

Toward Incremental Knowledge Correction for Agents in Complex Environments

Douglas J. Pearson and John E. Laird
Artificial Intelligence Laboratory
The University of Michigan
1101 Beal Ave.
Ann Arbor, Michigan 48109-2122
E-mail: dpearson@umich.edu, laird@umich.edu
FAX: (313) 763-1260 Tel: (313) 747-1761

Abstract

In complex, dynamic environments, an agent's domain knowledge will rarely be complete and correct. Existing deliberate approaches to domain theory correction are significantly restricted in the environments where they can be used. These systems are typically not used in agent-based tasks and rely on declarative representations to support non-incremental learning. This research investigates the use of *procedural* knowledge to support deliberate incremental error correction in complex environments. We describe a series of domain properties that constrain the error correction process and that are violated by existing approaches. We then present a procedural representation for domain knowledge which is sufficiently expressive, yet tractable. We develop a general framework for error detection and correction and then describe an error correction system, IMPROV, that uses our procedural representation to meet the constraints imposed by complex environments. Finally, we test the system in two sample domains and empirically demonstrate that it satisfies many of the constraints faced by agents in complex and challenging environments.

KEYWORDS : Machine Learning, Theory Revision, Error Detection, Error Correction, Domain Theory, Procedural Knowledge, Autonomous Agents

1 Introduction

In complex, dynamic environments, an agent's domain knowledge (its *domain theory*) will rarely be complete and correct. To succeed, an agent must have the ability to extend and correct its domain theory as it interacts with its environment. Figure 1 shows the basic processing steps of a planning system that performs error correction. A non-learning system would have only some combination of planning and execution. For learning, the system must be able to detect an error during execution. If an error is detected, the agent must determine which aspect of its knowledge is in error (credit assignment) and then correct that knowledge.

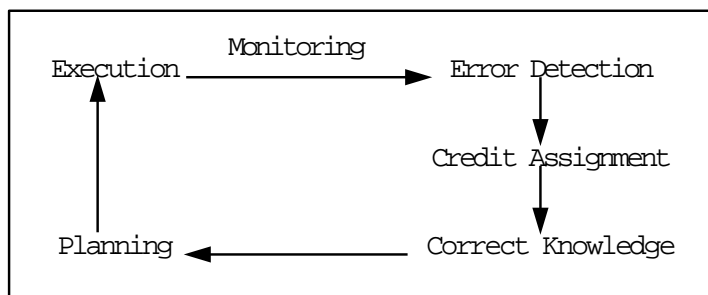


Figure 1: Deliberate Domain Theory Correction

Existing domain theory correction systems use *declarative* representations of operators as their domain knowledge. These representations, in turn, support *deliberate* error correction strategies (e.g. EITHER [Ourston and Mooney, 1990], EXPO [Gil, 1991], OCCAM [Pazzani, 1988], CLIPS-R [Murphy and Pazzani, 1994], FOIL [Quinlan, 1990]). In deliberate error correction, the system detects errors without any special signal from the environment (such as a payoff function, or a training signal), and corrects its internal planning knowledge of its actions by determining which operator was in error and then modifying it. Deliberate systems also typically consider multiple training episodes when making a correction. These approaches can be contrasted with *implicit* approaches for correcting behavior, such as Classifiers [Holland, 1986], Backpropagation [Rumelhart *et al.*, 1986], Q-learning [Watkins and Dayan, 1992], that are restricted to only detecting errors when an additional training signal is provided and are limited to correcting errors in execution knowledge, rather than planning knowledge which restricts them to purely reactive systems. They also consider only one training instance at a time when making a correction which, we will show, makes for less effective learning than considering multiple episodes at once.

Unfortunately, deliberate learners are currently restricted to simple domains and thus the agents are unable to act (or learn or plan) effectively in domains with the characteristics listed in Figure 2. For example, their representation may not be expressive enough to include conditional actions (D2) or actions with duration (D3). Similarly, their algorithms for credit assignment typically assume that sensors are error-free (D5), that immediate feedback is available (D9), and that the target domain theory never changes (D8).

<u>Agent, Domain and Task Characteristics</u>	<u>Examples from Driving</u>
D1: An action may apply in many disjunctive states D2: An action can have conditional effects. D3: An action can have duration. D4: An action may have unobservable effects. D5: A sensor may be noisy. D6: A task may have arbitrary duration. D7: A task may require a timely response. D8: A domain property may change over time. D9: A domain may not have immediate feedback.	Brakes can be applied to slow the car, to stop, or to signal a tailgater. The brakes have different effects depending on the weather conditions. A car does not stop instantly when the brakes are applied. Sudden braking will wear out the car's tires. A motorcycle may appear to be a bicycle, or the speedometer may be broken. A taxi must perform an arbitrary number of picks ups and deliveries. A car must avoid a vehicle that swerves in front of it. A car's tires can wear out, making braking take more time. Pushing on the accelerator may not lead to failure until a sharp turn is reached.

Figure 2: Agent, Domain, and Task Characteristics

There are at least two possible responses to this problem. One response would be to try to extend the current approaches by enriching their representations for action and extending their current techniques. However, there is a danger that as these representations are enriched (in response to D1, D2, D3) and as the number of operators grows, the cost of full first-order access to the preconditions and actions of all operators (as in the “glass box” assumption of PRODIGY [Carbonell *et al.*, 1991]) will be prohibitive during error correction.

We are pursuing an alternative in which we develop approaches to planning, error detection and correction which depend only on limited and efficient *procedural* access to the preconditions and actions of an operator. The agent only has access to preconditions by knowing when they have been satisfied for a particular operator, and can only access the actions by executing the operator. Neither the preconditions nor the actions can be directly examined.

The remainder of this paper has three main sections. The first section presents a representation of domain theories, based on production rules, that meets the requirements in Figure 2. The second section presents an incremental, domain-independent, deliberate error detection and correction system, called IMPROV, that has been designed to meet the constraints of Figure 2. Although IMPROV is implemented within a specific architecture, it is a general method for correcting operator preconditions when the operators are procedurally encoded. The third section presents the results to date, which include that IMPROV corrects its domain theory incrementally, becomes *faster* (as well as more accurate) as it learns, corrects knowledge in a noisy environment, modifies its domain knowledge as the target domain theory changes over time, and makes more efficient use of training instances than pure incremental learners that process only a single instance at a time.

2 Domain Theory Representation

Most deliberate error correction schemes represent their domain theories as sets of independent operators, where each operator corresponds to a different action an agent can perform in the world. These operators are used for planning as well as for execution of actions in the world. 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.

Typically, in deliberate error correction systems, an operator is represented declaratively using a STRIPS-style preconditions and actions, where the actions consist of add and delete lists. This representation is usually restricted so that the operators lack conditional effects (D2), or lack duration (D3), or are assumed to have only conjunctive conditions (D1). Additionally, these systems assume a declarative representation is always available. However, this is not always the case. For example, the precondition for moving a robot arm might be expressed as :

$$\text{if } x^7 + x^2y^5 + xy^2 + 7x - 12 > 0 \text{ then move-arm}$$

As this equation has no analytic solution, it is not clear how to declaratively represent when the operator should be chosen. The agent is limited to executing the expression for particular values, to determine if the precondition is satisfied.

We are exploring the use of a largely procedural representation in an effort to provide the necessary expressiveness (D1-D3) while restricting the access to the knowledge to avoid intractability. The approach also releases us from the assumption that a declarative representation is always available. Our hypothesis, born out in our implementation, is that declarative access to just the names of operators is sufficient for error correction. This type of restricted access is provided by Soar [Rosenbloom *et al.*, 1993], the architecture in which IMPROV is implemented.

In Soar, an operator's preconditions are defined by rules which test for situations in which the operator should be selected, and therefore include control knowledge. Once selected, an operator's actions are performed by "operator implementation" rules which support actions with conditional effects or duration. The sufficiency of this representation for constraints D1-D3 (and efficient matching [Doorenbos, 1993]) has been demonstrated for complex, real-time domains including control of simulated aircraft [Pearson *et al.*, 1993] and tactical air combat [Laird *et al.*, 1995; Tambe *et al.*, 1995].

An IMPROV agent will use this operator knowledge to both plan and behave in its world. Errors in the planning knowledge lead to errors in behavior, so it is planning knowledge that IMPROV learns to correct. In IMPROV, planning does not create a monolithic plan. Instead, situation dependent control rules are learned that will propose the operators when appropriate during execution [Laird and Rosenbloom, 1990]. For example, a possible "plan" to drive through an intersection (where the rules are preconditions for the operators) is shown in Figure 3.

An important 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 by using control rules learned from its most recent planning experience, but as a result of some unexpected change in the environment, control rules from an much earlier planning experience might become relevant and guide the agent. Thus, in a well-trained agent, planning will be the exception, with only enough planning to fill in for novel situations [Laird and Rosenbloom, 1990].

IF distance-to-intersection(medium) AND light(red) AND engine(running)	IF distance-to-intersection(medium) AND light(yellow) AND engine(running)
THEN choose operator(set-speed 10)	THEN choose operator(set-speed 10)
IF distance-to-intersection(close) AND light(red) AND engine(running)	IF distance-to-intersection(close) AND light(green) AND engine(running)
THEN choose operator(stop)	THEN choose operator(set-speed 30)

Figure 3: Precondition rules forming a procedural plan

To create a plan, IMPROV does not search through a space of plans because of the restrictions it has on access to its operators. IMPROV could potentially use a variety of state-space planning methods. However, because we expect that the search can be usefully biased by prior experience, and because the method should be complete, we’ve adopted a planning method called *Uncertainty-Bounded Iterative Deepening* (UBID)¹. UBID is an extension of the iterative deepening planning method. UBID bounds an iteration based on the uncertainty associated with a plan, rather than its length. The *uncertainty* of a plan is the sum of the uncertainty associated with each operator in that plan. These uncertainties are derived from the learning component of the system and essentially represent the number of times an operator was useful in similar situations. The details of the planning method are not important. 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. Additionally, UBID is complete, which guarantees that during error correction IMPROV will eventually discover the correct path to the goal, if one exists.

3 Domain Theory Correction

To perform domain theory correction, we must attend to four issues: (1) which knowledge errors will be corrected, (2) how these errors are detected, (3) how the operator(s) responsible for the error are determined (credit assignment), and (4) how the error is corrected.

3.1 Knowledge Error Types

As the agent’s domain theory consists of the preconditions and actions of its operators, the range of possible errors are defined in terms of that operator knowledge. An operator’s preconditions may be overgeneral (missing conditions that should be present) or overspecific (containing unnecessary conditions which should not be present).² The actions may

¹UBID was developed in collaboration with Scott Huffman

²The situation where a precondition is just plain incorrect can be seen as a combination of an overgeneral condition (since the correct condition is missing) and an overspecific condition (as the incorrect condition is present). IMPROV also corrects these errors.

similarly contain additional effects, which do not occur when the operator is executed, or be missing effects which do occur. Finally, it is possible for an operator to be completely missing.

Currently, IMPROV is guaranteed to converge to the correct knowledge for both over-specializations and overgeneralizations of operator preconditions.³ Thus, the emphasis of the remaining parts of this section are on how to identify and correct such errors. How this approach extends to correcting operator actions is discussed in the section on future work.

3.2 Error Detection

Errors can either be detected during planning or during execution. Errors during planning cannot be detected without additional knowledge, for example that certain states are impossible. Without this extra knowledge the agent can only detect errors in its planning knowledge during execution. In some way, the agent must compare its planning knowledge to the execution and detect when they are inconsistent.

Most, if not all, deliberate error correction systems make this comparison directly, by comparing the declarative model of the operator to the results of execution. That is, before the next operator in the plan is applied, its preconditions are checked and if they are not satisfied, an error is assumed to have happened earlier in the plan. Similarly, after an operator is applied, its actions are checked and if the expected changes are not found in the external state, an error is assumed.

The comparison cannot be made directly when actions have unobservable effects (D4) or when the system is restricted to only procedural access. An alternative might be to just monitor if the current plan is being followed. However, in “distributed” plans, such as IMPROV uses, there is no single plan for a given goal. Instead there can be collections of rules from many planning episodes that contribute control.

As a result, IMPROV relies on detecting that the agent no longer knows how to make progress towards the goal. More precisely, if at any point the agent reaches a state where no operator is proposed for the current goal, an error is assumed. The comparison between planning knowledge and execution is therefore made indirectly, through the rule matcher. Consider the example of driving a car through an intersection using the plan shown in Figure 3. If the light turns red as the agent approaches then the agent will execute `set-speed(10)`, `stop`, `set-speed(30)` (as the light turns back to green). 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 (namely `engine(running)`). As no other operator will be proposed, this will be recognized as an error.

This method only detects errors if 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. In complex, nondeterministic worlds, 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),

³Assuming the agent’s representation is sufficient to represent the correct operator preconditions, that actions do not destructively modify the environment and that a solution path exists.

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.

The above approach fails if the agent inadvertently cycles back to an earlier part of a 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 still has not changed gear. This problem is complicated by the fact that the agent may move to a state in which the only differences are irrelevant to the current task (the pedestrians on the sidewalk have moved). The key is to recognize that the task-relevant parts of the state have been repeated. For the purposes of error detection, states are grouped into equivalence classes; two states being equal if the same operator precondition matches both states. Loops are detected if the agent returns to a state in a previously visited equivalence class. IMPROV efficiently calculates these equivalence classes by explaining, at each step during execution, why the next operator in the operator is being selected. This explanation leads, through Soar’s EBL method *chunking* [Laird *et al.*, 1986], to a new rule being learned which matches exactly the precondition of the operator being chosen. Whenever one of these rules fire, a cycle is detected. This method allows the same operator to be selected multiple times during the course of a goal, as long as different preconditions are used to select it each time. Iterative tasks are still possible, even when there is no observable change in the external state (such as turning a finely threaded screw), as long as there is a change to the internal state (such as a counter).

3.3 Credit Assignment

Once an error is detected, the agent must determine which aspect of its domain knowledge was in error; that is which operator is incorrect and how it is incorrect. Because it is not always possible to detect an error immediately following a mistake, the agent does not know exactly which operator is incorrect. Even, if a specific operator has been identified, it is still a problem to identify in what way the preconditions of the operator are wrong, that is, which preconditions should be added, or which preconditions should be dropped. This is especially true when the domain state contains a large number of irrelevant features, and when the preconditions themselves are large disjunctive sets (D1).

3.3.1 Identifying which operators are incorrect

Since most existing deliberate error correction systems assume explicit monitoring combined with complete and immediate sensing, they can assume that the current operator is the one that failed. IMPROV’s approach is to compare the failed attempt’s operator sequence and a successful solution, and use the differences to determine the operators that are in error. Specifically, once an error is detected, IMPROV continues to attempt to solve the problem, by generating and executing a series of plans until it eventually succeeds⁴. Once a successful plan has been executed, IMPROV recalls for each state in the successful plan

⁴If actions do not destructively modify the world, UBID’s complete planning method ensures a successful plan will be found, if one exists.

the operator that would have been chosen in the original (incorrect) plan, and compares it to the operator used in the successful plan. The differences are assumed to be the operators with incorrect knowledge. For example, if the incorrect plan for crossing an intersection is `set-speed(10)`, `stop`, `set-speed(30)` (when the light changes) and the successful plan turns out to be *change-gear(down)*, `set-speed(10)`, `stop`, `set-speed(30)` then the conditions for *change-gear* and `set-speed(10)` should be changed. *Change-gear* should be generalized to be used in this task, while `set-speed 10` should be specialized so it is not chosen until after the car is in a lower gear. This comparison can be achieved without a declarative operator representation. The sequence of states that make up the successful plan are recorded as the agent executes the plan. IMPROV then observes which operators are proposed (i.e. have matching preconditions) for each state in turn and compares them to the operators that made up the successful operator sequence.

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 learning typically assigns most blame for a failure to the final step (`set-speed 30`, in the example). It is very difficult for such a system to discover that the true correction is much earlier in the plan, while still maintaining the final `set-speed(30)` operator, which is required in any successful plan.

3.3.2 Identifying errors in operator preconditions

Once incorrect operators have been detected and a successful plan has been found, the system can attempt to identify whether the operator preconditions must be either specialized or generalized. IMPROV's method for detecting errors gives an indication whether the operator needs to be specialized (because it was incorrectly selected for a given situation) or generalized (because it was not selected for a given situation). However, unless the system has access to some additional source of knowledge, such as a complete causal theory, it is impossible for the system to determine deductively exactly which features need to be added or dropped. Instead the system must rely on its accumulated experience and inductively guess. As stated earlier, we assume that there are several constraints on this process. First, it must be incremental because there may be arbitrary numbers of training instances (D6) and there are temporal constraints on the agent's behavior (D7). Second, it must be resistant to noise (D5), but also be able to adjust to changes in the underlying domain (D8). Previous deliberate approaches violate some or all of these constraints.

Our approach is to train an incremental inductive category learner on instances of operators succeeding or failing. The category being learned is which operator is the correct one to use for the current state and goal. 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. As the inductive

Main Control Loop

```
do
  O := Compute-Proposal(G,S)
  result := Execute(O)

  if (result = error-detected) then
    Make-Correction(G,S)
  end

while (result ≠ reached-goal)
```

```
Procedure Make-Correction(G,S)
begin
  /* Search for the correct plan */
  while (current-state ≠ G)
    Pi := UBID(G,current-state)
    do
      Oij := Next-step-in-plan(Pi)
      Sij := current-state
      result := Execute(Oij)
      if (result = error-detected) then Ri := error-detected
      if (result = reached-goal) then {Ri := reached-goal ; success = i }
      while (result ≠ error-detected and result ≠ reached-goal)

    /* Identify key differences between correct and incorrect plans */
    /* Train inductive module */
    foreach Sj in Psuccess
      important-features = Differences(S1j,S2j,S3j,...,Ssuccess,j)
      for i := 1 to success
        if (Ri == error-detected) then
          Train-SCA-negative-instance(G,Sij,Oij,important-features)
        if (Ri == reached-goal) then
          Train-SCA-positive-instance(G,Sij,Oij,important-features)

    /* Correct the operator precondition knowledge */
    foreach Sj in Psuccess
      O-current := Compute-Proposal(G,Sj)
      O-new := Test-SCA(G,Sj)
      if (O-new ≠ O-current) then
        Learn-Reject-Operator(G,Sj,O-current)
        Learn-Propose-Operator(G,Sj,O-new)

end
```

Figure 4: Basic algorithm for correcting operator preconditions

learner must be incremental and able to represent disjunctive sets of preconditions (D1) we have used the symbolic category learner SCA [Miller, 1991; Miller, 1993], which can learn arbitrarily complex categories⁵. SCA is also incremental, tolerant of noise and is guaranteed to converge to the correct category, if that category can be represented in its description language.

The full correction algorithm is summarized in Figure 4. For simplicity in understanding the method, the algorithm is described using iteration, however it is implemented using several levels of recursion because the plans are not represented declaratively. Similarly, many of the data structures are not explicitly recorded in the agent's working memory, but are instead recorded as procedural rules which recall the correct values into working memory at the appropriate stages of the algorithm. However, these distinctions are not critical to understanding the important aspects of the algorithm.

During a correction episode, IMPROV searches for a correct plan, $P_{success}$. This is done

⁵In the limit SCA can represent each exemplar individually [Miller, 1991].

by generating a series of plans, P_i , using the UBID planning method mentioned earlier. UBID generates plans in decreasing order of probability of successfully reaching the goal. Each plan is tested by recording the current state, S_{ij} , and then executing each operator, O_{ij} , in turn. The operator will either lead to an error, the goal or to neither. If either an error or the goal is detected, the plan is complete and the result recorded as R_i , otherwise IMPROV continues with the next operator in the plan. Once IMPROV finds a successful plan, the inductive learner, SCA, is trained on all of the instances, positive and negative, found during the search for a successful plan (see Section 3.3.2). This training is biased by the features found to be different between the failed plan states and those in the successful plan (as described below). Finally, the operator’s precondition knowledge is updated to match the results of the induction (see Section 3.3.3).

IMPROV does not train the inductive learner as soon as an instance of an operator leading to an error has been found. Instead, the instances are recorded until a successful plan is found, at which point the whole set of positive and negative instances are passed to the induction method. Introducing this delay before learning allows IMPROV to more accurately identify which operators are incorrect (as described in Section 3.3.1) and, as we’ll show, assists in identifying the important features for training the inductive learner.

Any incremental method 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. We will help to clarify this property of the system with a simple example. Consider an agent which tries to cross an intersection and fails (see Figure 5(a)).

(a) Attempt One : Set-speed(30)	Speed(40)	My-Vehicle(Car)	Other(Ambulance)	Light(Green) -> Failure
(b) Attempt Two: Set-speed(30)	Speed(40)	My-Vehicle(Car)	Other(Car)	Light(Green) -> Succeeds

Figure 5: Identifying the important features

An incremental inductive learner training on just this first instance would probably not correctly identify the cause of the failure. However, if the learner waits and considers both instances together, it is clear that the failure was due to the fact that the other vehicle at the intersection was an ambulance (and therefore should have been given precedence). This determination can only be made in the context of the two instances. IMPROV takes advantage of this general property by delaying its training until a successful instance has been found. As the number of instances IMPROV considers during each training episode will not increase over the life of the agent, the learning is still incremental, satisfying the constraints D5 and D6. It should be noted that this information simply forms an additional bias for the inductive learner⁶.

⁶We assume that some of the properties in the domain’s representation are causally related to the success of the operator (for example, speed and the types of vehicles involved). An alternative representation, such as a graphical one based on polygons, would not have this property and this bias would not assist, and could hinder, learning.

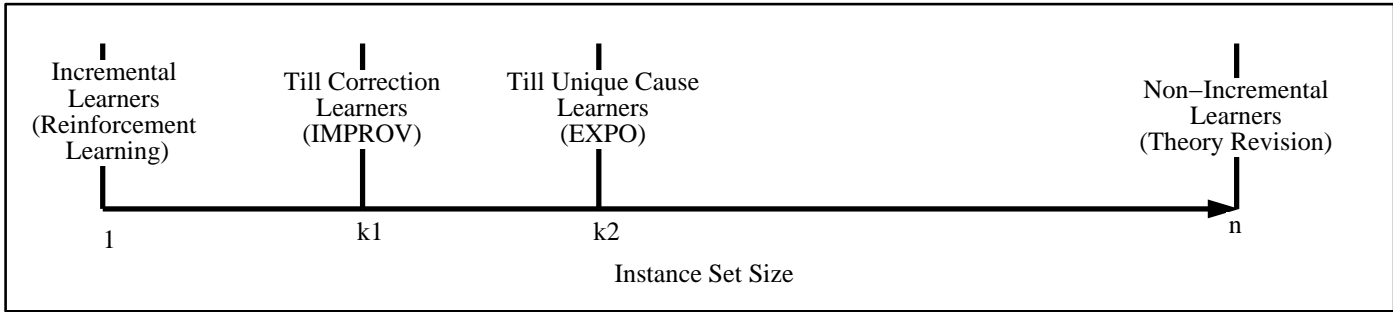


Figure 6: Range of systems based on instance set size

If training is delayed even further, until a unique cause has been identified, then this would result in a system which is actively experimenting (such as the simple induction in EXPO[Gil, 1991]). Thus IMPROV and EXPO define two interesting, k -incremental learners along the spectrum from pure incremental learners to pure non-incremental learners (Figure 6). 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 means the agents must be more careful in their learning. To continue our example from above: a system which did not delay its credit assignment might have attributed the cause of the failure (Figure 5(a)) to the light being green. This incorrect learning would then make it much harder for the system to discover its error in future as it would expect the correct plan (Figure 5(b)) to fail, making it unlikely to execute it in the environment. A passive learner, however, would be no more or less likely to be presented with the second plan, no matter what it learned as a result of seeing the first plan. This would allow the passive learner to more easily recover from any early, incorrect learning. The important point here, is that in agent-based systems which act in the world, it is more important to assign credit correctly than it is to assign credit quickly (as we will demonstrate in our experimental results).

3.3.3 Changing the Precondition Knowledge

When the operator is represented declaratively, the correction system can directly modify the operator knowledge. As knowledge in IMPROV is represented procedurally, it cannot be modified directly, as this would violate the restriction of only accessing the knowledge through execution. IMPROV therefore corrects the knowledge by learning additional rules that correct the decision about which operator to select. A rule is learned to reject the original (incorrect) operator and another rule is learned which suggests the new (correct) operator. This approach is explained in detail in Laird 1988 [Laird, 1988].

<u>Attributes</u>	<u>Number of possible values</u>	<u>Sample Initial Knowledge</u> (+3x2)	<u>Sample Target Knowledge</u>		
Distance-to-Intersection	5	Close	Close	Close	Close
My-vehicle-isa	5		Car		
My-vehicle-color	5				
My-road-sign	1	Signal	Signal	Signal	Signal
My-light-color	3		Red	Green	Green
Gear	4				
Weather	5				
Road-Quality	5				
Other-road-user-isa	5			Police	Ambulance
Other-sidewalk-user-isa	5				
Other-road-sign	1				
Operator :		set-speed (30)	set-speed(0)	set-speed(0)	set-speed(0)

Figure 7: Driving Domain Theory Examples

4 Results

IMPROV has been implemented in two test domains. The first was the toy domain of a robot in the blocks world, which provided a good initial test-bed during development. The second domain is a driving simulation, which allows us to test all of the constraints (D1-D9) which we used to characterize complex environments.

The task, for the agent in the robot domain, is simply to align two blocks on a table. The blocks have different characteristics and the agent must learn which of the blocks can be successfully picked up. The task for the agent in the driving domain is to successfully cross an intersection. There are other agents in the environment, both on the roads and sidewalks, and processes (such as the traffic light) which change independent of the agent’s actions. Figure 7 shows an example of a learning experiment. The attributes considered during learning are shown on the left, along with the range of values each attribute can take. Then the initial knowledge given to the agent is shown next (i.e. that the set-speed(30) operator should be chosen when the distance-to-intersection is close and the road-sign is a traffic signal). Finally, an example of a target theory is shown in the third column. 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.

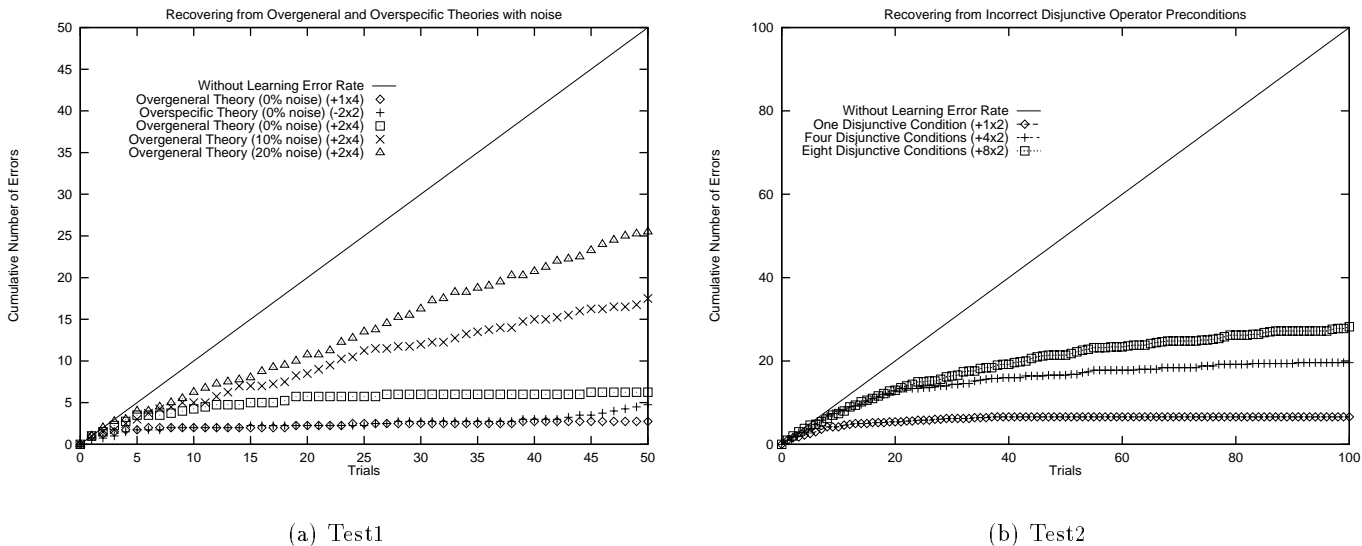


Figure 8: Recovery from different types of incorrect domain knowledge

4.1 Test 1: Overgeneral, Overspecific, Noise

Figure 8(a) shows the cumulative number of errors made by IMPROV over the course of 50 trials, for a range of target theories. The diagonal line is a reference, showing the number of errors that would be made if the system was not learning. The zero noise cases show that IMPROV can correct overgeneral or overspecific theories and quickly converges to the correct theory. The graph also shows that learning can tolerate noise⁷, although performance degrades.

4.2 Test 2: Learning Disjunctive Preconditions

Figure 8(b) shows the cumulative number of errors made as IMPROV learns a range of theories which include an increasing number of disjunctive terms. Disjunctive preconditions can present difficulties for some traditional learning methods and one of our constraints (D1) is that the theory may contain a number of disjunctive terms. This graph demonstrates that learning is harder as the number of disjuncts increases, but in each case IMPROV quickly converges towards the correct knowledge.

⁷10% noise indicates there is a 10% chance that a given attribute is incorrectly sensed for the duration of that trial.

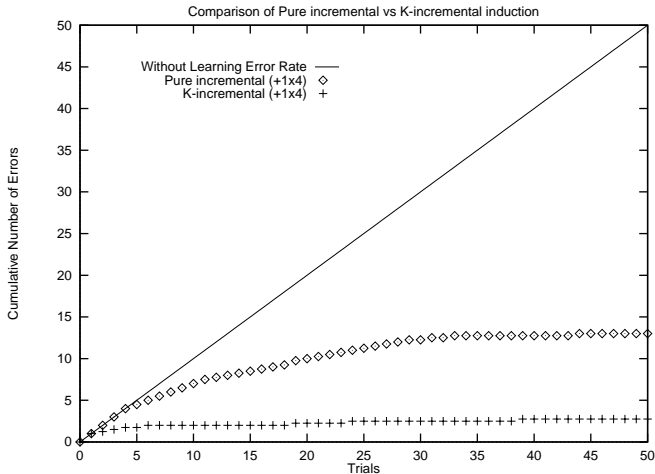


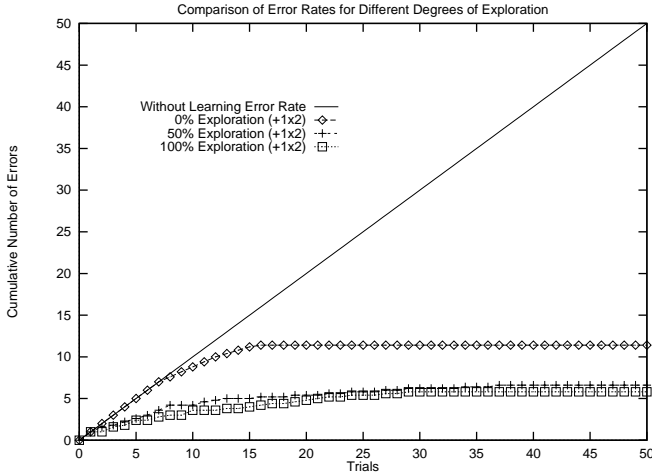
Figure 9: Test 3 – K-incremental learning vs Pure Incremental Learning

4.3 Test 3: Pure Incremental vs K-Incremental

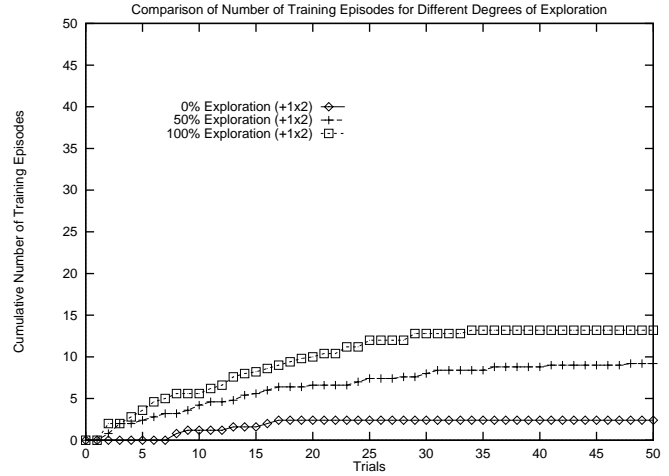
Figure 9 demonstrates the benefit of using a deliberate correction strategy that considers sets of k instances during training over a pure incremental learner. IMPROV was modified to train immediately after seeing each individual instance, rather than waiting and training on a set of instances, and thereby simulated a pure incremental system. The difference in the resulting error rates is substantial, confirming that the accuracy of the learning has been improved because the system delays its training until a successful plan has been found. This enhances the ability of the system in both credit assignment problems (which operators are wrong and what’s wrong with those operators).

4.4 Test 4: Exploration vs. Exploitation

Learning systems are often faced with the decision of how much to explore in an effort to gain new (and hopefully better) knowledge and how much to trust the knowledge they already have. This issue arises for IMPROV in domains with destructive modification, such as driving, during re-planning. In essence, the agent must decide if it has already, unsuccessfully, tried an action in *the same state* before. For example, the agent selects set-speed(30) at the point of reaching the intersection only to discover it later leads to an error. When the agent next comes upon an intersection, should it select set-speed(30) again? We tested three different approaches to resolving this question. First, the 0% exploration case, we required a complete match between the two states to avoid choosing the operator which lead to a failure again. Secondly, the 100% exploration case, we always selected a different operator over an operator which had been seen to fail before. Thirdly, the 50% exploration case, we required a partial match (half of the features or more must match between the two states) to avoid trying an operator which had previously failed. Figure 10(a) shows the error rates of the different approaches. Figure 10(b) shows the number of training



(a)



(b)

Figure 10: Test 4 - Performance of different exploration strategies

episodes (i.e. number of sets passed to SCA) that occurred for the different approaches. 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 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 case is based on only a small set of instances and therefore results in poor quality learning. The 50% exploration case produces the best overall results, with the lowest learning overhead for the same performance curve as the 100% exploration case. Thus the 50% exploration rate is the one used in the other trials.

4.5 Test 5: Tolerance of an Evolving Domain

Figure 11(a) illustrates that the induction is robust to changes in the target domain theory, for example when braking takes longer due to the wearing out of tires. In this test, two of the conditions in the target theory are changed to new values after the system has seen 50 trials. This causes the system to make more errors, but it then adjusts to the new theory. The baseline case shows the system's behavior when no change is made.

4.6 Test 6: Speeding up during Learning

Figure 11(b) shows the CPU time per trial over the life of the system while performing the previous evolving domain correction task. The spikes indicate when a correction had to be made. It should be noted that the time spent on each correction remains constant or decreases as the system learns and the theory becomes more complex. This is in contrast

to most machine learning algorithms, which become slower as the theory increases in complexity. Also, the frequencies of corrections decreases as the system’s theory becomes more accurate. These results help support the hypothesis that procedural representations lead to an efficient correction method.

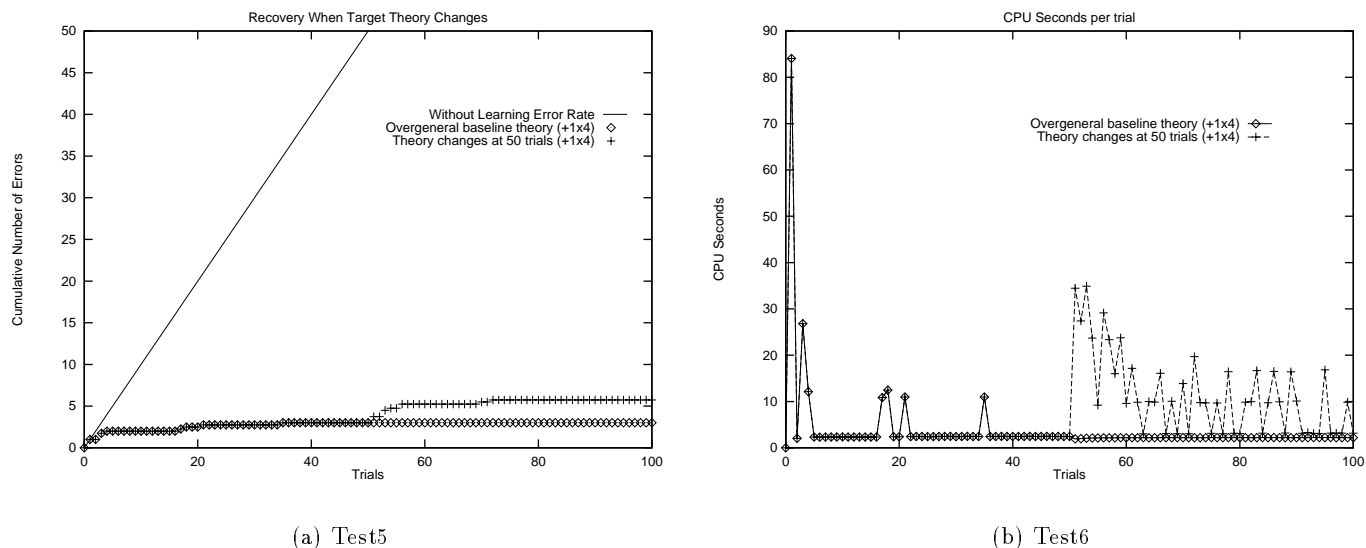


Figure 11: Tolerance of evolving domain and speed up during learning

5 Conclusions and Future Work

IMPROV can support an agent that learns, plans and executes its plans in the environment. We have shown that its deliberate, k-incremental correction strategy leads to better learning than implicit, pure incremental methods. It does this while using an expressive procedural representation that has been demonstrated to be effective in tasks that require complex domain knowledge. We have also developed a general framework for error detection and correction and discussed the contrasts between different approaches within this framework.

IMPROV’s correction method has been designed to work in complex and challenging domains, so it has been constrained in ways that mean its performance on simple domains will be slightly worse than that of existing correction systems. For example, its k-incremental learning method has access to less information, and therefore should be inferior, to any non-incremental algorithm. However, the learning cost in IMPROV is not proportional to the size of the theory or the size of the instance set which leads to better performance in time critical domains.

In future work, we will extend IMPROV to correct operator actions. Our approach will be to represent operators hierarchically, with sub-operators implementing super-operators (for example, the set-speed operator’s actions become a series of more primitive operators,

press-accelerator, press-brake and change-gear). This allows the problem of correcting the actions of a super-operator to be reduced to the problem of correcting the preconditions of a sub-operator which implements it. This will allow us to re-use IMPROV's precondition correction method in a recursive fashion to correct either operator preconditions or actions. The recursion will eventually terminate in a layer of primitive operators whose actions consist of modifying a single symbol and therefore can be guaranteed to be correct. This will allow all corrections to be reduced to the problem of correcting operator preconditions.

References

- [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.
- [Doorenbos, 1993] R. Doorenbos. Matching 100,000 learned rules. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*. AAAI, AAAI Press, 1993.
- [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.
- [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.
- [Laird and Rosenbloom, 1990] J. E. Laird and P. S. Rosenbloom. Integrating execution, planning, and learning in Soar for external environments. In *Proceedings of AAAI-90*, July 1990.
- [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.*, 1995] J. E. Laird, W. L. Johnson, R. M. Jones, F. Koss, J. F. Lehman, P. E. Nielsen, P. S. Rosenbloom, R. Rubinfeld, K. Schwamb, 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.
- [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.
- [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.
- [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, 1988] Michael J. Pazzani. Integrated learning with incorrect and incomplete theories. In *Proceedings of the International Machine Learning Conference*, pages 291–297, 1988.
- [Pearson *et al.*, 1993] D. J. Pearson, S. B. Huffman, M. B. Willis, J. E. Laird, and R. M. Jones. A symbolic solution to intelligent real-time control. *Robotics and Autonomous Systems*, 11:279–291, 1993.
- [Quinlan, 1990] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5(3):239–266, 1990.

- [Rosenbloom *et al.*, 1993] P. S. Rosenbloom, J. E. Laird, and A. Newell. *The Soar Papers: Research on Integrated Intelligence*. MIT Press, 1993.
- [Rumelhart *et al.*, 1986] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. In *Parallel Distributed Processing*, volume 1. MIT Press, Cambridge, MA, 1986.
- [Tambe *et al.*, 1995] M. Tambe, W. L. Johnson, R. M. Jones, F. Koss, J. E. Laird, P. S. Rosenbloom, and K. Schwamb. Intelligent agents for interactive simulation environments. *To appear in AI Magazine*, 1995.
- [Watkins and Dayan, 1992] Christopher J. C. H. Watkins and Peter Dayan. Technical note: Q-learning. *Machine Learning*, 8:279-292, 1992.