classification of errors
2. Detecting performance failures
3. Solving the current problem
4. Learning a general correction for the future
   (a) Credit assignment–Which operators are incorrect?
   (b) Credit assignment–How the operators are incorrect?
   (c) Changing the domain knowledge

Before explaining IMPROV's error correction and learning process, this chapter describes the induction method that underlies the later learning and credit assignment stages. The task of learning correct operator precondition knowledge can be seen as the task of learning a category. The category being learned is which operator is the correct one to use for the current state and goal. In this chapter we will discuss environmental properties that constrain the inductive learner and then present a learning method, SCA2, that is an extension of the symbolic category learner, SCA [Miller, 1991; Miller, 1993]. Readers who are not interested in the details of the inductive process can proceed to the next chapter.

The environmental properties we first considered in Chapter 2 constrain the design of the inductive learner. First, it must be incremental as there may be arbitrary numbers of training instances (E7) and there are temporal constraints on the agent's performance (E6). That is, the time to learn about a new instance should not increase as the total number of training instances seen increases. This is a constraint because the learner is embedded within the performance system. Time spent learning will slow the overall performance of the agent in the environment. Second, the learner must be tolerant of imperfect sensing and noise (E8) and be able to adjust to changes in environmental processes over time (E10). Third, its representation must be expressive enough to describe the large sets of disjunctive conditions required by complex tasks (E4). Finally, the learner should take advantage of additional knowledge sources or feedback when they are available (E11).

8.1 SCA2 – Functional Description

IMPROV uses SCA2 (an extension of SCA[Miller, 1993]) for inductive learning. SCA is an incremental, symbolic category learner that represents classification knowledge as production rules. SCA is a computational model that accurately models many human behaviors exhibited during category learning. For example, response times decrease as a result of learning, rather than increasing as is typical in most machine learning systems. SCA uses purely symbolic reasoning and learns categories by the acquisition of increasingly specialized prediction rules.

SCA does not support structured or relational objects, or the learning of negative classifications (e.g. X is *not* a ball). SCA2 uses the same basic method for induction as SCA, but extends learning to include structured objects and negative categories (which are important in learning knowledge about how to perform tasks). Further, SCA assumes that the knowledge available to guide inductions during training will still be present during performance. SCA2 removes this restriction, allowing the agent to use knowledge to guide training (e.g. from analyzing multiple attempts to correct an error) and then discard this extra knowledge (e.g. forget the previous attempts). Thus, performance is made independent of the knowledge used to guide induction. This allows SCA2 to use a wider range of additional knowledge to guide its inductive learning.

Learning in SCA2 is supervised, which means that training instances consist of a vector of state features along with a category label. The task is to correctly recall the category label when later test instances are presented. SCA2 incrementally learns prediction rules as it is trained. Initially, these rules are very general, testing only a few of the features from the training instances. As learning progresses, more specific rules are acquired that test more features.

8.1.1 Predictions

When making a prediction, SCA2 searches for the most specific prediction rule that matches the test instance. An example of a set of prediction rules is:

```
[speed 30]   -> predict accelerate operator
[speed 50]   -> predict maintain-speed operator
[speed 50, weather rain] -> predict brake operator
[speed 50, road freeway] -> predict accelerate operator
```

This set consists of two general rules (indicating that the agent should accelerate when traveling at 30mph and maintain speed at 50mph) and two more specific exceptions to the 50mph rule (to cover rain and freeway driving).

Given the instance:

```
[speed 50, car-color yellow, road freeway, behind nobody]
```

SCA2 would search for the most specific rule matching a subset of the features. In this case finding:

```
[speed 50, road freeway] -> predict accelerate operator
```

and so predicting the choice of the `accelerate` operator.

An interesting situation arises when the agent is on a freeway and it is raining. In this case SCA2 picks at random among the matching rules that are equally specific, leading it to either accelerate or brake.

8.1.2 Training

Further training leads SCA2 to learn more specific rules, resolving the conflict. Training occurs by first searching for the most specific rule that matches the training instance and predicts the same category. Then a new rule is built that includes all of the features from the rule that was matched and one new feature from the training instance.

For example, when presented with this training instance:

```
[speed 50, road freeway, weather rain, behind truck, car-color green; brake]
```

SCA2 would search for the most specific rule that matched the instance and predicted the same operator:

```
[speed 50, weather rain] -> predict brake operator
```

and then learn a new rule with an additional feature, such as:

```
[speed 50, weather rain, road freeway] -> predict brake operator
```

The key to the success of the induction is the choice of which feature to include. Features that are important in predicting the category should be included first. In this example, the color of the car being driven is probably irrelevant, while the weather conditions and the type of road are more important. If the concept to be learned is that you should brake when driving in the rain on a freeway, then learning the rule:

```
[speed 50, weather rain, car-color green] -> predict brake operator
```

will lead to more incorrect categorizations than the rule:

```
[speed 50, weather rain, road freeway] -> predict brake operator
```

An important capability of SCA2 is it's ability to use a wide range of different knowledge sources to guide this inductive step. Miller [1993] showed that when ID3's [Quinlan, 1986] information gain metric was used to select the feature, prediction accuracy was comparable to that of ID3. However, this metric is only one of a number of methods that can be used to select the feature to focus on during learning. In particular, when multiple instances are available for training, their differences can be used to guide the induction. Also, additional feedback (for example, from an instructor) can guide the choice of feature to focus on.

This inductive method leads to the following functional properties:

1. Learns incrementally

   The time to train on an instance is independent of the total number of training instances presented to the learner. Each training instance is used to construct a new rule and then the instance is discarded.

2. Performance does not slow with learning

   As SCA2 learns more prediction rules, the time taken in making a prediction either remains constant or declines. This is because SCA2 searches its rule base from specific to general rules. As new rules are added they are found more quickly during this search, ensuring that performance does not degrade as the SCA2 learns.

3. Expressive concept representation

   SCA2 can represent complex disjunctive and conjunctive categories. Each prediction rule represents a section of the space of instances. The more specific the rule, the smaller the section of the instance space represented. By combining multiple prediction rules, SCA2 can represent arbitrary sections of the instance space and represent complex disjunctive categories. In the limit, SCA2 can learn specific prediction rules for individual exemplars, allowing it to successfully represent any category that can be described in the representation language.

4. Additional knowledge can guide learning

   The inductive step is reduced to the problem of deciding which feature to include in new prediction rules. As SCA2 is fully encoded within a general problem solving architecture, the agent can use a wide range of knowledge based methods to select the feature. For example, learning could be guided by additional knowledge (such as causal theories) or extra feedback (such as an instructor). The benefits of these additional sources of knowledge are evaluated in our experiments in Section 10.5.

5. Noise tolerant

   Noisy instances can lead to incorrect prediction rules being learned. These incorrect rules can be overridden by learning more specific rules that mask the incorrect general rules. As the more specific rules are recalled first, this provides a method to override incorrect learning. This capability ensures SCA2 has a degree of tolerance for noise.

   Having learned an incorrect rule, as the result of noise, SCA2 can either learn a series of new, more specific rules each predicting the correct behavior or it can learn a single rule, indicating that the wrong prediction should not be made. For example, let's assume SCA2 has incorrectly learned that when driving behind a truck, you should accelerate. To recover from this, one approach is to learn a series of more specific rules, each suggesting the correct behavior. For example, when driving behind a truck in the rain, brake; when driving behind a truck on the freeway, brake etc. This can require SCA2 to learn a large number of rules to cover each of the cases. The alternative is to learn a rule indicating that the category should not be chosen. For example, when driving behind a truck, do not choose accelerate. Both approaches provide a degree of noise tolerance during learning.

   Both approaches also allow SCA2 to recover from noise in the features of the training instance (rather than noise in the category) using the same mechanisms. The noise tolerance is evaluated in Section 10.3.3.

8.2 SCA2 – Implementation

SCA2 is completely implemented within the Soar architecture. This is not an architectural extension, but is instead coded as production rules. The new inductive rules are learned through Soar's chunking mechanism for compiling general problem solving.

8.2.1 Knowledge Representation

Training instances are represented as a series of state and goal features along with a category label. In order to allow SCA2 to represent structured objects and relations, the features are represented by tuples containing an object identifier, an attribute and a value. For example, `(Obj1,color,red)` specifies that object1 is red. The values can themselves be identifiers, which allows relations to be represented. For example, `(Obj2,behind,Obj1)` indicates that object2 is behind object1. This representation is therefore more expressive than a simple, attribute-value representation.

Prediction rules test a set of features, with identifiers replaced by variables, and a counter indicating the total number of features in the set. Figure 8.1 shows an example. The predictions

```
IF   task(sca-prediction)
and  feature(goal,<c1>,drive-50)
and  feature(speed,50,<c1>)
and  feature(road,freeway,<r1>)
and  feature(behind,<c1>,truck)
and  feature(weather,rain)
and  count(5)

THEN  choose-op(predict,brake,acceptable,prediction318)
```

Figure 8.1: Example of a prediction rule in SCA2

consist of the name of an operator (e.g. `brake`), a *preference* (e.g. `acceptable`) and a unique constant (e.g. `prediction318`). The preference is either `acceptable` or `rejection` and indicates whether the operator should be chosen or avoided. The inclusion of this preference information allows SCA2 to learn *not* to select a particular operator in a given situation, for example, not to touch a hot stove. The unique constant is used to uniquely discriminate this prediction from other predictions (as multiple prediction rule may match at once) and is only used in training.

8.2.2 Predictions

SCA2's performance task is to correctly predict a category from a set of features. In IMPROV, the task is to predict an operator from a set of state and goal features. IMPROV can use a simple attention mechanism to select a subset of the entire state, if the state involves a very large number of features. This attention mechanism is domain specific and is assumed to include all of the features that are relevant to learning. This mechanism is not necessary, but does improve efficiency.

The implementation of SCA2's prediction method is shown in Figure 8.2. As more specific rules are more likely to be correct, SCA2 attempts to find the most specific rule that matches the input instance. This is achieved by counting the number of features in the input instance and then steadily decrementing this count. As the prediction rules include a test for the number of features in the rule, the effect is to recall specific rules first[1]. Thus, SCA2's search proceeds from specific to general. For example, the rule shown in Figure 8.1 will only match if the training instance includes all of the features in the rule and the counter has reached 5. A more specific rule, with a higher count, would be matched earlier. The main cycle is to steadily decrement the counter looking for a rule that matches the input instance. The first prediction found is the most specific rule that matches a subset of the features in the input instance.

If the prediction is for an acceptable preference (an operator that should be selected) then SCA2 attempts to build an instantiation for this operator. The instantiation consists of filling in the arguments for the operator (e.g. in grasping a block, filling in the name of the block to be grasped). This instantiation knowledge is domain specific and is required to be unique for a given input instance. No valid instantiation may exist (e.g. selecting grasp when there are no blocks

[1] An alternative would be to directly modify Soar's match algorithm to find the most specific rule. This would be more computationally efficient, but it is not clear that finding the most specific rule is appropriate in solving other problems unrelated to induction.

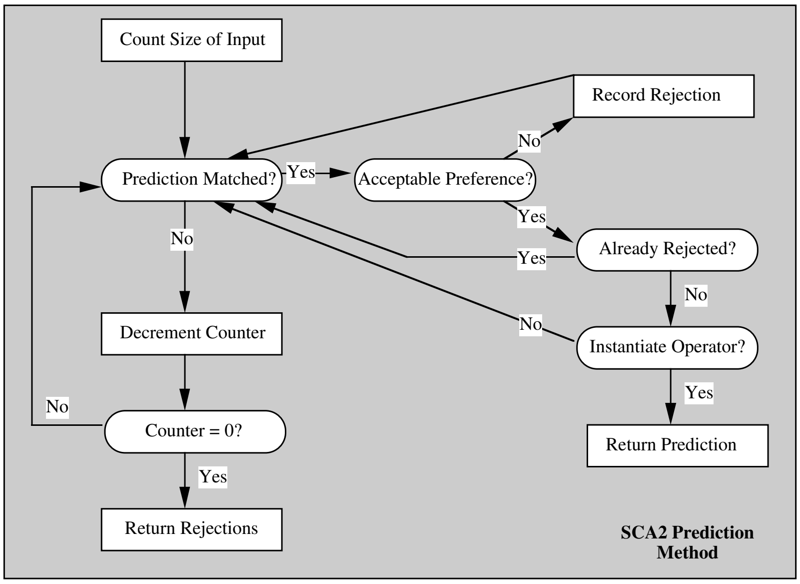


Figure 8.2: Prediction method for SCA2

present) in which case SCA2 returns to searching for another operator. If a valid instantiation exists, this is returned as SCA2's prediction.

If the prediction rule matched is for a rejection preference (the operator should not be selected) then SCA2 records the operator in a list of rejected operators and continues the search for a prediction. If a later prediction is for an operator that has already been rejected, the operator is ignored and the search continues. In the end (when the counter reaches 0), if only rejected operators have been found, they are returned. The effect of rejected operators is to effectively mask later predictions. As the rules are searched from specific to general, only a more specific rejection rule can mask an acceptable prediction rule.

The rejection mechanism can be adapted to generate successive predictions after an initial best match. For example, let's assume SCA2 makes a prediction having matched a rule with 7 features proposing operator `A`. To generate a second best prediction, SCA2 marks operator `A` as rejected from the start of its search. This will lead to another operator being predicted, for instance, operator `B` based on matching 5 features. This process can be then be repeated by adding `B` to the list of initially rejected operators and re-running SCA2's search. This ability turns out to be useful in allowing IMPROV to generate alternative operators after an operator is seen to fail.

SCA2 also associates an *uncertainty level* with each recalled operator. This value is used in searching for a plan using the UBID method from Section 4.3.1. The uncertainty level for a prediction is:

uncertainty = 10 - number of features matched, min 1.

with a bonus of 10 if the prediction was rejected, leading to a range of 1-10 for acceptable operators and 11-20 for rejected operators.

8.2.3 Training

SCA2 learns new rules by searching for an existing prediction rule that predicts the training category and then adding a new feature to specialize that rule. When combined with the search from specific to general rules during prediction, this leads SCA2 to incrementally improve the accuracy of its predictions.

SCA2's training method is summarized in Figure 8.3. The initial stage is a specific to general search for a prediction that matches the category in the training instance. For example, in training on the instance:

```
[speed 50, road freeway, weather rain, behind truck, car small; predict brake]
```

SCA2 recalls the most specific rule matching a subset of the features in the instance and sharing the same operator. The name of the operator must match, but the preference (rejection or acceptable) can be different. For instance, the matched rule might be:

```
[speed 50, weather rain] -> predict brake operator
```

This search is guaranteed to terminate as SCA2 always starts with an initial set of maximally general rules, testing no features, that predict each possible operator. Once a match has been found, SCA2 selects a feature to add to this rule. This choice is cast as selecting among `select-feature` operators (one for each attribute-value pair in the training instance). This leads to an operator tie impasse, causing Soar to generate the subgoal of selecting the correct operator (and feature). This allows the agent to employ its full, deliberate problem-solving mechanisms in selecting the feature. The specific knowledge used in making this selection is dependent on the particular SCA2 implementation. For example, in IMPROV the choice is based on differences found among sets of instances (see Section 9.1.3).

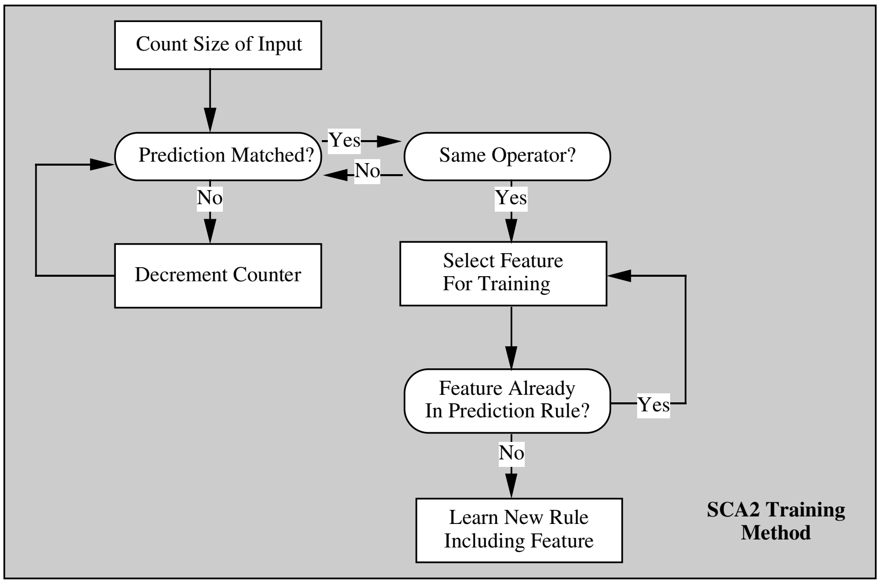


Figure 8.3: Training method for SCA2

Once a feature has been selected, SCA2 checks to see if the feature is already included in the rule that was matched[2]. For example, having matched the rule:

```
[speed 50, weather rain] -> predict brake operator
```

the selected feature for training should not be either of `speed 50` or `weather rain`. SCA2 does not have direct access to this rule, as it is limited to just procedural access to the rule base.

[2]Thanks to Seth Rogers for suggesting this optimization, previous versions of SCA2 computed the set of matched features first which is usually less efficient.

SCA2 tests whether the rule includes this feature by temporarily removing the feature from the training instance and examining whether the same prediction rule fires. If the rule still matches, the removed feature was not tested in the prediction rule and can be usefully added. If the rule no longer matches a different feature should be selected. The unique constant associated with the prediction operator (see Figure 8.1) is used to keep track of whether the prediction rule is still matching or not. As each feature is removed, SCA2 checks to see if this constant is still present in memory. This is necessary as multiple prediction rules might make the same prediction at once, so simply checking for the prediction is insufficient.

The final stage is to learn a new prediction rule (predicting the training category) that includes the new feature as an extra condition. This is achieved by matching the prediction rule in one of Soar's subgoals, testing the new feature and creating a new prediction operator as a result. Soar's chunking mechanism complies this processing into a new prediction rule (see Appendix A.2 for further details of this processing).

8.3 Discussion

SCA2 is one of several induction algorithms that could be included within IMPROV. IMPROV's target environments (Figure 2.1) constrain the induction algorithm's design in general ways. For example, the induction must be incremental (or on-line) rather than non-incremental (off-line) or IMPROV's performance will degrade over time as it learns new knowledge, violating E6. However, there is considerable flexibility within these constraints. SCA2 is particularly attractive as it can be fully implemented through production rules in the Soar architecture. This uniformity of representation and processing facilitates the inclusion of additional knowledge in guiding the induction, in the same manner that the rest of IMPROV can be improved through additional knowledge. For example, IMPROV (and SCA2) has been successfully integrated with Instructo-Soar [Huffman and Laird, 1994; Huffman, 1994], a system for learning from instructions [Pearson and Huffman, 1995]. This integration was greatly simplified as IMPROV, SCA2 and Instructo-Soar shared the same representation and processing model. In Section 10.5 we evaluate the benefits of this additional knowledge in improving learning.

SCA2's learning is limited to symbolic concept descriptions. The concepts can include relations (e.g. `on(<block1>,<block2>)`) but SCA2 will not discover or create new relations (e.g. given the coordinates of two blocks it will not construct the relation `on`). Additionally, it will not automatically create new numerical relationships between values. For example, to learn that you should maintain constant speed on reaching the speed limit, SCA2 would either have to start with the concept `at-speed-limit` or learn a series of pairs:

```
[speed 20, limit 20] -> predict maintain-speed
[speed 30, limit 30] -> predict maintain-speed
[speed 40, limit 40] -> predict maintain-speed
```

An interesting avenue for future research would be to extend SCA2 to include the capability to build new relations, potentially leading to faster and more compact learning. SCA2's initial, programmed, knowledge can include these relations, for example:

```
[speed <s>, limit <s>] -> predict maintain-speed
```

This allows SCA2's initial knowledge to include complex, intentional functions. If this knowledge was incorrect, SCA2 would then learn a symbolic correction, for explicit values (as shown above).

As SCA2 learns new rules, execution time in making a prediction should, on average, remain constant or decrease. This follows as the search progresses from specific rules to general rules. As the new rules will be more specific than existing rules, they will be matched more rapidly in future cases. This speed up is offset by an increase in match cost as rules are added, however empirical evidence indicates that for the latest version of Soar's matcher, the match time increases either very slowly or not at all as more rules are added [Doorenbos, 1993]. In our experience with IMPROV, no slow down is apparent as new rules are learned, but we have not explored learning especially large numbers of rules.

CHAPTER 9

Correcting Operator Preconditions

Having detected an error, the agent is faced with two, related problems:

1. Solving the current problem

   This is the problem of identifying the correct sequence of operators to use when faced with this goal. The agent must search the space of possible operator sequences to locate one that succeeds. For example, an agent may fail in its attempt to start a manual transmission car. It discovers that pressing the clutch and putting the car into neutral allows it to start the engine.

2. Learning a general correction for the future

   This is the problem of learning the important features of the task that lead to success or failure, allowing the agent to solve similar problems in the future. For example in starting a car, the agent may generalize from its attempts to conclude that the car can only be started in neutral.

In many systems, these two problems are folded together, with the agent learning from each success or failure and then replanning, leading it to select a new sequence of operators when faced with the same goal in the future. The learning leads the agent to search the space of possible plans. In this chapter we will argue that agent-based learning can be improved by keeping these two stages separate. The agent should first search for a successful plan and only then attempt to generalize the reasons for earlier plans succeeding or failing. As we described earlier, the generalization problem can be further broken down into 3 stages:

1. Credit assignment–Which operators are incorrect?

   Determining which operator, or operators, have the incorrect knowledge that led to the performance failure. If feedback and sensing are incomplete or delayed, the agent will not necessarily know what operator caused the failure. This leads to the problem of assigning credit for the failure to a step, or steps, in the plan.

2. Credit assignment–How the operators are incorrect?

   Having identified an operator that contributed to a performance failure, the agent must determine the part of the operator's knowledge that is incorrect. In this chapter, we will focus on errors in precondition knowledge, so this task becomes deciding which preconditions should be added or removed. The agent must decide what properties of the state or goal were important to the success or failure of this operator.

3. Changing the domain knowledge

   Having identified the cause of the error, the agent modifies its knowledge to avoid this error in future.

This chapter describes IMPROV's approach to these three issues for the task of correcting errors in operator preconditions. As we have discussed in earlier chapters (Section 2.2 and Figure 2.1), this operator precondition learning is constrained by properties of the environment. IMPROV learns planning knowledge rather than direct execution knowledge so that the agent can reason about tasks with large, dynamic state and goal spaces (E1 and E2). The corrected planning knowledge is then used to correct execution knowledge. IMPROV's precondition learning must be sufficiently expressive to learn large numbers of disjunctive preconditions in a form that can be efficiently accessed (E4). Learning may occur over multiple training episodes interleaved with long periods of other, unrelated problem solving (E5). The agent's performance should not become slower over time as the agent learns more information (E6 and E7). Learning must be tolerant of delayed, incomplete or noisy sensing (E8); changes to the environment not due to the agent's actions (E9); and environmental processes that evolve and change over time (E10). Finally, we would like the learner to take advantage of multiple forms of feedback and sources of knowledge, using whatever information is available in solving a given problem (E11). Each part of IMPROV's design, from knowledge representation, through error detection and now error correction is constrained by consideration of these environmental properties. In this chapter we will present one approach to meeting these criteria and in the next chapter (Chapter 10), we will evaluate the performance of the precondition learning. This is followed by Chapter 11 which explains how the precondition correction method can be extended to correct operator effects.

IMPROV's error recovery method involves two searches. First, a search of the space of plans, executing them in turn to discover a successful plan. Second, IMPROV searches the space of operator preconditions, using an inductive learning method to converge on the correct operator knowledge. Each search defaults to a weak method, with the knowledge learned through induction guiding the search for plans and the search for a plan then guiding future inductions. As we will demonstrate later, each search can be made stronger by the addition of domain specific knowledge to guide the search.

9.1 Functional Description

9.1.1 Solving the current problem

IMPROV's basic method for correcting operator preconditions is to generate new plans, execute the plans to test their correctness and then analyze the set of plans and their results to identify errors in the agent's operator knowledge (see Figure 9.1). This search can occur over a long time-span, spread across multiple problem solving episodes. Each time IMPROV detects a failure, or recalls an earlier failure in a similar situation, the agent generates a new plan and executes it, in an effort to avoid the failure. As this search may be spread across multiple problem solving episodes, all information and records about the search are stored as rules, rather than as data structures in working memory. This approach ensures that working memory remains bounded in size, which is important for efficiency because Soar's match cost is proportional to the size of working memory but can be independent of the number of rules. The rules that record information about this search are only recalled when the agent returns to a similar context. A measure of "similarity" is required because IMPROV cannot assume that the agent will return to an identical state. For example, in playing a card game, the identical hands may never been repeated. We discuss the merits of a number of different measures of similarity in describing the implementation (Section 9.3.1).

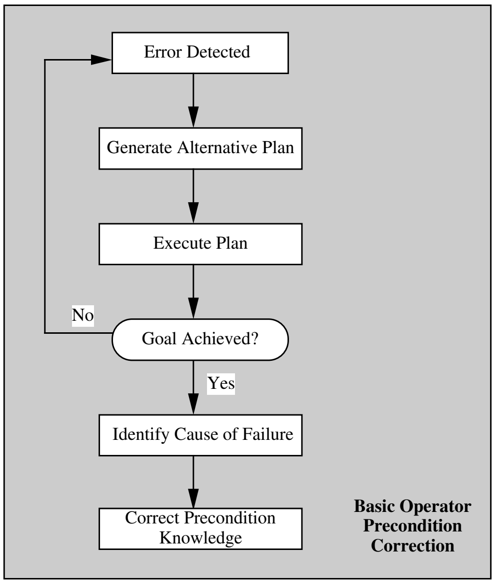


Figure 9.1: Basic operator precondition correction method

IMPROV searches for alternative plans in decreasing order of certainty as measured by UBID. IMPROV records the last operator that was executed before the failure and then uses UBID to generate a new plan, trying an alternative operator for the that final step. If the new plan fails a different alternative is tried. Once every option has been explored for a given step in the plan, alternatives are considered for the preceding step. This search is biased to explore plans with lower uncertainty first (as described in Section 4.3.1 on UBID). Plans that are similar to the original plan will have the lowest uncertainty, leading to a search through plans that are increasingly different from the initial, incorrect plan.

For example, Figure 9.2 shows an incorrect plan to drive across an intersection. The plan fails during execution as the agent fails to change gear and the engine stalls when the car comes to a halt. IMPROV generates alternative plans to cross the intersection. These plans include approaching the

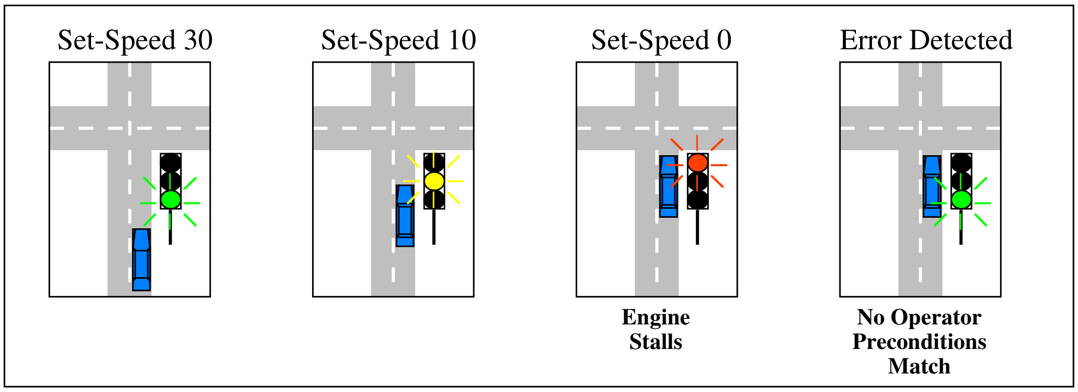


Figure 9.2: Initial incorrect plan to cross an intersection

intersection at different speeds or changing to different gears. As IMPROV is searching for an error in the preconditions of the operators that make up this plan, IMPROV temporarily assumes that the planning knowledge about the operators' effects is correct. IMPROV internally simulates the

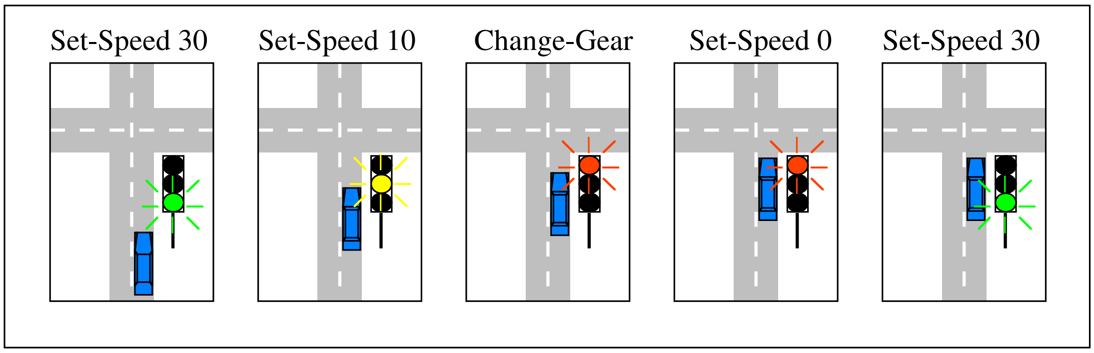


Figure 9.3: Alternative, correct plan to cross an intersection

operator sequence generated by UBID using the agent's current planning knowledge to determine the expected outcome of the plan. This allows IMPROV to reject plans that the agent's planning knowledge indicates would lead to failure. For example, turning onto a different street leads to a failure state. This allows IMPROV to only execute plausible plans rather than executing arbitrary operator sequences in search of a correct plan. Each plan that reaches the goal in the internal simulation is executed in the external environment, until the agent finds a successful plan. In the driving example, this search leads IMPROV to discover the successful plan shown in Figure 9.3. If a correct plan cannot be found at this level of operators, IMPROV proceeds to try to find a correction within the actions of the operators. This is done by applying the same correction process to the more primitive operators that implement the operators in the original plan (see Chapter 11).

This approach allows IMPROV to recover from either overgeneral or overspecific operator preconditions. Overgeneral operator preconditions lead to an operator being incorrectly included in a plan, while overspecific preconditions lead to the operator being incorrectly excluded from a plan. Recovering from overspecific preconditions is potentially more difficult as a standard planner would not generate a plan that required an operator whose preconditions were overspecific. As the plan is not created, there is no opportunity for the agent to execute it and realize the plan unexpectedly succeeds. For example, the agent learns to drive at the weekends and concludes that cars can only be driven on Saturday and Sunday. To discover this precondition is overspecific, the agent must attempt to execute a plan to drive the car at the weekends, even though this violates the agent's precondition knowledge for driving. IMPROV addresses this problem as UBID will generate all possible operator sequences including operators that the agent believes should not be chosen. These *rejected* operators are given a higher uncertainty, but they are still generated (see Section 4.3.2). Sequences that more closely agree with the agent's operator precondition knowledge will receive lower uncertainty and therefore will be generated earlier, but eventually all sequences (up to a certain maximum uncertainty level) will be generated.

Each plan is generated by UBID, simulated using the agent's planning knowledge and then executed in the external environment. After each plan is executed, IMPROV records the operators that were applied, the states before they were applied and the outcome of the plan. These records will later be used as training instances for the inductive learner. If the plan fails, IMPROV also builds a record to show that it attempted this plan (or more specifically the final operator in the plan). This record is used to indicate which plans have already been tried, causing IMPROV to generate a new alternative. The details of these records and the search are described in the section on the implementation (Section 9.3.1).

The search for an alternative plan is summarized in Figure 9.4. Notice that all learning is delayed until a successful plan has been found. Only then are the training instances analyzed and

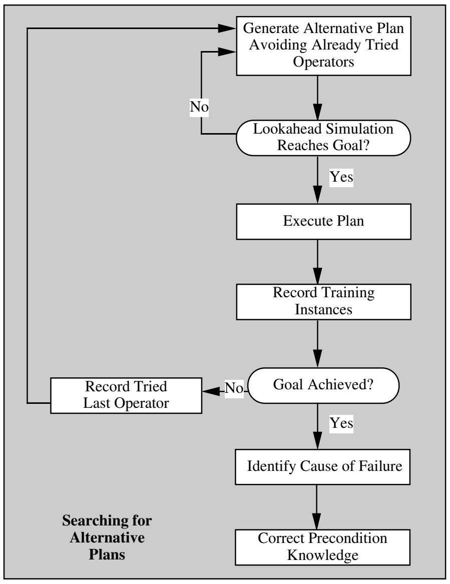


Figure 9.4: Searching for alternative plans

passed to the inductive learner, with the differences between the successful plan and the failed plans forming the basis of the operator precondition learning.

9.1.2 Credit assignment–Which operators are incorrect?

Delayed feedback, incompletely sensed, non-deterministic or multi-agent environments (E8 and E9) prevent an agent from assuming that the last operator executed led to a performance failure. In these situations, the agent is faced with a credit assignment problem of locating the operator(s) that caused the failure. IMPROV's approach is to compare the failed plans to a successful solution, and use the differences to determine the operators that are in error. For each state in the successful plan, IMPROV determines which operator would have been chosen using the original, possibly incorrect, planning knowledge applied to that state. This original operator is compared to the operator used in the successful plan. Any differences are taken to be the operators with incorrect knowledge.

Figure 9.5 shows an example. As the successful plan includes the additional operator, `change-gear`, the preconditions for this operator are overspecific and will be generalized to include it in plans to achieve this goal in future. Also, `set-speed 10` should be specialized so it is not chosen until after the car is in a lower gear. In general, an operator that is present in the successful plan but does not occur at the same point in the original plan, has overspecific preconditions and should be generalized. Similarly, an operator that is not present in the successful plan but is present in the original, has overgeneral preconditions that should be specialized. Where one operator has been replaced by another, this is a combination of an overgeneral operator (the original) and an overspecific one (the successful replacement).

Although this approach is not guaranteed to always identify the incorrect operators, it allows the agent to more accurately locate which operators are incorrect than weaker methods because IMPROV has access to more information (in the form of the successful plan). Traditional error correction methods consider only the incorrect plan during this credit assignment. Therefore, they must rely on a fixed bias. For instance, reinforcement learners using temporal difference assign most blame for a failure to the final step of a plan (`set-speed 0`, in the example). It is very

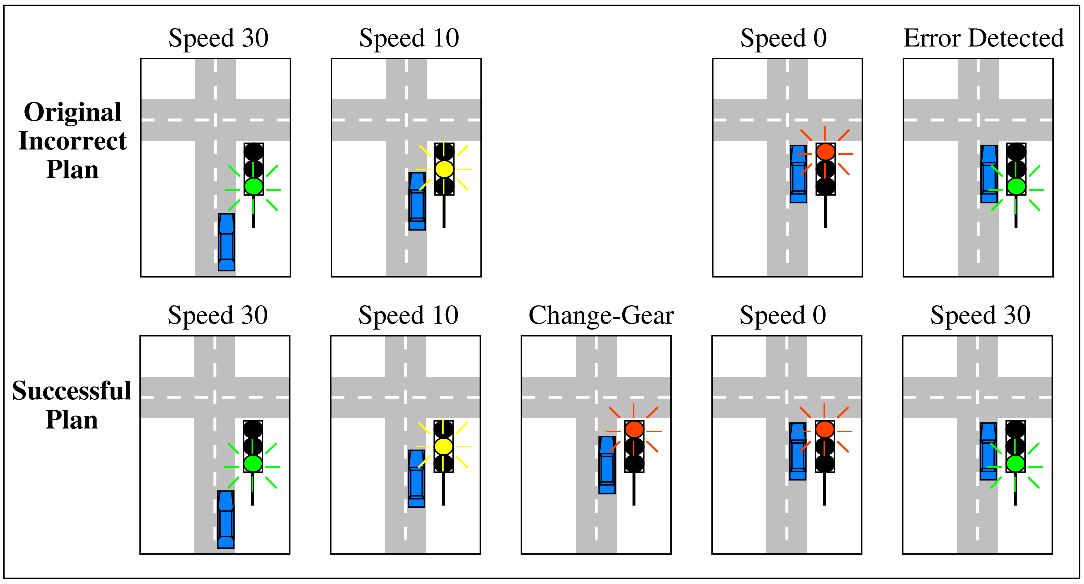


Figure 9.5: Differences between plans used for credit assignment

difficult for such a system to discover that the error is earlier in the plan, while still maintaining the final `set-speed 0` operator, which is required in any successful plan. This is an example of the benefit of separating the search for a correct plan from the task of generalizing that plan to future situations. The reinforcement learner would first have to learn not to choose `set-speed 0` and then have to unlearn that and learn to choose it again, in order to discover the correct plan.

9.1.3 Credit assignment–How the operators are incorrect?

Having determined the operators that are incorrect, IMPROV must decide how to specialize or generalize the precondition knowledge. Unless the agent has access to an additional source of knowledge, such as a complete causal theory, it is impossible to determine deductively exactly which features need to be added or dropped. Instead, the system must rely on its accumulated experience and inductively guess.

To achieve this induction, IMPROV trains the symbolic category learner SCA2 described in Chapter 8. SCA2 learns incrementally (E7), is tolerant of noise (E8), can represent large numbers of disjunctive operator preconditions (E4) and can be guided by multiple sources of knowledge (E11). The instances consist of state and goal information and whether a particular operator lead to success or failure. SCA2 learns the category of which operator is correct for the current state and goal. During training, SCA2 learns increasingly specific rules by adding one feature at a time to existing prediction rules. During prediction, the rule base of prediction rules is indirectly searched from specific to general, to identify the most specific rule matching a subset of the test instance. As a result, the preconditions of an operator are represented twice within the system. First, as rules for when to choose the operator, that can be executed efficiently. Secondly, as rules within the inductive learner, that are less efficient to access (requiring a deliberate search) but support learning well.

SCA2 is trained on all of the instances, positive and negative, found during the search for a successful plan. Learning is delayed until the successful plan has been found, allowing IMPROV to identify the incorrect operators (as described above) and also more accurately identify the important features that cause an operator to succeed or fail. IMPROV biases SCA2's learning to features of the state that are present in positive instances but missing, or different, in negative instances. To help clarify this, consider the simple example shown in Figure 9.6.

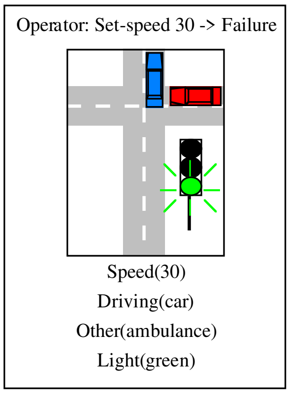


Figure 9.6: Credit assignment problem for a single failure instance

The agent crosses the intersection by choosing the `set-speed 30` operator and detects a failure. The problem is to decide which feature lead to the failure and there is little information available from examining just this failure instance. However, if the learner waits until a successful instance has been found (Figure 9.7), it is clear that the failure was probably due to the presence of an ambulance (which should have been given precedence). This determination can only be made in the context of the two instances.

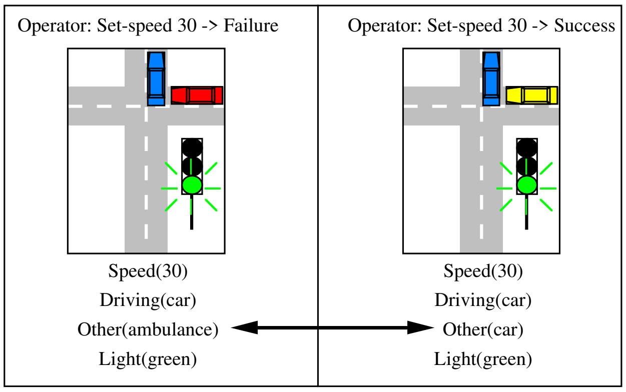


Figure 9.7: Differences between states used for credit assignment

The general property is that any incremental learner discards information about the instances it sees during training (or it's not incremental). IMPROV gains more information about what to discard from an instance by considering a set of instances. The number of instances IMPROV considers during each training episode only increase until it finds a successful plan, providing a bound that keeps learning incremental. This is *k-incremental* learning, as the learner considers sets of k instances at a time and while k varies, it does not grow over the life of the agent.

It is important to realize that the differences between multiple instances only form an additional bias for the inductive learner. If the bias is incorrect, learning will be slower but will still converge to the correct knowledge, assuming the category can be described correctly in the agent's representation language (see Chapter 8). These differences form a default, weak bias for the inductive learning. Additional, domain specific knowledge sources or guidance from an instructor can be easily incorporated into IMPROV's learning as an additional bias. In the next chapter we will evaluate the influence that adding domain specific knowledge has on learning accuracy.

IMPROV delays learning until a correct plan has been found. If training is delayed even further, until a unique cause of the failure has been identified, this results in a system that is actively experimenting (such as the induction that happens in EXPO [Gil, 1991; Gil, 1993]–see Chapter 3 for a discussion of related approaches to learning precondition knowledge). Thus IMPROV and EXPO define two interesting, k-incremental learners along the spectrum from pure incremental learners to pure non-incremental learners (Figure 9.8).

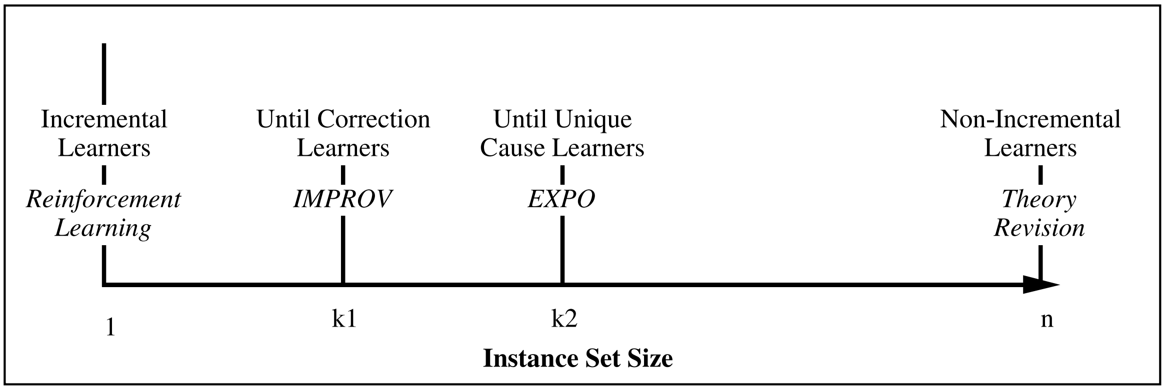


Figure 9.8: Variation of instance set size from incremental to non-incremental learners

K-incremental learning is related to incremental batch learners, such as RL [Clearwater *et al.*, 1989]. However, those learners train on a set of randomly selected instances and therefore are *passive* learners. IMPROV and EXPO are *actively* creating instances as they act in the world. This allows the learners to select instances that are closely related (e.g. repeated attempts at the same task), making learning easier but this also means the agents must be more careful in their learning. To return to the earlier example of failing while driving through an intersection (Figure 9.6). A system that does not delay its credit assignment may attribute the cause of the failure to the light being green. This incorrect learning will then make it much harder for the agent to discover its error. In future, it will expect the correct operator (Figure 9.7) to fail as the light is green. The agent will be unlikely to execute a plan to cross when the light is green and discover that it succeeds. A passive learner, however, would be no more or less likely to be presented with the correct plan, no matter what it learned as a result of seeing the first plan. This allows a passive learner to more easily recover from any early, incorrect learning. The important point is that in agent-based learning, it is more important to assign credit correctly than it is to assign credit quickly [Pearson, 1995].

9.1.4 Changing the domain knowledge

In systems where the operator knowledge is represented declaratively, the learner can directly modify the operator knowledge to make a correction. As IMPROV is restricted to procedural access to its knowledge, it cannot modify the knowledge directly. Instead of searching its rule-base for the incorrect knowledge and correcting individual rules, IMPROV learns additional rules that correct the decision about which operator to select. This method is discussed in detail in the section on implementation (Section 9.3.4).

9.2 Scope of Corrections

IMPROV's precondition correction method, including error detection, is summarized in Figure 9.9. Once an error is detected, IMPROV records information from the failure of that plan and

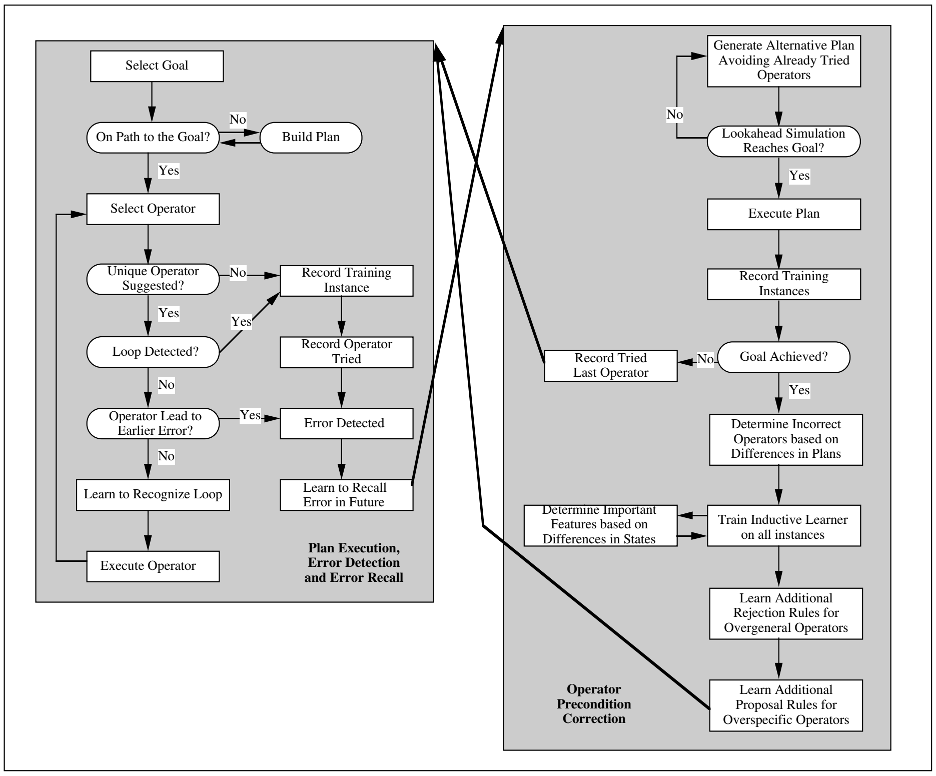
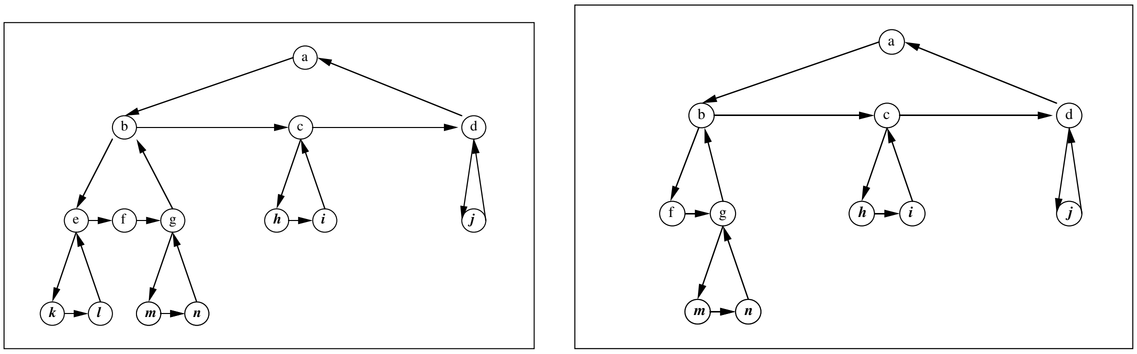


Figure 9.9: IMPROV's precondition correction method

then attempts a correction. There are three main components in the correction, searching for the correct plan, determining which operators are incorrect and then training an inductive learner to determine how they are incorrect. The search through the space of plans is guided by the knowledge gained during earlier inductive learning. Similarly, the search through the space of concepts made by the inductive learner is guided by the results of executing alternative plans. The credit assignment problems during learning are made easier to solve by delaying learning until after a correct plan has been found.

We will use a specific example to help clarify what corrections IMPROV can make. Figures 9.10-9.12 show a series of corrections to an agent's domain knowledge represented by an operator hierarchy (similar to the one for crossing an intersection in Figure 4.2). In these figures each node is an operator, with the italicized operators generating external motor commands when they are executed. The hierarchy is analogous to the typical top-down program decomposition used extensively in computer science, except that as in Figure 4.2, the order of the operators (or procedures) arises from operator precondition rules that select each operator in turn. Each of the subsequent corrections are made to the same original theory, showing how IMPROV can map this theory into a range of different theories.

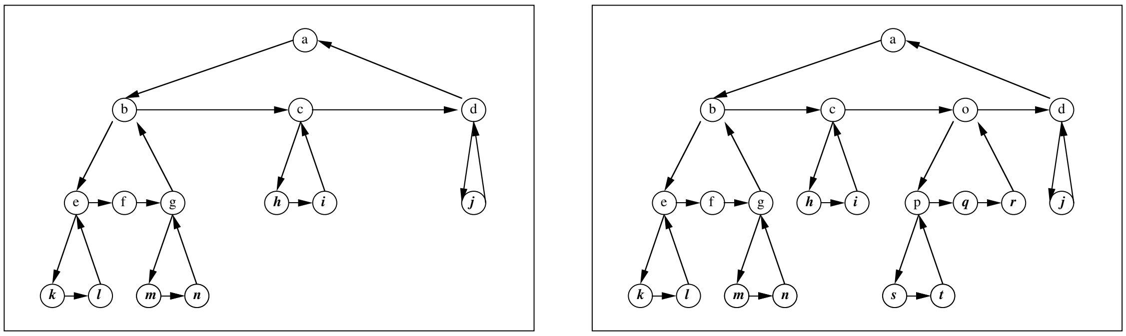
Figure 9.10 shows a correction where an operator's preconditions were initially overgeneral and were specialized by IMPROV. In Figure 9.10, operator `e` has overgeneral preconditions, leading it to be included in the plan to achieve `b` when it is unnecessary. For example, this could be the `set-speed 0` operator that is included in a plan to cross an intersection even when the light is



(a) Initial Theory (b) Final Theory

Figure 9.10: Specializing an overgeneral operator precondition

green. By specializing the preconditions of `e`, the operator is removed from the plan, along with its subtree (`k` and `l`).



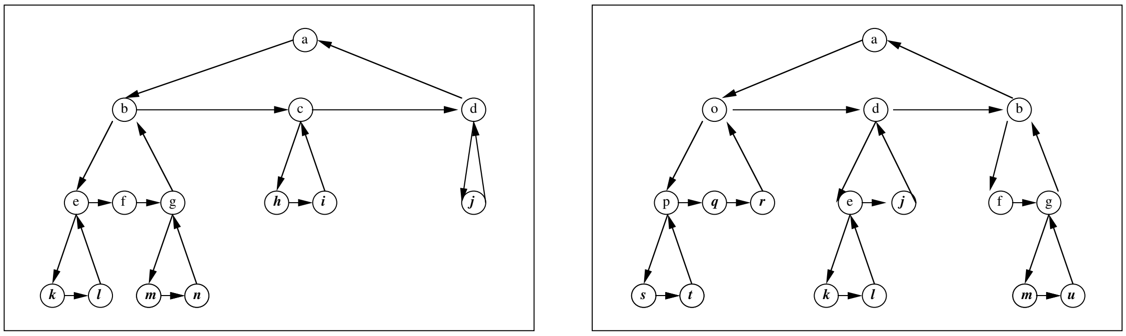
(a) Initial Theory (b) Final Theory

Figure 9.11: Generalizing an overspecific operator precondition

Figure 9.11 shows a complementary situation where the preconditions for operator `o` are initially overspecific, so it is not included in operator `a`. IMPROV generalizes the conditions, leading to that operator (and its subtree) being included in the plan to directly achieve operator `a`. This capability assumes that the agent already has operator `o` available, so that it can be included. If `o` doesn't exist, IMPROV will not create it. In that case, the only correction IMPROV could make would be to include the more primitive operators `p`, `q` and `r` as part of the plan to achieve operator `a`.

The important point to recognize is that operators can be added to the hierarchy by generalizing their preconditions and can be removed by specializing their preconditions. These changes can be made at any level and apply to complete operator subtrees.

Figure 9.12 shows a more complex example after a series of corrections, generalizing and specializing operators. The intention here is to show that by combining these primitive corrections of specializing and generalizing preconditions, the hierarchy of operators can be radically altered.



(a) Initial Theory (b) Final Theory

Figure 9.12: Correcting a series of operator precondition errors

9.3 Implementation

This section describes the details of how IMPROV's precondition learning method is implemented in Soar.

9.3.1 Solving the current problem

In generating alternative plans, the agent has to select a plan that has not been tried before. The issue of deciding whether a plan has been already executed becomes challenging when the problem solving episodes occur at widely different times. For example, in trying to cross an intersection the agent drives through a red light, leading to a failure. When approaching another intersection, the agent must decide whether to try the same plan again or attempt a new plan. The agent should still drive across the intersection in some situations. The key issue, then, for the implementation is determining when a plan has already been attempted in a *similar* situation and therefore that a new plan should be selected. IMPROV records all information about the search as rules, rather than as data structures in working memory to ensure that the memory requirements of the agent do not grow over time, slowing the agent's performance [1].

We have tested a series of alternative strategies for determining when a plan has already been tried in a similar situation. The strategies are as follows:

1. Same State / 0% Exploration

   The agent records the goal, all the features of the state, the operator that was executed and whether this lead directly to failure. An alternative is only generated when the agent returns to the same state during a future recovery attempt. This is also labeled 0% exploration as this leads to the lowest level of exploration as the agent only believes an action has already been tried, if it has been tried in an identical state.

2. Any State / 100% Exploration

   The agent only records the goal, the operator and the result, with no features from the state. An alternative is selected when the agent considers choosing the same operator in service of

[1] Soar's highly optimized match algorithm has time complexity that's proportional to the size of working memory, but under certain conditions is independent of the number of rules being matched[Doorenbos, 1993].

the same goal. This leads to a high degree of exploration, as the agent will always try an alternative action when faced with a goal it has earlier failed to achieve.

3. Partial State / 50% Exploration

   The agent records the goal, all of the state features, the operator and the result in the same manner as the Same State method above. However, an alternative is generated when half of the features match between the recorded state and the current state, rather than all of the features in the Same State method. This strategy falls between the very specific match of the first approach and the very general match of the second approach.

4. Learned State / Variable Exploration

   The agent records the goal, the features matched by the inductive learner, in selecting this operator, the operator and the result. An alternative is generated when the agent returns to a state matching the recorded state features. This strategy is similar to the partial state approach, but the degree of similarity required changes as the agent learns. Initially, when SCA2 has only been trained on a few instances, it will contain rules that match only a few features. As the agent learns more, the rules become more specialized and the agent requires a closer match to select an alternative operator. In essence, as the agent learns more about the domain, IMPROV will make finer distinctions in deciding when a state has already been visited. This means that an agent initially explores a lot and over time, explores less frequently.

For each strategy, the method is the same. When an failure is detected, the agent records the relevant information as described above. These records take the form of rules; an example is shown in Figure 9.13. During a correction attempt, IMPROV attempts to construct a plan using

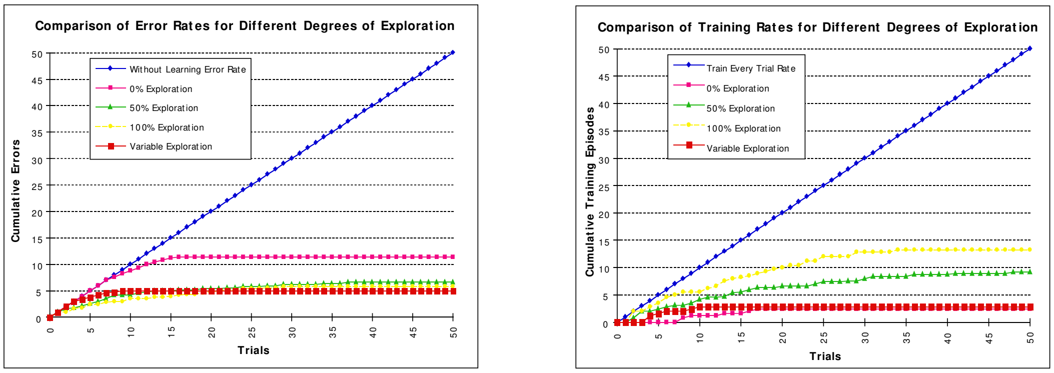
```
IF      goal(generate-plan)
and     error(error-code125)
and     task(cross-intersection)
and     engine(running)
and     light(red)
and     distance(close)
and     gear(high)

THEN  add already-tried(set-speed,30)
```

Figure 9.13: Rule showing when an operator has already been tried

UBID and hence the knowledge encoded in SCA2. During this stage, the rules for avoiding certain previous operators may be recalled, leading SCA2 to select other alternatives. It is important to realize that these rules test the current error instance. Once IMPROV has learned new rules to correct the agent's operator knowledge, this error instance will not be recalled (see Section 7.2 for details). Therefore these rules will only be recalled during the search for an alternative plan and then will be discarded after IMPROV corrects the agent's knowledge, ensuring that the search is separate from the long-term learning process.

Figure 9.14 shows a comparison of these different approaches, applied to a driving task. The details of these experiments are discussed more fully in Chapter 10. The first graph (Figure 9.14(a))



(a) Number of Errors

(b) Number of Training Episodes

Figure 9.14: Comparison of different strategies for exploring plan space

shows the cumulative number of errors made in completing a series of tasks. The second graph (Figure 9.14(b)) shows the cumulative number of training episodes (i.e. number of sets of instances passed to SCA2) over the same set of trials. The 0% exploration case leads to the most errors (as the system repeatedly makes the same mistakes) but results in very few training episodes. This is because each training episode consists of a large set of instances (the k in k-incremental learning is large) and results in high quality learning. The 100% exploration case leads to the most training episodes, but about the same number of errors as the 50% exploration case. This is because each training episode is based on only a small set of instances and therefore results in poor quality learning. The 50% exploration approach provides a balance between these extremes, with less learning overhead for the same number of errors as the 100% exploration case. However, the variable exploration approach outperforms all of the others. It leads to the same error rates as the higher levels of exploration (as the agent initially explores a lot) but with the low learning overhead of the 0% exploration case (as it later avoids exploration). Therefore, the variable exploration, or learned state, approach is the one used by default in IMPROV.

9.3.2 Credit assignment–Which operators are incorrect?

IMPROV decides which operators have incorrect knowledge by comparing the successful plan to the original, incorrect plan. Once a plan has been executed that achieves the current goal, IMPROV compares the operators that would have originally been chosen to the operators in the successful plan. When IMPROV is correcting an error, it temporarily records each state and operator along the path of the plan being executed. Then for each state in the successful plan, IMPROV determines which operator would have been chosen using the original, possibly incorrect, planning knowledge applied to that state. This original operator is compared to the successful operator found as a result of IMPROV's search through alternative plans.

Continuing the example of stalling a car from not changing gear, the top of Figure 9.15 shows the successful plan. The lower part of the figure shows the original, incorrect precondition knowledge and the operators that are recalled for each state in the successful plan. This approach provides an efficient way to compare plans, a process that is a potentially complex matching problem.

Differences between the successful plan and the original plan, indicate points where the agent's operator precondition knowledge should be changed. An operator that is absent from the correct

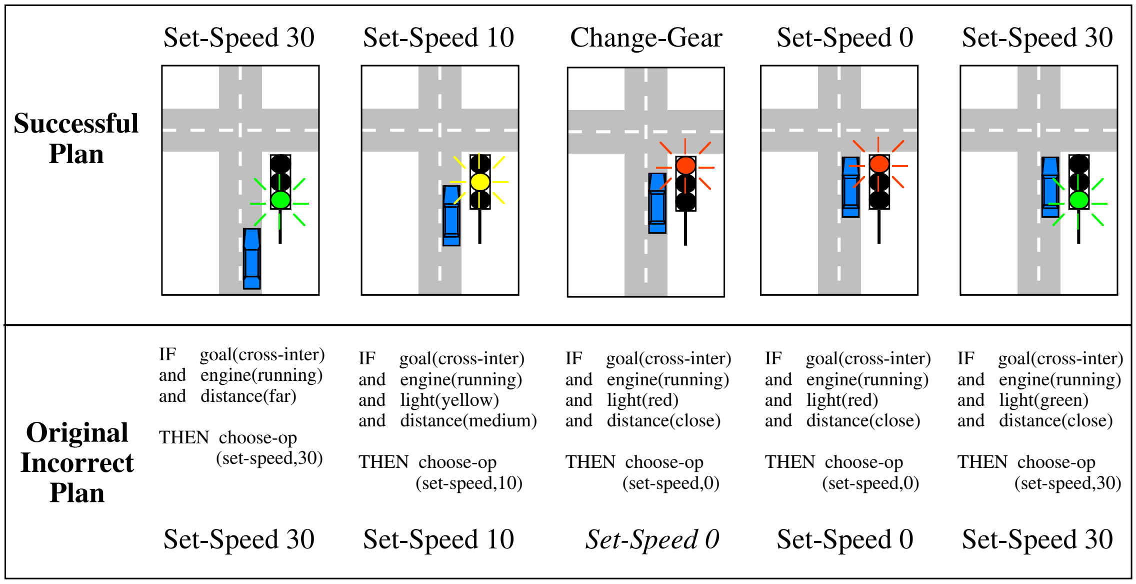


Figure 9.15: Comparing operators from successful and incorrect plans

plan should have its preconditions generalized, while an operator that is only included in the incorrect plan should have its preconditions specialized. In the example shown in Figure 9.15, `change-gear` is only present in the successful plan and should have its preconditions generalized, while `set-speed 0` is chosen too soon in the incorrect plan, and should have its preconditions specialized.

9.3.3 Credit assignment–How the operators are incorrect?

IMPROV determines the parts of the state and goal that led to success or failure of an operator by inductively learning which operator is appropriate for a given state and goal. The inductive learner is SCA2, which is described in detail in Chapter 8. As IMPROV builds and executes plans, in search of a successful path to the goal, it builds up a series of instances that are later used to train SCA2. The instances consist of the goal (currently limited to a single symbol–the name of the super-operator) and the state, with a category label of the operator that was chosen and whether the plan lead to success or failure.

The instances are recorded as rules (long-term memory), and are only recalled at the point when the agent is ready to train SCA2. An example is shown in Figure 9.16. The rule includes a test for the current error instance (error-code132). As IMPROV learns new precondition rules, this error code will no longer be recalled (see Chapter 7 for details), effectively causing the agent to discard the training instances related to that error. This ensures that the number of instances considered during training does not grow over the life of the agent and ensures that IMPROV remains a k-incremental learner.

Once a successful plan has been discovered, IMPROV trains SCA2 on all of the instances recorded for the current error. The learning is biased by differences between the states in an effort to more accurately locate the cause of an operator succeeding or failing. IMPROV selects the feature (attribute-value pair) in training SCA2 based on the following heuristics:

1. Present in success, missing from all failures

   The feature occurs in the state when the operator succeeded, but never when the operator

```
IF       error(error-code132)
and      operator(recall-instances)

THEN  add instance(<i>)
      add operator(<i>,change-gear))
      add result(<i>,success)
      add goal(<i>,cross-intersection)
      add feature(<i>,isa,car,<c>)
      add feature(<i>,gear, high,<c>)
      add feature(<i>,light red,<r>)
      add feature(<i>,engine,running,<c>) etc.
```

Figure 9.16: Rule to record a training instance

failed, e.g. a green traffic light.

2. Present in success, missing from some failures

   The feature occurs in the successful state and doesn't occur in some cases when the operator failed.

3. Features important later in the plan

   The feature occurs in the success and is missing from the failures at a later point in the plan. This helps IMPROV focus on causes that are only apparent close to the point of a failure and pass them back to earlier sections of the plan.

4. Fixed order of attribute importance

   IMPROV can include a domain specific list of attributes ranked in order of their expected importance in determining the success or failure of operators. For instance, when driving, this list might include that the position of cars is more important than their color.

5. Random

   If none of the previous heuristics have narrowed the choice of feature to a single entry, IMPROV picks randomly.

These biases act as a series of filters. An operator is proposed for each feature of the state, leading to a tie. Then these heuristics are applied in turn to reduce the set of operators, until the tie is broken and a single operator (and feature) is selected. This framework helps to demonstrate how additional knowledge sources can easily be added to guide IMPROV's learning. For example, if an instructor is present they could provide additional guidance in identifying important properties of the environment and this would become an additional bias.

9.3.4 Changing the domain knowledge

In systems where the operator knowledge is represented declaratively, the learner can directly modify the operator knowledge. As IMPROV is restricted to procedural access to its knowledge, it cannot modify the knowledge directly. Instead, IMPROV corrects the operator knowledge by learning additional rules that correct the decision about which operator to select. This is a general approach for modifying knowledge in Soar agents [Laird, 1988]. In IMPROV, a rule is learned to reject the original (incorrect) operator and another rule is learned which suggests the new (correct) operator. For example, to generalize the preconditions of the `change-gear` operator, IMPROV could learn the rule shown in Figure 9.17(a), and to specialize the preconditions for `set-speed 0`, IMPROV could learn the rule shown in Figure 9.17(b).

```
IF     goal(cross-intersection)
and    engine(running)
and    light(red)
and    distance(close)
and    gear(high)

THEN   choose-operator(change-gear,low)
```

(a) Generalize change-gear

```
IF     goal(cross-intersection)
and    engine(running)
and    light(red)
and    distance(close)
and    gear(high)
and    choose-operator(set-speed,0)

THEN   reject-operator(set-speed,0)
```

(b) Specialize set-speed 0

Figure 9.17: Possible rules to generalize and specialize operator preconditions

Unfortunately, this approach is insufficient as the rules learned in making a correction may themselves be wrong, as this is an inductive rather than deductive process. Having learned the rule shown in Figure 9.17(b), it would be impossible for IMPROV to later generalize the preconditions for `set-speed 0` (for example, so it's chosen when an emergency happens) as rules to reject an operator dominate rules that suggest that operator. If the agent learned the rule shown in Figure 9.18, with the intention of stopping in an emergency, the rule in Figure 9.17(b) would still prevent `set-speed 0` from being selected.

```
IF     goal(cross-intersection)
and    engine(running)
and    light(red)
and    distance(close)
and    status(emergency)

THEN   choose-operator(set-speed,0)
```

Figure 9.18: A rule that's intended to generalize set-speed 0

To overcome this problem, IMPROV uses the concept of *version numbers* for operator precondition and action knowledge. This additional symbol was first described in Chapter 7, in describing IMPROV's approach to discarding records of old errors. In learning rules that reject or suggest specific operators, IMPROV associates a version number with the operator. This allows IMPROV to create new rules that suggest a new version of an operator, replacing the old, rejected version. For the example of learning to generalize `change-gear` and specialize `set-speed 0` and then later to generalize `set-speed 0`; IMPROV would first learn the rules shown in Figure 9.19(a) and then the rule shown in Figure 9.19(b). Notice that the different version numbers ensure that the rejection rule does not match the most recent version of `set-speed 0`. Over time, early versions are masked by later versions. The rules are never deleted leading to a potential source of inefficiency for Soar's matcher. Our experiments to date have not shown any significant slow-down from these additional rules, but this may become more of an issue if a very large number of corrections are made to the same part of the agent's domain knowledge.

```
IF      goal(cross-intersection)
and     engine(running)
and     light(red)
and     distance(close)
and     gear(high)

THEN  choose-operator(change-gear,low,version 1)

IF      goal(cross-intersection)
and     engine(running)
and     light(red)
and     distance(close)
and     gear(high)
and     choose-operator(set-speed,0,version 1)

THEN  reject-operator(set-speed,0,version 1)
```

(a) Generalize change-gear and specialize set-speed 0

```
IF      goal(cross-intersection)
and     engine(running)
and     light(red)
and     distance(close)
and     status(emergency)

THEN  choose-operator(set-speed,0,version 2)
```

(b) Generalize set-speed 0

Figure 9.19: Actual rules to generalize and specialize operator preconditions

IMPROV computes the version numbers at the point of learning new precondition rules. Rejection rules, for specializing an operator's preconditions, test the existing version number, ensuring they are specific to the current version (see Figure 9.19(a)). In learning rules to generalize an operator's preconditions, IMPROV examines the existing operators that have already been suggested and rejected in the current state, and uses the next highest version number. For example, if `set-speed 0` versions 1-4 have already been suggested and rejected in the current state, version 5 would be used. As the decision about which version number to use is made locally, there is no need to maintain a global record of the current version numbers of operators.

As an operator's precondition version number is separate from the name of the operator, the agent's knowledge concerning the actions or effects of the operator are unaffected by a change in the precondition knowledge. This is a desirable property as it allows IMPROV to learn operator preconditions separately from learning operator effects. Version numbers are used to allow more recent precondition knowledge to override earlier, incorrect precondition knowledge and for no other purpose (see Appendix A.3 for further details on IMPROV's operator precondition learning method).

9.4 Discussion

IMPROV learns in constant time. That is, learning is not proportional to the current size of the agent's domain knowledge, nor does it increase over the life of the agent. The time to process a given set of training instances is, naturally, proportional to the size of that set, but the sets do not grow in size over the agent's life and the time per induction will remain constant or decrease[2]. A number of the design decisions related to IMPROV's learning were made to achieve this goal, such as only assuming procedural access to the agent's knowledge. Let us consider more globally, what the effects are of taking this approach to learning. The changes IMPROV makes to the agent's knowledge are local. That is, IMPROV does not immediately transfer a correction to the rest of the agent's knowledge. For example, if while stopping at a red light, the agent learns that braking in the rain takes twice as long as expected, the agent's knowledge will only be corrected with respect

[2]This is because SCA2 searches from specific to general while learning rules from general to specific.

to stopping at intersections. The problem of correcting the agent's knowledge in other situations that are affected by the new knowledge, is deferred until that new situation is encountered. For example, when the agent considers braking to slow down when leaving a freeway, it may recall the previous failure (see Chapter 7), leading it to replan on the basis of the new knowledge and learn a correction that covers freeway braking. In this way, the new knowledge is only transfered when the agent encounters this new situation. This is similar to *lazy evaluation* in functional programming, where a value is only computed when it is requested. IMPROV only transfers the local corrections on-demand, in response to new tasks. This spreads the work of making global corrections across multiple problem solving episodes, allowing the agent to learn and perform at a uniform, constant speed during each episode.

IMPROV's learning method is intentionally general and flexible. It makes few assumptions about the nature of the environment and is intended to be embedded within an autonomous agent that uses large bodies of knowledge in its reasoning. In return for this generality, IMPROV's basic learning mechanisms are weak. It uses a general purpose, state-based search method for planning, rather than a more advanced plan-space planner. It uses a weak search strategy to generate new plans in search of a successful plan, rather than stronger plan repair approaches. At each step, the lack of assumptions, about the agent's knowledge or the environment, limits IMPROV to weak learning. This is because it is precisely those assumptions that make so many existing learning methods inapplicable to the agent-based learning in complex environments that we are interested in. This means we would expect a knowledge-lean version of IMPROV to perform less well in simpler environments than existing machine learning approaches. For example, in domains where unlimited time is available for learning, non-incremental learning based on declarative access to the agent's knowledge should outperform IMPROV's learning. Similarly, if credit can accurately be assigned from examining a single training instance, then delaying learning (as IMPROV does) is unnecessary.

IMPROV also relies on the assumption that an initial, approximate domain theory is available. This initial domain knowledge guides planning and subsequent exploration. Without this knowledge, many problems would become intractable to IMPROV's search methods as the agent would be forced to rely on blind search. Let us look more closely at this assumption. The vast majority of machine learning systems do not make this assumption or require this initial bias. However, this seems somewhat surprising (at least to this researcher). It would seem that the vast majority of problems that people are interested in solving, are problems for which there is some or a great deal of knowledge available. In medical diagnosis, or intelligent vehicle control, or internet softbots, or robotic bomb-disposal systems, or expert systems for system configuration, to name just a few, the problem areas are well understood. The problem that leads to the knowledge acquisition bottleneck, I would suggest, is not the difficulty in supplying an approximate model of the problem domain, but in supplying one that is complete and correct. IMPROV will perform best in domains where there is already some level of human understanding of the processes and representations that are appropriate. IMPROV will perform less well on tasks that are generally classed as discovery tasks, where little is known about the problem or the correct behavior. Before leaving this point, it is also worth pointing out that almost every machine learning system assumes, or implicitly requires, the domain to be appropriately represented. If the wrong features are presented to the learner, there are very few systems that can recover (and even constructive induction systems are limited to a relatively simple combinations of existing features). I suggest that the space of problems that can be appropriately represented, largely overlaps the problems where approximate domain knowledge is available.

CHAPTER 10

Results and Evaluation

The goal of this research is to develop a learner capable of extending and correcting domain knowledge in complex environments. The degree of coverage of the different classes of knowledge errors, the character of the environment and the character and performance of the agent in learning and achieving its goals can all be used as criteria for evaluating the system. IMPROV has been implemented in a simulated robot domain, as an initial testbed, and in a simulated driving domain, allowing us to test IMPROV's performance on a wide range of complex environmental properties.

10.1 Experimental Domains

The task for the agent in the robot domain is to align blocks on tables (see Figure 10.1). The blocks have different characteristics and the agent must learn which of the blocks can be successfully picked up. The agent can move around the room and has a single gripper for manipulating the

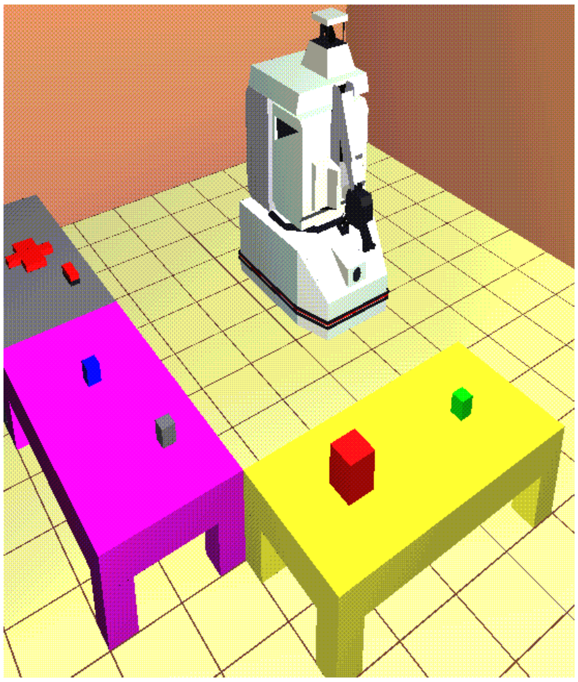


Figure 10.1: Robotic domain

blocks.

The task for the agent in the driving domain is to successfully cross an intersection (see Figure 10.2). There are other agents in the environment, both on the road and sidewalks, and processes (such as traffic lights) which change independently of the agent's actions [1]. The agent must learn the correct procedure for crossing the intersection. For example, the agent should stop for red lights or police cars. In both domains, the agent starts with sufficient initial knowledge to build a

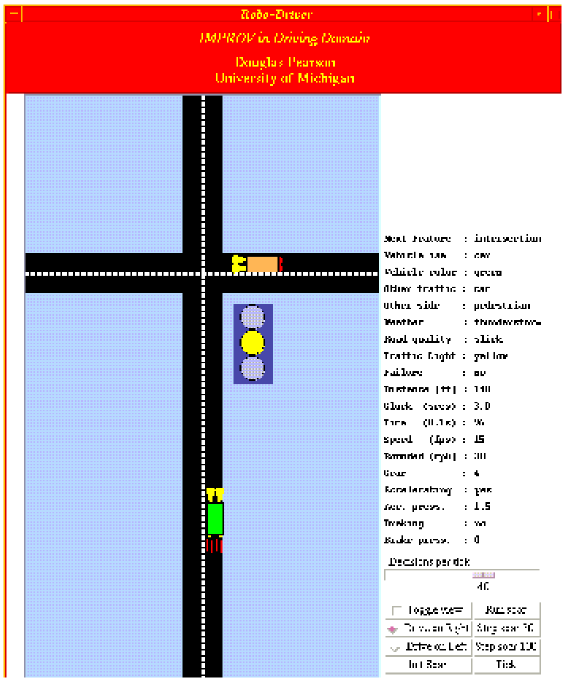


Figure 10.2: Driving domain

plan that it believes will succeed. This knowledge ensures that the agent is not performing a blind search, either for a correct plan or for the correct operator knowledge. In both cases, the search is usefully biased by the agent's initial knowledge.

10.2 Evaluation Criteria

Together, these two test domains have allowed us to evaluate the system's performance on a range of criteria:

- Coverage of different classes of errors

  IMPROV has been demonstrated correcting domain knowledge that initially included:

  - Overgeneral Operator Preconditions
  - Overspecific Operator Preconditions
  - Incomplete Operator Effects
  - Extraneous Operator Effects

  IMPROV has not been tested on, and has no ability to learn, when there are:

[1] For a more comprehensive, cognitive model of driving behavior in Soar, see [Aasman95] .

- Missing Operators

- Character of the Environment

  IMPROV has been demonstrated on environments that have:

  - Large state space [E1]

    The state space for the driving domain exceeds a billion distinct states, with certain parts of the state space (the other agents) being added and removed to produce dynamic changes to the representation.
  - Large goal space [E2]

    Neither of the goal spaces is particularly large, but the robot domain allows for in excess of 100 different goal states.
  - Actions with complex structure [E3]

    Each domain included many conditional actions and a number of sequential effects that occurred over time as a result of a single external action.
  - Actions that apply in a large range of states [E4]

    IMPROV has been tested on theories missing up to 8 disjunctive preconditions. The initial domain knowledge can include many more disjunctive preconditions, but we only explored learning up to an additional 8 disjuncts at one time.
  - Irreversible environmental changes [E5]

    The driving domain contains actions that cannot be reversed, with the agent only facing each problem situation once and being able to execute a single sequence of actions in an effort to achieve the goal.
  - Time-critical tasks [E6]

    The driving domain required the agent to act within a fixed time limit, as the environment changed asynchronously. The agent always had sufficient time for processing, but the time allowed was fixed ensuring the agent could not spend arbitrary time reasoning and would fail if it slowed after learning.
  - Long-term tasks and continual existence [E7]

    The agent wasn't given time to process information "off-line", it was continually acting in the environment.
  - Limited sensing [E8]

    Noise was introduced to the agent's sensors in the robot domain, leading it to incorrectly sense external features of the environment. Additionally, feedback that an action would lead to success or failure was delayed. These forms of noise largely hinder credit assignment during learning, although they can also make the search for a solution to the current goal more difficult.
  - Environmental changes independent of the agent [E9]

    The driving domain included multiple agents and other external processes that changed the environment asynchronously and without action by the agent.
  - Evolving environmental processes [E10]

    The underlying physics for the robot domain was modified after training IMPROV for a period in the environment. The agent was then required to re-learn the correct effects of its actions.

– Many forms of feedback [E11]

  In the driving domain we experimented with adding additional knowledge to guide IMPROV's learning. In additional work, IMPROV was integrated with a system for taking human instructions, Instructo-Soar [Huffman and Laird, 1993; Huffman, 1994], and demonstrated on some simple tasks in a modified robotic domain. This demonstrates IMPROV's ability to receive guidance either from an external source (an instructor) or from an internal source (additional knowledge) [Pearson and Huffman, 1995].

The experiments were not designed to compare IMPROV's learning rate or accuracy to that of other systems, but rather to evaluate the scope of IMPROV's learning and demonstrate its ability to learn in environments with a wide range of challenging properties. The key point to take from each experiment is the demonstration that IMPROV could perform in a particular type of environment, while using a deliberate, knowledge intensive and yet efficient learning method. We primarily measured IMPROV's performance in terms of the number of errors the agent made in performing a series of trial tasks. This is preferable to directly analyzing the accuracy of the rules that IMPROV has learned as the agent's domain knowledge is distributed across a large number of rules acting in concert as a result of IMPROV's incremental addition of rules during learning. Figure 10.3 shows an example of a learning experiment. The attributes considered during learning

Attributes	Number of possible values	Sample Initial Knowledge (+3x2)	Sample Final Knowledge		
Distance	5	Close	Close	Close	Close
Speed	10				
My-vehicle-isa	10		Car		
My-vehicle-color	10				
My-road-sign	2	Signal	Signal	Signal	Signal
My-light-color	3		Red	Green	Green
Gear	4				
Weather	10				
Road-Quality	10				
First-road-user-isa	10			Police	Ambulance
Sidewalk-user-isa	10				
Other-road-sign	2				
Operator:		set-speed(30)	set-speed(0)	set-speed(0)	set-speed(0)

Figure 10.3: Example domain knowledge in driving domain experiment

are shown on the left, along with the range of values each attribute can take. Next is shown part of the initial knowledge given to the agent (i.e. that the `set-speed 30` operator should be chosen when the distance is close and the road-sign is a traffic signal). Finally, an example of a target theory is shown in the third column. In this experiment, the agent must learn three exception cases to the initial theory's general rules. The target operator preconditions consist of 3 disjunctive terms, each containing two additional conjunctive terms that are missing in the initial knowledge. This example is labeled as +3x2. The + indicates that the initial theory is overgeneral and must add three disjunctive terms (each of two conjuncts) to reach the target theory. Overspecific initial theories are the converse, for example -3x2 would mean the agent started with the target theory shown and had to learn the initial theory. The robot domain contains a similar number of attributes and has test cases formulated in the same manner. Each experiment reflects the average results from 10 runs and each test case is designed so that the agent's initial, incorrect knowledge will lead to a failure if the agent does not learn. This makes it easier to identify the effect that learning is having on the agent's ability to perform the task. Without any learning, every trial would lead to an error.

10.3 Environmental Complexity

10.3.1 Overgeneral and Overspecific Preconditions

Figure 10.4 shows the cumulative number of errors made by IMPROV over the course of 50 trials, for a range of target domain theories. The diagonal line is a reference, showing the number

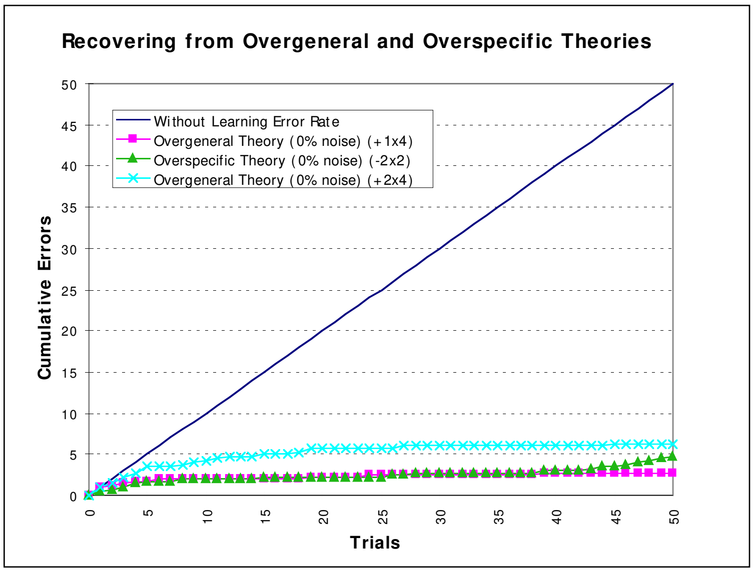


Figure 10.4: Correcting overgeneral and overspecific precondition knowledge

of errors that would be made if the system was not learning and providing a uniform reference point between the graphs. The graph shows that IMPROV can correct overgeneral and overspecific theories and quickly converges to a reasonable approximation of the correct theory, resulting in few total errors.

10.3.2 Disjunctive Preconditions

Figure 10.5 shows the cumulative number of errors made as IMPROV learns domain knowledge that includes an increasing number of disjunctive terms. Disjunctive preconditions can present difficulties for some existing learning methods and this graph demonstrates that learning is more difficult as the number of disjuncts increases, but in each case IMPROV quickly converges towards the correct knowledge.

10.3.3 Noise

Figure 10.6 demonstrates the effect of adding noise to the training data. In these experiments, the level of noise indicates the percentage chance that a given attribute is incorrectly sensed by IMPROV for the duration of that trial. The graph shows that noise significantly degrades the

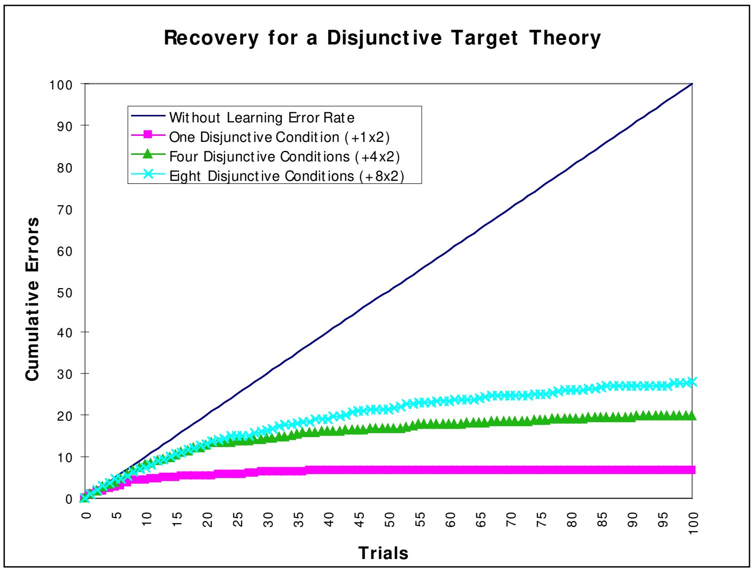


Figure 10.5: Learning disjunctive precondition knowledge

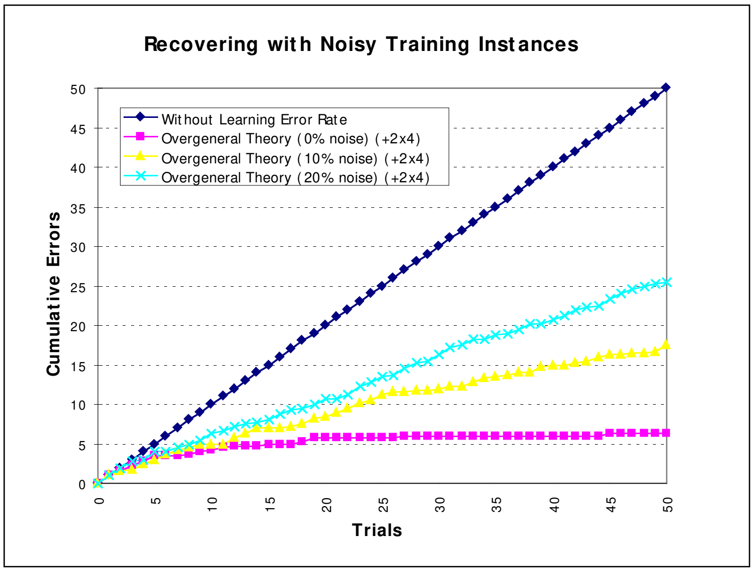


Figure 10.6: Learning from noisy training data

agent's performance, but does illustrate that the learning can tolerate the presence of noise.

10.3.4 Evolving Target Domain Model

Figure 10.7 illustrates that the learning is robust to changes in the target domain theory. In this test, the physics underlying the behavior of the environment changes after the agent has completed 50 trials. An example of this is when braking takes more time as tires wear out, or when part of a robot starts to malfunction. The unexpected change to the domain causes the agent to make more

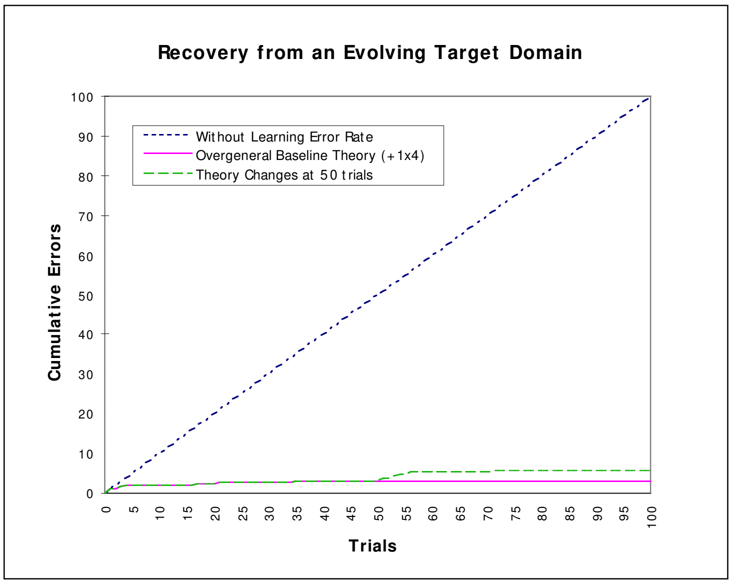


Figure 10.7: Learning as the target knowledge changes over time

errors, but it then adjusts to the new theory. The baseline case shows the behavior when no change is made to the domain.

10.3.5 Limited Time

Figure 10.8 shows the CPU time per trial over the life of the system, while performing on the previous evolving domain learning problem. The first, largest spike, includes the time for the agent to build an initial plan. The later spikes indicate when a correction had to be made. The time spent on each correction remains constant or decreases as the agent learns and the theory becomes more complex. This is in contrast to most machine learning algorithms, which become slower as the theory they are learning grows more complex. Also, the frequency of the corrections decreases as the system's theory becomes more accurate. These results help to support the hypothesis that limiting an agent to procedural access to its knowledge leads to an efficient correction method.

10.4 K-Incremental Learning

IMPROV's ability to assign credit for errors is improved by delaying learning until a success plan has been found. Figure 10.9 demonstrates the benefit of using this deliberate, analytic k-incremental approach over a pure incremental learner. IMPROV was modified to train immediately after seeing each individual instance, rather than waiting and training on a set of instances; thereby simulating a pure incremental system. Figure 10.9(a) shows the comparison in the driving domain,

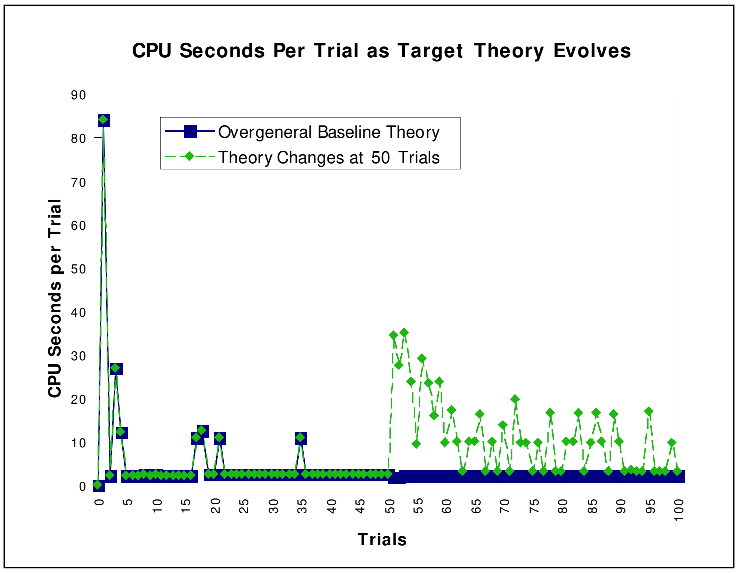
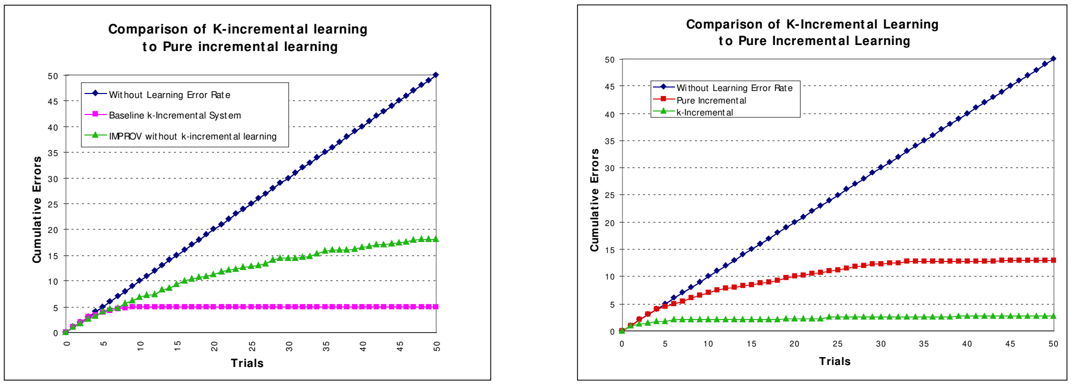


Figure 10.8: CPU time per trial as the agent learns

while Figure 10.9(b) shows the relative performance in the robot domain. The difference in the



(a) Driving Domain (b) Robot Domain

Figure 10.9: Comparison of k-incremental learning to pure incremental

resulting error rates is substantial, confirming that the accuracy of the learning has been improved by IMPROV delaying its training until a successful plan has been found.

10.5 Knowledge Directed Learning

IMPROV represents a weak, general purpose method for learning planning knowledge in a range of challenging environments. The learning can be made stronger by adding additional knowledge sources to guide the agent. This is analogous to making a weak search method stronger by adding additional knowledge [Laird and Newell, 1983]. This is a more flexible alternative to building a collection of domain specific methods and then selecting among them or integrating the results of their reasoning. IMPROV's use of a uniform knowledge representation, in a general purpose architecture, simplifies the addition of extra knowledge to guide learning. We have also experimented with knowledge that comes from an unstructured source, such as an instructor [Pearson and Huffman, 1995]. These instructions are presented in English and provide guidance for individual parts of the learning (rather than complete examples of correct behavior). For example,

"Think about your speed."

could be used to guide the inductive learning phase. In this section we demonstrate how IMPROV's error detection, searching to find a successful plan and inductive learning can all be enhanced by the addition of extra knowledge. In these experiments, the additional knowledge was only accessed procedurally, in keeping with the assumptions underlying IMPROV's design.

10.5.1 Error Detection

Figure 10.10 demonstrates the benefit of adding additional knowledge to indicate that an error is about to occur. For example, in the driving domain, this could be an instructor who shouts "Look Out!" prior to a performance failure. For this test, the early indication takes the form of a signal prior to the agent taking an action that would lead to failure. This leads IMPROV to record the action as a failure, and generate an alternative plan instead. The graph shows a

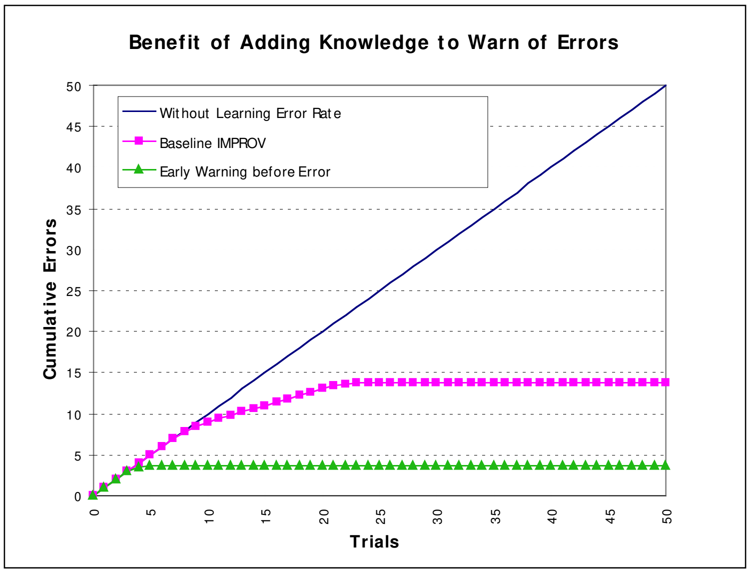


Figure 10.10: Knowledge added to detect errors early

comparison between a baseline IMPROV agent and the same agent with this additional, early detection knowledge. As expected, this signal leads the agent to make substantially fewer errors during performance. The important point in this, and the other knowledge-directed experiments, is not the size of reduction in the error rate. This, after all, depends directly on the quality of the additional knowledge provided. Instead, the important point is that additional knowledge can be easily added to achieve this improvement in performance.

10.5.2 Replanning

Figure 10.11 illustrates the benefit of providing the agent with additional knowledge to guide the generation of alternative plans. For example, this knowledge could be general purpose plan repair knowledge or could be domain specific guidance. In this test, the agent was given the additional knowledge that selecting a speed of 0 was different from selecting any other speed. This leads the driving agent to first consider coming to a complete halt at the intersection after failing to cross it successfully. In this task, this is often the appropriate behavior and therefore this knowledge is a useful heuristic. The graph again shows a comparison between a baseline IMPROV agent and the

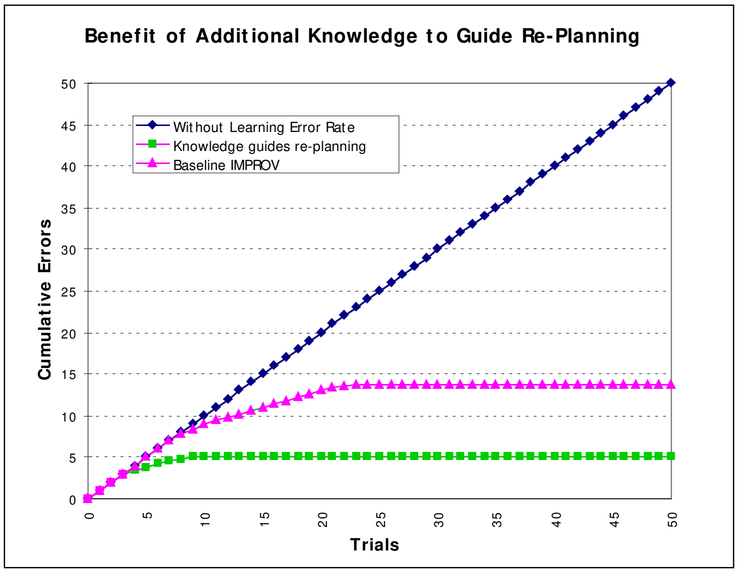


Figure 10.11: Knowledge added to guide replanning after failure

same agent with the additional knowledge to guide it in generating alternative plans after a failure. As this additional knowledge is useful, the agent's performance is significantly improved. Naturally, if the additional guidance during replanning was unhelpful, the agent's performance would degrade. Again, the important point is that the additional knowledge can be easily added. In this example, the two agents differ by just a single rule.

10.5.3 Induction

Figure 10.12 evaluates the effect of adding additional knowledge to guide the agent's inductive learning. This could be a causal theory, a naive physical model or some domain specific heuristics indicating the important features of the domain. In this test, the knowledge guided the inductive learner by indicating the important features of the task. Without this additional knowledge, the agent has no particular reason to focus its learning on these features; with the additional knowledge the agent can learn the correct causes of the failure as soon as it finds a successful plan. The graph shows that the presence of the additional guiding knowledge improves the learning, leading the agent to make fewer errors. The benefit to learning is only as good as the quality of the additional bias. The idea underlying IMPROV's design is rather than trying to develop a powerful learner that performs well on all tasks (a goal many machine learning researchers would consider unattainable), IMPROV instead represents a general learning method whose performance can easily be enhanced by the addition of domain specific knowledge. In many practical learning problems, domain experts can provide guidance to the properties of the environment that they believe are important. This is a much simpler task than correctly specifying the agent's behavior in that environment. It is much easier to know that traffic lights are important in driving than it is to describe the correct behavior for crossing a busy intersection.

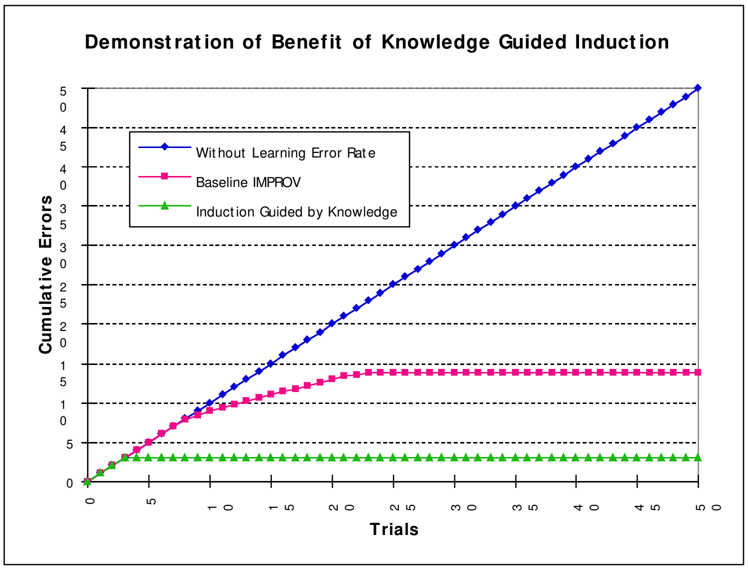
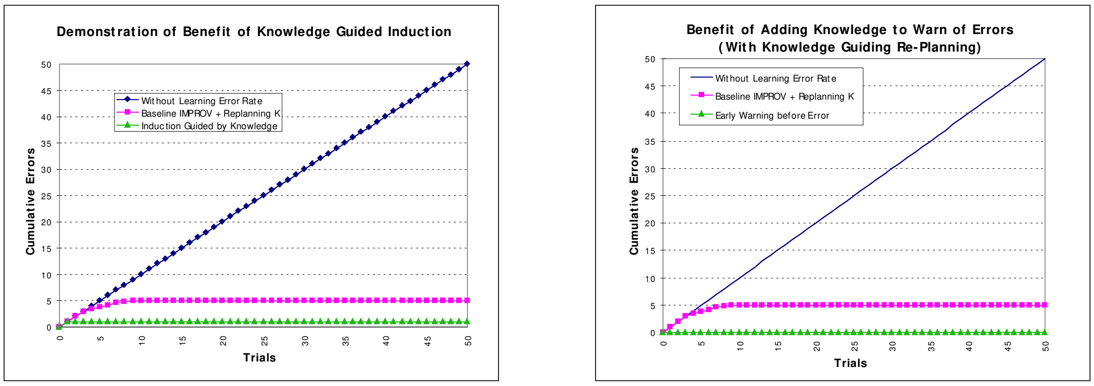


Figure 10.12: Knowledge added to guide inductive learning

10.5.4 Combining Knowledge Sources

Figure 10.13 shows examples of combining two of the knowledge sources. In the first experiment, the agent is given the knowledge to guide the search for an alternative plan and the knowledge to guide inductive learning. As Figure 10.13(a) shows, the agent makes a single mistake, is guided to the correct action and then learns the correct cause, leading it to avoid future mistakes. In the second experiment, the agent is given the knowledge to indicate when an error is about to occur and knowledge to guide the replanning. As Figure 10.13(b) shows, the agent detects that it is about to make a mistake and correctly avoids the error. The agent will continue to train after this, as it lacks causal knowledge about the failure, but the agent makes no further execution errors. These



(a) Replanning and Induction

(b) Early Error Detection and Replanning

Figure 10.13: Combinations of knowledge added to guide learning

examples demonstrate that with sufficient additional knowledge IMPROV's weak searches, for a correct plan and for the correct operator preconditions can both be made deductive problems that require no search.

CHAPTER 11

Correcting Operator Effects

In this chapter, we will describe IMPROV's approach to learning the effects of operator actions. First, we discuss the range of effects that an action can produce and the ways an agent's knowledge of these effects can be wrong. Then we describe some existing approaches to learning operator effects and discuss the scope of effects they can learn. Finally, we present IMPROV's approach and demonstrate how a hierarchical operator representation allows IMPROV to learn complex operator effects by correcting the precondition knowledge of operators lower in the hierarchy.

11.1 The Problem

An agent can select the actions it will take either through an exhaustive policy or by using planning knowledge about the effects of its actions to determine a plan that it expects to succeed. Agents that only learn and maintain an execution policy, primarily reinforcement learners, only learn precondition knowledge. The agent learns a policy which represents when to select an action. Therefore, this policy encodes precondition knowledge. These systems typically do not learn a model of the effects of taking an action. If they perform internal simulations, they usually assume a complete and correct model for the effects of taking an action (e.g Samuel's checker player[Samuel, 1959]) and limit learning to determining the correct conditions for when to take that action.

Agents that learn planning knowledge to model the effects of taking an action may have errors in that knowledge (as described in Section 5.2):

- Incomplete Effects

  The agent is not aware of all of the effects of executing an external action.

- Extraneous Effects

  The agent believes that the external action produces more effects than it really does.

The case where an effect is simply incorrect is a combination of an incomplete effect (the correct effect is missing) and an extraneous effect (the wrong effect is present).

The effects of an action may have a complex structure. For example, actions may produce *sequential* effects, where changes occur over time, or may be *conditional* on the current state of the world (E3). In explaining IMPROV's approach to learning these complex effects of operator actions we will consider a specific example taken from the driving domain. The example is the effects of a single external action, pressing the brake pedal in a car. In this domain, external commands indicate when to press a pedal, with the pedal remaining pressed until another command is issued to release the pedal. Figure 11.1 shows an example of an agent's initial, incorrect knowledge for the effects of braking. This figure shows that braking affects two properties, the rate of deceleration of the

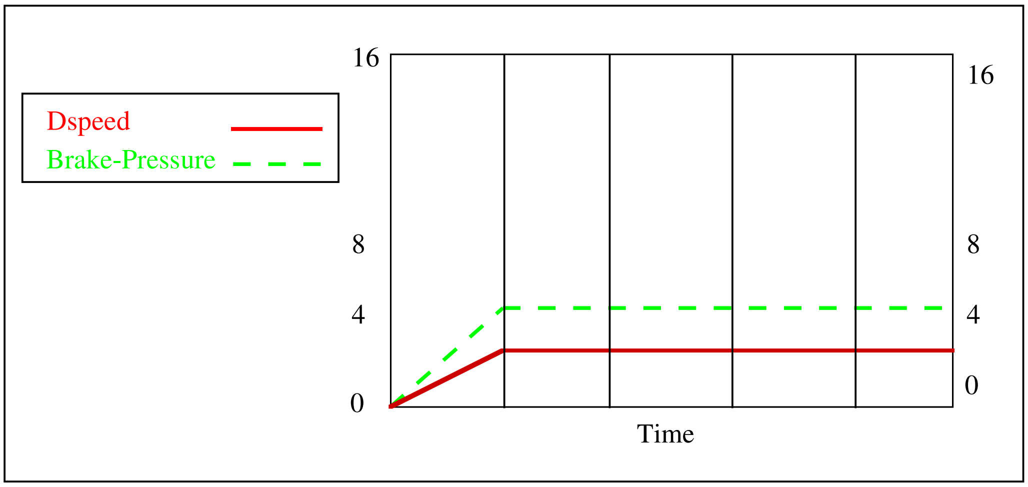


Figure 11.1: Initial model for effects of braking

car (`Dspeed`) and the pressure on the brake pedal (`Brake-Pressure`). The agent's initial planning knowledge indicates that the brake-pressure rises immediately to 4, while the rate of deceleration also rises immediately to 3 and then both remain constant. The correct model for the effects of this action, in this situation, is shown in Figure 11.2 The pressure on the brake pedal increases

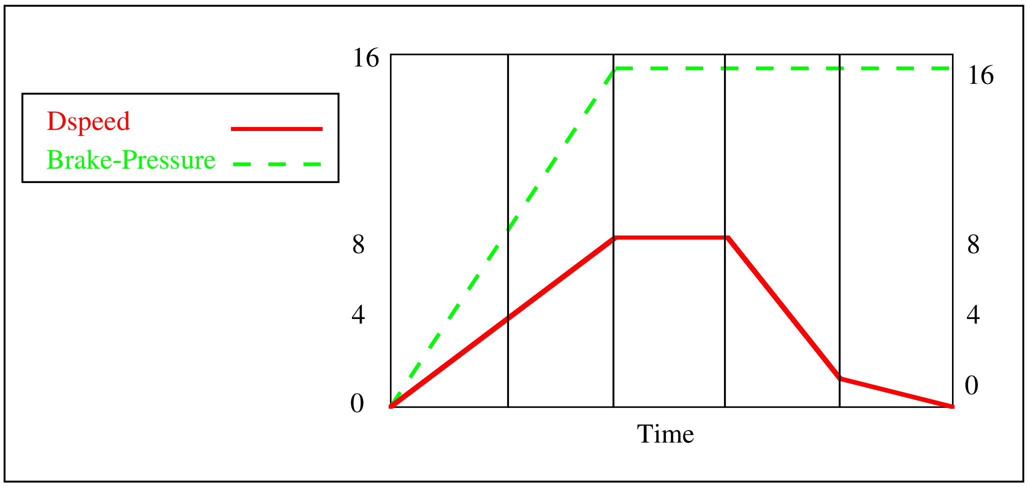


Figure 11.2: Correct model for effects of braking

over time (not in a single step as before) and then remains constant. The rate of deceleration also rises, levels off and then drops to zero as the car comes to a halt. This is an interesting example as the effects vary over time as the result of a single external action and the effect of braking is conditional on the current state of the vehicle. In general, any action can produce a sequence of effects (e.g. knocking over the first of a line of dominoes).

11.2 Related Approaches to Learning Operator Effects

Many existing learners find complex effects, such as those in the braking example (Figure 11.2) difficult to represent and learn. Most representations (as we will shortly describe in detail) are limited to modeling the effects of actions as a single set of state features that exist after completing the action. This form of representation leads these learners to use two different approaches to learning operators that produce effects over time.

The first approach is to model the action at the grain-size of effects that persist after the action has completed. In the example of braking, the effects would be limited to the eventual change in speed, without representing the changing rates of deceleration. However, this approach is insufficient for certain tasks. In the braking example, modeling the rate of deceleration may be important in deciding whether it's safe to brake on ice. More extreme examples include walking in a circle, where the agent finishes in the same position that it started from. This action produces no effect on the final state, but the agent requires a model of where it will be walking to decide if a collision will occur. In short, the intermediate stages of an action with duration may be important in solving certain tasks.

The second approach is to assume that all actions can be decomposed into a series of primitive actions, each producing a single state transition. In the braking example, the external command language would consist of tapping the brake repeatedly, each time increasing the rate of deceleration. This is also an unsatisfactory approach for a number of reasons. Firstly, a series of primitive actions is not equivalent to a single action that produces multiple effects as small actions can be interrupted while a single, large action cannot. The decision about whether an action is interruptible or not, should depend on the dynamics of the environment not on the design of the learner. Secondly, it assumes that the agent's interface to the environment can be modified and engineered to meet the requirements of the learning system. Philosophically, this is undesirable as engineering the problem constrains the applicability of the intelligent agent. More practically, it may be costly or even impossible to decompose the agent's actions. For example, consider clicking a mouse on a computer, that starts a script to produce an arbitrarily long sequence of effects or knocking over the first in a line of dominoes. It is not clear how these events can be modeled as a series of primitive actions without confusing the concepts of the actions an agent can take (the initial push) and the effects that the action produces (the sequence of falling dominoes).

In the remainder of this section we will examine some of the existing approaches to learning operator effects and discuss the range of operator effects they can model.

11.2.1 LIVE, EXPO and OBSERVER

LIVE [Shen, 1989; Shen, 1994], EXPO [Gil, 1992; Gil, 1994] and OBSERVER [Wang, 1995; Wang, 1996] are three, related approaches to learning planning knowledge to model the effect of actions. In each system, operators are represented in a STRIPS-like form, with a list of state features (or postconditions) that are added, and another list of features that are removed, by applying the operator. Effects are learned on the basis of differences between the state before and after an action. In LIVE, features which disappear after an operator has been applied become preconditions, while new features become actions. This initial model is refined based on the differences between the current state and the state before an earlier application of the operator. In OBSERVER, the differences before and after an action is taken, the *delta-state*, are matched against the current operator effects. If the current operator's effects do not account for all of the changes in the delta-state, the effects are generalized or additional, conditional effects are added. EXPO uses a similar approach, based on computing the changes to the state after applying an operator. EXPO simplifies the inductive problem by designing a series of experiments to determine which effects are the result of which action, improving the subsequent learning.

Each of these approaches is unable to represent the class of effects shown in Figure 11.2 and is therefore unable to correctly learn this class of actions. As the STRIPS-like representation only includes a single set of effects it is not possible to model the sequence of effects in the braking example.

11.2.2 Reinforcement Learning

As we mentioned above, most reinforcement learning approaches involve only learning an execution policy, rather than learning models for the effects of taking actions. As a result, these methods learn operator preconditions (in our formalism) and not operator effects. This means the methods are either not used on planning tasks or require a complete and correct model for the effects of each action.

An interesting exception is the work of Mahadevan on learning action models for a robotic domain [Mahadevan, 1992; Mahadevan and Connell, 1991]. Q-learning [Watkins, 1989; Watkins and Dayan, 1992] is used to learn the expected reward of taking each possible action in any given state. As we discussed in Section 4.1 this large representation of states and actions is only appropriate for relatively small state spaces. The system augments this Q-learning with a method for learning the expected state after taking an action. Actions are recorded in tuples mapping from a pre-action state to a post-action state. Both pre and post-action states are vectors describing the probabilities that each state feature will be present before or after the action has been taken. The agent learns the effects of each action by updating the probability vectors in the post-action state after observing the effects of an action. The new vector is a weighted sum of the old vector and the observed state after taking an action. The models for the effects of actions are then used in a lookahead search to guide the agent.

This action model representation is essentially equivalent to STRIPS-style operators with the addition of probabilities for each precondition and effect. IMPROV's representation does not provide explicit support for probabilistic effects (although we will discuss later how IMPROV has a certain level of implicit support for probabilities). However, in Mahadevan's system it is assumed that complex effects, such as the braking example, can be decomposed into a series of primitive actions, each producing a single effect.

11.2.3 MSDD

Multi-Stream Dependency Detection (MSDD) [Oates and Cohen, 1996] is a learning method that has been applied to learning planning operator preconditions and effects. Operators are represented as tuples consisting of a state vector (or *context*) for the operator precondition, an action, and a state vector describing the effect of the action in that context along with a probability that the effect occurs. The same action may be represented as a set of operators, which allows multiple effects to be represented. This representation is similar to the one used by Mahadevan, except that each effect is associated with its own condition, allowing conditional effects to be represented explicitly. MSDD learning occurs by searching for repeated patterns over a series of states. Values that are frequently seen to occur after taking an action become the effects of that action.

As MSDD searches for any patterns of state features across multiple states this results in a very large search space. MSDD employs a number of filters in an effort to prune this space, but the method seems best suited to domains where the assumption is that most attributes of the state will change value from one state to the next. In this class of environments, methods, such as IMPROV, or other learners that focus on changes between states to direct their learning, will perform poorly and MSDD may be more appropriate. However, in environments where most state features remain constant from one time step to the next, it is not clear that addressing this larger problem of recognizing arbitrary patterns will scale well to large environments. It is interesting to note that MSDD examines sequences of states and therefore could potentially detect sequential effects that occur over time (such as in our braking example). However, when the effects are only transitory (such as deceleration in our example) the planning operator representation is insufficient to express this. MSDD operators can model actions that produce multiple, conditional effects, but all effects are modeled as occurring at the end of the action. Thus, learning would either conclude that braking produces zero deceleration (the final state) or a range of conflicting decelerations (from the sequence of states).

11.2.4 TRAIL

TRAIL [Benson, 1995; Benson and Nilsson, 1996] learns planning knowledge in the form of teleo-operators (or TOPS). Operators consist of a possibly disjunctive, precondition (or *preimage*), an action, a single intended effect (or *postcondition literal*) and a set of probabilistic side effects. The action is continuously executed until the postcondition literal is achieved. Operators are learned for each action and postcondition literal combination. The action is repeated until either the postcondition is achieved or a time limit is exceeded and the operator is deemed to have failed. TRAIL analyzes the sequence of states that the agent passes through in executing the operator. State features that have different values, at some point during execution, from their final value once the operator has completed are classed as *side effects*. This analysis of a sequence of states allows TRAIL to learn about actions that take time to produce an effect. TRAIL's modeling also appears well suited to environments that are non-deterministic and where actions produce monotonic effects over the course of their application. The effects must be monotonic because the TOP representation is again insufficient to record the sequential information that is identified during learning. In the braking example, TRAIL would apparently conclude that each rate of deceleration (-4, -8, -1) is a side effect of executing the operator.

11.3 Correcting Operator Effects – Functional Description

IMPROV's approach to correcting operator effects is to model the effects as a sequence of more primitive operators and then correct the preconditions of those primitive operators using the precondition method described in Chapter 9. This representation is highly expressive, allowing IMPROV to model conditional effects as well as sequential effects that occur over time. This representation also allows IMPROV to use a single learning mechanism to learn operator preconditions and operator effects, two problems that have traditionally been seen as different. This is a simple, powerful idea yet it can be a difficult concept to grasp so we will describe it from a number of angles.

The first observation is that generalizing and specializing operator preconditions provides a general method for adding and removing operators from an operator hierarchy. The previous chapter includes many examples of this (Section 9.2). Earlier, we introduced the idea that goals and plans are represented as high level operators. For example, the goal of crossing an intersection and the plan to achieve it can be represented as a high level `cross-intersection` operator, whose termination condition is that the intersection has been crossed and whose implementation is a planned series of motor level operators (e.g. `brake` and `accelerate`). We can extend that same concept to the motor level operators, seeing them as a goal to be achieved and a plan to achieve them. In this case, the goal is to execute an external motor command, the execution implementation knowledge is to send that motor command and the planning implementation knowledge is a series of primitive operators that together model the effects of the motor level operator. These primitive operators are called *single-effect operators* as they each model one expected effect of taking an action. This representation can be extended to cover multiple actions that occur simultaneously (e.g. turning and braking at the same time) by representing the combined actions as a single, compound operator (e.g. `Turn-and-Brake`) [Covrigaru, 1992].

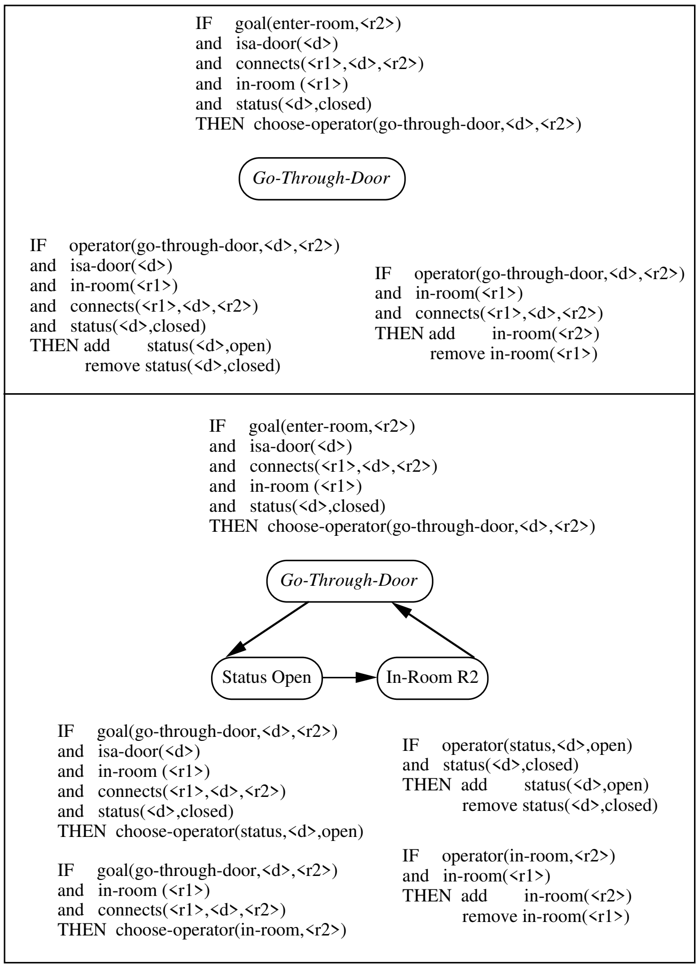


Figure 11.3: Mapping an operator's effects to single-effect operators

Figure 11.3 shows an example of representing the planning knowledge for `Go-Through-Door` either as direct, operator implementation knowledge (in the top of the figure) or as a pair of more primitive operators (in the lower part of the figure). Notice that the knowledge about whether a particular effect should occur moves into the operator preconditions for the primitive operators. Time is modeled through a series of "T" operators that indicate the point when time advances. Figure 11.4 shows a simple example of a `Brake` operator with `T` operators included (for a full example see Figure 11.6). IMPROV's default model is to assume that state transitions occur at the same rate as sensing. The `T` operators indicate when sensing will occur and when a transition is expected to occur. This representation allows IMPROV to represent multiple effects that occur simultaneously (see Figure 11.6 for an example).

With the hierarchical representation of Figure 11.3, the problem of determining the correct *effects* of `Go-Through-Door` becomes the problem of determining the correct *preconditions* of `Status` and `In-Room`. Extraneous effects can be removed by specializing operator preconditions, while missing effects can be added by generalizing an operator's preconditions. For example, when going

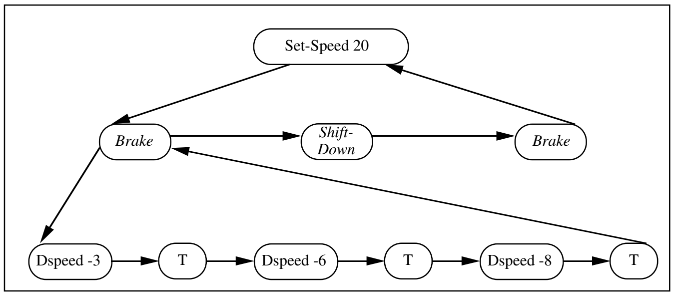


Figure 11.4: Modeling of temporal transitions

through a revolving door, the door's status is not changed by passing through it. This is an example of an extraneous effect: `(status,<d>,open)` does not occur when the external command to go through a door is sent. In this hierarchical representation, this extraneous effect is removed by specializing the preconditions of the `status open` operator, to include the additional condition that the door is not a revolving door. We will shortly look at another example in more detail.

The general result is:

> *The planned effects of an operator can be represented as a series of more primitive operators. This allows the problem of correcting operator actions to be reduced to the problem of correcting operator preconditions.*

This result is important as the problem of learning operator actions has traditionally been seen as completely separate from the problem of learning operator preconditions. Furthermore, correcting complex actions with duration or that produce conditional, sequential effects has been seen as difficult to solve, leading to the prevalence of correction methods that limit actions to simple, instantaneous effects. The rich representation of operator effects as a series of primitive operators, allows IMPROV to represent and correct complex actions that produce conditional or sequential effects. The sequence of primitive, single-effect operators can model much more complex behavior than the traditional STRIPS-like models of actions as discrete, instantaneous transitions. The re-use of the precondition correction method also allows the agent to re-use knowledge for learning preconditions on problems for learning effects (e.g. a causal theory indicating important properties of the domain). Finally, from a research perspective, this duality allows us to develop capabilities motivated for one problem and transfer them to the other. For example, having developed an approach for learning conditional effects, this is identical to learning conditional branches in plans. They are both problems of learning conditional operator preconditions at different levels of an operator hierarchy.

This method for correcting operator effects is very general as it can be applied to any level of the hierarchy. As we will discuss in the implementation section, it may be appropriate to have different biases for the different levels of the hierarchy, but the same general method is essentially applied throughout. As IMPROV corrects operator actions by relying on a lower level of operators, it is important to realize that this process will always terminate. The lowest level of operators in any IMPROV operator hierarchy consist of *single-effect* operators that manipulate a single symbol. For example, `Dspeed` operators only modify the rate of change of the agent's speed. As these operators modify only one symbol, their effects are *guaranteed* to be correct. For example, the `Dspeed -3` operator is the operator that produces a rate of change of speed of -3. Errors in the agent's knowledge are the result of using the wrong sequence of single-effect operators, not because the effects of those operators are wrong. For example, if braking produces a final deceleration of -12 not -10, then the `Dspeed -10` operator should not be used to model braking, while the `Dspeed -12` operator should be, and this change is achieved by changing their preconditions.

11.4 Correcting Operator Effects – Implementation

To more fully illustrate IMPROV's learning of operator effects and how IMPROV's precondition correction method is re-used we will describe IMPROV's behavior on the braking example in detail. This will also provide an opportunity to describe the places where different biases are used to guide learning.

The method for learning operator preconditions is explained in Chapter 9 and consists of the following stages:

1. Knowledge Representation
2. Detecting performance failures
3. Solving the current problem
4. Learning a general correction for the future
   (a) Credit assignment–Which operators are incorrect?
   (b) Credit assignment–How the operators are incorrect?
   (c) Changing the domain knowledge

11.4.1 Knowledge Representation

The first step is to model the effects of a motor level operator as a series of more elementary *single-effect* operators, as shown in Figure 11.5. This hierarchy of operators models the incorrect

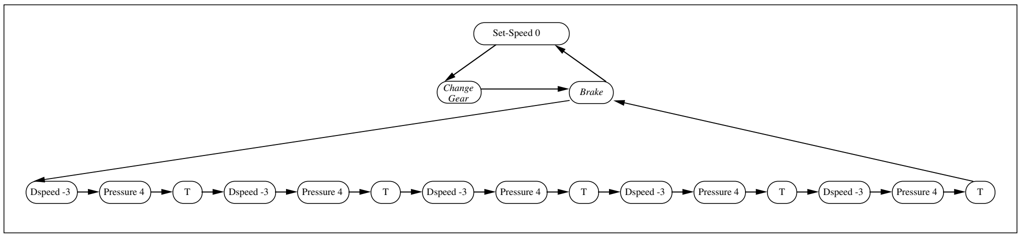


Figure 11.5: IMPROV's initial representation for the effects of braking

braking behavior shown in Figure 11.1, where braking immediately leads to a constant rate of deceleration and a steady pressure on the brake pedal. The effects of the `Brake` operator are modeled by the sequence of `DSpeed` and `Brake-Pressure` operators at the lowest level of the hierarchy. The lowest level operators are *single-effect* operators as they each modify a single symbol in the internal planning representation. They play no direct role in execution, although they are useful for error detection (see Section 6.1.1).

IMPROV also assumes there is explicit feedback indicating when the external command is still producing changes in the environment. This is an important assumption as it significantly simplifies the learning problem. Without this feedback, the learner would have to decide whether arbitrarily delayed external effects could still be the result of the agent's actions. For example, without this assumption it would be difficult to decide when the action of knocking over the first in a line of dominoes had stopped producing effects.

IMPROV corrects the agent's knowledge to produce the operator hierarchy shown in Figure 11.6. This hierarchy encodes the braking behavior shown in Figure 11.2, where braking takes time to

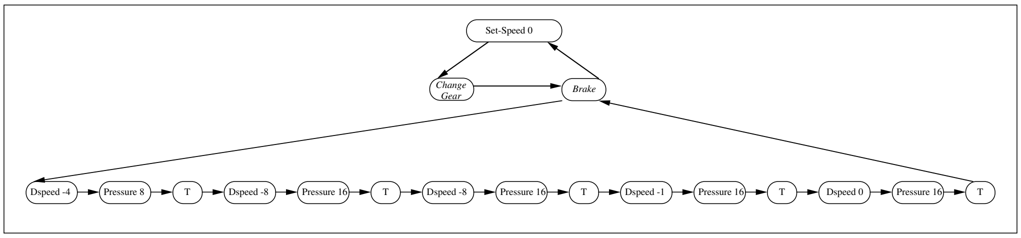


Figure 11.6: IMPROV's final representation for the effects of braking

reach full deceleration and then lessens as the car comes to a halt.

The mapping from the initial representation to the final representation is achieved by changing the preconditions of the single-effect operators. Preconditions are specialized to remove an existing operator and they are generalized to include a new operator. For example, `DSpeed -3` has overgeneral preconditions (as it should not be included in the effects of `Brake`), while `DSpeed -8` is overspecific and should be generalized to include it in the implementation. In this example, IMPROV learns this correct model after a single failure and a single attempt at finding the correct model. This is because the credit assignment problems are simplified in this example because the action produces deterministic effects and because the initial model is close to the correct behavior. In general, the search for the correct model could require many trials, in the same way that learning operator preconditions for motor level operators may require many trials to correctly solve the credit assignment problems.

11.4.2 Detecting performance failures

IMPROV detects a performance failure when no operator (or multiple operators) are suggested during execution after building a plan. This principle is applied at each level of the operator hierarchy, including the single-effect operators. In the braking example, the agent initially has knowledge indicating when the `DSpeed -3` operator should be chosen in modeling the effects of braking (Figure 11.7). During execution, the car decelerates at a faster rate than expected. The

```
IF    goal(brake)
and   dspeed(<c>,0)
and   isa(<c>,car)

THEN  choose-operator(dspeed,-3,version 1)


IF    goal(brake)
and   dspeed(<c>,-3)
and   isa(<c>,car)

THEN  choose-operator(dspeed,-3,version 1)
```

Figure 11.7: Incorrect operator preconditions for a single-effect operator

second precondition rule does not match as expected, so no operator is selected signaling a performance failure. IMPROV records the state, operator and result (a failure) and learns to avoid this operator in a future search for an alternative, successful implementation.

To date, the agent's modeling of time is left to domain-specific knowledge. The `T` operators, that indicate when time advances, can be implemented to include a time limit for each effect to be seen. If the limit is exceeded this indicates a performance failure. However, the current version of IMPROV has no support for learning these time limits. While waiting for an effect to occur, IMPROV will repeatedly select the same single-effect operator and subsequent `T` operator. IMPROV does not attempt to detect loops at the lowest, single-effect level as otherwise any delay would be detected as a loop and signaled as a failure. For the purposes of this braking example, we will assume that the sequential effects are deterministic and occur reliably.

In the current implementation of IMPROV, when a failure is detected at the lowest level, IMPROV does not immediately halt execution and start to search for the correct knowledge. This is because the agent may still achieve its goals, even though it has incorrect knowledge about the effects of its operators. Instead of halting execution, IMPROV records that an error occurred, as usual, but then continues the execution. In future, when IMPROV selects `Brake` during planning or execution, IMPROV recalls the error and only then starts searching for a correction.

11.4.3 Solving the current problem

Having detected an error, IMPROV will search for an alternative sequence of single-effect operators to implement the motor level operator. The search begins at this lowest level as `Brake` was the super-operator when the error was detected. IMPROV generates alternative implementations, uses them to replan when to brake, executes them and observes whether they lead to an error or if the plan succeeds.

In correcting motor-level operator effects, IMPROV uses an additional bias in guiding its search. When a failure is detected, IMPROV records the values that are observed changing during the failure. For example, that `DSpeed` changed to -4, rather than -3. This information is recorded along with the information indicating that `DSpeed -3` has already been tried. In correcting operator effects, the search for the correct operator sequence starts with this observed behavior, rather than from the original, incorrect plan.

In this way, the differences between states that are used as the basis of learning in many existing systems, are used as search control in IMPROV. If the observed values are not sufficient to model the correct effects, IMPROV will continue to search for the correct planning knowledge, in the same manner as described earlier in Section 9.1.1, searching in decreasing order of similarity to the initial attempt.

In our test domain, the braking example is sufficiently simple that the sequence of changes to `DSpeed` and `Brake-Pressure` observed during the course of the first, failed braking are all that need to be considered. In this case, IMPROV will immediately generate the correct implementation as the first alternative. However, the important point here is that the differences between the states during the braking maneuver are only a bias for the search. This means that when a more complex implementation is required, IMPROV can continue to search for the correct implementation.

11.4.4 Credit assignment–Which operators are incorrect?

Once IMPROV has found a sequence of single-effect operators that lead to achieving the agent's current goal, IMPROV assigns credit in the same way as for learning operator preconditions. IMPROV recalls each state in the successful sequence and compares the operator that was chosen in the original sequence to the one in the successful sequence. Operators only present in the original are specialized, to remove them, while operators only present in the successful sequence are generalized, to include them. Figure 11.8 shows the comparison between the original planning effects and the successful effects (which is done on a state by state basis).

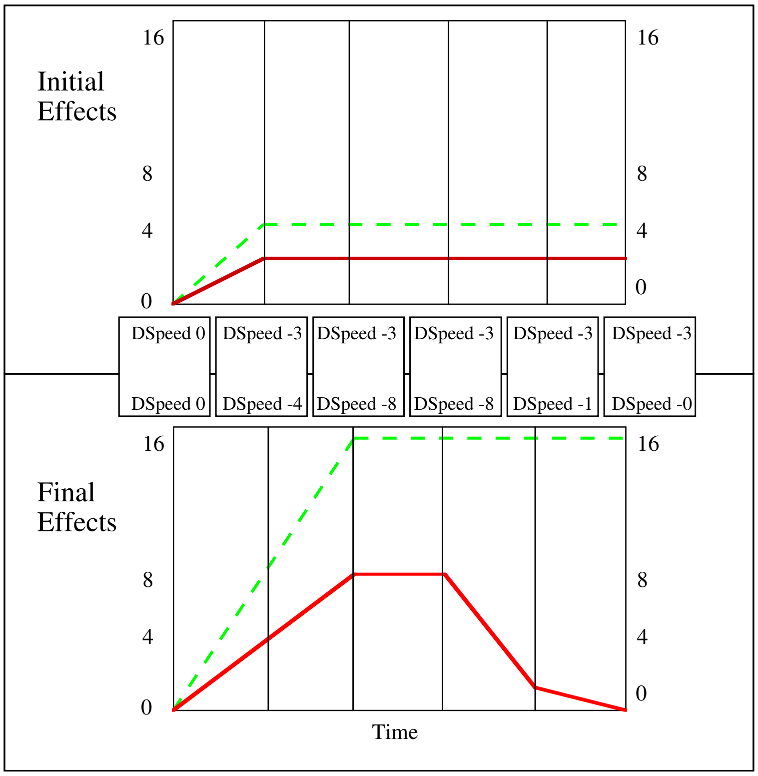


Figure 11.8: Comparison between original and successful effects

11.4.5 Credit assignment–How the operators are incorrect?

IMPROV determines how to specialize or generalize operator preconditions by training its inductive learner, SCA2. The examples of successful or incorrect effects collected during the search for correct behavior are used as positive and negative training instances of single-effect operators.

The inductive learner is described in Chapter 8 and the training process is described in Section 8.1. IMPROV biases its learning to focus on the differences between the states in the incorrect sequence of effects and in the successful sequence. When learning about single-effect operator preconditions, IMPROV prefers differences that share the same attribute as the attribute for the successful single-effect operator. This leads IMPROV to learn rules that describe how a particular attribute changes, such as `DSpeed 0 -> DSpeed -4` (examples of these rules are shown in the next section). This heuristic reflects the intuition that actions are frequently selected to change an existing state feature.

Once again, this bias only guides the inductive learning. If the bias is inappropriate, or if additional features are required to learn when the particular effect occurs, the errors that remain in the agent's planning knowledge after learning will lead to more performance failures. When those failures occur, IMPROV will train SCA2 further and learn additional features until it determines the correct conditions for when a particular effect occurs.

11.4.6 Changing the domain knowledge

IMPROV corrects the operator precondition knowledge for single-effect operators by learning additional rules that change the decision about when the single-effect operator is chosen. Operator are generalized by learning new selection rules and specialized by learning new rejection rules. Figure 11.9 shows an example of rules learned to correct the knowledge for the first step in the effects

```
IF      goal(brake)
and     dspeed(<c>,0)
and     isa(<c>,car)
and     choose-operator(dspeed,-3,version 1)

THEN  reject-operator(dspeed,-3,version 1)

IF      goal(brake)
and     dspeed(<c>,0)
and     isa(<c>,car)

THEN  choose-operator(dspeed,-4,version 1)
```

Figure 11.9: Corrected operator precondition knowledge for single-effect operators

of the `Brake` operator. The preconditions for `DSpeed -3` are specialized, while the preconditions for `DSpeed -4` are generalized. The corrected planning knowledge is used to correct the agent's execution behavior through replanning (see Figure 11.10). The new plan, based on the corrected planning knowledge, will lead IMPROV to brake at the right time, so that it stops before, rather than half way across, an intersection.

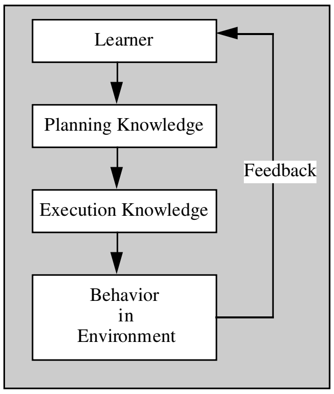


Figure 11.10: Correcting external behavior by correcting planning knowledge

In learning operator effects, IMPROV can be run in a mode where knowledge from low-level operators is compiled into higher level macro-operators. Rather than executing the sequence of single-effect operators to determine the effects of a motor-level operator, the effects are compiled into knowledge for the motor-level operator that can be executed directly. In the example of the `Brake` operator, when the operator is selected during planning, rules would immediately fire to model the sequential effects of braking. This is more efficient than decomposing the motor-level operator into a sequence of primitive single-effect operators and executing each in turn as there is an overhead associated with subgoaling and applying operators. This knowledge compilation process can be applied to each level of the operator hierarchy, resulting in subplans or complete plans that include knowledge of the effects the plan will produce when executed.

There is an ongoing debate in the machine learning community over the merits of knowledge compilation. There is the potential for the additional rules to slow down the rule matcher by a greater amount than is saved during planning through compiled macro-operators. This is an example of the *utility problem* [Minton, 1990]. The extra rules may not be used sufficiently often to produce an overall performance improvement. Considerable research has been directed at examining this problem (e.g. [Gratch and DeJong, 1992; Wray *et al.*, 1996; Minton, 1996]). In this research project we have not directly examined this trade-off. However, on certain problems, the Soar matcher has been shown to maintain a constant speed as up to 100,000 new rules are learned [Doorenbos, 1993]. This result suggests that knowledge compilation in Soar may be beneficial. However, as this remains a matter for debate, IMPROV can either be run in a mode where knowledge is compiled, or where full problem decomposition occurs.

When the agent compiles operator effects at a higher level of the operator hierarchy, IMPROV requires an additional mechanism to correct errors in the compiled knowledge. As the agent cannot directly access and modify this compiled knowledge, IMPROV takes an approach similar to that used for operator preconditions. A version number is associated with the effects of each operator. The compiled effects of the operator are only recalled when the operator's effects version number matches. An example is shown in Figure 11.11. The first rule assigns an effects version number

```
IF      operator(<o>,set-speed,20)
and     planning(yes)

THEN add operator(<o>,effects 1)


IF      operator(<o>,set-speed,20,effects 1)
and     speed(<c>,<old>)
and     isa(<c>,car)

THEN  add     speed(<c>,20)
      delete speed(<c>,<old>)
```

Figure 11.11: Compiled operator effects, indexed by version number

to the `Set-Speed 20` operator, and the second rule shows an effect of the operator, changing the speed.

When IMPROV detects a performance failure, it deletes the effects version number, forcing IMPROV to use the full operator hierarchy. This provides the necessary structure for IMPROV to conduct its search for an alternative sequence of operator effects. Once a correct sequence of primitive, single-effect operators has been discovered, IMPROV generates a new effects version number (in the same manner as for operator preconditions, Section 9.3.4) and learns a new set of effects. This provides a method for IMPROV to replace the incorrect knowledge without directly locating and modifying the rules that are incorrect, saving a potentially expensive search process.

11.5 Discussion

Non-deterministic effects

IMPROV provides no explicit support for non-deterministic effects, but it does provide a degree of implicit support. All non-deterministic effects can be seen as deterministic effects that depend on properties of the environment that the agent is unable to sense[1]. For example, when a robot occasionally drops a pen, this is a result of the friction between pen and gripper, the force applied etc. If these properties cannot be measured, the effect appears non-deterministic. This distinction provides an insight into IMPROV's approach to non-determinism. As SCA2 is trained, it learns progressively more and more specialized rules in an effort to describe the conditions for when an effect occurs. If the features that determine when an effect will occur cannot be sensed, SCA2 will record other irrelevant features in its rules. In the pen example, the learner might record the color of the pen, the position of the robot, whether the paper has lines or not etc. For simplicity, let's assume for the moment that these irrelevant features are uniformly distributed over the instances (i.e. each pen color is equally likely). Now, let's assume that the agent drops the pen 20% of the time. In this case, SCA2 will eventually have 20% of its rules testing these irrelevant features and predicting that the pen is dropped, and 80% predicting that the pen is held. These ratios will occur because SCA2 learns one rule per training instance. During performance, if the irrelevant features are uniformly distributed, there is a 20% chance that SCA2 will predict the pen will be dropped and an 80% chance that it will predict that the pen is held. Notice that this is different from recalling the different probabilities. Rather, the probabilities describe the likelihood of different properties being recalled during planning. In turn this means the same state-space, UBID method can be used for planning, but now the results of planning will vary, depending on the values of otherwise irrelevant features (such as pen color).

To date, we have not evaluated the benefits and disadvantages of this approach to modeling non-deterministic effects. The problem with the current implementation is that the number of features in the domain is fixed and these features usually will not be uniformly distributed. This means that eventually SCA2 will learn rules that test all of the state features and predict each possible outcome. In this situation, in the pen example, the agent would conclude that the pen would be dropped 50% of the time. To achieve the correct behavior when SCA2 reaches the limit of available state features, IMPROV should create new, extra features (which play no part in the domain) with values that are uniformly distributed. By basing learning on these extra features, the implicit non-deterministic effects should be correctly modeled. The creation of these extra features remains as future work.

Strengths and Weaknesses

IMPROV uses a simple model of time and provides no direct support for learning how long effects typically take to occur. By default, IMPROV will wait indefinitely for an action to produce effects. Each time the environment is sensed, if no changes have occurred, the same single-effect operators will be suggested and no failure will be detected. The method for detecting failures in operator actions should include a method for learning the expected time for an effect to occur. Additionally, IMPROV assumes the temporal extent of operator actions is provided as feedback for the learner. This is an additional weakness in IMPROV's reasoning about time.

More generally, IMPROV's learning is based on a very expressive representation for operator effects. Modeling effects as a sequence of operators allows for disjunctive and conjunctive conditional effects and sequential effects that occur over time. As we have shown, without this expressive representation, certain classes of effects cannot be represented, let alone learned. With increasing expressiveness there is a danger that there will be a loss of tractability. Indeed, analyzing a sequence of single-effect operators to determine the conditions for when a particular effect would occur could easily be intractable. IMPROV avoids this potential problem through its use of limited, procedural

[1]If we ignore quantum mechanical issues; a reasonable assumption in most domains.

access. However, as a result, its learning is weaker than other approaches to learning operator effects. This is because the learner has access to less information as it cannot directly access a model of the current operator effects and therefore cannot use this to build experiments or otherwise direct learning. We would expect that methods which make more assumptions about the environment or have greater access to the agent's knowledge will learn more quickly than IMPROV, but these methods will be unusable in many of the complex domains that IMPROV has been developed for.

CHAPTER 12

Conclusions

IMPROV is a method for learning planning knowledge for the preconditions and effects of operators in complex environments. The agent is only assumed to have procedural access to its domain knowledge to ensure learning does not slow down as the agent's knowledge grows. IMPROV learns operator preconditions by generating and executing sequences of operators until the agent's goals are achieved. IMPROV then trains an inductive learner on the instances of successful and unsuccessful operators and the states where the operators were chosen. The method for correcting operator preconditions is re-used to learn knowledge about the effects of operator actions. The effects of motor-level operators are represented as a series of more primitive single-effect operators. This representation is very expressive, allowing conditional and sequential effects to be modeled. Errors in the effects of a motor-level operator are corrected by changing the preconditions of the single-effect operators to add or remove an effect. The complete method is summarized in about 10 pages in Section 1.1.

12.1 Contributions

The main contributions of this thesis are:

1. Presenting a framework for learning planning knowledge

   The thesis presents a framework for correcting errors in planning knowledge. Each stage also includes a discussion of how environmental properties constrain the design of the learner. The stages in the framework are:

   - Classification of knowledge level errors
   - Classification of performance failures
   - Detecting performance failures
   - Solving the current problem
   - Learning a general correction for the future
     - Credit assignment–Which operators are incorrect?
     - Credit assignment–How the operators are incorrect?
     - Changing the domain knowledge

   As we better understand the constraints placed by the environment on each stage of this learning process, it will become possible to independently replace or modify sections of the learner. This goal is not fully realized in this thesis, but it does serve to establish a number of the constraints. For example, the inductive learner could be substituted for another, that provided the same capabilities as SCA2, and the remainder of the learning methods would still be applicable.

2. Presenting a method for learning planning knowledge in complex environments

   IMPROV is a method for learning planning knowledge. The ability to plan allows it to solve tasks with large state and goal spaces. IMPROV's knowledge representation is sufficiently expressive to represent complex effects of action that are conditional and occur over time. IMPROV's learning can span multiple training episodes, allowing it to perform in environments where actions are irreversible. We have demonstrated that IMPROV learns incrementally and the system's execution time does not grow as the agent learns more knowledge. IMPROV is tolerant of noise, exogenous events and domains that change and evolve over time.

3. Casting learning as a weak method within a general problem solving architecture

   IMPROV represents a weak method for learning domain knowledge. We have demonstrated that additional, domain specific knowledge can be added to guide error detection, the search for correct behavior or learning. The implementation of IMPROV within the general problem solving architecture of Soar allows the agent to use arbitrary knowledge and problem solving methods to guide its learning. Additionally, as IMPROV has been developed within an architecture that has been applied to many other tasks, we can be confident that IMPROV's learning can be integrated with those other tasks because the architecture was not designed specifically for correcting knowledge. For example, a system for learning knowledge from natural language instruction, Instructo-Soar [Huffman, 1994], was successfully integrated with IMPROV in about two weeks; producing an agent that could recover from incorrect instructions or use instruction to guide recovery [Pearson and Huffman, 1995].

4. Presenting K-Incremental learning

   IMPROV collects sets of training instances prior to learning. This approach provides the learner with access to more information than is normally available to incremental learners, while still exhibiting incremental learning performance. By examining sets of training instances, the learner can more accurately assign credit for failures and successes. We have experimentally demonstrated that this k-incremental approach leads to improved performance.

5. Demonstrating that operator effects can be learned by correcting operator preconditions

   This thesis demonstrates that the effects of operators can be modeled by a series of more primitive operators. This representation is highly expressive, supporting conditional and sequential effects that occur over time. Additionally, the representation of operator effects through a hierarchy of operators allows IMPROV to reduce the problem of correcting operator effects to correcting operator preconditions. Thus, IMPROV uses a single learning method for correcting any part of its domain knowledge.

6. Demonstrating that procedural access is sufficient for learning operator knowledge

   IMPROV demonstrates that procedural access is all that is required to support deliberate error correction, where the learner explicitly identifies and corrects errors in the agent's knowledge. This restriction allows IMPROV to use expressive representations and model complex actions while keeping learning tractable and efficient.

12.2 Future Work

There are a number of specific extensions that could be made to IMPROV's learning methods:

- Knowledge level errors

  IMPROV currently has no mechanisms for learning new operators or learning new state representations. This prevents IMPROV from generating novel decompositions to solve otherwise intractable problems. To learn new operators, one approach is to determine when the addition of an operator leads to better problem decompositions.

- Improved planning and plan repair

  IMPROV currently relies on a general purpose search method (UBID) for planning and the generation of alternative plans. It would be interesting to investigate alternative planning methods that are more goal directed (e.g. means-ends planning) as well as identifying useful domain independent heuristics for guiding the generation of alternative plans (e.g. considering the dimensions of the plan space being searched).

- Removing initial knowledge

  IMPROV currently assumes the agent starts with approximate planning knowledge. This knowledge focuses the generation of plans and subsequent learning. It would be interesting to provide IMPROV with deliberate exploration or experimentation knowledge and explore how far we could remove the requirement for an initial approximate theory.

- Searching across multiple levels of an operator hierarchy

  IMPROV does not currently search across, or reason about, multiple levels of the operator hierarchy in correcting an error. For example, an agent traveling by car from Los Angeles to New York, detects a failure when it enters Miami. IMPROV currently starts searching for an error at the same level it was detected (for example, having a conflict over choosing the next city to drive to). However, in general we would like IMPROV to be able to reason about the level to search at for a correction and preferably define a search across multiple levels at once. This would allow the agent to attempt a number of corrections at one level, before attempting to find a correction at a different level; and then perhaps returning to the first level.

- Improved temporal reasoning

  IMPROV currently has no methods for learning about the expected duration of actions. IMPROV could be extended to include more direct temporal reasoning and learning. This would allow IMPROV to learn time-limits for when an effect should occur, which currently must be pre-programmed. It might also allow us to remove the assumption that feedback is provided to specify when actions are still producing effects in the environment.

- Evaluation on non-deterministic environments

  IMPROV's approach to non-deterministic events is to learn rules in proportion to the probability of the event occurring. The performance of this approach has not yet been thoroughly evaluated on non-deterministic environments. To date, non-determinism has been limited to noise.

More generally, there are at least two interesting avenues for future research on IMPROV. One is to implement IMPROV's theoretical model using a completely different underlying implementation. For example, exploring how well IMPROV can be implemented using neural networks for knowledge representation, or using an alternative inductive learner. This would quickly clarify the distinction between IMPROV's functional level and implementation details. The second avenue is to embed IMPROV within a more complete intelligent agent, developed for a real world problem domain. This would allow us to investigate the issues of integrating IMPROV with other planning and learning methods. It would also lead to a greater understanding of useful domain specific heuristics for guiding IMPROV's learning and the way knowledge intended for other uses within the agent could be used to assist learning.

APPENDICES

APPENDIX A

Examples of IMPROV's Execution Traces

This appendix contains a number of execution traces from running IMPROV in the simulated driving domain. The traces show the sequence of operators and subgoals selected during problem solving, some of the rules that are learned and comments to explain some of the processing.

A Soar trace consists of a series of decisions. Each decision is either the selection and application of a new operator, or an impasse and creation of a new subgoal. Operators appear as:

```
O: O7 (operator name)
```

while impasses (and their associated subgoals) appear as:

```
==>S: S8 (state no-change)
```

`S8` in this example refers to the newly created problem state, in the new subgoal. The traces include vertical bars to show the extent of each subgoal. The learning of a new rule is shown as:

```
Building chunk-44
```

Other messages are simply included to help explain the purpose of the agent's current reasoning. The traces have been edited at points to reduce their length and make them more readable.

A.1 Example Trace of Planning

Figures A.1-A.3 show an example of IMPROV learning a new plan to approach and then cross an intersection. It includes an example of the rules learned that will guide the agent during execution.

```
soar> run

   0: ==>S: S1  ( )
```

Top Level Operator

```
   1: ...O: O1 (cross-intersection)
   2: ..|==>S: S2 (operator no-change) ( )
```

Loop detection (ignore for now)

```
   3: ..|...O: O2 (comprehend)
   4: ..|..|==>S: S3 (operator no-change) ( )
   5: ..|..|..|==>S: S4 (operator conflict) ( )
   6: ..|..|==>S: S5 (state no-change) ( )
   7: ..|..|...O: O7 (make-candidate1)    (type no-attend)
```

There is no plan, so impasse occurs, leading to planning.

```
   8: ..|..|...O: O8 (make-candidate)
   9: ..|..|...O: O9 (attend) (I4 vehicle green)
  10: ..|..|...O: O10 (make-set)
  11: ..|..|...O: O12 (find-candidate) (driving green vehicle)
```

Select an operator based on closest match to previous experiences.

```
  12: ..|..|..|==>S: S7 (operator no-change) ( )
  13: ..|..|..|...O: O14 (count-set)
  14: ..|..|..|...O: O32 (abstract 9)
  15: ..|..|..|...O: O34 (abstract 8)
  16: ..|..|..|...O: O36 (abstract 7)
  17: ..|..|..|...O: O38 (abstract 6)
```

Inductive Learner (SCA2) searching for best match to current state.

```
  18: ..|..|..|...O: O40 (abstract 5)
  19: ..|..|..|...O: O42 (abstract 4)
  20: ..|..|..|...O: O44 (abstract 3)
  21: ..|..|..|...O: O46 (abstract 2)
                Matched on rule simulated-chunk*propose*drive-501*15
  22: ..|..|..|...O: O49 (predict propose drive-50)
  23: ..|..|..|..|==>S: S8 (operator no-change) ( )
```

Found drive-50. Now try to instantiate the operator (fill in parameters).

```
  24: ..|..|..|..|...O: O61 (inst-attend)
  25: ..|..|..|..|...O: O51 (full-inst I5)
Building chunk-1
                Found an instantiation for operator drive-50
  26: ..|..|..|...O: O63 (return-result)
Building chunk-2
  27: ..|..|...O: O65 (activate-candidate drive-50)
```

drive-50 selected as first try for the first plan step.

```
Best overall        : drive-50 uncertainty 8 attending to green vehicle
Best when attend to : green vehicle is drive-50 with uncertainty 8
  28: ..|..|...O: O66 (set-uncertainty-limit)
Set uncertainty limit to 40
  29: ..|..|...O: O67 (evaluate-object I5 (drive-50) )
Uncertainty limit now 32
```

Initial uncertainty is 40. Reduced to 32 by drive-50's uncertainty level of 8.

```
  30: ..|..|..|==>S: S9 (operator no-change) ( )
  31: ..|..|..|...O: L1 (drive-50)
```

Apply drive-50 in simulated world. For this example agent has compiled effects of drive-50 (or this would lead to impasse and search)

```
  32: ..|..|..|..|==>S: S10 (state no-change) ( )
```

Now search for the next step in the plan.

```
  33: ..|..|..|..|...O: O68 (make-candidate1)
  34: ..|..|..|..|...O: O69 (make-candidate)
  35: ..|..|..|..|...O: O70 (attend) (I4 vehicle green)
  36: ..|..|..|..|...O: O71 (make-set)
  37: ..|..|..|..|...O: O73 (find-candidate) (driving green vehicle)

  ...
```

Figure A.1: Trace of planning – part 1

```
  ...

  52: ..|..|..|..|..|..|...O: O124 (return-result)
Building chunk-4
  53: ..|..|..|..|..|...O: O126 (activate-candidate drive-30)
Best overall        : drive-30 uncertainty 6 attending to green vehicle
Best when attend to : green vehicle is drive-30 with uncertainty 8
  54: ..|..|..|..|..|...O: O127 (set-uncertainty-limit)
Set uncertainty limit to 32
  55: ..|..|..|..|..|...O: O128 (evaluate-object I6 (drive-30) )
Uncertainty limit now 26
  56: ..|..|..|..|..|==>S: S14 (operator no-change) ( )
  57: ..|..|..|..|..|..|...O: L2 (drive-30)
  58: ..|..|..|..|..|..|..|==>S: S15 (state no-change) ( )

  ...

  79: ..|..|..|..|..|..|..|...O: O187 (activate-candidate drive-20)
Best overall        : drive-20 uncertainty 7 attending to green vehicle
Best when attend to : green vehicle is drive-20 with uncertainty 8
  80: ..|..|..|..|..|..|..|...O: O188 (set-uncertainty-limit)
Set uncertainty limit to 26
  81: ..|..|..|..|..|..|..|...O: O189 (evaluate-object I7 (drive-20) )
Uncertainty limit now 19
  82: ..|..|..|..|..|..|..|..|==>S: S19 (operator no-change) ( )
  83: ..|..|..|..|..|..|..|..|...O: L3 (drive-20)
  84: ..|..|..|..|..|..|..|..|..|==>S: S20 (state no-change) ( )

  ...

 105: ..|..|..|..|..|..|..|..|..|...O: O248 (activate-candidate drive-10)
Best overall        : drive-10 uncertainty 5 attending to green vehicle
Best when attend to : green vehicle is drive-10 with uncertainty 8
 106: ..|..|..|..|..|..|..|..|..|...O: O249 (set-uncertainty-limit)
Set uncertainty limit to 19
 107: ..|..|..|..|..|..|..|..|..|...O: O250 (evaluate-object I8 (drive-10) )
Uncertainty limit now 14
 108: ..|..|..|..|..|..|..|..|..|..|==>S: S24 (operator no-change) ( )
 109: ..|..|..|..|..|..|..|..|..|..|...O: L4 (drive-10)
 110: ..|..|..|..|..|..|..|..|..|..|...O: O251 (reached-goal)
Building chunk-9
Building chunk-10
 111: ..|..|..|..|..|..|..|..|..|...O: O252 (correction)
Existing correction level for operator drive-10 is 0
Operator drive-10 will have new correction level 1
 112: ..|..|..|..|..|..|..|..|..|...O: O253 (propose-operator)
Building chunk-11
 113: ..|..|..|..|..|..|..|..|...O: I8 (drive-10)
 114: ..|..|..|..|..|..|..|..|...O: O254 (reached-goal)
Building chunk-12
Building chunk-13
 115: ..|..|..|..|..|..|..|...O: O255 (correction)
Existing correction level for operator drive-20 is 0
Operator drive-20 will have new correction level 1

...
```

Considering drive-30 as the next step.

Uncertainty limit counts down as apply operators. If reaches 0 then have reached limit.

Uncertainty limit down to 19 after drive-20.

If below 0 this would lead to backtracking looking for other paths and eventually an increase in the initial limit (40)

After applying drive-10 detect that have reached the goal state for cross-intersection.

Learn chunks, one per operator in the plan. These new rules will guide the agent during execution.

Figure A.2: Trace of planning – part 2

```
...
 115: ..|..|..|..|..|..|...O: O255 (correction)
Existing correction level for operator drive-20 is 0
Operator drive-20 will have new correction level 1
 116: ..|..|..|..|..|..|...O: O256 (propose-operator)
Building chunk-14
 117: ..|..|..|..|..|...O: I7 (drive-20)
 118: ..|..|..|..|..|...O: O257 (drive-10)
 119: ..|..|..|..|..|...O: O258 (reached-goal)
Building chunk-15
Building chunk-16
```

Learning rules for each step in the plan in reverse order from the goal.

```
 120: ..|..|..|..|...O: O259 (correction)
Existing correction level for operator drive-30 is 0
Operator drive-30 will have new correction level 1
 121: ..|..|..|..|...O: O260 (propose-operator)
Building chunk-17
 122: ..|..|..|...O: I6 (drive-30)
 123: ..|..|..|...O: O261 (drive-20)
 124: ..|..|..|...O: O262 (drive-10)
 125: ..|..|..|...O: O263 (reached-goal)
Building chunk-18
Building chunk-19
```

Correction (version) level is used later to let IMPROV change the agent's knowledge without locating the incorrect rule directly (ignore for now).

```
 126: ..|..|...O: O264 (correction)
Existing correction level for operator drive-50 is 0
Operator drive-50 will have new correction level 1
 127: ..|..|...O: O266 (propose-operator)
Building chunk-20
 128: ..|...O: I5 (drive-50)
Building chunk-21
 129: ..|..|==>S: S26 (operator no-change) ( )

 Measurement 1 : Applying drive-50 in the real world
```

Planning is complete and operator drive-50 is selected for execution. In the future, planning will not be required for this task as the rules that guide the agent will still be present.

```
sp {chunk-20
    :chunk
    (state <s1> ^desired <d1> ^problem-space <p1> ^io-state <i1>)
    (<d1> ^direction straight-ahead ^name cross)
    (<p1> ^name driving)
    (<i1> ^object-dynamic <o1> ^car <o1>)
    (<o1> ^distance long ^object-static <o2>)
    (<o2> ^isa vehicle)
    -->
    (<s1> ^operator <o3> +)
    (<o3> ^speed 50 + ^name drive-50 +
          ^class drive + ^leads-to-goal t + ^version 1 +)
}
```

A slightly simplified example of one of the rules used to guide the agent in executing the plan.

Note that the choice of operator is made based on the current goal (<d1>) and state (<i1>).

Figure A.3: Trace of planning – part 3

A.2 Example Trace of SCA2 Inductive Learner

Figure A.4 shows an example of SCA2 predicting an operator from a set of 10 state features. Figures A.5-A.6 show an example of SCA2 training on a new instance and learning a new, more specific prediction rule.

```
 336: ..|..|...O: O618 (find-candidate) (driving green vehicle)
 337: ..|..|..|==>S: S130 (operator no-change) ( )
 338: ..|..|..|...O: O620 (count-set)
 339: ..|..|..|...O: O638 (abstract 9)
 340: ..|..|..|...O: O640 (abstract 8)
 341: ..|..|..|...O: O642 (abstract 7)
 342: ..|..|..|...O: O644 (abstract 6)
 343: ..|..|..|...O: O646 (abstract 5)
 344: ..|..|..|...O: O648 (abstract 4)
 345: ..|..|..|...O: O650 (abstract 3)
 346: ..|..|..|...O: O652 (abstract 2)
              Matched on rule simulated-chunk*propose*drive-101*6
 347: ..|..|..|...O: O655 (predict propose drive-10)
 348: ..|..|..|..|==>S: S131 (operator no-change) ( )

 349: ..|..|..|..|...O: O667 (inst-attend)
 350: ..|..|..|..|...O: O657 (full-inst I9)
Building chunk-27
              Found an instantiation for operator drive-10
 351: ..|..|..|...O: O669 (return-result)
Building chunk-28
 352: ..|..|...O: O671 (activate-candidate drive-10)
```

Count the number of features, in this case 10.

Progressively decrement the counter, looking for a prediction rule to match.

This leads to the specific to general search.

Prediction rule matches suggesting drive-10 as the operator.

Fill in the paramters for the operator (can fail here, in which case the search continues).

Return the operator as SCA2's prediction: drive-10.

Figure A.4: Trace of SCA2 making a prediction

```
544: ..|..|..|..|...O: O1139 (train) (propose drive-20)
545: ..|..|..|..|..|==>S: S244 (operator no-change) ( )
546: ..|..|..|..|..|...O: O1144 (count-set)
547: ..|..|..|..|..|...O: O1162 (abstract 9)
548: ..|..|..|..|..|...O: O1164 (abstract 8)
549: ..|..|..|..|..|...O: O1166 (abstract 7)
550: ..|..|..|..|..|...O: O1168 (abstract 6)
551: ..|..|..|..|..|...O: O1170 (abstract 5)
552: ..|..|..|..|..|...O: O1172 (abstract 4)
553: ..|..|..|..|..|...O: O1174 (abstract 3)
554: ..|..|..|..|..|...O: O1176 (abstract 2)
                Matched on rule simulated-chunk*propose*drive-101*6
555: ..|..|..|..|..|...O: O1179 (predict propose drive-10)
556: ..|..|..|..|..|..|==>S: S245 (operator no-change) ( )

557: ..|..|..|..|..|..|...O: O1191 (inst-attend)
558: ..|..|..|..|..|..|...O: O1181 (full-inst I17)
Building chunk-111
                Found an instantiation for operator drive-10
559: ..|..|..|..|..|...O: O1178 (abstract 1)
560: ..|..|..|..|..|...O: O1199 (predict propose drive-50)
561: ..|..|..|..|..|..|==>S: S246 (operator no-change) ( )

562: ..|..|..|..|..|..|...O: O1211 (inst-attend)
563: ..|..|..|..|..|..|...O: O1201 (full-inst I18)
                Found an instantiation for operator drive-50
Building chunk-112
564: ..|..|..|..|..|...O: O1198 (predict propose drive-30)
565: ..|..|..|..|..|..|==>S: S247 (operator no-change) ( )

566: ..|..|..|..|..|..|...O: O1224 (inst-attend)
567: ..|..|..|..|..|..|...O: O1214 (full-inst I19)
Building chunk-113
                Found an instantiation for operator drive-30
568: ..|..|..|..|..|...O: O1197 (predict propose drive-20)
568: ..|..|..|..|..|...O: O1197 (predict propose drive-20)
569: ..|..|..|..|..|..|==>S: S248 (operator no-change) ( )

570: ..|..|..|..|..|..|...O: O1237 (inst-attend)
571: ..|..|..|..|..|..|...O: O1227 (full-inst I20)
                Found an instantiation for operator drive-20
Building chunk-114
572: ..|..|..|..|..|...O: O1154 (test-if-matched isa vehicle)
Matched isa vehicle
573: ..|..|..|..|..|...O: O1153 (test-if-matched color green)
574: ..|..|..|..|..|...O: O1152 (test-if-matched type car)
575: ..|..|..|..|..|...O: O1151 (test-if-matched distance crossing)
576: ..|..|..|..|..|...O: O1150 (test-if-matched weather raining)
577: ..|..|..|..|..|...O: O1149 (test-if-matched road-quality slick)
578: ..|..|..|..|..|...O: O1148 (test-if-matched sign green)
579: ..|..|..|..|..|...O: O1147 (test-if-matched other-road none)
580: ..|..|..|..|..|...O: O1146 (test-if-matched other-sidewalk none)
581: ..|..|..|..|..|...O: O1145 (test-if-matched other-sign light)
Distance is 0 Speed is 7 and Dspeed is 0 Clock is 44.5
```

First stage in training is searching for an existing prediction for drive-20.

Search from specific to general. finding the most specific prediction rule that matches a subset of the features and is for drive-20.

Finds a prediction rule for drive-10. Doesn't match the target operator so keeps searching.

(This could be more efficient).

Finds a prediction rule for drive-20.

Determine which features are already matched by the existing prediction rule.

Do this by removing features one at a time and seeing if the rule still matches, then putting them back in.

In this version of SCA2, the matched features are calculated first, then one unmatched feature is chosen for learning.

Figure A.5: Trace of SCA2 during training – part 1

```
 582: ..|..|..|..|..|..|==>S: S249 (operator tie) ( )
Found an attribute which is different in all envs: sign so focusing on it
Found an attribute which is different in all envs: other-road so focusing on it
Found an attribute which is different in all envs: other-sidewalk so focusing on it
 583: ..|..|..|..|..|..|..|==>S: S250 (operator tie) ( )
Using the static ordering.
 584: ..|..|..|..|..|..|...O: O1259 (feature-attend) (sign)
Learning about the sign green feature.
 585: ..|..|..|..|..|...O: O1250 (select-feature) (sign green)
 586: ..|..|..|..|..|...O: O1263 (replace 2)
 587: ..|..|..|..|..|..|==>S: S251 (state no-change) ( )
 588: ..|..|..|..|..|..|...O: O1270 (predict propose drive-10)
 589: ..|..|..|..|..|..|..|==>S: S252 (operator no-change) ( )
 590: ..|..|..|..|..|..|..|...O: O1282 (inst-attend)
 591: ..|..|..|..|..|..|..|...O: O1272 (full-inst I21)
Building chunk-115
 592: ..|..|..|..|..|..|...O: O1269 (abstract 1)
 593: ..|..|..|..|..|..|...O: O1290 (predict propose drive-50)
 594: ..|..|..|..|..|..|..|==>S: S253 (operator no-change) ( )
 595: ..|..|..|..|..|..|..|...O: O1302 (inst-attend)
 596: ..|..|..|..|..|..|..|...O: O1292 (full-inst I22)
Building chunk-116
 597: ..|..|..|..|..|..|...O: O1289 (predict propose drive-30)
 598: ..|..|..|..|..|..|..|==>S: S254 (operator no-change) ( )
 599: ..|..|..|..|..|..|..|...O: O1315 (inst-attend)
 600: ..|..|..|..|..|..|..|...O: O1305 (full-inst I23)
Building chunk-117
 601: ..|..|..|..|..|..|...O: O1288 (predict propose drive-20)
 602: ..|..|..|..|..|..|..|==>S: S255 (operator no-change) ( )
 603: ..|..|..|..|..|..|..|...O: O1328 (inst-attend)
 604: ..|..|..|..|..|..|..|...O: O1318 (full-inst I24)
Building chunk-118
 605: ..|..|..|..|..|..|...O: O1330 (return-result)
Building chunk-119
Building chunk-120
 606: ..|..|..|..|...O: O1239 (return-result)
Building chunk-121
```

SCA2 selects the feature to add to the existing prediction rule.

3 features are present in the success and different in all 3 failures. The static order ranks the attributes of the domain in decreasing order of expected importance.

SCA2 increments the counter by 1, subgoals to create a new prediction rule.

In the subgoal, SCA2 searches for the drive-20 prediction again.

Once drive-20 is found, Soar explains why drive-20 was chosen and includes an additional test for the new feature (sign green--which means a green traffic light).

This explanation leads to a new rule: chunk-120.

```
sp {chunk-120
    :chunk
    (state <s1> ^problem-space <p1> ^task-goal <t1> ^object <o1> ^desired <d1>)
    (<p1> ^name recognize)
    (<t1> ^name cross ^direction straight-ahead)
    (<o1> ^count 2 ^features <f1>)
    (<f1> ^feature <f2> ^feature <f3>)
    (<f2> ^value green ^attribute sign  ^object <s1>)
    (<f3> ^value vehicle ^attribute isa ^object <s2>)
    -->
    (<s1> ^operator <o2> + ^operator <o2> =)
    (<o2> ^name predict + ^preference propose + ^operator <o3> + ^id id1686 +)
    (<o3> ^class drive + ^speed 20 + ^name drive-20 +)
}
```

Will only match when count reaches 2.

This is the new condition.

Figure A.6: Trace of SCA2 during training – part 2

A.3 Example Trace of IMPROV's Precondition Learning

Figure A.7 shows the initial detection of an execution failure and learning a rule to recall the error in future. Figure A.8 shows how IMPROV learns to avoid using the same plan when it next tries to cross this intersection. Figures A.9-A.15 show the behavior of the system on a later trial. In between IMPROV has tried a number of other plans, each of which has failed. However, on this attempt, IMPROV finally discovers a plan that succeeds and therefore proceeds to correct the agent's knowledge. The final figure (A.15) includes two of the rules that are learned, indicating that when there is a red light, the car should stop, rather than driving on through the intersection. This is a simple example, but serves to demonstrate the main elements of IMPROV's learning method.

```
Sending command tick-simulation
Distance is -21 Speed is 5 and Dspeed is 0 Clock is 22.5

 Measurement 4 : Error detected for operator drive-10
Building chunk-22
Building chunk-23
Building chunk-24
 329: ..|...O: O611 (error-detected)

sp {chunk-22
    :chunk
    (state <s1> ^operator <o1>)
    (<o1> ^name drive-10 ^version 1)
    -->
    (<o1> ^error <w1> +)
    (<w1> ^error-code constant334 +)
}
```

Once an error is detected, learn rules including the unique error-code which defines this error instance.

Figure A.7: Recording initial error

```
 Measurement 4 : Error detected for operator drive-10
Building chunk-22
Building chunk-23
Building chunk-24
```

Learn to avoid the final step in the plan, drive-10 in this case.

```
 329: ..|...O: O611 (error-detected)
 330: ..|...O: O610 (recall-previous)
 331: ..|..|==>S: S128 (operator no-change) ( )
```

Recall the state before the final operator was applied. (The last state is kept in memory).

```
Building chunk-25
Building chunk-26
 332: ..|..|...O: O612 (make-candidate1)    (type no-attend)
 333: ..|..|...O: O613 (make-candidate)
 334: ..|..|...O: O615 (attend) (I4 vehicle green)
 335: ..|..|...O: O616 (make-set)
 336: ..|..|...O: O618 (find-candidate) (driving green vehicle)
 337: ..|..|..|==>S: S130 (operator no-change) ( )
 ...

 351: ..|..|..|...O: O669 (return-result)
Building chunk-28
 352: ..|..|...O: O671 (activate-candidate drive-10)
Best overall        : drive-10 uncertainty 0 attending to green vehicle
Best when attend to : green vehicle is drive-10 with uncertainty 0
```

Use SCA2 to generate the same operator, based on SCA2's knowledge.

```
 353: ..|..|...O: O674 (train) (reject drive-10)
 354: ..|..|..|==>S: S132 (operator no-change) ( )
 ...
```

Now learn to reject (i.e. not select) this operator during future searches for alternative plans when return to a similar state.

```
 379: ..|..|..|..|==>S: S134 (operator tie) ( )
 380: ..|..|..|..|..|==>S: S135 (operator tie) ( )
Using the static ordering.
 381: ..|..|..|..|...O: O728 (feature-attend) (color)
Learning about the color green feature.
 ...

 391: ..|..|..|...O: O717 (return-result)
Building chunk-34
Building chunk-35
Building chunk-36
 392: ..|...O: O759 (attempt-correction)
Starting a new attempt at a correction.
```

Now select the feature to focus learning on. IMPROV is forced to guess, choosing the color of the agent's car (we gave it a poor set of default knowledge to demonstrate the learning).

```
sp {chunk-33
   :chunk
   (state <s1> ^error-code constant334 ^problem-space <p1>
        ^object <o1> ^task-goal <t1> ^desired <d1>)
   (<p1> ^name recognize)
   (<t1> ^name cross ^direction straight-ahead)
   (<o1> ^count 3 ^features <f1>)
   (<f1> ^feature <f2> ^feature <f3> ^feature <f4>)
   (<f2> ^value green ^attribute color ^object <s2>)
   (<f3> ^value crossing ^attribute dist ^object <s2>)
   (<f4> ^value vehicle ^attribute isa ^object <s2>)
   -->
   (<s1> ^operator <o2> + ^operator <o2> =)
   (<o2> ^name predict ^preference reject ^operator <o3> ^id id406)
   (<o3> ^class drive ^speed 10 ^name drive-10)
}
```

Example of a prediction rule, testing 3 state features and suggesting that drive-10 should not be chosen.

Note : Tests ^error-code constant334 so only chosen while processing this error.

Once the error has been corrected, this rule will not match.

Figure A.8: Learning to avoid last operator in plan

Recall the error and start looking for a correction.

```
128: ..|...O: O216 (error-detected)
129: ..|...O: O217 (attempt-correction)
Starting a new attempt at a correction.
130: ..|..|==>S: S67 (operator conflict) ( )
131: ..|..|...O: O220 (make-candidate1)    (type no-attend)
132: ..|..|...O: O221 (make-candidate)
133: ..|..|...O: O222 (attend) (I4 vehicle green)
134: ..|..|...O: O223 (make-set)
135: ..|..|...O: O225 (find-candidate) (driving green vehicle)
```

Choose an operator as the first step in the plan to avoid the failure.

These chunks recall all of the previous attempts made in a similar state.

```
Firing chunk-29
Firing chunk-45
Firing chunk-62
Firing chunk-72
Firing chunk-53
Firing chunk-35
136: ..|..|..|==>S: S72 (operator no-change) ( )
137: ..|..|..|...O: O230 (count-set)
```

SCA2 search from specific to general looking for a prediction that's not already been tried.

```
138: ..|..|..|...O: O248 (abstract 9)
139: ..|..|..|...O: O250 (abstract 8)
140: ..|..|..|...O: O252 (abstract 7)
141: ..|..|..|...O: O254 (abstract 6)
142: ..|..|..|...O: O256 (abstract 5)
143: ..|..|..|...O: O258 (abstract 4)
144: ..|..|..|...O: O260 (abstract 3)
```

Learned to avoid drive-10 before.

```
Firing chunk-33
145: ..|..|..|...O: O263 (predict reject drive-10)
146: ..|..|..|..|==>S: S73 (operator no-change) ( )
147: ..|..|..|..|...O: O274 (inst-attend)
148: ..|..|..|..|...O: O275 (full-inst all)
               Found an instantiation for operator drive-10
Firing chunk-73
149: ..|..|..|...O: O262 (abstract 2)
Firing chunk-51
Firing chunk-70
               Matched on rule simulated-chunk*propose*drive-101*6
150: ..|..|..|...O: O280 (predict reject drive-30)
```

And to avoid drive-30

```
151: ..|..|..|..|==>S: S74 (operator no-change) ( )
152: ..|..|..|..|...O: O292 (inst-attend)
153: ..|..|..|..|...O: O293 (full-inst all)
Building chunk-74
               Found an instantiation for operator drive-30
154: ..|..|..|...O: O279 (predict reject drive-50)
```

And already tried drive-50 too.

```
155: ..|..|..|..|==>S: S75 (operator no-change) ( )

156: ..|..|..|..|...O: O305 (inst-attend)
157: ..|..|..|..|...O: O306 (full-inst all)
               Found an instantiation for operator drive-50
Building chunk-75
```

Already marked this for rejection, so drive-10 not chosen.

```
158: ..|..|..|...O: O281 (predict propose drive-10)
159: ..|..|..|..|==>S: S76 (operator no-change) ( )
160: ..|..|..|..|...O: O319 (inst-attend)
161: ..|..|..|..|...O: O309 (full-inst I5)
Building chunk-76
               Found an instantiation for operator drive-10
162: ..|..|..|...O: O278 (abstract 1)
```

Figure A.9: Finding correct behavior and generalizing it – part 1

```
 163: ..|..|..|...O: O327 (predict propose drive-50)
Distance is 16 Speed is 8 and Dspeed is 0 Clock is 12.5
 164: ..|..|..|..|==>S: S77 (operator no-change) ( )
 165: ..|..|..|..|...O: O339 (inst-attend)
 166: ..|..|..|..|...O: O329 (full-inst I6)
                Found an instantiation for operator drive-50
```

This message shows the external environment has just updated--shows that environment is asynchronous with reasoning. Makes reasoning time limited.

```
 167: ..|..|..|...O: O326 (predict propose drive-30)
 168: ..|..|..|..|==>S: S78 (operator no-change) ( )
 169: ..|..|..|..|...O: O352 (inst-attend)
 170: ..|..|..|..|...O: O342 (full-inst I7)
                Found an instantiation for operator drive-30
```

These operators have alreadt been tried so they're not chosen.

```
 171: ..|..|..|...O: O323 (predict propose drive-00)
 172: ..|..|..|..|==>S: S79 (operator no-change) ( )
 173: ..|..|..|..|...O: O364 (inst-attend)
 174: ..|..|..|..|...O: O365 (full-inst I8)
                Found an instantiation for operator drive-00
Building chunk-79
 175: ..|..|..|...O: O367 (return-result)
Building chunk-80
 176: ..|..|...O: O369 (activate-candidate drive-00)
Best overall       : drive-00 uncertainty 14 attending to green vehicle
Best when attend to : green vehicle is drive-00 with uncertainty 14
 177: ..|..|...O: O370 (set-uncertainty-limit)
Set uncertainty limit to 60
```

Finds drive-00, which hasn't already been tried, so it's returned as SCA2's prediction.

```
 178: ..|..|...O: O371 (evaluate-object I8 (drive-00) )
 179: ..|..|..|==>S: S80 (operator no-change) ( )
 180: ..|..|..|...O: L1 (drive-00)
 181: ..|..|..|..|==>S: S81 (operator conflict) ( )
 182: ..|..|..|..|...O: O375 (make-candidate1)    (type no-attend)
 183: ..|..|..|..|...O: O376 (make-candidate)
 184: ..|..|..|..|...O: O377 (attend) (I4 vehicle green)
 185: ..|..|..|..|...O: O378 (make-set)
 186: ..|..|..|..|...O: O380 (find-candidate) (driving green vehicle)

...

 226: ..|..|..|..|..|...O: O522 (return-result)
Building chunk-88
 227: ..|..|..|..|...O: O524 (activate-candidate drive-20)
Best overall       : drive-20 uncertainty 14 attending to green vehicle
Best when attend to : green vehicle is drive-20 with uncertainty 14
 228: ..|..|..|..|...O: O525 (set-uncertainty-limit)
Set uncertainty limit to 46
 229: ..|..|..|..|...O: O526 (evaluate-object I12 (drive-20) )
```

Now see if we can complete a plan using drive-00 as the first step.

```
 230: ..|..|..|..|..|==>S: S94 (operator no-change) ( )
 Measurement 2 : Applying eval-obj during planning
 231: ..|..|..|..|..|...O: L2 (drive-20)
 232: ..|..|..|..|..|...O: O529 (reached-goal)
Building chunk-89
Building chunk-90
 233: ..|..|..|..|...O: O530 (correction)
Existing correction level for operator drive-20 is 0
Operator drive-20 will have new correction level 1
 234: ..|..|..|..|...O: O531 (propose-operator)
Building chunk-91
Building chunk-92
```

Following drive-00 with drive-20 reaches the goal according to current planning knowledge. So we'll try this plan.

Figure A.10: Finding correct behavior and generalizing it – part 2

```
 235: ..|..|..|...O: I12 (drive-20)
 236: ..|..|..|...O: O532 (reached-goal)
Building chunk-93
Building chunk-94
```

Record the current state before executing drive-00 in the environment.

```
 237: ..|..|...O: O534 (observe) (drive-00)
 238: ..|..|..|==>S: S97 (operator no-change) ( )
Firing chunk-11
```

Execute drive-00 (takes many steps and sub-operators)

```
 239: ..|..|..|...O: O82 (drive-00)
 240: ..|..|..|..|==>S: S98 (operator no-change) ( )
 Measurement 1 : Applying drive-00 in the real world
 241: ..|..|..|..|...O: O539 (comprehend)
 242: ..|..|..|..|..|==>S: S99 (operator no-change) ( )
Distance is 8 Speed is 8 and Dspeed is 0 Clock is 13.5
 244: ..|..|..|..|...O: O540 (brake)
Sending command brake
 245: ..|..|..|..|..|==>S: S101 (operator no-change) ( )
 246: ..|..|..|..|..|...O: O548 (moving)
Sending command tick-simulation
Distance is 6 Speed is 4 and Dspeed is -4 Clock is 14.0
 247: ..|..|..|..|...O: O547 (comprehend)
 248: ..|..|..|..|..|==>S: S102 (operator no-change) ( )
 250: ..|..|..|..|...O: O550 (moving)
Sending command tick-simulation
Distance is 6 Speed is 0 and Dspeed is -4 Clock is 14.5
 251: ..|..|..|..|...O: O556 (comprehend)
 252: ..|..|..|..|..|==>S: S104 (operator no-change) ( )
 254: ..|..|..|..|...O: O557 (release-pedal)
Sending command release-pedal
 255: ..|..|..|..|..|==>S: S106 (operator no-change) ( )
 256: ..|..|..|..|..|...O: O566 (moving)
Sending command tick-simulation
Distance is 6 Speed is 0 and Dspeed is 0 Clock is 15.0

...

 356: ..|..|..|..|...O: O736 (moving)
Sending command tick-simulation
 *** Event light is now green ***
Distance is 6 Speed is 0 and Dspeed is 0 Clock is 27.5
```

Now decide on the step after drive-00...

```
 357: ..|..|..|..|==>S: S157 (operator conflict) ( )
 358: ..|..|..|..|...O: O744 (make-candidate1)    (type no-attend)
 359: ..|..|..|..|...O: O745 (make-candidate)
 360: ..|..|..|..|...O: O746 (attend) (I4 vehicle green)
 361: ..|..|..|..|...O: O747 (make-set)
 362: ..|..|..|..|...O: O749 (find-candidate) (driving green vehicle)

...

 398: ..|..|..|..|..|...O: O849 (predict propose drive-20)
 399: ..|..|..|..|..|..|==>S: S169 (operator no-change) ( )

 400: ..|..|..|..|..|..|...O: O889 (inst-attend)
 401: ..|..|..|..|..|..|...O: O879 (full-inst I16)
               Found an instantiation for operator drive-20
Building chunk-107
 402: ..|..|..|..|..|...O: O891 (return-result)
Building chunk-108
```

Again, select drive-20 to follow drive-00. In general, this may no longer be appropriate--it depends on what actually happened in executing drive-00.

Figure A.11: Finding correct behavior and generalizing it – part 3

```
 403: ..|..|..|..|...O: O893 (activate-candidate drive-20)
Best overall       : drive-20 uncertainty 14 attending to green vehicle
Best when attend to : green vehicle is drive-20 with uncertainty 14
 404: ..|..|..|..|...O: O894 (set-uncertainty-limit)
Set uncertainty limit to 60
 405: ..|..|..|..|...O: O895 (evaluate-object I16 (drive-20))
Firing chunk-90
 406: ..|..|..|..|...O: O897 (observe) (drive-20)
 407: ..|..|..|..|..|==>S: S172 (operator no-change) ( )
Firing chunk-92
 408: ..|..|..|..|..|...O: C146 (drive-20)
 409: ..|..|..|..|..|..|==>S: S173 (operator no-change) ( )
 Measurement 1 : Applying drive-20 in the real world
 410: ..|..|..|..|..|..|...O: O902 (comprehend)
 411: ..|..|..|..|..|..|..|==>S: S174 (operator no-change) ( )
 413: ..|..|..|..|..|..|...O: O903 (accelerate)
Sending command accelerate
 414: ..|..|..|..|..|..|..|==>S: S176 (operator no-change) ( )
 415: ..|..|..|..|..|..|..|...O: O911 (moving)
Sending command tick-simulation
Distance is 6 Speed is 1 and Dspeed is 0 Clock is 28.5
 416: ..|..|..|..|..|..|...O: O910 (comprehend)
 417: ..|..|..|..|..|..|..|==>S: S177 (operator no-change) ( )
 419: ..|..|..|..|..|..|...O: O913 (moving)
Sending command tick-simulation
Distance is 6 Speed is 2 and Dspeed is 1 Clock is 29.0
 420: ..|..|..|..|..|..|...O: O919 (comprehend)
 421: ..|..|..|..|..|..|..|==>S: S179 (operator no-change) ( )
 423: ..|..|..|..|..|..|...O: O920 (moving)
Sending command tick-simulation
Distance is 4 Speed is 3 and Dspeed is 1 Clock is 29.5
 424: ..|..|..|..|..|..|...O: O926 (comprehend)
 425: ..|..|..|..|..|..|..|==>S: S181 (operator no-change) ( )
 427: ..|..|..|..|..|..|...O: O927 (moving)
Sending command tick-simulation
Distance is 3 Speed is 4 and Dspeed is 2 Clock is 30.0
 428: ..|..|..|..|..|..|...O: O933 (comprehend)
 429: ..|..|..|..|..|..|..|==>S: S183 (operator no-change) ( )
 431: ..|..|..|..|..|..|...O: O934 (moving)
Sending command tick-simulation
Distance is 0 Speed is 6 and Dspeed is 2 Clock is 30.5
 432: ..|..|..|..|..|..|...O: O940 (comprehend)
 433: ..|..|..|..|..|..|..|==>S: S185 (operator no-change) ( )
 435: ..|..|..|..|..|..|...O: O941 (moving)
Sending command tick-simulation
Distance is -2 Speed is 7 and Dspeed is 2 Clock is 31.0
 436: ..|..|..|..|..|..|...O: O947 (comprehend)
 437: ..|..|..|..|..|..|..|==>S: S187 (operator no-change) ( )
 439: ..|..|..|..|..|..|...O: O948 (release-pedal)
Sending command release-pedal
 440: ..|..|..|..|..|..|..|==>S: S189 (operator no-change) ( )
 441: ..|..|..|..|..|..|..|...O: O955 (moving)
Sending command tick-simulation
Distance is -6 Speed is 7 and Dspeed is 0 Clock is 31.5

...
```

Decide if drive-20 will lead to the goal, using planning knowlegde. We can use a compiled chunk here (chunk-90) because we're still working on the same error. In general (as happened above), have to go through the planning.

Execute drive-20, again it takes a while to complete...

Figure A.12: Finding correct behavior and generalizing it – part 4

```
 541: ..|..|..|..|..|..|...O: O1125 (moving)
Sending command tick-simulation
Distance is 0 Speed is 7 and Dspeed is 0 Clock is 44.0
 542: ..|..|..|..|..|..|==>S: S240 (operator conflict) ( )
 543: ..|..|..|..|..|..|...O: O1138 (terminate-observation)
Building chunk-109
Building chunk-110
 That worked.
  Observation of I16 (drive-20) is partial-success
 544: ..|..|..|..|...O: O1139 (train) (propose drive-20)
```

Reached the goal--successfully crossed the intersection. Now it's time to learn.

Train on the successful drive-20 operator.

```
...

 582: ..|..|..|..|..|..|==>S: S249 (operator tie) ( )
Found an attribute which is different in all envs: sign so focusing on it
Found an attribute which is different in all envs: other-road so focusing on it
Found an attribute which is different in all envs: other-sidewalk so focusing on it
 583: ..|..|..|..|..|..|..|==>S: S250 (operator tie) ( )
Using the static ordering.
 584: ..|..|..|..|..|..|...O: O1259 (feature-attend) (sign)
Learning about the sign green feature.
Learning about the sign green feature.
 585: ..|..|..|..|..|...O: O1250 (select-feature) (sign green)
 I think that worked because of the sign.
```

Decide that the reason it succeeded was because the light changed. Naturally, this could be wrong, but in this case it's the true cause.

```
...

 605: ..|..|..|..|..|..|...O: O1330 (return-result)
Building chunk-119
Building chunk-120
 606: ..|..|..|..|..|...O: O1239 (return-result)
Building chunk-121
Building chunk-122
Building chunk-123
Building chunk-124
```

Build a new SCA2 prediction rule.

```
 607: ..|..|..|..|...O: O1332 (learn-correction)
 608: ..|..|..|..|..|==>S: S257 (operator no-change) ( )
Firing chunk-92
 609: ..|..|..|..|..|..|==>S: S258 (operator conflict) ( )
 610: ..|..|..|..|..|..|...O: O1336 (make-candidate1)    (type no-attend)
 611: ..|..|..|..|..|..|...O: O1337 (make-candidate)
 612: ..|..|..|..|..|..|...O: O1338 (attend) (I4 vehicle green)
 613: ..|..|..|..|..|..|...O: O1339 (make-set)
 614: ..|..|..|..|..|..|...O: O1341 (find-candidate) (driving green vehicle)
```

Now return to the state before drive-20 was applied and learn a new proposal rule based on the new SCA2 prediction rule.

```
...

 634: ..|..|..|..|..|..|...O: O1411 (activate-candidate drive-20)
Best overall        : drive-20 uncertainty 13 attending to green vehicle
Best when attend to : green vehicle is drive-20 with uncertainty 13
 635: ..|..|..|..|..|..|...O: O1412 (set-uncertainty-limit)
Set uncertainty limit to 60
 636: ..|..|..|..|..|..|...O: O1413 (evaluate-object I26 (drive-20) )
Firing chunk-90
```

Go back to SCA2 and generate the drive-20 prediction. This is needed so this chunks right.

Figure A.13: Finding correct behavior and generalizing it – part 5

```
 637: ..|..|..|..|..|..|...O: O1414 (correction)
Existing correction level for operator drive-20 is 1
Operator drive-20 will have new correction level 2
 638: ..|..|..|..|..|..|...O: O1416 (propose-operator)
Building chunk-128
Building chunk-129
Building chunk-130
Building chunk-131
```

Learn the new operator precondition rule.

```
 639: ..|..|..|..|...O: O1331 (terminate-observation)
Building chunk-132
Building chunk-133
 That worked.
  Observation of I8 (drive-00) is partial-success
 640: ..|..|...O: O1417 (train) (propose drive-00)
 Measurement 3 : Training to propose drive-00 operator
```

Now repeat the same steps for drive-00.

```
...

 678: ..|..|..|..|==>S: S275 (operator tie) ( )
Found an attribute which is different in poss cause: sign so focusing on it
 679: ..|..|..|..|...O: O1537 (feature-attend) (sign)
Learning about the sign red feature.
Learning about the sign red feature.
 680: ..|..|...O: O1528 (select-feature) (sign red)
 I think that worked because of the sign.
```

Decide to add the light as the feature to add to the operator preconditions.

```
...

Building chunk-142
Building chunk-143
 701: ..|..|..|...O: O1517 (return-result)
Distance is 0 Speed is 7 and Dspeed is 0 Clock is 46.0
Building chunk-144
 702: ..|..|...O: O1610 (learn-correction)
 703: ..|..|..|==>S: S282 (operator no-change) ( )
Firing chunk-11
```

Reconstruct the state before drive-00 was chosen and learn a new operator precondition rule.

```
 704: ..|..|..|..|==>S: S283 (operator conflict) ( )
 705: ..|..|..|..|...O: O1614 (make-candidate1)     (type no-attend)
 706: ..|..|..|..|...O: O1615 (make-candidate)
 707: ..|..|..|..|...O: O1616 (attend) (I4 vehicle green)
 708: ..|..|..|..|...O: O1617 (make-set)
 709: ..|..|..|..|...O: O1619 (find-candidate) (driving green vehicle)

...

 724: ..|..|..|..|..|...O: O1674 (return-result)
Building chunk-146
 725: ..|..|..|..|...O: O1676 (activate-candidate drive-00)
Best overall       : drive-00 uncertainty 13 attending to green vehicle
Best when attend to : green vehicle is drive-00 with uncertainty 13
 726: ..|..|..|..|...O: O1677 (set-uncertainty-limit)
Set uncertainty limit to 60
 727: ..|..|..|..|...O: O1678 (evaluate-object I35 (drive-00) )
Firing chunk-94
```

Figure A.14: Finding correct behavior and generalizing it – part 6

```
 728: ..|..|..|..|...O: O1679 (correction)
Existing correction level for operator drive-00 is 0
Operator drive-00 will have new correction level 1
 729: ..|..|..|..|...O: O1681 (propose-operator)
Building chunk-147
Building chunk-148
Building chunk-149
Building chunk-150
 730: ..|..|...O: O1609 (terminate-observation)
Building chunk-151
 731: ..|...O: O1133 (reached-goal)
****************************************
 Measurement 5 : Task completed successfully
****************************************
```

Learn the new precondition rule for drive-00 and learn not to select drive-10 in this situation.

```
sp {chunk-149
    :chunk
    (state <s1> ^problem-space <p1> ^desired <d1> ^operator <o1> +
         ^io-state <i1>)
    (<p1> ^name driving)
    (<d1> ^direction straight-ahead ^name cross)
    (<o1> ^name drive-10 ^version 1)
    (<i1> ^object-dynamic <o2> ^car <o2>)
    (<o2> ^distance crossing ^sign red ^object-static <o3>)
    (<o3> ^isa vehicle)
    -->
    (<s1> ^operator <o1> -)
}
```

The rule indicating that drive-10 should not be chosen when about to cross an intersection and the light is red.

```
sp {chunk-150
    :chunk
    (state <s1> ^problem-space <p1> ^desired <d1> ^io-state <i1>)
    (<p1> ^name driving)
    (<d1> ^direction straight-ahead ^name cross)
    (<i1> ^object-dynamic <o1> ^car <o1>)
    (<o1> ^distance crossing ^sign red ^object-static <o2>)
    (<o2> ^isa vehicle)
    -->
    (<s1> ^operator <o3> +)
    (<o3> ^speed 0 +     ^name drive-00 +
         ^class drive + ^leads-to-goal t + ^version 1 +)
}
```

The rule indicating that drive-00 should be chosen in the same situation.

Figure A.15: Finding correct behavior and generalizing it – part 7

A.4 Example Trace of IMPROV Learning Operator Effects

Figures A.16-A.20 show a trace of an initial attempt to brake in time for an intersection. This trace shows the detection of the original performance failure and the subsequent recording of the single-effects operators that should be avoided in future and the values to initially try in their place. Then Figures A.21-A.24 show a trace of a second run. During this attempt to brake, IMPROV selects an alternative sequence of single-effect operators to model the effects of braking, leading the agent to brake at the correct time. The agent then learns to use this new model in future. Figure A.25 shows the final behavior of the system on a third run, using the now correct knowledge.

```
Setup for this run:
   Soar car is a green car initial speed 32 mph.
   The weather is sunny.
   The road quality is excellent
   The sign is a stop light (traffic light).
   Change cycle is  { 0.0 4.99 yellow } { 5.0 24.99 red } { 25.0 59.99 green }
   The other road user is a car.
   The sidewalk user is a pedestrian.
   The other sign is a stop sign.
Soar started at Mon Oct  7 12:08:00 EDT 1996.
soar> run
run

   0: ==>S: S1  ( )
  Simulation updating.
Distance is 173 Speed is 14 and Dspeed is 0 Clock is 0.5
Finished.
   1: ...O: O1 (drive)
   2: ..|==>S: S2 (operator no-change) ( )

   3: ..|...O: O2 (maintain)

...

  19: ..|==>S: S3 (state no-change) ( )

  20: ..|...O: O21 (make-candidate1)    (type no-attend)
  21: ..|...O: O22 (make-candidate)
  22: ..|...O: O23 (attend) (I4 vehicle green)
  23: ..|...O: O24 (make-set)
  24: ..|...O: O26 (find-candidate) (top-ps green vehicle)
  25: ..|...O: O29 (activate-candidate stop-in-time)
  26: ..|...O: O30 (set-uncertainty-limit)
  27: ..|...O: O31 (evaluate-object O27 (stop-in-time) )
  28: ..|..|==>S: S5 (operator no-change) ( )
  29: ..|..|...O: L1 (stop-in-time)
  30: ..|..|..|==>S: S6 (operator no-change) ( )
  31: ..|..|..|..|==>S: S7 (state no-change) ( )
  32: ..|..|..|..|...O: O32 (make-candidate1)    (type no-attend)
  33: ..|..|..|..|...O: O33 (make-candidate)
  34: ..|..|..|..|...O: O34 (attend) (I4 vehicle green)
  35: ..|..|..|..|...O: O35 (make-set)
  36: ..|..|..|..|...O: O37 (find-candidate) (driving green vehicle)
  37: ..|..|..|..|...O: O41 (activate-candidate stop)
  38: ..|..|..|..|...O: O42 (set-uncertainty-limit)
  39: ..|..|..|..|...O: O43 (evaluate-object O38 (stop) )
  40: ..|..|..|..|..|==>S: S9 (operator no-change) ( )
  41: ..|..|..|..|..|...O: L2 (stop)
  42: ..|..|..|..|..|..|==>S: S10 (operator no-change) ( )
  43: ..|..|..|..|..|..|..|==>S: S11 (state no-change) ( )
  51: ..|..|..|..|..|..|..|...O: O55 (evaluate-object O50 (brake) )
  52: ..|..|..|..|..|..|..|..|==>S: S13 (operator no-change) ( )
  53: ..|..|..|..|..|..|..|..|...O: L3 (brake)
  54: ..|..|..|..|..|..|..|..|..|==>S: S14 (operator no-change) ( )
  55: ..|..|..|..|..|..|..|..|..|..|==>S: S15 (state no-change) ( )
  85: ..|..|..|..|..|..|..|..|..|..|...O: O516 (evaluate-object O493 (dspeed-3) )
  86: ..|..|..|..|..|..|..|..|..|..|..|==>S: S19 (operator no-change) ( )
  87: ..|..|..|..|..|..|..|..|..|..|..|...O: L4 (dspeed-3)
  88: ..|..|..|..|..|..|..|..|..|..|..|..|==>S: S20 (state no-change) ( )
 118: ..|..|..|..|..|..|..|..|..|..|..|..|...O: O977 (evaluate-object O954 (brake-pressure5) )
 119: ..|..|..|..|..|..|..|..|..|..|..|..|..|==>S: S24 (operator no-change) ( )
 120: ..|..|..|..|..|..|..|..|..|..|..|..|..|...O: L5 (brake-pressure5)
 121: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|==>S: S25 (state no-change) ( )
 152: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O1461 (evaluate-object O1438 (tick) )
 153: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|==>S: S29 (operator no-change) ( )
 154: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: L6 (tick)
Internal SIM had distance 61 speed 14 time 8.5
Internal SIM now has distance 55.5 speed 11 time 9.0
 155: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|==>S: S30 (state no-change) ( )
```

Trace of IMPROV learning the correct effects of a braking. Please note that these traces have been extensively edited to reduce the total length (which was excessively long and detailed). Watch the numbers at the left (the decision cycle counter) to see where sections have been omitted.

Initial planning to stop the car at an intersection.

Problem is decomposed down to the brake operator and then dspeed and brake-pressure operators to model the effects of braking.

Figure A.16: Correcting effects, first run – part 1

```
...

 702: ..|..|..|..|..|..|..|..|..|..|..|...O: O7219 (test-goal)
Building chunk-59
 703: ..|..|..|..|..|..|..|..|..|..|...O: O7220 (correction)
 704: ..|..|..|..|..|..|..|..|..|..|...O: O7221 (propose-operator)
Building chunk-60
 705: ..|..|..|..|..|..|..|..|..|...O: O493 (dspeed-3)
 706: ..|..|..|..|..|..|..|..|..|...O: O7222 (brake-pressure5)
 707: ..|..|..|..|..|..|..|..|..|...O: O7223 (tick)
 708: ..|..|..|..|..|..|..|..|..|...O: O7224 (dspeed-3)
 709: ..|..|..|..|..|..|..|..|..|...O: O7225 (brake-pressure5)
```

Planning takes many steps and finally terminates with a "plan" or model of the effects of implementing the brake operator.

```
 710: ..|..|..|..|..|..|..|..|..|...O: O7226 (tick)
 711: ..|..|..|..|..|..|..|..|..|...O: O7227 (dspeed-3)
 712: ..|..|..|..|..|..|..|..|..|...O: O7228 (brake-pressure5)
 713: ..|..|..|..|..|..|..|..|..|...O: O7229 (tick)
 714: ..|..|..|..|..|..|..|..|..|...O: O7230 (dspeed-3)
 715: ..|..|..|..|..|..|..|..|..|...O: O7231 (brake-pressure5)
 716: ..|..|..|..|..|..|..|..|..|...O: O7232 (tick)
 717: ..|..|..|..|..|..|..|..|..|...O: O7233 (dspeed-3)
 718: ..|..|..|..|..|..|..|..|..|...O: O7234 (brake-pressure5)
 719: ..|..|..|..|..|..|..|..|..|...O: O7235 (tick)
We've figured out how to brake correctly.
Building chunk-76
 720: ..|..|..|..|..|..|..|...O: O7239 (correction)
 721: ..|..|..|..|..|..|..|...O: O7240 (propose-operator)
 722: ..|..|..|..|..|..|...O: O50 (brake)
Internal SIM had distance 61 speed 14 time 8.5
```

In the initial plan, the agent brakes too soon, stopping before the intersection. This plan is therefore rejected and the agent searches for an alternative.

This is all planning, no actions have yet been taken in the outside world.

```
Internal SIM now has distance 55.5 speed 11 time 9.0
Internal SIM now has distance 51.0 speed 8 time 9.5
Internal SIM now has distance 48.5 speed 5 time 10.0
Internal SIM now has distance 47.0 speed 2 time 10.5
Internal SIM now has distance 47.0 speed 0.0 time 11.0
That plan would stop me short of the maximum distance at 45
Building chunk-94
 723: ..|..|..|..|...O: O7243 (make-candidate1)    (type no-best)
 724: ..|..|..|..|...O: O40 (make-candidate)
 725: ..|..|..|..|...O: O7245 (find-candidate) (driving green vehicle)
 726: ..|..|..|..|...O: O7248 (activate-candidate maintain)
 727: ..|..|..|..|...O: O7249 (evaluate-object O7246 (maintain) )
 728: ..|..|..|..|..|==>S: S90 (operator no-change) ( )
```

The agent considers coasting for a short distance before braking.

```
 729: ..|..|..|..|..|...O: L19 (maintain)
 730: ..|..|..|..|..|..|==>S: S91 (operator no-change) ( )
 731: ..|..|..|..|..|..|...O: O7250 (coast)
 732: ..|..|..|..|..|..|==>S: S92 (state no-change) ( )
 733: ..|..|..|..|..|..|...O: O7251 (make-candidate1)    (type no-attend)
 734: ..|..|..|..|..|..|...O: O7252 (make-candidate)
 735: ..|..|..|..|..|..|...O: O7253 (attend) (I4 vehicle green)
 736: ..|..|..|..|..|..|...O: O7254 (make-set)
 737: ..|..|..|..|..|..|...O: O7256 (find-candidate) (driving green vehicle)
 738: ..|..|..|..|..|..|...O: O7260 (activate-candidate stop)
 739: ..|..|..|..|..|..|...O: O7261 (set-uncertainty-limit)
 740: ..|..|..|..|..|..|...O: O7262 (evaluate-object O7257 (stop) )
 741: ..|..|..|..|..|..|..|==>S: S94 (operator no-change) ( )
 742: ..|..|..|..|..|..|..|...O: L20 (stop)
 743: ..|..|..|..|..|..|..|..|==>S: S95 (operator no-change) ( )
 744: ..|..|..|..|..|..|..|..|...O: O7263 (brake)
```

Note that the effects of "brake" have now been compiled into the brake operator, making planning much more efficient (as there is no need to decompose the operator into the primitive dspeed and brake-pressure single-effect operators).

```
That plan would stop me short of the maximum distance at 40
Building chunk-112
 745: ..|..|..|..|..|..|...O: O7266 (make-candidate1)    (type no-best)
 746: ..|..|..|..|..|..|...O: O7259 (make-candidate)
 747: ..|..|..|..|..|..|...O: O7268 (find-candidate) (driving green vehicle)
 748: ..|..|..|..|..|..|...O: O7271 (activate-candidate maintain)
 749: ..|..|..|..|..|..|...O: O7272 (evaluate-object O7269 (maintain) )
 750: ..|..|..|..|..|..|..|==>S: S96 (operator no-change) ( )
 751: ..|..|..|..|..|..|..|...O: L21 (maintain)
 752: ..|..|..|..|..|..|..|..|==>S: S97 (operator no-change) ( )
 753: ..|..|..|..|..|..|..|..|...O: O7273 (coast)
 754: ..|..|..|..|..|..|..|..|==>S: S98 (state no-change) ( )
```

Figure A.17: Correcting effects, first run – part 2

```
...

 810: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O7332 (brake)
That looks like a plan which should work!
Building chunk-166
 811: ..|..|..|..|..|..|..|..|..|..|..|..|...O: O7335 (correction)
 812: ..|..|..|..|..|..|..|..|..|..|..|..|...O: O7336 (propose-operator)
Building chunk-167
 813: ..|..|..|..|..|..|..|..|..|..|..|...O: O7326 (stop)
```

Eventually the UBID search locates a plan for stopping at the intersection after a number of "coast" operators followed by a "brake".

```
...

That looks like a plan which should work (top-level)!
We've figured out how to stop.
Building chunk-289
 866: ..|...O: O7384 (correction)
 867: ..|...O: O7386 (propose-operator)
Building chunk-290
 868: ...O: O27 (stop-in-time)
 869: ..|==>S: S130 (operator no-change) ( )
 870: ..|...O: O7388 (maintain)
Sending command tick-simulation  Simulation updating.
Distance is 54 Speed is 14 and Dspeed is 0 Clock is 9.0
```

IMPROV starts to execute the plan in the environment.

```
 871: ..|...O: O7390 (maintain)
Sending command tick-simulation  Simulation updating.
Distance is 47 Speed is 14 and Dspeed is 0 Clock is 9.5
 872: ..|...O: O7392 (maintain)
Sending command tick-simulation  Simulation updating.
Distance is 40 Speed is 14 and Dspeed is 0 Clock is 10.0
 873: ..|...O: O7394 (maintain)
Sending command tick-simulation  Simulation updating.
Distance is 33 Speed is 14 and Dspeed is 0 Clock is 10.5
 874: ..|...O: O7396 (stop)
 875: ..|..|==>S: S131 (operator no-change) ( )
 879: ..|..|...O: O7398 (brake)
Sending command brake
 880: ..|..|..|==>S: S134 (operator no-change) ( )
 881: ..|..|..|...O: O7407 (dspeed-3)
 882: ..|..|..|...O: O7408 (brake-pressure5)
 883: ..|..|..|...O: O7409 (tick)
Sending command tick-simulation  Simulation updating.
Distance is 28 Speed is 10 and Dspeed is -4 Clock is 11.0
Missing dspeed. Expected -3 saw -4.
Missing brake-pressure. Expected 5 saw 8.
 884: ..|..|..|..|==>S: S135 (state no-change) ( )
Error detected for operator O7398
 Measurement 4 : Error detected for operator brake
```

A different rate of deceleration (dspeed) and brake-pressure are observed than are expected, leading to a state no-change (i.e. no operator proposed). IMPROV detects this lack of an operator as an error (the unexepected deceleration is not itself seen as an error).

```
Building chunk-291
Building chunk-292
Building chunk-293
Building chunk-294
Building chunk-295
Building chunk-296
Building chunk-297
 885: ..|..|..|...O: O7418 (error-detected)
Building chunk-298
Building chunk-299
 886: ..|..|..|...O: O7417 (recall-action)
 887: ..|..|..|...O: O7416 (recall-action)
 888: ..|..|..|...O: O7425 (recall-previous)
 889: ..|..|..|..|==>S: S136 (operator no-change) ( )
 890: ..|..|..|..|...O: O7428 (make-candidate1)
 891: ..|..|..|..|...O: O7429 (make-candidate)
 892: ..|..|..|..|...O: O7431 (attend) (I4 vehicle green)
 893: ..|..|..|..|...O: O7432 (make-set)
 894: ..|..|..|..|...O: O7434 (find-candidate) (single green vehicle)
 921: ..|..|..|..|...O: O7908 (activate-candidate brake-pressure5)
```

IMPROV records that brake can lead to an error and records the previously seen effects (dspeed -4 instead of -3, for example).

Figure A.18: Correcting effects, first run – part 3

```
 922: ..|..|..|..|...O: O7911 (train) (reject brake-pressure5)
 984: ..|..|..|..|..|..|...O: O8484 (return-result)
 985: ..|..|..|..|..|...O: O8338 (return-result)
Building chunk-314
Building chunk-315
```

IMPROV learns (temporarily) not to choose brake-pressure 5 in the same situation in future.

```
sp {chunk-314
   :chunk
   (state <s1> ^operator <o1> ^error <e1> ^desired <d1>)
   (<o1> ^type sca ^prediction <p1> ^task-space <t1> ^attend-set <a1>)
   (<p1> ^attribute proposal)
   (<t1> ^name single)
   (<e1> ^watch-constant constant1794)
   (<a1> ^feature <f1> ^feature <f2> ^feature <f3>)
   (<f1> ^attribute brake-pressure ^value 0 ^orig state ^source <s2>)
   (<f2> ^attribute sent-brake-command ^value yes ^orig state ^source <s2>)
   (<f3> ^value vehicle ^attribute isa ^orig state ^source <s2>)
   -->
```

While in SCA2, IMPROV learns a rule to recall all of the state features in future (as usual, this is part of k-incremental learning--note that the rule tests for the presence of this error)

```
   (<o1> ^previous-failures <p2> + ^previous-failures <p2> &)
   (<p2> ^state <s3> + ^operator <o2> +)
   (<s3> ^feature  <f4> + & ^feature  <f5> + & ^feature  <f6> + & ^feature <f7>  + &
       ^feature  <f8> + & ^feature  <f9> + & ^feature <f10> + & ^feature <f10> + &
       ^feature <f12> + & ^feature <f13> + & ^feature <f14> + & ^feature <f15> + &
       ^feature <f16> + & ^feature <f17> + & ^feature <f18> + & ^feature <f19> + &
       ^feature <f20> + & ^generated-results <g1> +)
   (<f4> ^source <s2> + ^attribute sent-brake-command + ^value yes +)
   (<f5> ^source <s2> + ^attribute other-sign        + ^value stop +)
   (<f6> ^source <s2> + ^attribute accelerating      + ^value no +)
   (<f7> ^source <s2> + ^attribute braking           + ^value no +)
   (<f8> ^source <s2> + ^attribute other-sidewalk    + ^value pedestrian +)
   (<f9> ^source <s2> + ^attribute other-road        + ^value car +)
   (<f10> ^source <s2> + ^attribute sign             + ^value red +)
   (<f11> ^source <s2> + ^attribute fps              + ^value 14 +)
   (<f12> ^source <s2> + ^attribute brake-pressure   + ^value 0 +)
   (<f13> ^source <s2> + ^attribute dspeed           + ^value 0 +)
   (<f14> ^source <s2> + ^attribute road-quality     + ^value excellent +)
   (<f15> ^source <s2> + ^attribute weather          + ^value sunny +)
   (<f16> ^source <s2> + ^attribute gear             + ^value 4 +)
   (<f17> ^source <s2> + ^attribute feet             + ^value 35 +)
   (<f18> ^source <s2> + ^attribute type             + ^value car +)
   (<f19> ^source <s2> + ^attribute color            + ^value green +)
   (<f20> ^source <s2> + ^attribute isa              + ^value vehicle +)
   (<g1>  ^brake-pressure 8 + ^dspeed -4 +)
   (<o2>  ^value 5 + ^attribute brake-pressure +
        ^class single + ^name brake-pressure5 +)
}
Building chunk-316
 986: ..|..|..|...O: O7426 (recall-previous)
 987: ..|..|..|..|==>S: S151 (operator no-change) ( )
 988: ..|..|..|..|...O: O8487 (make-candidate1)    (type no-attend)
 989: ..|..|..|..|...O: O8488 (make-candidate)
 990: ..|..|..|..|...O: O8490 (attend) (I4 vehicle green)
 991: ..|..|..|..|...O: O8491 (make-set)
 992: ..|..|..|..|...O: O8493 (find-candidate) (single green vehicle)
1019: ..|..|..|..|...O: O8968 (activate-candidate dspeed-3)
1020: ..|..|..|..|...O: O8971 (train) (reject dspeed-3)

...
```

This rule also includes the values that changed on the state during the last sensing of the state. In this case (since we're making things easy for this trial) the only values to change are brake-pressure 8 and dspeed -4. This knowledge is later used to bias the search for a correct model of the effects of braking.

The process is repeated for dspeed -3.

Figure A.19: Correcting effects, first run – part 4

```
1084: ..|..|..|...O: O9548 (attempt-correction)
1085: ..|..|..|..|==>S: S168 (operator conflict) ( )
1086: ..|..|..|..|...O: O9551 (make-candidate1)    (type no-attend)
1087: ..|..|..|..|...O: O9552 (make-candidate)
1088: ..|..|..|..|...O: O9553 (attend) (I4 vehicle green)
1089: ..|..|..|..|...O: O9554 (make-set)
1090: ..|..|..|..|...O: O9556 (find-candidate) (single green vehicle)
1113: ..|..|..|..|...O: O10009 (activate-candidate dspeed-3)
1114: ..|..|..|..|...O: O10011 (set-uncertainty-limit)
1115: ..|..|..|..|...O: O10010 (evaluate-object O9988 (dspeed-3) )
1116: ..|..|..|..|..|==>S: S172 (operator no-change) ( )
1585: ..|..|..|..|...O: O15289 (observe) (dspeed-3)
1586: ..|..|..|..|..|==>S: S230 (operator no-change) ( )
1587: ..|..|..|..|..|...O: C505 (dspeed-3)
1588: ..|..|..|..|..|..|==>S: S231 (operator conflict) ( )
1589: ..|..|..|..|..|..|...O: O15295 (make-candidate1)    (type no-attend)
1590: ..|..|..|..|..|..|...O: O15296 (make-candidate)
1591: ..|..|..|..|..|..|...O: O15297 (attend) (I4 vehicle green)
1592: ..|..|..|..|..|..|...O: O15298 (make-set)
1593: ..|..|..|..|..|..|...O: O15300 (find-candidate) (single green vehicle)
1616: ..|..|..|..|..|..|...O: O15753 (activate-candidate brake-pressure5)
1617: ..|..|..|..|..|..|...O: O15754 (set-uncertainty-limit)
1618: ..|..|..|..|..|..|...O: O15755 (evaluate-object O15732 (brake-pressure5) )
1632: ..|..|..|..|..|..|...O: O15769 (observe) (brake-pressure5)
1633: ..|..|..|..|..|..|..|==>S: S238 (operator no-change) ( )
1634: ..|..|..|..|..|..|..|...O: C516 (brake-pressure5)
1679: ..|..|..|..|..|..|..|..|...O: O16274 (observe) (tick)
1680: ..|..|..|..|..|..|..|..|..|==>S: S246 (operator no-change) ( )
1681: ..|..|..|..|..|..|..|..|..|...O: C527 (tick)
Sending command tick-simulation
Distance is 27 Speed is 2 and Dspeed is -8 Clock is 11.5
Missing dspeed. Expected -3 saw -8.
Missing brake-pressure. Expected 5 saw 16.
1718: ..|..|..|..|..|..|..|..|..|..|...O: O16772 (observe) (dspeed-3)
1719: ..|..|..|..|..|..|..|..|..|..|..|==>S: S254 (operator no-change) ( )
1720: ..|..|..|..|..|..|..|..|..|..|..|...O: C540 (dspeed-3)
1756: ..|..|..|..|..|..|..|..|..|..|..|..|...O: O17248 (observe) (brake-pressure5)
1757: ..|..|..|..|..|..|..|..|..|..|..|..|..|==>S: S262 (operator no-change) ( )
1758: ..|..|..|..|..|..|..|..|..|..|..|..|..|...O: C551 (brake-pressure5)
1794: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O17749 (observe) (tick)
1795: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|==>S: S270 (operator no-change) ( )
1796: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: C562 (tick)
Sending command tick-simulation
Distance is 27 Speed is 0 and Dspeed is -2 Clock is 12.0
Missing dspeed. Expected -3 saw -2.
Missing brake-pressure. Expected 5 saw 16.
1797: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|==>S: S271 (operator conflict) ( )
1798: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O17786 (terminate-observation)
1799: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O17788 (terminate-observation)
  Observation of O17219 (brake-pressure5) is partial-failure
1800: ..|..|..|..|..|..|..|..|..|..|..|..|...O: O17790 (train) (reject brake-pressure5)
1801: ..|..|..|..|..|..|..|..|..|..|..|..|..|==>S: S272 (operator no-change) ( )
1854: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O18321 (return-result)
1855: ..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O18196 (return-result)
1856: ..|..|..|..|..|..|..|..|..|..|..|..|...O: O18322 (terminate-observation)
  Observation of O16743 (dspeed-3) is partial-failure
1857: ..|..|..|..|..|..|..|..|..|..|...O: O18324 (train) (reject dspeed-3)
1921: ..|..|..|..|..|..|..|..|..|..|...O: O18900 (terminate-observation)
1922: ..|..|..|..|..|..|..|..|...O: O18902 (terminate-observation)
  Observation of O15732 (brake-pressure5) is partial-failure
1923: ..|..|..|..|..|..|...O: O18904 (train) (reject brake-pressure5)
1979: ..|..|..|..|..|..|...O: O19436 (terminate-observation)
1980: ..|..|..|..|...O: O19438 (train) (reject dspeed-3)
2044: ..|..|..|..|...O: O20014 (terminate-observation)
2045: ..|..|..|...O: O20015 (attempt-correction)
Already stationary.
Soar halted at Mon Oct  7 12:11:43 EDT 1996.

System halted.
```

Having recorded the initial error, dspeed and brake-pressure values, IMPROV proceeds to attempt to find a correction.

IMPROV proceeds to execute the remainder of the brake operator effects (e.g. dspeed -3) and monitors the eventual outcome of the plan....

... recording the observed values as it goes.

Eventual the "braking" is complete and IMPROV records all of the original single-effect operators as failure instances and trains to avoid them in future (temporarily), which will cause the next run of IMPROV to select an alternative implementation for the brake operator.

Figure A.20: Correcting effects, first run – part 5

```
soar> init-soar
init-soar

Conditions for a failure are :
and   soar(current) == intersection
and   events(light,value) == red
and   soar(distance) <= -20
and   soar(distance) >= -60
```

This is a second run attempting to stop correctly at the same intersection on a future occassion.

```
Setup for this run:
   Soar car is a green car initial speed 32 mph.
   The weather is sunny.
   The road quality is excellent
   The sign is a stop light (traffic light).
   Change cycle is  { 0.0 4.99 yellow } { 5.0 24.99 red } { 25.0 59.99 green }
   The other road user is a car.
   The sidewalk user is a pedestrian.
   The other sign is a stop sign.
Soar started at Mon Oct  7 12:12:04 EDT 1996.
soar> run
run

   0: ==>S: S1  ( )
  Simulation updating.
Distance is 173 Speed is 14 and Dspeed is 0 Clock is 0.5
Finished.
   1: ...O: O1 (drive)
   2: ..|==>S: S2 (operator no-change) ( )
   3: ..|...O: O2 (maintain)
Firing chunk-290
  19: ...O: O21 (stop-in-time)
  20: ..|==>S: S3 (operator no-change) ( )
Firing chunk-245
```

When the agent starts the "stop" maneuver, it recalls the earlier error and starts to look for a way to avoid the error this time.

```
  21: ..|...O: O22 (error-detected)
  22: ..|...O: O25 (attempt-correction)
  23: ..|..|==>S: S4 (operator conflict) ( )
  24: ..|..|...O: O28 (make-candidate1)    (type no-attend)
  25: ..|..|...O: O29 (make-candidate)
  26: ..|..|...O: O30 (attend) (I4 vehicle green)
  27: ..|..|...O: O31 (make-set)
  28: ..|..|...O: O33 (find-candidate) (driving green vehicle)
  29: ..|..|...O: O37 (activate-candidate stop)
  30: ..|..|...O: O38 (set-uncertainty-limit)
  31: ..|..|...O: O39 (evaluate-object O34 (stop) )
  32: ..|..|..|==>S: S6 (operator no-change) ( )
```

The reasons for selecting stop and brake have not changed and they are still chosen.

```
  33: ..|..|..|...O: L1 (stop)
  34: ..|..|..|..|==>S: S7 (operator no-change) ( )
  35: ..|..|..|..|..|==>S: S8 (operator conflict) ( )
  36: ..|..|..|..|..|...O: O46 (make-candidate1)    (type no-attend)
  37: ..|..|..|..|..|...O: O47 (make-candidate)
  38: ..|..|..|..|..|...O: O48 (attend) (I4 vehicle green)
  39: ..|..|..|..|..|...O: O49 (make-set)
  40: ..|..|..|..|..|...O: O51 (find-candidate) (controls green vehicle)
  41: ..|..|..|..|..|...O: O55 (activate-candidate brake)
  42: ..|..|..|..|..|...O: O56 (set-uncertainty-limit)
  43: ..|..|..|..|..|...O: O57 (evaluate-object O52 (brake) )
  44: ..|..|..|..|..|..|==>S: S10 (operator no-change) ( )
  45: ..|..|..|..|..|..|...O: L2 (brake)
  46: ..|..|..|..|..|..|..|==>S: S11 (operator no-change) ( )
Building chunk-452
Firing chunk-60
  47: ..|..|..|..|..|..|..|..|==>S: S12 (operator conflict) ( )
  48: ..|..|..|..|..|..|..|..|...O: O64 (make-candidate1)    (type no-attend)
  49: ..|..|..|..|..|..|..|..|...O: O65 (make-candidate)
  50: ..|..|..|..|..|..|..|..|...O: O66 (attend) (I4 vehicle green)
  51: ..|..|..|..|..|..|..|..|...O: O67 (make-set)
  52: ..|..|..|..|..|..|..|..|...O: O69 (find-candidate) (single green vehicle)
```

However, the implementation of brake is now different...

Figure A.21: Second run – part 1

```
  52: ..|..|..|..|..|..|..|..|...O: O69 (find-candidate) (single green vehicle)
Firing chunk-314
Firing chunk-327
  53: ..|..|..|..|..|..|..|..|..|==>S: S16 (operator no-change) ( )
  54: ..|..|..|..|..|..|..|..|..|...O: O73 (count-set)
  55: ..|..|..|..|..|..|..|..|..|...O: O148 (abstract 16)
   ...
  68: ..|..|..|..|..|..|..|..|..|...O: O434 (abstract 3)
Firing chunk-312
Firing chunk-325
  69: ..|..|..|..|..|..|..|..|..|...O: O458 (predict reject dspeed-3)
  73: ..|..|..|..|..|..|..|..|..|...O: O457 (predict reject brake-pressure5)
  77: ..|..|..|..|..|..|..|..|..|...O: O456 (abstract 2)
  88: ..|..|..|..|..|..|..|..|..|...O: O628 (predict propose dspeed-4)
  92: ..|..|..|..|..|..|..|..|..|...O: O686 (return-result)
  93: ..|..|..|..|..|..|..|..|...O: O688 (activate-candidate dspeed-4)
  94: ..|..|..|..|..|..|..|..|...O: O689 (set-uncertainty-limit)
  95: ..|..|..|..|..|..|..|..|...O: O690 (evaluate-object O667 (dspeed-4) )
  96: ..|..|..|..|..|..|..|..|..|==>S: S22 (operator no-change) ( )
  97: ..|..|..|..|..|..|..|..|..|...O: L3 (dspeed-4)
  98: ..|..|..|..|..|..|..|..|..|..|==>S: S23 (operator conflict) ( )
  99: ..|..|..|..|..|..|..|..|..|..|...O: O694 (make-candidate1)    (type no-attend)
 100: ..|..|..|..|..|..|..|..|..|..|...O: O695 (make-candidate)
 101: ..|..|..|..|..|..|..|..|..|..|...O: O696 (attend) (I4 vehicle green)
 102: ..|..|..|..|..|..|..|..|..|..|...O: O697 (make-set)
 103: ..|..|..|..|..|..|..|..|..|..|...O: O699 (find-candidate) (single green vehicle)
Firing chunk-314
Firing chunk-327
 140: ..|..|..|..|..|..|..|..|..|..|...O: O1297 (activate-candidate brake-pressure8)
 141: ..|..|..|..|..|..|..|..|..|..|...O: O1298 (set-uncertainty-limit)
 142: ..|..|..|..|..|..|..|..|..|..|...O: O1299 (evaluate-object O1276 (brake-pressure8) )
 143: ..|..|..|..|..|..|..|..|..|..|..|==>S: S32 (operator no-change) ( )

 ...

 529: ..|..|..|..|..|..|..|..|...O: O5431 (propose-operator)
 530: ..|..|..|..|..|..|..|...O: O667 (dspeed-4)
 531: ..|..|..|..|..|..|..|...O: O5432 (brake-pressure8)
 532: ..|..|..|..|..|..|..|...O: O5433 (tick)
 533: ..|..|..|..|..|..|..|...O: O5434 (dspeed-8)
 534: ..|..|..|..|..|..|..|...O: O5435 (brake-pressure16)
 535: ..|..|..|..|..|..|..|...O: O5436 (tick)
 536: ..|..|..|..|..|..|..|...O: O5438 (dspeed-2)
 537: ..|..|..|..|..|..|..|...O: O5439 (brake-pressure16)
 538: ..|..|..|..|..|..|..|...O: O5440 (tick)
 539: ..|..|..|..|..|...O: O5444 (correction)
 540: ..|..|..|..|..|...O: O5445 (propose-operator)
 541: ..|..|..|..|...O: O52 (brake)
Firing chunk-452
Firing chunk-527
Firing chunk-528
Firing chunk-529
Internal SIM had distance 61 speed 14 time 8.5
Internal SIM now has distance 56.0 speed 10 time 9.0
Firing chunk-530
Firing chunk-531
Firing chunk-532
Internal SIM had distance 56 speed 10 time 9.
Internal SIM now has distance 55.0 speed 2 time 9.5
Building chunk-547
Building chunk-548
Building chunk-549
Internal SIM had distance 55 speed 2 time 9.5
Internal SIM now has distance 55.0 speed 0 time 10.0
That plan would stop me short of the maximum distance at 55
We've figured out how to stop.
Building chunk-550
```

This time, IMPROV recalls the previous failed attempt to model brake and avoids the earlier model (dspeed -3 and brake-pressure 5)

It then selects dspeed -4 on the basis of the information in chunk-314 and chunk-327 (see the earlier part of this trace to see where that information is learned).

IMPROV then attempts to build a plan on the basis of this new model for the effects of "brake".

The other steps are similarly chosen to match the values observed during the failed attempt to brake.

The combined sequence becomes the new effects for the brake operator.

```
sp {chunk-527
   :chunk
   (state <s1> ^level lookahead ^operator <o1> ^io-state <i1>)
   (<o1> ^name brake ^actions 2 ^desired <d1>)
   (<d1> ^name brake ^fps 0)
   (<i1> ^car <o2> ^new <n1>)
   (<o2> ^dspeed 0 ^object-static <o3>)
   (<o3> ^isa vehicle)
   -->
   (<n1> ^dspeed -4 +)
}
```

The new effects indicate that the car will stop short of the intersection if the agent brakes immediately. Again, the agent must coast for a while before braking.

Figure A.22: Second run – part 2

```
542: ..|..|...O: O5448 (make-candidate1)    (type no-best)
543: ..|..|...O: O36 (make-candidate)
544: ..|..|...O: O5450 (find-candidate) (driving green vehicle)
545: ..|..|...O: O5453 (activate-candidate maintain)
546: ..|..|...O: O5454 (evaluate-object O5451 (maintain) )
547: ..|..|..|==>S: S102 (operator no-change) ( )
548: ..|..|..|...O: L12 (maintain)
549: ..|..|..|..|==>S: S103 (operator no-change) ( )
550: ..|..|..|..|...O: O5458 (coast)
Internal SIM had distance 61 speed 14 time 8.5
Internal SIM now has distance 54.0 speed 14 time 9.0
551: ..|..|..|..|==>S: S104 (operator conflict) ( )
552: ..|..|..|..|...O: O5462 (make-candidate1)    (type no-attend)
553: ..|..|..|..|...O: O5463 (make-candidate)
554: ..|..|..|..|...O: O5464 (attend) (I4 vehicle green)
555: ..|..|..|..|...O: O5465 (make-set)
556: ..|..|..|..|...O: O5467 (find-candidate) (driving green vehicle)
557: ..|..|..|..|...O: O5471 (activate-candidate stop)
558: ..|..|..|..|...O: O5472 (set-uncertainty-limit)
559: ..|..|..|..|...O: O5473 (evaluate-object O5468 (stop) )
560: ..|..|..|..|..|==>S: S106 (operator no-change) ( )
561: ..|..|..|..|..|...O: L13 (stop)
Internal SIM had distance 54 speed 14 time 9.
Internal SIM now has distance 49.0 speed 10 time 9.5
576: ..|..|..|..|..|..|...O: O5485 (brake)
That plan would stop me short of the maximum distance at 50
577: ..|..|..|..|...O: O5499 (make-candidate1)    (type no-best)
578: ..|..|..|..|...O: O5470 (make-candidate)
579: ..|..|..|..|...O: O5501 (find-candidate) (driving green vehicle)
580: ..|..|..|..|...O: O5504 (activate-candidate maintain)
585: ..|..|..|..|..|..|...O: O5509 (coast)
Internal SIM had distance 54 speed 14 time 9.
Internal SIM now has distance 47.0 speed 14 time 9.5
596: ..|..|..|..|..|..|..|...O: L16 (stop)
That plan would stop me short of the maximum distance at 40
618: ..|..|..|..|..|..|..|...O: L18 (maintain)
619: ..|..|..|..|..|..|..|..|==>S: S121 (operator no-change) ( )
620: ..|..|..|..|..|..|..|..|...O: O5560 (coast)
671: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: L23 (stop)
703: ..|..|..|..|...O: O5663 (propose-operator)
Building chunk-599
Building chunk-600
Building chunk-601
704: ..|..|..|...O: O5502 (maintain)
707: ..|..|..|...O: O5666 (maintain)
710: ..|..|..|...O: O5669 (maintain)
713: ..|..|..|...O: O5672 (maintain)
716: ..|..|..|...O: O5674 (stop)
Firing chunk-451
Firing chunk-541
Firing chunk-542
Firing chunk-543
Internal SIM had distance 26 speed 14 time 11.
Internal SIM now has distance 21.0 speed 10 time 11.5
Retracting chunk-579
Firing chunk-556
Firing chunk-557
Firing chunk-558
Internal SIM had distance 21 speed 10 time 11.5
Internal SIM now has distance 20.0 speed 2 time 12.0
Firing chunk-568
Firing chunk-569
Firing chunk-570
Internal SIM had distance 20 speed 2 time 12.
Internal SIM now has distance 20.0 speed 0 time 12.5
That looks like a plan which should work!
Building chunk-606
```

IMPROV considers adding coast operators before the brake operator.

Notice that the new effects of the brake operator have been compiled, so the operator can be applied in a single step without decomposing the operator down to the level of primitive effects.

Eventually IMPROV discovers a new plan, using the new model of the brake operator.

Figure A.23: Second run – part 3

```
717: ..|..|...O: O5676 (observe) (maintain)
719: ..|..|..|...O: C341 (maintain)
734: ..|..|..|..|...O: O5706 (observe) (maintain)
751: ..|..|..|..|..|..|...O: O5739 (observe) (maintain)
768: ..|..|..|..|..|..|..|..|...O: O5774 (observe) (maintain)
```

IMPROV executes the new plan in the external environment to see if it works.

```
797: ..|..|..|..|..|..|..|..|..|..|..|..|...O: O5837 (observe) (stop)
798: ..|..|..|..|..|..|..|..|..|..|..|..|..|==>S: S172 (operator no-change) ( )
799: ..|..|..|..|..|..|..|..|..|..|..|..|..|...O: C380 (stop)
810: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O5860 (observe) (brake)
811: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|==>S: S178 (operator no-change) ( )
812: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: C387 (brake)
Sending command brake
813: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|==>S: S179 (operator no-change) ( )
874: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O6512 (observe) (dspeed-4)
875: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|==>S: S193 (operator no-change) ( )
932: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O7139 (observe) (brake-pressure8)
933: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|==>S: S206 (operator no-change) ( )
984: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O7690 (observe) (tick)
985: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|==>S: S218 (operator no-change) ( )
```

In this relatively simple example, the new plan is immediately successful, leading IMPROV to learn a permanent correction to the effects of brake.

```
1044: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O8352 (observe) (dspeed-8)
1310: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O11436 (terminate-observation)
 That worked.
  Observation of O10798 (brake-pressure16) is partial-success
1311: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O11438 (train) (propose brake-pressure16)
1376: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O12199 (return-result)
Building chunk-671
Building chunk-672
1377: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O12034 (return-result)
Building chunk-673
1378: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O12201 (learn-correction)
1379: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|==>S: S311 (operator no-change) ( )
1380: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|==>S: S312 (operator conflict) ( )
1381: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O12207 (make-candidate1)    (type no-attend)
1382: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O12208 (make-candidate)
1383: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O12209 (attend) (I4 vehicle green)
1384: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O12210 (make-set)
1385: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O12212 (find-candidate) (single green vehicle)
1421: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O12754 (activate-candidate brake-pressure16)
1422: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O12755 (set-uncertainty-limit)
1423: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O12756 (evaluate-object O12733 (brake-pressure16) )
1424: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O12757 (correction)
1425: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O12759 (propose-operator)
Building chunk-679
Building chunk-680
Building chunk-681
Building chunk-682
1426: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O12200 (terminate-observation)
  Observation of O10172 (dspeed-2) is partial-success
1427: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O12760 (train) (propose dspeed-2)
1506: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O13589 (learn-correction)
1558: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O13588 (terminate-observation)
1562: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O14175 (train) (propose brake-pressure16)
1629: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O14938 (learn-correction)
1677: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O14937 (terminate-observation)
1678: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O15496 (train) (propose dspeed-8)
1757: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O16325 (learn-correction)
1809: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O16324 (terminate-observation)
1813: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O16910 (train) (propose brake-pressure8)
1880: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O17673 (learn-correction)
1928: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O17672 (terminate-observation)
2008: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O19061 (learn-correction)
2060: ..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|..|...O: O19060 (terminate-observation)

...

2082: ..|...O: O11391 (test-goal)
Car stopped successfully.
Soar halted at Mon Oct  7 12:16:19 EDT 1996.
```

Once learning is complete, there is no more work to do.

Figure A.24: Second run – part 4

```
soar> init-soar
soar> run
  0: ==>S: S1  ( )
  1: ...O: O1 (drive)
  2: ..|==>S: S2 (operator no-change) ( )
```

Third run, learning is complete and the agent zips through the problem.

```
  3: ..|...O: O2 (maintain)
  4: ..|...O: O2 (maintain)
  5: ..|...O: O2 (maintain)
  6: ..|...O: O2 (maintain)
  7: ..|...O: O2 (maintain)
  8: ..|...O: O2 (maintain)
 *** Event light is now yellow ***
 *** Event other_road is now car ***
 *** Event other_sidewalk is now pedestrian ***
  9: ..|...O: O2 (maintain)
 10: ..|...O: O2 (maintain)
 11: ..|...O: O2 (maintain)
 12: ..|...O: O2 (maintain)
 13: ..|...O: O2 (maintain)
 14: ..|...O: O2 (maintain)
 15: ..|...O: O2 (maintain)
 16: ..|...O: O2 (maintain)
 17: ..|...O: O2 (maintain)
 18: ..|...O: O2 (maintain)
 *** Event light is now red ***
 19: ...O: O21 (stop-in-time)
 20: ..|==>S: S3 (operator no-change) ( )
 21: ..|...O: O23 (maintain)
 22: ..|...O: O26 (maintain)
 23: ..|...O: O29 (maintain)
 24: ..|...O: O32 (maintain)
```

The correct plan to brake (with just the right delay before braking) is recalled and chosen. (Lots of chunks fire here, but I've delected them for clarity).

```
 25: ..|...O: O35 (maintain)
 26: ..|...O: O37 (stop)
 27: ..|..|==>S: S4 (operator no-change) ( )
 31: ..|..|...O: O40 (brake)
Sending command brake
 32: ..|..|..|==>S: S7 (operator no-change) ( )
```

The agent starts to brake in the external world...

```
Firing chunk-60
Firing chunk-526
Firing chunk-774
Firing chunk-798
Firing chunk-524
Firing chunk-797
 33: ..|..|..|...O: O53 (dspeed-4)
Firing chunk-522
Firing chunk-773
 34: ..|..|..|...O: O52 (brake-pressure8)
```

And models the effects of braking as the car is slowing.

```
Firing chunk-519
 35: ..|..|..|...O: O55 (tick)
Distance is 21 Speed is 10 and Dspeed is -4 Clock is 11.5
 36: ..|..|..|...O: O59 (dspeed-8)
 37: ..|..|..|...O: O58 (brake-pressure16)
 38: ..|..|..|...O: O64 (tick)
 39: ..|..|..|...O: O67 (dspeed-2)
 40: ..|..|..|...O: O66 (brake-pressure16)
 41: ..|..|..|...O: O75 (tick)
```

No errors are detected, so the goal is achieved and the task complete.

```
 42: ..|..|..|...O: O80 (test-goal)
We're done--and it worked
Soar halted at Mon Oct  7 12:21:07 EDT 1996.

System halted.
```

Figure A.25: Third run

APPENDIX B

The Operator Proposal Problem Space

The majority of IMPROV's learning and control occurs in the operator proposal problem space. This appendix is a very brief, technical description of the role of each operator in the problem space. This appendix is primarily intended for people who are moderately familiar with Soar.

The operators, and their typical order of execution, are as follows:

```
make-candidate1
make-candidate
attend
make-set
find-candidate
activate-candidate
set-uncertainty-limit
evaluate-object
observe
train
correction
propose-operator
```

The general role of this problem space is to select a task operator as the next step in a plan to achieve the current goal. This task decomposes into the problem of learning the correct conditions for choosing particular operators and therefore maps to IMPROV's main learning task.

1. Make-candidate1 and make-candidate

   These two operators are used to signal that a new task operator should be generated; that is, a new *candidate*. The fact that two operators are used, is just a historical artifact.

2. Attend

   The attend operator selects a subset of the state. This subset is used as the initial feature set for SCA2. SCA2 must return a unique operator for a given attention set. Consider, a `pick-up` operator. If IMPROV attends to a particular green block, SCA2 may return `pick-up(green-block)`, while if IMPROV attends to a different blue block, SCA2 could then return `pick-up(blue-block)`. This attention mechanism allows IMPROV to avoid generating a thousand `get-book` operators when faced with a library.

3. Make-set

This operator converts the attended subset of the state from a graph representation to a flat set of features. This set representation is more appropriate for SCA2.

4. Find-candidate

   This is the interface to SCA2's performance mechanism. Find-candidate leads to SCA2 predicting a particular operator to match a given state and goal.

5. Activate-candidate

   After SCA2 generates a candidate operator, IMPROV can either select that operator immediately or generate an alternative attention set and generate other cadnidates. The decision is based on the uncertainty level returned by SCA2. The decision to activate a candidate is made when a reasonable candidate has been located or a maximum number of attention shifts have been made.

6. Set-uncertainty-limit

   This operator controls the setting of the current bound for the UBID search. The value is usually derived from a supergoal and then reduced by the uncertainty of the current operator.

7. Evaluate-object

   This operator leads IMPROV to internally simulate the current candidate operator and a series of subsequent operators. The evaluation will either be success (because the goal is reached) or failure (because a known failure state is reached) or UBID's uncertainty bound will be exceeded. The evaluate-object operator leads to a recursive series of operator proposal problem spaces.

8. Observe

   This operator is chosen when a candidate task operator has been selected for execution in the external world. The observe operator records the state before the task operator is applied and then subgoals and executes the task operator in the external world. It is very similar to the evaluate-object operator as it leads to a recursive stack of operator proposal problem spaces and eventually either a success or failure result.

9. Train

   This operator is the second interface point to SCA2 and is used during training. SCA2 will learn a new prediction rule as a result of the train operator. Some of the new rules are effectively short-term (as they test the current error, which will eventually be discarded) while others are long-term (as they do not test the current error).

10. Correction

    The correction operator computes the appropriate version number to use for the new operator precondition rule.

11. Propose-operator

    This final operator creates a new operator proposal (or operator precondition) rule and may also create new rules to reject other, incorrect operators. This operator produces the final, long-term correction to the agent's knowledge.

BIBLIOGRAPHY

BIBLIOGRAPHY

[Aasman, 1995] Jannes Aasman. *Modelling Driver Behavior in Soar.* PhD thesis, Rijksuniversiteit Groningen, 1995.

[Asker, 1994] Lars Asker. Improving accuracy of incorrect domain theories. In *Proceedings of the Eleventh International Conference on Machine Learning*, pages 19–27, 1994.

[Baffes and Mooney, 1993] Paul T. Baffes and Raymond J. Mooney. Symbolic revision of theories with m-of-n rules. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 1135–1140, 1993.

[Benson and Nilsson, 1996] Scott Benson and Nils J. Nilsson. Reacting, planning, and learning in an autonomous agent. In Stephen Muggleton, Donald Michie, and Koichi Furukawa, editors, *Machine Intelligence*, volume 14. Oxford University Press, 1996.

[Benson, 1995] Scott Benson. Inductive learning of reactive action models. In *Proceedings of the Twelth International Conference on Machine Learning*, pages 47–54, 1995.

[Booker *et al.*, 1989] L. B. Booker, D. E. Goldberg, and J. H. Holland. Classifier systems and genetic algorithms. *Artificial Intelligence*, 40:235–282, 1989.

[Cain, 1991] Timothy Cain. The ductor: A theory revision system for propositional domains. In *Proceedings of the Eighth International Conference on Machine Learning*, pages 485–489, 1991.

[Carbonell and Gil, 1987] Jaime G. Carbonell and Yolanda Gil. Learning by experimentation. In *Proceedings of the International Workshop on Machine Learning*, pages 256–265, 1987.

[Carbonell *et al.*, 1991] Jaime G. Carbonell, Craig A. Knoblock, and Steven Minton. Prodigy: An integrated architecture for planning and learning. In Kurt VanLehn, editor, *Architectures for Intelligence.* Lawrence Erlbaum, 1991.

[Clearwater *et al.*, 1989] Scott H. Clearwater, Tze-Pin Cheng, Haym Hirsh, and Bruce G. Buchanan. Incremental batch learning. In *Proceedings of the International Workshop on Machine Learning*, pages 366–370, 1989.

[Covrigaru, 1992] Arie Covrigaru. *Emergence of meta-level control in multi-tasking autonomous agents.* PhD thesis, University of Michigan, Department of Electrical Engineering and Computer Science, 1992.

[DeJong and Mooney, 1986] Gerald F. DeJong and Raymond J. Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1(2):145–176, 1986.

[Doorenbos, 1993] Robert B. Doorenbos. Matching 100,000 learned rules. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 290–296, 1993.

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

[Gil, 1991] Yolanda Gil. A domain-independent framework for effective experimentation in planning. In *Proceedings of the International Machine Learning Workshop*, pages 13–17, 1991.

[Gil, 1992] Yolanda Gil. *Acquiring Domain Knowledge for Planning by Experimentation.* PhD thesis, Carnegie Mellon University, August 1992.

[Gil, 1993] Yolanda Gil. Efficient domain-independent experimentation. In *Proceedings of the International Conference on Machine Learning*, pages 128–134, 1993.

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

[Gratch and DeJong, 1992] Jonathan Gratch and Gerald DeJong. Composer: A probabilistic solution to the utility problem in speed-up learning. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 235–240, 1992.

[Gupta, 1987] Ajay Gupta. Explanation-based failure recovery. In *Proceedings of the National Conference on Artificial Intelligence*, pages 606–610, 1987.

[Hall, 1986] R.J. Hall. Learning by failing to explain. In *Proceedings of the National Conference on Artificial Intelligence*, pages 568–572, Philadelphia, PA, August 1986.

[Hall, 1988] Robert J. Hall. Learning by failing to explain: Using partial explanations to learn in incomplete or intractable domains. *Machine Learning*, 3:45–78, 1988.

[Hammond *et al.*, 1990] Kristian Hammond, Timothy Converse, and Charles Martin. Integrating planning and acting in a case-based framework. In *Proceedings of the Eighth National Conference on Artificial Intelligence*, pages 292–297, Boston, Mass., 1990. AAAI Press.

[Hammond, 1986] Kristian J. Hammond. Learning to anticipate and avoid planning problems through the explanation of failures. In *Proceedings of the National Conference on Artificial Intelligence*, pages 556–560, 1986.

[Hammond, 1989] Kristian Hammond. Chef. In C. K. Riesbeck and R. C. Schank, editors, *Inside case-based reasoning.* Erlbaum, Hillsdale, NJ, 1989.

[Hard *et al.*, 1990] D. Hard, S. Anderson, and P. Cohen. Envelopes as a vehicle for improving the efficiency of plan execution. In *Proceedings of DARPA Workshop on Innovative Approaches to Planning, Scheduling and Contol*, pages 71–76, 1990.

[Holland, 1986] John H. Holland. Escaping brittleness: The possibilities of general-purpose learning algorithms applied to parallel rule-based systems. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An artificial intelligence approach, Volume II.* Morgan Kaufmann, 1986.

[Howe and Cohen, 1991] Adele E. Howe and Paul R. Cohen. Failure recovery: A model and experiments. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 801–808, July 1991.

[Huffman and Laird, 1993] Scott B. Huffman and John E. Laird. Learning procedures from interactive natural language instructions. In P. Utgoff, editor, *Machine Learning: Proceedings of the Tenth International Conference*, 1993.

[Huffman and Laird, 1994] S. B. Huffman and J. E. Laird. Learning from highly flexible tutorial instruction. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, Seattle, WA, 1994.

[Huffman *et al.*, 1993] Scott B. Huffman, Douglas J. Pearson, and John E. Laird. Correcting imperfect domain theories: A knowledge-level analysis. In Susan Chipman and Alan L. Meyrowitz, editors, *Foundations of Knowledge Acquisition: Cognitive Models of Complex Learning*. Kluwer Academic, 1993. Also available as technical report number CSE-TR-114-91, University of Michigan Department of Electrical Engineering and Computer Science, November 1991.

[Huffman, 1994] S. B. Huffman. *Instructable autonomous agents*. PhD thesis, University of Michigan, Dept. of Electrical Engineering and Computer Science, 1994.

[Laird and Newell, 1983] J.E. Laird and A. Newell. A universal weak method: Summary of results. In *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, pages 771–773, August 1983.

[Laird *et al.*, 1986] John E. Laird, Paul S. Rosenbloom, and Allen Newell. Chunking in soar: The anatomy of a general learning mechanism. *Machine Learning*, 1(1):11–46, 1986.

[Laird *et al.*, 1987] John E. Laird, Allen Newell, and Paul S. Rosenbloom. Soar: An architecture for general intelligence. *Artificial Intelligence*, 33(1):1–64, 1987.

[Laird *et al.*, 1995] J. E. Laird, W. L. Johnson, R. M. Jones, F. Koss, J. F. Lehman, P. E. Nielsen, P. S. Rosenbloom, R. Rubinoff, K. B. Schamb, M. Tambe, J. Van Dyke, M. van Lent, and R. E. Wray. Simulated intelligent forces for air: The Soar/IFOR project 1995. In *Proceedings of the Fifth Conference on Computer Generated Forces and Behavioral Representation*, pages 27–36, May 1995.

[Laird, 1988] John E. Laird. Recovery from incorrect knowledge in Soar. In *Proceedings of the National Conference on Artificial Intelligence*, pages 618–623, August 1988.

[Maclin and Shavlik, 1996] R. Maclin and J. W. Shavlik. Creating advice-taking reinforcement learners. *Machine Learning*, 22:251–282, 1996.

[Mahadevan and Connell, 1991] Sridhar Mahadevan and Jonathan Connell. Automatic programming of behavior-based robots using reinforcement learning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 768–773, 1991.

[Mahadevan, 1992] Sridhar Mahadevan. Enhancing transfer in reinforcement learning by building stochastic models of robot actions. In *Proceedings of the Ninth International Workshop on Machine Learning*, pages 290–299, 1992.

[McCloskey and Cohen, 1989] Michael McCloskey and Neal J. Cohen. Catastrophic interference in connectionist networks: The sequential learning problem. *The Psychology of Learning and Motivation*, 24:109–165, 1989.

[Miller, 1991] Craig M. Miller. A constraint-motivated model of concept formation. In *The Thirteenth Annual Conference of the Cognitive Science Society*, pages 827–831, 1991.

[Miller, 1993] Craig M. Miller. *A model of concept acquisition in the context of a unified theory of cognition.* PhD thesis, The University of Michigan, Dept. of Computer Science and Electrical Engineering, 1993.

[Minton *et al.*, 1989] Steven Minton, Jaime G. Carbonell, Craig A. Knoblock, Daniel R. Kuokka, Oren Etzioni, and Yolanda Gil. Explanation-based learning: A problem-solving perspective. *Artificial Intelligence*, 40:63–118, 1989.

[Minton, 1990] Steven N. Minton. Quantatative results concerning the utility of explanation-based learning. *Artificial Intelligence*, 42:363–392, 1990.

[Minton, 1996] Steven Minton. Is there any need for domain-dependent control information? a reply. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 855–862, 1996.

[Mitchell *et al.*, 1986] Tom M. Mitchell, R. M. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1, 1986.

[Mitchell, 1982] Tom M. Mitchell. Generalization as search. *Artificial Intelligence*, 18(2):203–226, 1982.

[Muggleton and Feng, 1992] Stephen Muggleton and Cao Feng. Efficient induction of logic programs. In Stephen Muggleton, editor, *Inductive Logic Programming.* Academic Press, New York, NY, 1992.

[Murphy and Pazzani, 1994] Patrick M. Murphy and Michael J. Pazzani. Revision of production system rule-bases. In *Proceedings of the International Conference on Machine Learning*, pages 199–207, 1994.

[Oates and Cohen, 1996] Tim Oates and Paul R. Cohen. Searching for planning operators with context-dependent and probabilistic effects. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 863–868, 1996.

[Ourston and Mooney, 1990] Dirk Ourston and Raymond J. Mooney. Changing the rules: A comprehensive approach to theory refinement. In *Proceedings of the National Conference on Artificial Intelligence*, pages 815–820, 1990.

[Pazzani and Kibler, 1992] Michael Pazzani and Dennis Kibler. The role of prior knowledge in inductive learning. *Machine Learning*, 9:57–94, 1992.

[Pazzani *et al.*, 1991] Michael J. Pazzani, Clifford A. Brunk, and Glenn Silverstein. A knowledge-intensive approach to learning relational concepts. In *Proceedings of the Eighth International Workshop on Machine Learning*, pages 432–436, 1991.

[Pazzani, 1988] Michael J. Pazzani. Integrated learning with incorrect and incomplete theories. In *Proceedings of the International Machine Learning Conference*, pages 291–297, 1988.

[Pazzani, 1991] Michael Pazzani. Learning to predict and explain: An integration of similarity-based, theory driven, and explanation-based learning. *Journal of the Learning Sciences*, 1(2):153–199, 1991.

[Pearson and Huffman, 1995] Douglas J. Pearson and Scott B. Huffman. Combining learning from instruction with recovery from incorrect knowledge. In *Machine Learning Conference Workshop on Agents that learn from other agents.*, 1995. Available from http://ai.eecs.umich.edu/people/douglasp/homepage.html.

[Pearson and Laird, 1996] Douglas J. Pearson and John E. Laird. Toward incremental knowledge correction for agents in complex environments. In Stephen Muggleton, Donald Michie, and Koichi Furukawa, editors, *Machine Intelligence*, volume 15. Oxford University Press, 1996. In press.

[Pearson *et al.*, 1993a] Douglas J. Pearson, Scott B. Huffman, Mark B. Willis, John E. Laird, and Randolph M. Jones. Intelligent multi-level control in a highly reactive domain. In F. C. A. Groen, S. Hirose, and C. E. Thorpe, editors, *Proceedings of the Third International Conference on Intelligent Autonomous Systems*, Burke, VA, 1993. IOS Press.

[Pearson *et al.*, 1993b] Douglas J. Pearson, Scott B. Huffman, Mark B. Willis, John E. Laird, and Randolph M. Jones. A symbolic solution to intelligent real-time control. *Robotics and Autonomous Systems*, 11:279–291, 1993.

[Pearson, 1995] Douglas J. Pearson. Active learning in correcting domain theories: Help or hindrance. In *AAAI Symposium on Active Learning*, 1995. Available from http://ai.eecs.umich.edu/people/douglasp/homepage.html.

[Quinlan, 1986] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[Quinlan, 1990] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

[Rajamoney and DeJong, 1987] Shankar Rajamoney and Gerald DeJong. The classification, detection and handling of imperfect theory problems. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 205–207, 1987.

[Ratcliff, 1990] Roger Ratcliff. Connectionist models of recognition memory: Constraints imposed by learning and forgetting functions. *Psychological Review*, 97(2):285–308, 1990.

[Reece and Tate, 1994] Glen A. Reece and Austin Tate. Synthesizing protection monitors from causal structure. In Kristian Hammond, editor, *Proceedings of the Second International Conference on Artificial Intelligence Planning Systems*, pages 146–151, 1994.

[Richards and Mooney, 1991] Bradley L. Richards and Raymond J. Mooney. First-order theory revision. In *Proceedings of the Eighth International Conference on Machine Learning*, pages 447–451, 1991.

[Rumelhart *et al.*, 1986] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propogation. In *Parallel Distributed Processing*, volume 1. MIT Press, Cambridge, MA, 1986.

[Samuel, 1959] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal on Research and Development*, 3:210–229, 1959.

[Sanborn and Hendler, 1988] J. Sanborn and J. Hendler. A model of reaction for planning in dynamic environments. *Artificial Intelligence in Engineering*, 3(2):95–102, 1988.

[Shen and Simon, 1989] Wei-Min Shen and Herbert A. Simon. Rule creation and rule learning through environmental exploration. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 675–680, Detroit, Michigan, 1989.

[Shen, 1989] W. Shen. *Learning from the Environment Based on Percepts and Actions.* PhD thesis, Cagnegie-Mellon University, 1989.

[Shen, 1994] Wei-Min Shen. *Autonomous Learning from the Environment.* Computer Science Press, W. H. Freeman and Company, New York, NY, 1994.

[Sutton, 1988] Richard S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[Sutton, 1990] Richard S. Sutton. Integrated architectures for learning, planning and reacting based on approximating dynamic programming. In *Proceedings of the Seventh International Conference on Machine Learning*, pages 216–224, 1990.

[Tambe *et al.*, 1995] M. Tambe, W. L. Johnson, R. M. Jones, F. Koss, J. E. Laird, P. E. Rosenbloom, and K. B. Schwamb. Intelligent agents for interactive simulation environments. *AI Magazine*, 1995.

[Tesauro, 1992] Gerald Tesauro. Temporal difference learning of backgammon strategy. In *Proceedings of the Ninth International Conference on Machine Learning*, pages 451–457, 1992.

[VanLehn, 1987] Kurt VanLehn. Learning one subprocedure per lesson. *Artificial Intelligence*, 31(1):1–40, 1987.

[Wang, 1995] Xuemei Wang. Learning by observation and practice: An incremental approach for planning operator acquisition. In *Proceedings of the Twelth International Conference on Machine Learning*, pages 549–557, 1995.

[Wang, 1996] Xuemei Wang. *Learning Planning Operators by Observation and Practice.* PhD thesis, Carnegie Mellon University, 1996.

[Watkins and Dayan, 1992] Christopher J. C. H. Watkins and Peter Dayan. Technical note: Q-learning. *Machine Learning*, 8:279–292, 1992.

[Watkins, 1989] Christopher J. C. H. Watkins. *Learning from delayed rewards.* PhD thesis, University of Cambridge, England, 1989.

[Wray *et al.*, 1996] Robert Wray, John Laird, and Randolph Jones. Compilation of non-contemporaneous constraints. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence*, pages 771–778, Portland, OR, August 1996.